CS 3431 Database Systems Notes

Christopher Myers

June 13, 2020

Contents

1 Intro

Database systems are systems designed to store, index, and retrieve data according to the queries and commands given to it. In practice, this usually means that the database is disk-based, since most databases are too big to fit into memory, and that the database is designed to be persistent across instances of the programs and users that need the data. The overwhelming majority of online services rely on databases to serve their many users, whether that means banks, airlines, universities, or even bio-informatics. Regular files are usually not used directly because a good database engine (or database management system, DBMS for short) has features that can enhance the usability, speed, and storage usage of the system. For example, a sophisticated database might provide a faster way to search through records using some form of indexing or hashing, while using compression to ensure that redundant data isn't stored.

Common features include:

- Data constraints (e.g. make sure numbers are entered into a field)
- Compression
- Query execution planning (for efficiency)
- Caching (?)
- Relationships between data
- Concurrent multi-user access
- Access control
- Crash recovery[1] & transaction rollback

Usually, databases are interacted with using a query language called SQL, or **S**tructured **Q**uery **L**anguage. SQL is broken down into four different parts: DML (Data Manipulation Language), DDL (Data Definition Language), DCL (Data Control Language), and TCL (Transaction Control Language). Unlike most other common programming languages, SQL is a non-procedural language; procedural languages allow the user to specify what data is required and how to get it, while non-procedural languages like SQL leave the "how" part up to the database engine. SQL is designed to run on a relational database, the most common type of database today — these databases are based on tables with data contained in the rows, all bound together by relations between the tables.

2 Conceptual Design & ER Models

Database designs start with a conceptual design, or something that shows the relationships between different data that needs storing, entities that might contain that data, and the relationships between them. More specifically, you need an ER diagram, or **E**ntity **R**elationship diagram, which often has some crossovers with UMLİn ER diagrams, databases are modeled as collections of entities and the relationships between them, where an entity is an object that exists and is distinguishable from other objects. Entities have attributes (properties with a domain), and entity sets are just groups of entities. Attributes can be broken down into different types — primitive attributes are usually things that store a single value (numbers, strings, booleans, dates, etc.), while composite attributes can be divided into sub-attributes, where each sub-attribute is a primitive. Derived attributes are attributes computed from others, and as a super-type of attribute, you can have multi-value attributes, where an attribute has many values of some given type, be it primitive, composite, or derived. Naturally, there are different types of relationships possible. In general, relationships are any kind of association or connection between entity sets, and a relationship type is a class of relationships. Usually, relationships are represented using a diamond along the connection line between entities in an ER diagram.

[1] Note that this might be a little redundant depending on the use case, modern journaling filesystems generally ensure that all operations are atomic

Consider a simple example of suppliers, consumers, and products. Suppliers have a name and location, products have a name and number, and consumers have a name and location. Some suppliers have established contracts to supply a certain product to a particular consumer for specially negotiated prices at some given quantity. What you end up with is three entities and two relationships: suppliers, consumers, and products are each entities, and there are relationships between suppliers & products and products & consumers. The original attributes (name, location, number) are defined as primitive attributes. This relies on binary relationships (i.e. the relationship involves only two entities).

Relationships can also be recursive, where a relationship exists between an entity and the same entity. For example, you might have a "Part" entity that has sub-parts that are also Parts. That means that any given Part could play either the role of a sub-part *or* a super-part, and that a whole lineage (or tree, if it's a one-to-many relationship) of Parts could be established.

2.1 Keys and Constraints

To facilitate these relationships, databases usually rely on keys, or attributes of the same type that two entities share. For example, a flight might have a flight number, and an airplane ticket would also have a flight number, so you could use the flight number as a key to link tickets to flights. A super key of an entity set is a set of one or more attributes whose values uniquely determine each entity. A candidate key of an entity set is then a minimal super key, i.e. a key with the fewest number of attributes. Someone's full name might be a super key, for instance, but if their SSN is also a super key, it would then be a candidate key. So, in actual database software, choose a candidate key and call it a "primary key". Primary keys don't have to be globally unique, just locally unique to that entity set. Only primary keys are modeled in ERD.

