Databases

Muchang Bahng

Fall 2024

Contents

This is a course on database languages (SQL), database systems (Postgres, SQL server, Oracle, MongoDB), and data analysis.

Definition 0.1 (Data Model)

A **data model** is a notation for describing data or information, consisting of 3 parts.

1. *Structure of the data.* The physical structure (e.g. arrays are contiguous bytes of memory or hashmaps use hashing). This is higher level than simple data structures.
2. *Operations on the data.* Usually anything that can be programmed, such as **querying** (operations that retrieve information), **modifying** (changing the database), or **adding/deleting**.
3. *Constraints on the data.* Describing what the limitations on the data can be.

There are two general types: relational databases, which are like tables, and semi-structured data models, which follow more of a tree or graph structure (e.g. JSON, XML).

1 Relational Algebra

The most intuitive way to store data is with a *table*, which is called a relational data model, which is the norm since the 1990s.

Definition 1.1 (Relational Data Model)

A **relational data model** is a data model where its structure consists of

1. **relations**, which are two-dimensional tables.
2. Each relation has a set of **attributes**, or columns, which consists of a name and the data type (e.g. int, float, string, which must be primitive).[a]
3. Each relation is a set[b] of **tuples** (rows), which each tuple having a value for each attribute of the relation. Duplicate (agreeing on all attributes) tuples are not allowed.

So really, relations are tables, tuples are rows, attributes are columns.

Definition 1.2 (Schema)

The **schema** of a relational database just describes the form of the database, with the name of the database followed by the attributes and its types.

```
Beer (name string, brewer string)
Serves (bar string, price float)
...
```

Definition 1.3 (Instance)

The entire set of tuples for a relation is called an **instance** of that relation. If a database only keeps track of the instance now, the instance is called the **current instance**, and **temporal databases** also keep track of the history of its instances.

SQL (Structured Query Language) is the standard query language supported by most DBMS. It is **declarative**, where the programmer specifies what answers a query should return,but not how the query should be executed. The DBMS picks the best execution strategy based on availability of indices, data/workload characteristics, etc. (i.e. provides physical data independence). It contrasts to a **procedural** or an **operational** language like C++ or Python. One thing to note is that keywords are usually written in uppercase

[a] The attribute type cannot be a nonprimitive type, such as a list or a set.

[b] Note that since this is a set, the ordering of the rows doesn't matter , even though the output is always in some order.

by convention.

Definition 1.4 (Primitive Types)

The primitive types are listed.

1. *Characters.* `CHAR(n)` represents a string of fixed length n, where shorter strings are padded, and `VARCHAR(n)` is a string of variable length up to n, where an endmarker or string-length is used.
2. *Bit Strings.* `BIT(n)` represents bit strings of length n. `BIT VARYING(n)` represents variable length bit strings up to length n.
3. *Booleans.* `BOOLEAN` represents a boolean, which can be `TRUE`, `FALSE`, or `UNKNOWN`.
4. *Integers.* `INT` or `INTEGER` represents an integer.
5. *Floating points.* `FLOAT` or `REAL` represents a floating point number, with a higher precision obtained by `DOUBLE PRECISION`.
6. *Datetimes.* `DATE` types are of form `'YYYY-MM-DD'`, and `TIME` types are of form `'HH:MM:SS.AAAA'` on a 24-hour clock.

1.1 Tables, Attributes, and Keys

Before we can even query or modify relations, we should know how to make or delete one.

Theorem 1.1 (`CREATE TABLE`, `DROP TABLE`)

We can create and delete a relation using `CREATE TABLE` and `DROP TABLE` keywords and inputting the schema.

```
CREATE TABLE Movies(
  name CHAR(30),
  year INT,
  director VARCHAR(50),
  seen DATE
);

DROP TABLE Movies;
```

What if we want to add or delete another attribute? This is quite a major change.

Theorem 1.2 (`ALTER TABLE`)

We can add or drop attributes by using the `ALTER TABLE` keyword followed by

1. `ADD` and then the attribute name and then its type.
2. `DROP` and then the attribute name.

```
ALTER TABLE Movies ADD rating INT;
ALTER TABLE Movies DROP director;
```

Theorem 1.3 (`DEFAULT`)

We can also determine default values of each attribute with the `DEFAULT KEYWORD`.

```
ALTER TABLE Movies ADD rating INT 0;
...
```

```sql
CREATE TABLE Movies(
  name CHAR(30) DEFAULT 'UNKNOWN',
  year INT DEFAULT 0,
  director VARCHAR(50),
  seen DATE DEFAULT '0000-00-00'
);
```

Definition 1.5 (Key)

A set of attributes $\mathcal{K}$ form a **key** for a relation

1. if we do not allow two tuples in any relation instance to have the same values in *all* attributes of the key (i.e. in general).
2. no proper subset of $\mathcal{K}$ can also be a key for *any* relation instance, that is, $\mathcal{K}$ is *minimal*.

A relation may have multiple keys, but we typically pick one as the **primary key** and underline all its attributes in the schema, e.g. Address($\underline{\text{street}}$, city, state, $\underline{\text{zip}}$).

While we can make a key with a set of attributes, many databases use artificial keys such as unique ID numbers for safety.

Example 1.1 (Keys of User Relation)

Given the schema *User(uid, name, age)*,

1. *uid* is a key of *User*
2. *age* is not a key (not an identifier) even if the relation at the current moment all have different ages.
3. $\{$*uid, name*$\}$ is not a key (not minimal)

Theorem 1.4 (PRIMARY KEY, UNIQUE)

There are multiple ways to identify keys.

