Databases

Muchang Bahng

Fall 2024

Contents

This is a course on database languages (SQL, XML, JSON) and database management systems (Postgres, MongoDB).

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). We'll cover in the following order:

1. The theory of relational algebra.
2. Practical applications with SQL.
3. Theory and practice of XML.
4. Theory and practice of JSON.

1 Design Theory for Relational Databases

Okay, we know all that is needed to define a database, but not how to work practically well with it yet. Given some set of data or some structure, we must find an optimal way to store it, within either one or multiple relations. How should we do this? The first step is to categorize the potential problems.

1.1 Anomalies and Decomposition

Definition 1.1 (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 (which violates referential integrity).
3. *Deletion.* If a set of values becomes empty, we may lose other information as a side effect (another violation of referential integrity).

Example 1.1 (Redundancies)

Let's try to see where these anomalies came from. Consider a non-trivial FD $\mathbf{a} \mapsto \mathbf{b}$ where $\mathbf{a}$ is not a superkey. Since $\mathbf{a}$ is not a superkey, there are attributes, say $\mathbf{c}$ that are not functionally determined by $\mathbf{a}$. Therefore, there are multiple combinations of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ which have the same $\mathbf{a}, \mathbf{b}$ but not $\mathbf{c}$, leading to redundancy.

$\mathbf{a}$	$\mathbf{b}$	$\mathbf{c}$
x	y	z_1
$\vdots$	$\vdots$	$\vdots$
x	y	z_{100}

Table 1: Redundant information in $\mathbf{a}$ and $\mathbf{b}$ attributes.

If $T_{\mathbf{a}}, T_{\mathbf{b}}$ are both 4-byte integers, we have just wasted 400 bytes of space. It seems that most of the redundancy comes from using a and b too many times. If we had two relations R and S.

To eliminate these anomalies, we want to **decompose** relations, which involve splitting R into two new relations R_1, R_2.

Definition 1.2 (Decomposition)

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

1. $\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$ (not necessarily disjoint)
2. $R_1 = \pi_{\mathbf{A}_1}(R)$
3. $R_2 = \pi_{\mathbf{A}_2}(R)$

There are two types of decomposition:

1. **lossy** decomposition of R to R_1, R_2 means that joining R_1, R_2 does not give us R.
2. **lossless** decomposition indeed gives us back R.

Theorem 1.1 (Decomposition)

Any decomposition R_1, R_2 of R satisfies

$$R \subset R_1 \bowtie R_2 \tag{1}$$

Proof.

If $R_1.\mathbf{A} \cap R_2.\mathbf{A} = \emptyset$, then $R \subset R_1 \times R_2$ for sure. If there is some overlap, then given some $r \in R$, we will be able to find $r_1 \in R_1$ and $r_2 \in R_2$, and they will definitely join in their overlapping attribute, giving R.

Therefore, we should be worried about if our decomposition will create *new* tuples since it will never delete relevant ones.

Example 1.2 (Lossy and Lossless Decomposition)

Consider the relation

X	Y	Z
a	b	c_1
a	b	c_2
a_1	b	c_2

Projecting to (X, Y) and (X, Z) gives us a lossless decomposition.

X	Y
a	b
a_1	b

X	Z
a	c_1
a	c_2
a_1	c_2

X	Y	Z
a	b	c_1
a	b	c_2
a_1	b	c_2

While projecting to (X, Y) and (Y, Z) gives us a lossy one since we get (a_1, b, c_1) when joining the decompositions, which is not in the original relation.

X	Y
a	b
a_1	b

Y	Z
b	c_1
b	c_2

X	Y	Z
a	b	c_1
a	b	c_2
a_1	b	c_1
a_1	b	c_2

Generally, the intuition behind trying to find a lossless decomposition is this: For a value of a certain attribute X that is in both decompositions R_1 and R_2, we want only one instance of a value on one side. For example,

1. in the lossless decomposition, notice how for overlapping attribute **X**, for each instance a, a_1, we had 1 of each on the left relation, and however many on the right.
2. in the lossy decomposition, notice how we have two b's on the left and two b's on the right for the lossy decomposition. Rather, we want something like 1 b vs multiple b's.

1.1.1 Boyce-Codd Normal Form

This decomposition eliminated the redundancy anomaly, and for attributes **X** and **Z**, it eliminated the deletion and update anomalies. Let's formalize the conditions needed to decompose such a relation, and how we should actually decompose it.

Definition 1.3 (BNCF)

BCNF is defined in two equivalent ways:

1. 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.
2. Given a relation R with a set of nontrivial functional dependencies $F = \{\mathbf{a} \mapsto \mathbf{b}\}$, it is in BCNF if for every $\mathbf{a}$, $\mathbf{a}^+$ is the set of all attributes of R.

Note that since there are multiple keys, $\mathbf{a}$ does not always have to include the same key.

Example 1.3 (Non-BNCF Form)

The table below is not in BCNF form since

$$(\texttt{title}, \texttt{year}) \mapsto (\texttt{length}, \texttt{genre}, \texttt{studioName}) \tag{2}$$

Is a functional dependency where the LHS is not a superkey (the key is `title`, `year`).

title	year	length	genre	studioName	starName
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Table 2: Movie Data

Example 1.4 (BNCF Form)

However, if we decompose this into the following tables, both satisfy BCNF.

title	year	length	genre	studioName
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	231	drama	MGM
Wayne's World	1992	95	comedy	Paramount

Table 3: Simplified Movie Data

title	year	starName
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

Table 4: Movie Titles, Years, and Stars

Algorithm 1.1 (Constructing BCNF of a Relation)

To actually construct this, we act on BCNF violations. 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
$$R_1 = \pi_{\mathbf{a}, \mathbf{a}^+}(R) \tag{3}$$
2. We don't want any loss in data, so we take the rest of the attributes not in the closure and define
$$R_2 = \pi_{\mathbf{a}, \mathbf{r}-\mathbf{a}^+}(R) \tag{4}$$

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.