Any entity set that does not have a primary key is referred to as a weak entity, as it totally depends on another entity. These entities must be bound to other entities as they don't have a reason to exist otherwise. For example, in a life insurance database, a registrant's beneficiaries could be considered weak entities, since they have no reason to exist if the registrant does not exist in the database.[2] Weak entities have a discriminator, or partial key, that uniquely identifies it given its identifying entity; the primary key of a weak entity set is then the composition of the primary key of the identifying entity set plus the weak entity set's discriminator.

When establishing relationships, be careful to eliminate redundancies. Instead of storing separate type data for entities that tend to share the same data, for instance, create a Type entity and give it a relationship to the original entity.

3 The Relational Model

The relational model is the way a conceptual design is turned into an actual database. The set of allowed values (...data types) for an attribute is called the attribute domain. Tables are also known as "relations". An instance is the current schema plus the current tuples in the relation, where the tuples are contents of the columns (attributes). Instances frequently change as new tuples are inserted or existing tuples are deleted or updated. Relations have arity (or degree, the number of attributes it has) and cardinality, or the number of tuples in the relation.

Note that relations are unordered — tuple order is not important and column order is not important, so relations should be thought of as a set (or bag) of tuples, not a list. Keys still apply here: super keys are a subset of attributes that uniquely identify each tuple, candidate keys are minimal super keys, and primary keys are still any one of the candidate keys. Once again, there can only be one primary key. If a relation has a primary key, it is a set and not a bag, since it implies there is no duplication.

To create these relations, SQL is used. In SQL there are many data types, such as:

- char (n) — fixed length character string
- varchar2(n) — variable length character string with maximum size

[2] In such cases, it is often useful to add the ON DELETE CASCADE setting, so the database will automagically delete them when their associated entity gets vaporized.

- int — integer
- Real — floating point number
- Number (p,s) — number with precision p and scale s

Tables can be dropped or altered like so:

```
DROP TABLE ExampleTable;
ALTER TABLE ExampleTable ADD DemoRow varchar(127);
ALTER TABLE ExampleTable DROP COLUMN DemoRow; --column drop not allowed by most DBMSs
```

SQL allows you to set integrity constraints, which must be true for any instance of the database. There are three types of key constraints: primary key (must be unique, referenced by foreign keys), unique (cannot have duplicates in the system), and foreign key (must reference some entity's primary key value). An example of this is shown below.

```
CREATE TABLE STUDENTS (
   sid char(20) PRIMARY KEY,
   name char(20) NOT NULL,
   login char(20) UNIQUE,
   age integer,
   gpa real DEFAULT 0
);
```

This demonstrated table creation and a few more constraints. Note that *everything* is null by default until otherwise specified. Not shown above is a value constraint that allows you to control the actual data input, e.g. restrict a "grade" attribute to A, B, or C.

Foreign keys are a little more tricky. These keys state that their value must match the value of the specified primary key in some entity. If the foreign key doesn't match some entity's foreign key, an integrity violation exception will occur and the value will be rejected. This sort of behavior is useful when establishing relationships between entities, and can even be used to guide the database into removing other data. For example:

```
CREATE TABLE Node (
   id char(32) PRIMARY KEY
   posx int,
   posy int
);

CREATE TABLE Edge (
   src char(32),
   dst char(32),
   FOREIGN KEY (src) REFERENCES Node(id) ON DELETE CASCADE,
   FOREIGN KEY (dst) REFERENCES Node(id) ON DELETE CASCADE,
   CONSTRAINT UNIQ_EDGE(src,dst)
);
```

In the above relation, Edges link Nodes together and cannot exist without corresponding Nodes. In the event that one of the Nodes is deleted, the edge entity will be deleted. Alternatives to cascade deleting include setting values to null or simply setting "ON DELETE NO ACTION".