1. Use the PRIMARY KEY keyword to make name the key. It can be substituted with UNIQUE.

```sql
CREATE TABLE Movies(
  name CHAR(30) PRIMARY KEY,
  year INT,
  director VARCHAR(50),
  seen DATE
);
```

2. Use the PRIMARY KEY keyword, which allows you to choose a combination of attributes as the key. It can be substituted with UNIQUE.

```sql
CREATE TABLE Movies(
  name CHAR(30),
  year INT,
  director VARCHAR(50),
  seen DATE,
  PRIMARY KEY (name, year)
);
```

1.2 Relational Algebra

We've talked about the structure of the data model, but we still have to talk about operations and constraints. We will focus on the operations here, which can be introduced with *relational algebra*, which gives a powerful way to construct new relations from given relations. Really, SQL is a syntactically sugared form of relational algebra.

The reason we need this specific query language dependent on relational algebra is that it is *less* powerful than general purpose languages like C or Python. These things can all be stored in structs or arrays, but the simplicity allows the compiler to make huge efficiency improvements.

An algebra is really just an algebraic structure with a set of operands (elements) and operators.

Definition 1.6 (Relational Algebra)

A relational algebra consists of the following operands.

1. Relations R, with attributes A_i.
2. Operations.

It has the following operations.

1. *Set Operations.* Union, intersection, and difference.
2. *Removing.* Selection removes tuples and projection removes attributes.
3. *Combining.* Cartesian products, join operations.
4. *Renaming.* Doesn't affect the tuples, but changes the name of the attributes or the relation itself.

Let's take a look at each of these operations more carefully, using the following relation.

bar	beer	price
The Edge	Budweiser	2.50
The Edge	Corona	3.00
Satisfaction	Budweiser	2.25

Figure 1: The example relation, which we will denote `serves`, which we will use to demonstrate the following operations.

Definition 1.7 (Set Operations)

Given relations R and S which must have the same schema (if not, just apply a projection), we can do the following set operations.

1. Union. $R \cup S$.
2. Intersection. $R \cap S$, which can be written also as $R - (R - S), S - (S - R)$, and surprisngly $R \bowtie S$.[a]
3. Difference. $R - S$.

Definition 1.8 (Selection)

The **selection** operator σ_p filters the tuples of a relation R by some condition p. It must be the case that p is deducible by looking only at that row.

$$\sigma_p R \tag{1}$$

This is analogous to the `WHERE` keyword.

[a] The natural join will check for all attributes in each schema, but sine we assumed that they had the same schema, it must check for equality over all attributes.

```sql
SELECT *
FROM relation
WHERE
  p_1 AND p_2 AND ... ;
```

Definition 1.9 (Projection)

The **projection** operator π_L filters the attributes of a relation R, where L is a subset of R's attributes.

$$\pi_L R \tag{2}$$

Note that since this operates on sets, if the projection results in two tuples mapping to the same projected tuple, then this repeated element is deleted. This is simply the `SELECT` keyword.

```sql
SELECT
  bar,
  beer
FROM beers;
```

Now let's talk about operations between two relations.

Definition 1.10 (Cartesian Product)

The **cartesian product** $S \times R$ of two relations is the relation

$$S \times R = \{(s \in S, r \in R)\} \tag{3}$$

which has a length of $|S| \times |R|$. It is commutative (so tuples are not ordered, despite its name), and if S and R have the same attribute name n, then we usually prefix it by the relation to distinguish it: $S.n, R.n$. In SQL, we can do it in one of two ways.

```sql
SELECT *
FROM table1
CROSS JOIN table2;

SELECT *
FROM table1, table2;
```

Definition 1.11 (Theta-Join)

The **theta-join** with **join condition/predicate** p gives

$$R \bowtie_p S = \sigma_p(R \times S) \tag{4}$$

1. If p consists of only equality conditions, then it is called an **equi-join**.
2. If p is not specified, i.e. we write $R \bowtie S$, called a **natural join**. The p is automatically implied to be

$$R.A = S.A \tag{5}$$

   for all $A \in R.att \cap S.att$. Duplicate columns are always equal by definition and so one is removed, unlike equijoin, where duplicate columns are kept.

There are other types of joins that we will use.

Example 1.2 (Simple Filter)

Find all the addresses of the bars that Ben goes to.

name	address
The Edge	108 Morris Street
Satisfaction	905 W. Main Street

Table 1: Bar Information

drinker	bar	times_a_week
Ben	Satisfaction	2
Dan	The Edge	1
Dan	Satisfaction	2

Table 2: Frequents Information

We do the following.

$$\pi_{\text{address}}\big(\text{Bar} \bowtie_{\text{name=bar}} \sigma_{\text{drinker=Dan}}(\text{Frequents})\big) \tag{6}$$

Finally, we look at renaming.

Definition 1.12 (Renaming)

Given a relation R,

1. $\rho_S R$ means that you are chaning the relation name to S.
2. $\rho_{(A_1, \ldots, A_n)} R$ renames the attribute names to $(A_1, \ldots, A_n)$.
3. $\rho_{S(A_1, \ldots, A_n)} R$ renames the relation name to S and the attribute names to $(A_1, \ldots, A_n)$.

It does not really adding any processing power. It is only used for convenience.

1.3 Additional Operations

Definition 1.13 (Monotone Operators)

An operator $O(R)$ is monotone with respect to input R if increasing the size (number of rows/tuples) of R does not decrease the output relation O.

$$R \subset R' \implies O(R) \subset O(R') \tag{7}$$

Example 1.3 ()

Let's go through to see if each operator is monotone.

1. *Selection is monotone.*
2. *Projection is monotone.*
3. *Cross Product is monotone.*
4. *Join is monotone.*
5. *Natural Join is monotone.*
6. *Union is monotone.*
7. *Intersection is monotone.*

8. *Difference $R - S$ is monotone* w.r.t. R but not monotone w.r.t. S.