To insert data, more SQL statements are used, namely INSERT (insert an entire record), DELETE (to delete an entire record), and UPDATE (to modify an existing record).

3.1 Transferring ER Schema to Relational Schema

Transferring an ER schema to a relational schema is usually fairly easy:

- Entities are encoded as tables
- The number of columns to each table will correspond to the number of attributes to an entity
- Column names correspond to attribute names
- Many-to-many relationships are turned into tables (sometimes called junction tables)
- For other relationships, if a relationship has a key, it maps to a separate table. Otherwise, it doesn't.
- One-to-many relationships have a primary key on the "one" side and a corresponding foreign key on the "many" side.
- Recursive relationships can be implemented by applying the one-to-many rule
- If a relationship lacks a key, it doesn't get turned into a table, its attributes just get absorbed into one of the entities involved.

4 Relational Algebra

4.1 Basics

Relational algebra is a system of mathematics based on set operations that work on sets or bags. The basic operators are:

- Set operations $\cup \cap -$,
- Select σ
- Project π
- Cartesian project $\times$
- Rename ρ

In the end, query compilation, optimization, and execution all rely on relational algebra. Note that the basic set operations (union, intersection, and difference) only apply to union-compatible relations. Relations are union compatible if they have the same attributes with the same types. Since sets cannot contain two of the same item, a union between two identical sets will result in the original set. The difference over the sets just subtracts common elements, and intersection returns only common elements. Selection will select elements from a relation based on some condition, where the condition is put in a subscript (note that conditions can be very complex). The project operator takes the form of $\pi_{A1,A2,A3...An}(R)$ and returns all tuples in R, but only for the attributes given. Although unsophisticated, this operator can be used to rename commands. The renaming operator ρ simply renames relation names or attributes. The Cartesian product is a little more complicated. During computation, it appends the first row of the first relation to *all* the rows of the second relation, then repeats that process for all the rest of the rows in the relation. Cartesian products are frequently used when joining tables together.

Operators can be used together to build complex expressions. For example, you might use the select operator on a Cartesian product to join two relations together, but only when a certain condition holds (ex. one relation's foreign key matches another's primary key).

4.2 Joins & Grouping

Suppose you have two tables R and S, and you want to join them on some condition. Two operations relate to this, natural join and theta join.

Natural join, or $\bowtie$, can be used to join two tables together on basis of specified attributes with equal values (in SQL this would be an inner join). This is equivalent to taking the Cartesian product, selecting rows from it based on equal values, and projecting the results. Note that this can result in the same instance

of one row being joined to multiple rows from the other table; this is normal and is an effect of the Cartesian product.

The alternative to natural joins are theta joins, which take the cross product of two tables (?!!) with a given condition. These are written as $\bowtie_C$ where C is the condition, and are logically equivalent to $\sigma_C(R \times S)$. Although it might seem like it, this is *not* equivalent to an inner join. There's another operator that can be useful to express complex queries on multiple lines: assignment, or $\leftarrow$.

The grouping operator γ can be used to group rows together based on attributes with the same value and an operation to perform on them. For example, you could go through a list of loans owned by customers and group them together by customer name, summing their balances together.

Generally speaking, it's a good idea to minimize usage of the Cartesian product, since it's a very expensive operation. It's also a good idea to select *before* performing any other operation, since select can trim down the data set that later operations have to perform. For example, although it's valid to project, then select, it's a good idea to select before projecting since the project operator will have less work to do.

5 Structured Query Language (SQL)

SQL is a language designed around working on relational databases, which are based on the same concepts of tables, relations, keys, and the various operations from relational algebra.

5.1 Basic Queries

DML is the section of SQL dedicated to insertion, deletion, and updating of data. The main keywords are SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BYThe most basic query is a simple "SELECT ... FROM ..." statement, which will return a list of attributes from a given table. This can be modified and enhanced by applying qualifiers, such as below:

```
SELECT * FROM Student WHERE sName='Greg' AND address='320FL';
```

This will return all attributes from the Student table where the attribute "sName" is "Greg" and the address is "320FL". SQL also supports the renaming of tables as needed:

```
SELECT ID AS StudentID FROM Student WHERE sName='Greg';
```

Sometimes direct values can be assigned and values can be concatenated:

```
SELECT ' Name: ' || sName AS info, 0 AS gpa
FROM STUDENT
WHERE address='320FL'
```

There are different operators available to help make simple calculations and to refine queries. Comparison operators are based on either numerical comparison or typical lexicographic ordering. The LIKE keyword can be used for pattern matching.

Set operations among union-compatible tables also exist in SQL, as UNION, INTERSECT, and EXCEPTThese are all set operators, so turn them into bag operators, simply append ALL to the end of the keyword. Joins can also be ported over from relational algebra; theta joins can be applied by selecting from multiple tables and applying a condition, and there's even a direct "NATURAL JOIN" function available in SQLNote that natural joins are only available between tables that have at least one pair of columns with the same name and type. In the result, the common column only appears once.

SQL does have a few additional features that cannot be found in relational algebra, most notably ordering. You can append a ORDER BY to a SELECT statement, and can provide several columns to sort by. The engine first looks to the first column given, then the second, and so on. ASC and DESC can be specified on each column for ascending and descending order, respectively. Another useful feature is SELECT DISTINCT, whereby only distinct entires for the given columns are provided.

Finally, SQL has facilities for aggregations and aggregate functions. The keyword here is GROUP BY (again, optional on SELECT statements). This one comes at the absolute end of a statement, but you

need to have a group function*s* on columns at the top as part of the SELECT. Here's a giant list of demo statements:

```
SELECT pNumber, address, count(sName) AS cnt, sum(sNumber) AS sum
FROM Student
WHERE sNumber > 1
GROUP BY pNumber, address;

SELECT pNumber, COUNT(sName)
FROM Student
GROUP BY pNumber
HAVING SUM(sNumber) > 2 /* similar to a WHERE clause, but for GROUP BY */

SELECT borrower.customer_name, SUM(loan.amount) AS total_loan
FROM borrower,loan
WHERE borrower.loan_number = loan.loan_number
GROUP BY customer_name
HAVING SUM(loan.amount) > 20000; /* can't reference total_loan yet, it doesn't exist */

SELECT branch_city
FROM Branch
GROUP BY branch_city
HAVING COUNT(branch_name) > 3

/* Generic template for SELECT statements: */
SELECT <projection list>
FROM <table names>
WHERE <conditions>
GROUP BY <grouping columns>
HAVING <grouping conditions>
ORDER BY <columns to sort with>
```

SELECT statements can also be used in more complicated situations, for example in the deletion of all rows returned by a SELECT.

```
UPDATE Registration
SET grade = "B"
WHERE sNumber IN (
   SELECT sNumber FROM Student WHERE sName='something'
);
```

5.2 Subqueries

Queries can be ran on more than just previously declared tables, they can be run on the results of queries too. The syntax is fairly simple: instead of a table after a FROM statement, just write another query in parentheses, then give it a name after the closing parenthesis by simply writing "AS" followed by a valid identifier. Example:

```
SELECT D.customer_name, Num_Loan, Num_ACC
FROM
   (SELECT customer_name, count(*) AS Num_Acc
   FROM Depositor
   GROUP BY customer_name) AS D,
   (SELECT customer_name, count(*) AS Num_Loan,
   FROM Borrower
   GROUP BY customer_name0 AS B
WHERE D.customer_name = B.customer_name;
```