1.4 Derived Operations

Example 1.4 (Getting maximum of an attribute)

Lec 1 6 min.

Notice that the $\max_{att}$ operator is *not* monotone, since the old answer is overwritten. Generally, whenever we want to construct a non-monotone operator, we want to use the set difference since the composition of monotones is monotone.

You should determine when to project, before or after the difference.

1.5 Constraints

Like mathematical structures, relational databases would not be very useful if they didn't have any structure on them. One important structure are *constraints*, which can also be written in relational algebra.

Definition 1.14 (Set Constraints)

There are two ways in which we can use relational algebra to express constraints. If R and S are relations, then

1. $R = \emptyset$ constrains R to be empty.
2. $R \subset S$ constrains R to be a subset of S.[a]

Definition 1.15 (Referential Integrity Constraints)

One way that we can use this is through *referential integrity* constraints, which asserts that a value appearing as an attribute r in relation R also should appear in a value of an attribute s in relation S. That is,

$$\pi_r(R) \subset \pi_s(S) \tag{8}$$

Definition 1.16 (Key Constraints)

If we have the key $\mathbf{k} = (k_1, \ldots, k_m) \subset \mathbf{r}$ of a relation R, we can express this constraint as

$$\sigma_{R_1.\mathbf{k}=R_2.\mathbf{k} \text{ and } R_1.\mathbf{k}' \neq R_2.\mathbf{k}'}(R_1 \times R_2) = \emptyset \tag{9}$$

where $\mathbf{k}' = \mathbf{r} - \mathbf{k}$. That is, if we took the cross of R with itself, we shouldn't find any tuple that match in the keys but doesn't match in the non-key attributes.

Definition 1.17 (Domain Constraints)

We can also constrain the domain of a certain attribute r of relation R. Let $C(r)$ be the constraint. Then,

$$\sigma_{\text{not } C(r)}(R) = \emptyset \tag{10}$$

[a] Note that this is technically unnecessary, since we can write $R - S = \emptyset$. We can also write $R = \emptyset \iff R \subset \emptyset$.

2 Design Theory for Relational Databases

2.1 Functional Dependencies

Now we introduce the concept of functional dependencies (FD), which will transition nicely into keys.

Definition 2.1 (Functional Dependency)

Given a relation R with attributes $\mathbf{r}$, let $\mathbf{a} = (a_1, \ldots, a_n), \mathbf{b} = (b_1, \ldots, b_m) \subset \mathbf{r}$. Then, the constraint

$$\mathbf{a} \mapsto \mathbf{b} \tag{11}$$

also called **a functionally determines b**, means that if two tuples agree on $\mathbf{a}$, then they must agree on $\mathbf{b}$. We say that R satisfies a FD f or a set of FDs F if this constraint is satisfied.

From this, we can see that the term "functional" comes from a literal function being defined on the input $\mathbf{a}$.

Lemma 2.1 (FDs as Key Constraints)

Note that the functional dependency $\mathbf{a} \mapsto \mathbf{b}$ also implies the key constraint

$$\sigma_{R_1.\mathbf{a}=R_2.\mathbf{a} \text{ and } R_1.\mathbf{b}\neq R_2.\mathbf{b}'}(R_1 \times R_2) = \emptyset \tag{12}$$

Definition 2.2 (Superkey)

A set of attributes $\mathbf{k}$ of a relation R is called a **superkey** if

$$\mathbf{k} \mapsto \mathbf{r} - \mathbf{k} \tag{13}$$

If no $\mathbf{k}' \subset \mathbf{k}$ functionally determines $\mathbf{r}$, then it is a key.

2.1.1 Structure on Spaces of Functional Dependencies

To introduce additional structure, we will introduce two spaces.

1. Given a relation R, let us consider the set of all FDs $F = F(R)$ on R. This is clearly a large set, which increases exponentially w.r.t. the number of attributes in R.
2. Let us denote the set of all relations R satisfying F as R_F, which is an infinite set.

Theorem 2.1 (Axioms)

Let's prove a few properties of FDs, which have nice structure.

1. *Splitting and Combining.* The two sets of FDs are equal.

$$\{\mathbf{a} \mapsto \mathbf{b}\} \iff \{\mathbf{a} \mapsto b_i \mid i = 1, \ldots, m\} \tag{14}$$

2. *Trivial FDs.* Clearly elements of $\mathbf{a}$ uniquely determines its own attributes.

$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a} \mapsto \mathbf{b} - \mathbf{a} \tag{15}$$

or can also be written as

$$\mathbf{b} \subset \mathbf{a} \implies \mathbf{a} \mapsto \mathbf{b} \tag{16}$$

3. *Augmentation.*

$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a}, \mathbf{c} \mapsto \mathbf{b}, \mathbf{c} \tag{17}$$

4. *Transitivity.* If $\mathbf{a} \mapsto \mathbf{b}, \mathbf{b} \mapsto \mathbf{c}$, then

$$\mathbf{a} \mapsto \mathbf{c} \tag{18}$$

Proof.

Trivial.

It is also possible to put a partial order on F.

Definition 2.3 (Partial Order)

Given two FDs f and g, consider the set of all relations R satisfying f and g, denoted as R_f and R_g.

1. Then $f \implies g$ iff $R_f \subset R_g$.
2. $f \iff g$ iff $R_f = R_g$.

Moreover, we can use this structure on F to induce structure on the set of attributes $\mathbf{r}$.

Definition 2.4 (Closure of Attributes)

The **closure** of $\mathbf{r}$ under a set of FDs F is the set of attributes $\mathbf{b}$ s.t.

$$R_F = R_{\mathbf{b}} \tag{19}$$

We denote this as $\mathbf{b} = \mathbf{r}^+$.