Subqueries can also be used in WHERE clauses as long as the subquery returns a table with only one column; this allows you to compare some attribute of a possibly-selected row against an entire table of values, rather than just one static value. Example:

```
SELECT sNumber, sName
FROM Student
WHERE pNum = (
   SELECT pNumber
   FROM Professor
   WHERE pname='MM');
```

Other operations than "=" and associated math operators can be used as well, namely IN and NOT IN. To each operation you can apply the ALL or ANY modifiers, which predictably say that the operation has to be true for either all the results of the subquery or only some of them. Here's another example, one where the goal is to find the student ID taking the largest number of courses:

```
SELECT sID, count(courseID) as cnt
FROM Registration
GROUP BY sID
HAVING count(courseID) = (
   SELECT MAX(cnt) AS mx
   FROM (
      SELECT count(courseID) AS cnt
      FROM Registration
      GROUP BY sID) q
   );
```

5.3 NULL

NULL can cause headaches. Any operation involving a NULL *also* returns NULL, so operations like 2+NULL might seem like they should return 2, but actually return NULL. When NULL values are contained in a table, the behavior varies with the DBMS. For example, some engines might say that NULL evaluates as equal to anything, which might be undesirable behavior. This can be remedied by using IS NOT NULL:

```
SELECT sNumber
FROM Student
WHERE address IS NOT NULL AND address='something';
```

Another solution is the NVL function, which will return its first argument if it's not null, or the second argument otherwise. Example:

```
SELECT sNumber, NVL(address, ''N/A'')
FROM Student;
```

When performing aggregates, NULL is ignored, but it *is* considered a separate group for the purposes of GROUP BY.

5.4 Views

In SQL, views are queries that are registered (stored) inside the database, and they can be treated just like regular tables. For example:

```
CREATE VIEW <name> AS <statement>;
DROP VIEW <name>;
```

Any query can be made into a view. These provide a sort of logical data independence — the base table may change, but the view will always stay the same. Code that queries a table that has been changed may need to be changed itself, but code that queries a view won't change because view output will stay the same. Another reason might be to hide data to prevent access to certain columns (like those containing sensitive info, like social security numbers).

Of note is that views are only stored with their definition, *not* the data — that's obtained at runtime. The actual view schema is really only the columns produced by the SELECT statement.

5.5 Triggers

Some application constraints are too complex to model as actual constraints on individual tables. For instance, it might be too complex to set the restraint that age must be derived from the date-of-birth field, or to set a constraint that the sum of all loans must not exceed 100k. Therefore, triggers are used.

Triggers are just procedures that run automatically when certain events occur in the database engine. Some of the things they can do include checking values, filling in values, changing records, or even commit/roll-back transactions. They have three components: an event, a condition (optional), and an action. In general, when the event occurs and the condition is met, the action is performed. Events can be insert/update/delete, triggers can be activated before or after the event, and the granularity can either be for each row or for the entire statement.

The syntax looks something like this:

```
CREATE TRIGGER <name>
BEFORE|AFTER INSERT|UPDATE|DELETE ON <tablename>
FOR EACH ROW | FOR EACH STATEMENT
WHEN <condition>
/* code */
```

For example:

```
CREATE TRIGGER abc
BEFORE INSERT ON Students
/* code */
```

This trigger will run before any inserts performed on the "Students" table. Actions can be performed below, such as update statements. The third line from the template will set the granularity of the trigger — the UPDATE statement might be activated once per statement, or it might activate once for each and every row change. Note that the "WHEN" (condition) part of a trigger is completely optional. Also, in the action section of the trigger, there are two references to the new values of inserted/deleted/updated records and the old values of deleted/updated records, called ":NEW" and ":OLD", respectively.

5.5.1 Examples

Example 1 If the employee salary increased by more than 10%, make sure the "rank" field is not null and its value has changed, otherwise reject the update.

```
CREATE TRIGGER EmpSal
BEFORE UPDATE ON Employee
FOR REACH ROW
BEGIN
   IF (:new.salary > (:old.salary * 1.1) THEN
      IF (:new.rank IS NULL OR :new.rank = :old.rank)) THEN
         RAISE_APPLICATION_ERROR(-20004, ‘‘rank field not correct’’);
      END IF;
   END IF;
END;
/ /* needed to execute the trigger creation (?) */
```

Triggers always execute in this order: before statement, before row, event (row-level), after row, after trigger. However, the entire transaction is regarded as one unit, so raising an application error will actually cancel the whole transaction. That is, if you update 500 employees at once but only one raises an error, *none* of the employees will actually be updated.

Row level triggers check individual values and can update them, and thus have access to the ":new" and ":old" vectors, but statement level triggers don't since there's no specific row for them to be applied to. In fact, statement level triggers apply to the entire statement no matter how many records are being update; usually, they're used for verification before or after statements.

Example 2 If the employee salary increased by more than 10%, increment the rank field by one.

```
CREATE TRIGGER EmpSal2
BEFORE UPDATE ON Employee
FOR EACH ROW
BEGIN
   IF (:new.salary > (:old.salary * 1.1) THEN
      :new.rank := :old.rank + 1;
   END IF;
END;
/
```

Note that the ":=" operator is required to set one thing equal to another; instead of having a "=" for assignment and a "==" for comparison like most other languages, SQL uses "=" for comparison and ":=" for assignment, since comparison happens more frequently.

Example 3 If the newly inserted record in Employee has a null hireDate field, fill it in with the current date.

```
CREATE TRIGGER EmpHireDate
BEFORE INSERT ON Employee
FOR EACH ROW
BEGIN
   IF (:new.hireDate IS NULL) THEN
      :new.hireDate := current_date;
   END IF;
END;
/
```

An alternate approach is to use "SELECT sysdate INTO temp FROM dual" to create a temporary variable containing the date, then set the date field equal to it. This may work on systems where "current_date" is not valid.

Example 4 Keep the bonus attribute in Employee always 3% of the salary.

```
CREATE TRIGGER EmpBonus
BEFORE INSERT OR UPDATE ON Employee
FOR EACH ROW
BEGIN
   :new.bonus = :new.salary * 0.3;
END;
/
```

It's import to note that you can declare variables and temporary tables inside a trigger; triggers define *procedures*, not just simple queries, so it's perfectly valid to have multiple statements between the BEGIN and END of a trigger. For instance, you might select columns from certain rows into a temporary table (variable) that could then be queried.

5.6 Cursors

Cursors are SQL constructs point to a single row from a select query. In essence, they're ways of getting just one row at a time, as opposed to getting all rows from the query at once. This is especially useful for triggers. They look something like this:

```
CURSOR name IS query
```

For example: '

```
CURSOR HighSalEmp IS
   SELECT empID, name, salary
   FROM Employee
   WHERE salary > 100000
```

Example 1 There are two tables, Customer and Product. When inserting a new customer, put in the Marketing table the customer ID along with the products labeled "On sale". Use a trigger:

```
CREATE TRIGGER NewCust
AFTER INSERT ON Customer
FOR EACH ROW
DECLARE
   pid number;
   CURSOR C1 IS SELECT product_id FROM Product WHERE label = ''OnSale'';
BEGIN
   open C1;
   LOOP
      FETCH C1 INTO pid;
      IF (C1\%FOUND) THEN
         INSERT INTO Marketing(Cust_id, Product_id) VALUES (:new.Id, pid);
      END IF;
      EXIT WHEN C1\%NOTFOUND;
   END LOOP;
   close C1;
END;
```

Alternatively, use a FOR loop:

```
CREATE TRIGGER NewCust
AFTER INSERT ON Customer
FOR EACH ROW
DECLARE
   CURSOR C1 IS SELECT product_id FROM Product WHERE label=''OnSale'';
BEGIN
   FOR rec IN C1 LOOP
      INSERT INTO Marketing(Cust_id, Product_id) VALUES (:new.Id, rec.product_id);
   END LOOP;
END;
```

For obvious reasons, this notation may come much more naturally, and it's shorter anyways.