To actually compute the closure, we take a greedy approach by starting with $\mathbf{r}$ and incrementally adding attributes satisfying F until we cannot add any more. If we want to know where one FD $f : \mathbf{a} \mapsto \mathbf{b}$ follows from a set F, we can always just compute the closure of $\mathbf{a}$ w.r.t. F. Alternatively, we can also use the axioms above to derive all implications.

2.1.2 Projections of Functional Dependencies

If we have a relation R with a set of FDs F, and we project $R' = \pi_{\mathbf{r}'}(R)$, then the set of FDs F' that hold for R' consists of

1. The FDs that follow from F, and
2. involve only attributes of R.

2.2 Anomalies and Decomposition

Definition 2.5 (Anomaly)

Beginners often try to cram too much into a relation, resulting in **anomalies** of three forms.

1. *Redundancies.* Information repeated unnecessarily in several tuples.
2. *Updates.* Updating information in one tuple can leave the same information unchanged in another.
3. *Deletion.* If a set of values becomes empty, we may lose other information as a side effect.

To eliminate these anomalies, we want to **decompose** relations, which involve splitting $\mathbf{r}$ to schemas of two new relations R_1, R_2.

Definition 2.6 (Decomposition)

Given relation $R(\mathbf{r})$, we can decompose R into two relations $R_1(\mathbf{r}_1)$ and $R_2(\mathbf{r}_2)$ such that

1. $\mathbf{r} = \mathbf{r}_1 \cup \mathbf{r}_2$
2. $R_1 = \pi_{r_1}(R)$
3. $R_2 = \pi_{r_2}(R)$

Example 2.1 ()

Notice how this decomposition eliminated all 3 anomalies. Now, let's formalize the conditions needed to decompose such a relation, and how we should actually decompose it.

2.3 Boyce-Codd Normal Form

Here is a simple condition under which the anomalies above are guaranteed not to exist.

Definition 2.7 (BNCF)

A relation R is in **BNCF** iff whenever there is a nontrivial FD $\mathbf{a} \mapsto \mathbf{b}$, it is the case that $\mathbf{a}$ is a superkey for R.

Example 2.2 (Non-BNCF Form)

Example 2.3 (BNCF Form)

Theorem 2.2 (Any 2-Attribute Relation Satisfies BNCF)

Any 2-attribute relations is in BCNF. Let's label the attributes a, b and go through the cases.

1. There are no nontrivial FDs, meaning that $\{A, B\}$ is the only key. Then BCNF must hold since only a nontrivial FD can violate this condition.
2. $a \mapsto b$ holds but not $b \mapsto a$, meaning that a is the only key. Thus there is no violation since a is a superkey.
3. $b \mapsto a$ holds but not $a \mapsto b$. This is symmetric as before.
4. Both hold, meaning that both a and b are keys. Since any FD has at least one of a, b on the left, this is satisfied.[a]

Therefore, we want to decompose a relation R into a set of relations $R_1, \ldots, R_n$ where each R_i is in BNCF *and* the data in the original relation can be reconstructed from the set of R_i's, i.e. there is *lossless decomposition*. It is this second condition that prevents us from just trivially decomposing every relation into 2-attribute relations, which we will elaborate later.

Given that we have found a FD $\mathbf{a} \mapsto \mathbf{b}$ that doesn't satisfy BCNF (i.e. $\mathbf{a}$ is not a superkey) of relation R, we decompose it into the following R_1 and R_2.

1. We want $\mathbf{a}$ to be a superkey for one of the subrelations, say R_1. Therefore, we have it satisfy $\mathbf{a} \mapsto \mathbf{a}^+$, which is satisfied by definition, and set
$$R_1 = \pi_{\mathbf{a}, \mathbf{a}^+}(R) \tag{20}$$
2. We don't want any loss in data, so we take the rest of the attributes not in the closure and define
$$R_2 = \pi_{\mathbf{a}, \mathbf{r} - \mathbf{a}^+}(R) \tag{21}$$

We keep doing this until every subrelation satisfies BCNF. This is guaranteed to terminate since we are decreasing the size of the relations until all attributes are superkeys.

[a] Note that BCNF only requires *some* key to be contained on the left side, not that all keys are.

3 Design Models

Now we will talk about the design of databases from scratch. Recall that

1. A database is a collection of relations.
2. Each relation schema has a set of attributes.
3. Each attribute has a name and domain (type)
4. Each relation instance contains a set of tuples.

Let's reintroduce everything now in the language of ER diagrams.

3.1 The Entity-Relationship Model and Cons

The first step is to designate a primary key for each relation. The most obvious application of keys is allowing lookup of a row by its key value. A more practical application of keys are its way to link key IDs for one relation to another key ID of a different relation. For example, we may have two schemas *Member(uid, gid)* and *Group(gid)*, and we can join these two using the condition `Member.gid = Group.gid`.

Definition 3.1 (Entity-Relationship Model)

This is done through an **E/R diagram**.

1. An **entity** is an object. An **entity set** is a collection of things of the same type, like a relation of tuples of a class of objects, represented as a *rectangle*.
2. A **relationship** is an association among entities. A **relationship set** is a set of relationships of the same type (among same entity sets), represented as a *diamond*.
3. **Attributes** are properties of entities or relationships, like attributes of tuples or objects, represented as *ovals*. Key attributes are underlined.

Example 3.1 (E/R Diagram)

Let us model a social media database with the relations

1. *Users(uid, name, age, popularity)* recording information of a user.
2. *Member(uid, gid, from)* recording whether a user is in a group and when they first joined.
3. *Groups(gid, name)* recording information of group.

The ER diagram is shown below, where we can see that the Member relation shows a relationship between the two entities Users and Groups.