5.7 Stored Procedures & Functions

If views are ways of storing data, procedures and functions are ways of storing code. These obey PL/SQL, or Procedural Language SQL (the same language used inside triggers). These look something like this:

```
CREATE [OR REPLACE] PROCEDURE <name> (<parameters>)
[IS|AS]
   <declarations>
BEGIN
   <executable section>
EXCEPTION
   <exception section>
END <procedure name>;
/
```

For example:

```
CREATE PROCEDURE remove_emp(employee_id number) IS
tot_emps number;
BEGIN
   DELETE FROM Employees
   WHERE Employee.employee_id = remove_emp.employee_id;
   tot_empts:=tot_emps-1;
END remove_emp;
/
```

This procedure allows the caller to delete an employee. Not sure why you'd want this as a procedure, but there it is. To call, use it like a function from any other language, but with "EXEC" in front of it. Here's a more complicated example:

```
CREATE PROCEDURE OpeningBal(p_type IN string) AS
   cursor C1 IS
      SELECT productId, name, price
      FROM Products
      WHERE type = p_type;
BEGIN
   FOR rec in C1 LOOP
      INSERT INTO Temp VALUES(rec.productId, rec.nam,e rec.price);
   END LOOP;
END;
/
```

These stored procedure can set output variables and they can alter tables, but they don't return a value. For that, you need a stored function. These are identical in syntax to procedures except they have "FUNCTION" in place of "PROCEDURE", and before IS they have a "RETURN <type>". Since they have a return value, stored functions can actually be used in a select statement (as part of WHERE, HAVING, or in the projection list). Example:

```
CREATE FUNCTION MaxNum() RETURN number AS
   num1 number;
BEGIN
   SELECT MAX(sNumber) INTO num1 FROM Student;
   RETURN num1;
END;
```

6 Functional Dependencies

Functional dependencies are just links between pieces of data that state that data A relies on data B. For instance, if a table has a primary key "ID" and an "address" column, it can be said that the "ID" field functionally determines the "address" field. In general, FD's require that th value for a certain set of attributes determines uniquely the value for another set of attributes — effectively just a generalization of keys.[3] They're represented as $A1, A2, \cdots An \rightarrow B1, B2, \cdots Bn$, where the left side depends on the right side (this does not say that the left side computes the right side). For example, you might say that SSN $\rightarrow$ name, DoB, address — this means that every time you see an SSN, the values name, DoB, and address should all be the same, carried over from other occurrences of SSN. The message that should be taken away from this is that any key (primary or candidate) or superkey of a relation functionally determines all the attributes of the relation.

In formal math: Let R be a relation schema where $\alpha \subseteq R$ and $\beta \subseteq R$ (a and b are subsets of R's attributes), the functional dependency $\alpha \rightarrow \beta$ holds on R if an only if, for any legal instance of R, whenever two tuples $t1$ and $t2$ agree on the attributes α, they also agree on attributes β. That is, $t1[\alpha] = t2[\alpha] \rightarrow t1[\beta] = t2[\beta]$. A given key K is a superkey for relation schema R if and only if $K \rightarrow R$ (K determines all attributes of R) and there is no $a \subset K, a \rightarrow R$.

Example: Consider the relation Student (SSN, Fname, Mname, Lname, DoB, address, age, admissionDate). If SSN is the primary key, then SSN $\rightarrow$ Fname, Mname, Lname, DoB, address, age, admissionDate. If instead you know that the three names together form a key, then Fname, Mname, Lname $\rightarrow$ SSN, DoB, address, age, admissionDate.

Functional dependencies have other useful properties. There's the usual transitive property where $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. Augmentation is when $A \rightarrow B$, then $AZ \rightarrow BZ$. The union property is where $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$. Finally, the reverse of the union property also applies in the case of the decomposition property.

6.1 Closure of a Set of Functional Dependencies

Given a set F of functional dependencies, there are other functional dependencies that can be inferred based on F. For example, if $A \rightarrow B$ and $B \rightarrow C$, then it can be inferred that $A \rightarrow C$. The closure set of F is then F+, i.e. the set of all functional dependencies that can be inferred by F. As the size of F increases, the complexity of calculating these closure sets increases dramatically.

Attribute closure, on the other hand, is relatively simple. The attribute closure is when, given a set of functional dependencies, all attributes X that are determined by A are computed. To compute it, just recursively apply the transitive property. Note that A can be either a single attribute or a set of attributes. The algorithm looks something like this:

1. Let X = {A1, A2, ... An}
2. If there exists a functional dependency $B1, B2, \cdots Bm \rightarrow C$ such that every $Bi \in X$, then $X = X \cup C$.
3. Repeat step 2 until no more attributes can be added.

6.2 Lossy & Lossless Decomposition

Sometimes relations must be decomposed into multiple relations, whether for some refactoring purpose or because the original design was faulty. When performing these decompositions, a reference to the primary key of the split-off relation must be maintained in the original relation. Failure to do so will result in lossy decomposition, i.e. data will be lost. Ideally, once decomposition is complete, a natural join between the new relations should exactly reproduce the original. In reality, with real SQL and tables, decompositions are usually performed using distinct selects and natural joins (to verify the decomposition matches the original data set).

[3] Note that keys imply a functional dependency, but simple uniqueness constraints don't.

6.3 Normalization

Normalization is a way of systematically eliminating anomalies from a database design, basically just a st of rules to avoid bad schema design. They decide whether a particular relation R is in a "good" form; if so, it's fine, if not, the relation is transformed in some way. This "good" form is known as "normal form", and when relations are in normal form it is known that certain types of problems have been avoided or minimized.

Of the normal forms, the further down the list you go, the more redundant the data becomes, but as you go up the normal forms, the number of tables increases along with the complexity. For instance, a database in first and second normal forms will be more complex but less redundant than a database in normal forms 1 through 5. Anything that isn't in at *least* first normal form is said to be not normalized. Each normal form builds upon the last.

First Normal form (1NF) An attribute domain is atomic if its elements are considered to be indivisible units, i.e. primitive types like integers or strings. A table is considered to be in 1NF if all the fields contain only scalar values, so arrays aren't allowed. For example, if a table contains a "phone number" attribute that contains a list of phone numbers in each row, the table is *not* in 1NF.

Boyce-Codd Normal Form (BCNF, also known as 3.5NF) A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all function dependencies $F+$ of the form $\alpha \to \beta$ where $a \subseteq R$ and $\beta \subseteq R$, then at least one of the following holds: $\alpha \to \beta$ is trivial (such as β is a subset of α) or α is a superkey for R.[4]

BCNF can be tricky in some cases because of dependency preservation. Ideally, all functional dependencies should be preserved once a decomposition process (to normalize a table into some normal form) is complete. Unfortunately, it's not always possible to decompose a relation to meet BCNF requirements while maintaining all functional dependencies. Instead, the third normal form may be used, although at cost of being slightly weaker.

Third Normal Form (3NF) 3NF has three conditions, but only one has to be true. Relation R is in 3NF if, for every functional dependency $\alpha \to \beta$ where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \to \beta$ is trivial (i.e. $\beta \subseteq \alpha$)
- α is a superkey of R
- Each attribute in $\beta - \alpha$ is a part of a candidate key (primate attribute)

Relations can be in 3NF but not in BCNF.

[4]Candidate keys are also superkeys. Superkeys are just any set of attributes that can uniquely identify a row in a table