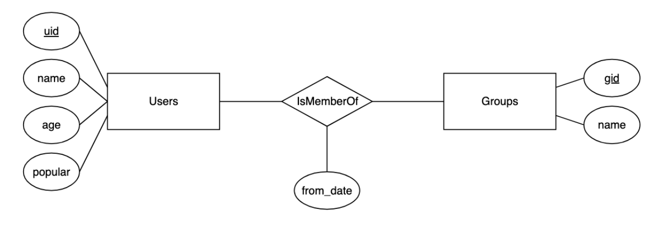

Figure 2: Social media database ER diagram.

Note that the *from* attribute must be a part of the Member relation since it isn't uniquely associated with a user (a user can join multiple groups on different dates) or a group (two users can join a group on different dates).

Therefore, we can associate an entity set and a relationship as relations. A minor detail is that relationships aren't really relations since the tuples in relations connect two entities, rather than the keys themselves, so some care must be taken to convert the entities into a set of attributes.

Therefore, we must determine if a relation models an entity or a relationship. There could also be multiple relationship sets between the same entity sets, e.g. if *Member* and *Likes* associates between *Users* and *Groups*. However, within a relationship set, there is an important set.

Theorem 3.1 ()

In a relationship set, each relationship is uniquely identified by the entities it connects.

If there is an instance that someone joins, leaves, and rejoins a group, then we can modify our design by either:

1. overwritting the first date joined
2. making another relation *MembershipRecords* which has a date also part of the key, which will capture historical membership.

3.1.1 Multiplicity of Binary Relationships

Definition 3.2 (Multiplicity of Relationships)

Given that E and F are entity sets,

1. *Many-many*: Each entity in E is related to 0 or more entities in F and vice versa. There are no restrictions, and we have *IsMemberOf(uid, gid)*.

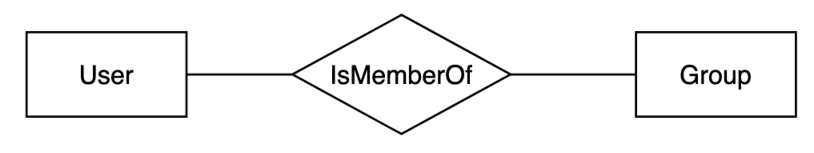

Figure 3

2. *Many-One*: Each entity in E is related to 0 or 1 entity in F, but each entity in F is related to 0 or more in E. If E points to F, then you can just think that this is an injective function, and we have *IsOwnedBy(gid, uid)*. If we have a rounded arrow, this means that for each group, its owner *must* exist in *Users* (so no 0).

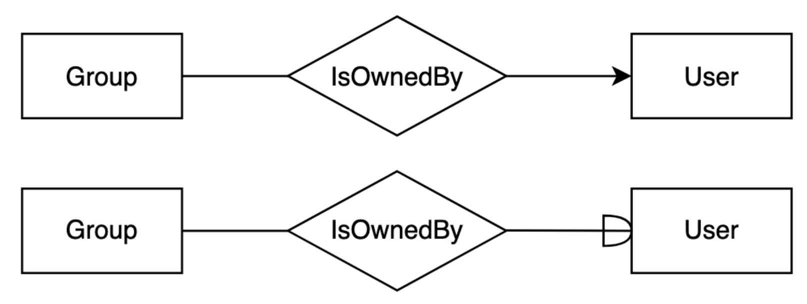

Figure 4

3. *One-One*: Each entity in E is related to 0 or 1 entity in F and vice versa. We have *IsLinkedTo(uid, twitter_ uid)* or *IsLinkedTo(uid, twitter_ uid)*.

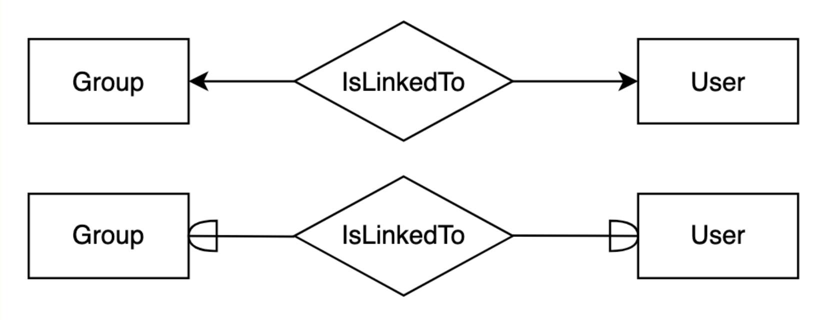

Figure 5

You may notice that multiplicity and functional dependence are very similar that is. If we have two relations R, S and have a relationship pointing from R to S, then this states the FD $\mathbf{r} \mapsto \mathbf{s}$! Say that the keys are $\mathbf{k}_R, \mathbf{k}_S$, respectively. Then, we have

$$\mathbf{k}_R \mapsto \mathbf{r} \mapsto \mathbf{s} \mapsto \mathbf{k}_S \tag{22}$$

Example 3.2 (Movie Stars)

Given the relations

1. *Movies(<u>title</u>, <u>year</u>, length, name)*
2. *Stars(<u>name</u>, address)* of a movie star and their address.
3. *Studios(<u>name</u>, address)*
4. *StarsIn(<u>star_name</u>, <u>movie_name</u>, <u>movie_year</u>)*
5. *Owns(studio_name, <u>movie_name</u>, <u>movie_year</u>)*

We have the following ER diagram

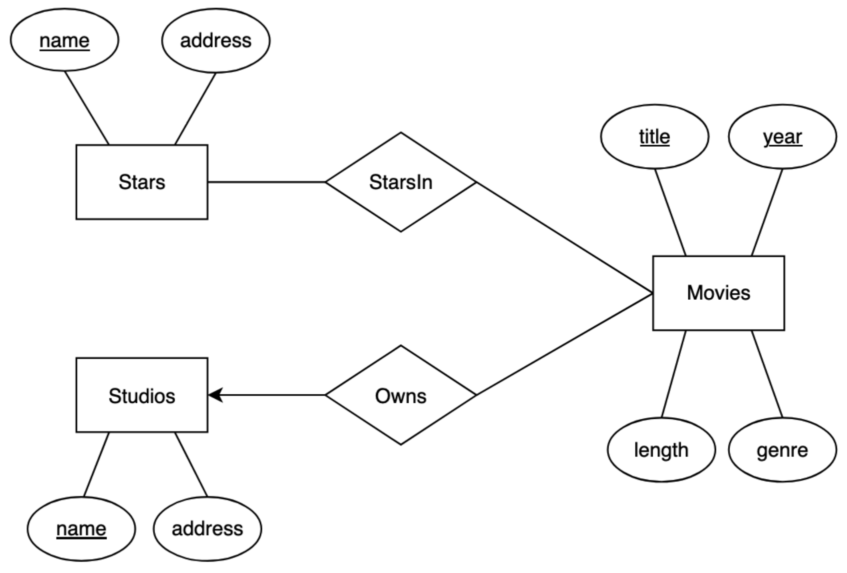

Figure 6: Movie stars.

Example 3.3 (Relationship within Itself)

Sometimes, there is a relationship of an entity set with itself. This gives the relations

1. *Users(<u>uid</u>, ...)*
2. *IsFriendOf(<u>uid1</u>, <u>uid2</u>)*
3. *IsChildOf(<u>child_uid</u>, parent_uid)*

This can be modeled by the following. Note that

1. users have no limitations on who is their friend.
2. assuming that all parents are single, a person can have at most one parent, so we have an arrow.

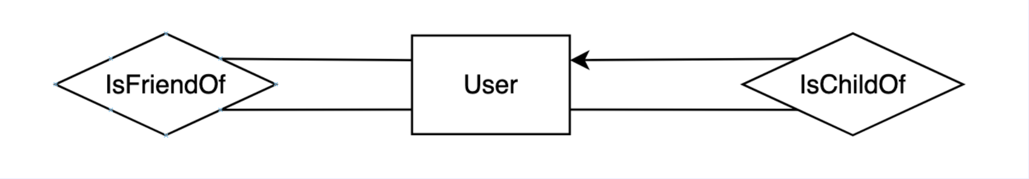

Figure 7

3.1.2 Multiplicity of Multiway Relationships

Sometimes, it is necessary to have a relationship between 3 or more entity sets. It can be confusing to contruct the relations with the necessary keys. A general rule of thumb for constructing the relation of a relationship is

1. Everything that the arrows point into are not keys.
2. Everything else are keys. So the arrow stumps are keys.

Example 3.4 (Movie Stars)

Suppose that we wanted to model *Contract* relationship involving a studio, a star, and a movie. This relationships represents that a studio had contracted with a particular star to act in a particular movie. We want a contract to be owned by one studio, but one studio can have multiple contracts for different combinations of stars and movies. This gives the relations

1. *Stars(<u>name</u>, address)*
2. *Movies(<u>title, year</u>, length, name)*
3. *Studios(<u>name</u>, address)*
4. *Contracts(<u>star_name</u>, <u>movie_name</u>, studio_name)*

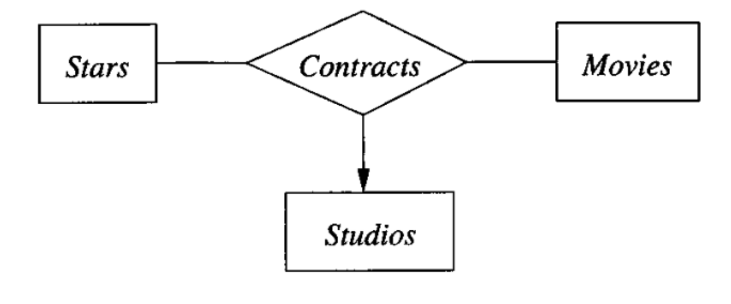

Figure 8

We can make this even more complex by modifying contracts to have a studio of the star and the producing studio.

1. *Contracts(star_ name, movie_ name, produce_ studio_ name, star_ studio_ name)*

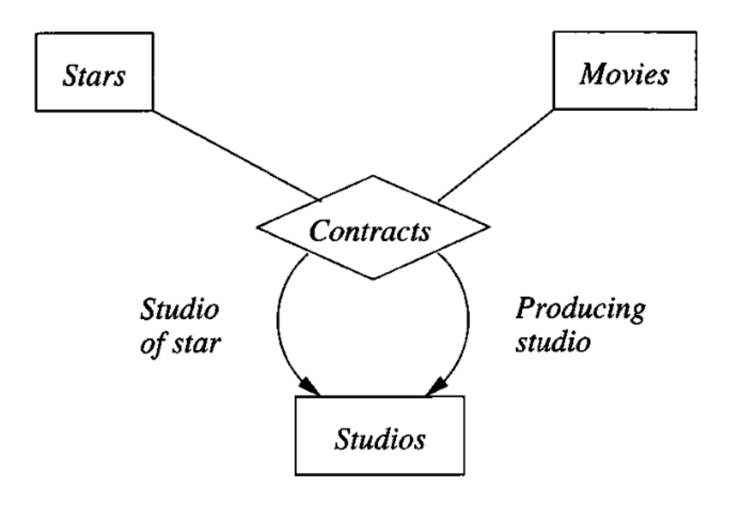

Figure 9

Note that contracts can also have attributes, e.g. salary or time period.

Example 3.5 (Social Media)

In a 3-ary relationship a user must have an initiator in order to join a group. In here, the *isMemberOf* relation has an initiator, which must be unique for each initiated member, for a given group.

1. *User(uid, ...)*
2. *Group(gid, ...)*
3. *IsMemberOf(member, initiator, gid)* since a member must have a unique pair of initiator/group that they are in.

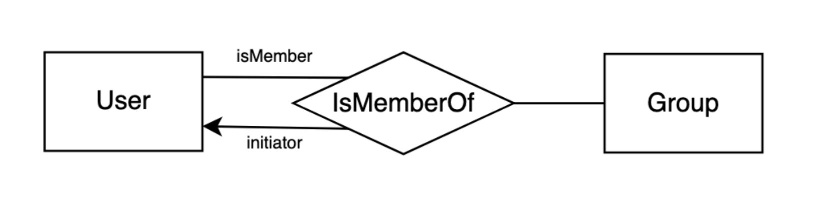

Figure 10

But can we model n-ary relationships with only binary relationships? Our intuition says we can't, for the same reasons that we get lossy decomposition into 2-attribute schemas when we try to satisfy BCNF.

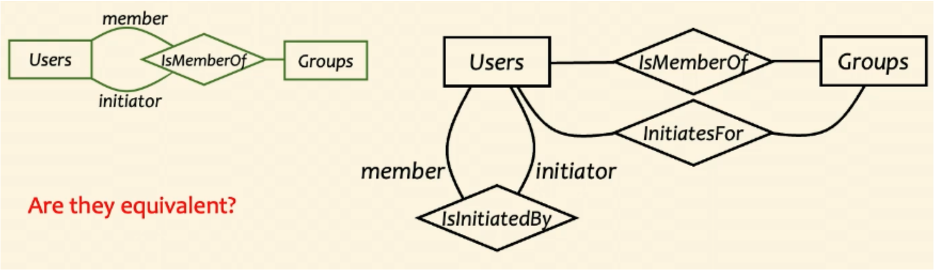

Figure 11: Attempt at reducing nary to binary ER relationships.

1. u1 is in both g1 and g2, so *IsMemberOf* contains both (u1, g1) and (u2, g2)
2. u2 served as both an initiator in both g1 and g2, so *InitiatesFor* contains both (g1, u2) and (g2, u2).
3. But in reality, u1 was initiated by u2 for g1 but not u2 for g2. This contradicts the information that you would get when joining the *IsMemberOf* and *InitiatesFor* relations.

Therefore, combining binary relations may generate something spurious that isn't included in the n-ary relationship.

3.1.3 Subclasses of Entity Sets

Sometimes, an entity set contains certain entities that have special properties not associated with all members of the set. We model this by using a **isa** relationship with a triangle.

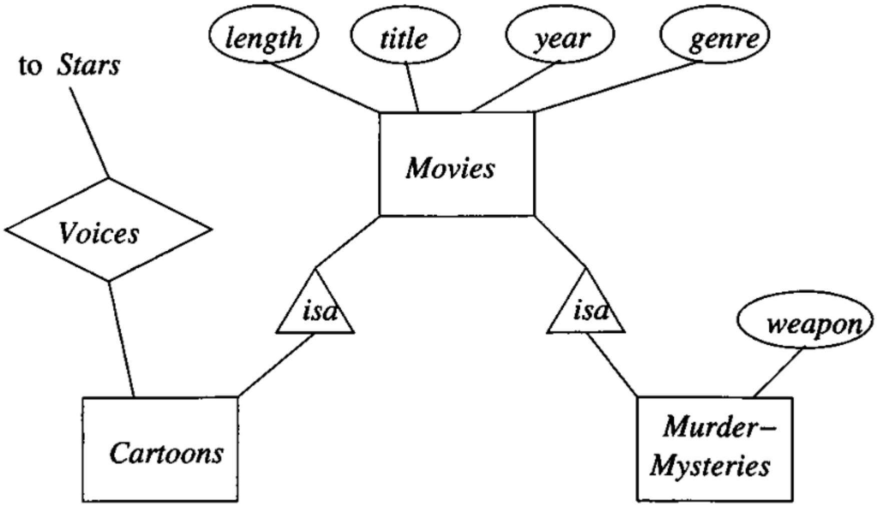

Figure 12: There are two types of movies: cartoons and murder-mysteries, which can have their own sub-attributes and their own relationships.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, and each component is inherited from its parent.

3.2 Design Principles

The first thing we should consider is the multiplicity, which is really context dependent. The second thing is redundancy, which we have mentioned through anomalies before.

3.3 Weak Entity Sets

It is possible for an entity set's key to be composed of attributes, some or all of which belong to another entity set. There are two reasons why we need weak entity sets.

1. Sometimes, entity sets fall into a hierarchy based on classifications unrelated to the *isa* hierarchy. If entities of set R are subunits of entities in set F, it is possible that the names of R-entities are not unique until we take into account the name of its S-entity.[1]
2. The second reason is that we want to eliminate multiway relationships, which are not common in practice anyways. These weak entity sets have no attributes and have keys purely from its supporting sets.

Definition 3.3 (Weak Entity Set)

A **weak entity set** R (double rectangles) depends on other sets. It is an entity set that

1. has a key consisting of 0 or more of its own attributes, and
2. has key attributes from **supporting entity sets** that are reached by many-one **supporting relationships** (double diamonds) from it to other sets S.

It must satisfy the following.

1. The relationship T must be binary and many-one from R to S.
2. T must have referential integrity from R to S (since these are keys and therefore must exist in supporting sets), which is why we have a rounded arrow.
3. The attributes that S supplies for the key of R must be key attributes of S, unless S is also weak, and it will get keys from its supporting entity set.
4. If there are several different supporting relationships from R to the same S, then each relationship is used to supply a copy of the key attributes of S to help form the key of R.

If an entity set supplies any attributes for its own key, then those attributes will be underlined.

Example 3.6 ()

To specify a location, it is not enough to specify just the seat number. The room number, and the building name must be also specified to provide the exact location. There are no extra attributes needed for this subclass, which is why a *isa* relationship doesn't fit into this.

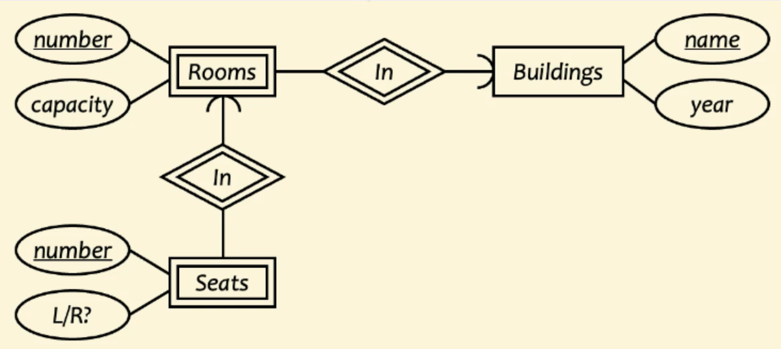

Figure 13: Specifying a seat is not enough to determine the exact location in a university. We must know the room number and the building to fully identify it. Note that we must keep linking until we get to a regular, non-weak entity.

[1] Think of university rooms in different buildings.

3.4 Translating ER Diagrams to Relational Designs

One a simple level, converting an ER diagram to a relational database schema is straightforward. Here are some rules we list.

Theorem 3.2 (Converting Entity Sets)

Turn each entity set into a relation with the same set of attributes.

Theorem 3.3 (Converting Relationships)

Replace a relationship by a relation whose attributes are the keys for the connected entity sets along with its own attributes. If an entity set is involved several times in a relationship, then its key attributes are repeated, so you must rename them to avoid duplication.

Theorem 3.4 (Reduce Repetition for Many-One Relationships)

We can actually reduce repetition for many-one relationships. For example, if there is a many-one relationship T from relation R to relation S, then $\mathbf{r}$ functionally determines $\mathbf{s}$, so we can combine them into one relation consisting of

1. all attributes of R.
2. key attributes of S.
3. Any attributes belonging to relationship T.

Theorem 3.5 (Handling Weak Entity Sets)

To build weak entity sets, we must do three things.

1. The relation for weak entity set W must include its own attributes, all key (but not non-key) attributes of supporting entity sets, and all attributes for supporting relationships for W.
2. The relation for any relationship where W appears must use the entire set of keys gotten from W and its supporting entity sets.
3. Supporting relationships should not be converted since they are many-one, so we can use the reduce repetition for many-one relationships rule above.

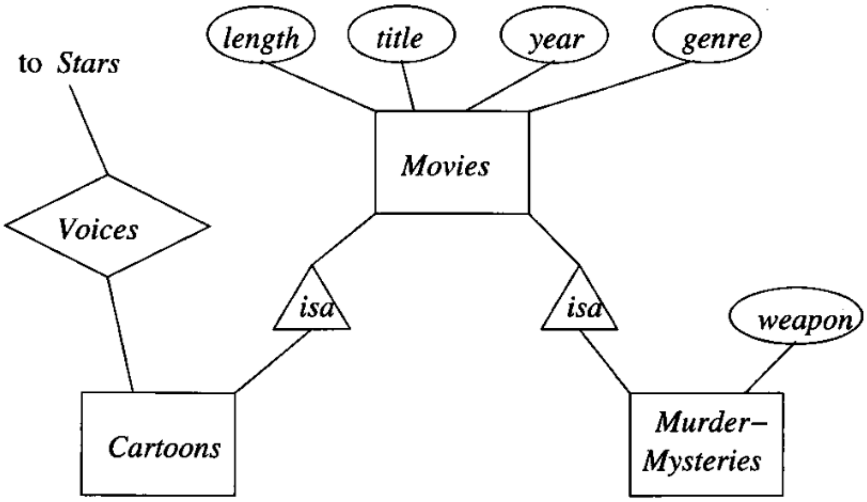

Figure 14: A figure of the movie hierarchy for convenience.

Theorem 3.6 (Converting Subclass Structures)

To convert subclass structure with a *isa* hierarchy, there are multiple ways we can convert them.

1. *E/R Standard.* For each entity set R in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to R. This gives us
   (a) *Movies(title, year, length, genre)*
   (b) *MurderMysteries(title, year, weapon)*
   (c) *Cartoons(title, year)*
2. *Object Oriented.* For each possible subtree that includes the root, create one relation whose schema includes all the attributes of all entity sets in the subtree.
   (a) *Movies(title, year, length, genre)*
   (b) *MoviesC(title, year, length, genre)*
   (c) *MoviesMM(title, year, length, genre, weapon)*
   (d) *MoviesCMM(title, year, length, genre, weapon)*
   Additionally, the relationship would be *Voices(title, year, starName)*.
3. *Null Values.* Create one relation for the entire hierarchy containing all attributes of all entity sets. Each entity is one tuple, and the tuple has null values for attributes the entity does not have. We would in here always have a single schema.
   (a) *Movie(title, year, length, genre, weapon)*

As you probably notice, each standard has pros and cons. The nulls approach uses only one relation, which is simple and nice. To filter out over all movies, E/R is nice since we only filter through *Movies*, whilst in OO we have to go through all relations. However, when we want to filter movies that are both Cartoons and Murder Mysteries, then OO is better since we can only select from *MoviesCMM* rather than having to go through multiple relations for ER or filter out with further selections in Null. Also, OO uses the least memory, since it doesn't waste space on null values on attributes.