Theorem 1.2 (Lossless Guarantee for BCNF)

If we decompose on a BCNF violation as stated above, then our decomposition is guaranteed to be a lossless join decomposition.

Proof.

An outline of the proof means that given a BCNF violation $\mathbf{a} \mapsto \mathbf{b}$, we must show that anything we project always comes back in the join
$$R \subset \pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a}, \mathbf{r}-\mathbf{a}}(R) \tag{5}$$
which is proved trivially, and anything that comes back in the join must be in the original relation
$$\pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a}, \mathbf{r}-\mathbf{a}}(R) \subset R \tag{6}$$
We can use the fact that $\mathbf{a} \mapsto \mathbf{b}$.

The lossless guarantee makes BCNF very attractive, but it comes with a tradeoff.

Theorem 1.3 (BCNF Does Not Preserve Functional Dependencies)

BCNF may not enforce equivalent functional dependencies over its decomposed relations compared to its original relation.

Proof.

We consider a counterexample. Given a schema $R(A, B, C)$ with FDs
$$f = AB \mapsto C, \quad g = C \mapsto B \tag{7}$$
When we decompose based on $C \mapsto B$ into $R_1(C, B), R_2(C, A)$, the join is still lossless because we kept C in both relations, and when we join the two on C, each C value will match exactly one B value (due to $C \to B$ in R_2). However, we cannot enforce $AB \to C$.

It is initially confusing since one may ask that if functional dependencies aren't preserved, then doesn't this mean lossy decomposition? Not exactly, since the lossless join property is independent of dependency preservation. The lossless join property ensures we can reconstruct the exact original relation through natural

joins, without getting any spurious tuples. Dependency preservation simply means we can't enforce a FD when inserting/updating. In fact, this is analogous to some functional dependencies being removed when projecting a relation.

1.1.2 Recovery and Chase Test

Now let's talk about properties of general decompositions.

Theorem 1.4 (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]

From this theorem stating that any 2-attribute relation satisfies BCNF, we can just trivially decompose all relations into relations $R_1, \dots, R_n$, and we are done, right? Not exactly, since to ensure lossless join, we must start with the original relation and *split only on the BCNF violations*. This is what makes it lossless, and simply splitting arbitrarily will lead to a lossy decomposition. Furthermore, even if it was lossless, it will destroy most functional dependencies too.

This motivates the need for a general test to see if a decomposition of R is lossless. That is, is it true that $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples t such that for all $i = 1, 2, \dots, k$, t projected onto the set of attributes S_i is a tuple in $\pi_{S_i}(R)$.
- Any tuple t in R is surely in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$. The reason is that the projection of t onto S_i is surely in $\pi_{S_i}(R)$ for each i, and therefore by our first point above, t is in the result of the join.
- As a consequence, $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$ when the FD's in F hold for R if and only if every tuple in the join is also in R. That is, the membership test is all we need to verify that the decomposition has a lossless join.

Theorem 1.5 (Chase Test for Lossless Join)

The **chase test** for a lossless join is just an organized way to see whether a tuple t in $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ can be proved, using the FD's in F, also to be a tuple in R. If t is in the join, then there must be tuples in R, say $t_1, t_2, \dots, t_k$, such that t is the join of the projections of each t_i onto the set of attributes S_i, for $i = 1, 2, \dots, k$. We therefore know that t_i agrees with t on the attributes of S_i, but t_i has unknown values in its components not in S_i.

We draw a picture of what we know, called a *tableau*. Assuming R has attributes $A, B, \dots$ we use $a, b, \dots$ for the components of t. For t_i, we use the same letter as t in the components that are in S_i, but we subscript the letter with i if the component is not in S_i. In that way, t_i will agree with t for the attributes of S_i, but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

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

Example 1.5 ()

Suppose we have relation $R(A, B, C, D)$, which we have decomposed into relations with sets of attributes $S_1 = \{A, D\}$, $S_2 = \{A, C\}$, and $S_3 = \{B, C, D\}$. Then the tableau for this decomposition is:

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

The first row corresponds to set of attributes A and D. Notice that the components for attributes A and D are the unsubscripted letters a and d. However, for the other attributes, b and c, we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple (a, b_1, c_1, d) represents a tuple of R that contributes to $t = (a, b, c, d)$ by being projected onto $\{A, D\}$ and then joined with other tuples. Since the B- and C-components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.
Similarly, the second row has the unsubscripted letters in attributes A and C, while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for $\{B, C, D\}$ and subscript 3 on a. Since each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters.

Remember that our goal is to use the given set of FD's F to prove that t is really in R. In order to do so, we "chase" the tableau by applying the FD's in F to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as t (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple t in the join of the projections was actually a tuple of R.

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurrences.

Example 1.6 ()

Let us continue with the decomposition of the previous example, and suppose the given FD's are $A \to B$, $B \to C$, and $CD \to A$. Start with the tableau:

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2
a_3	b	c	d

Since the first two rows agree in their A-components, the FD $A \to B$ tells us they must also agree in their B-components. That is, $b_1 = b_2$. We can replace either one with the other, since they are both subscripted. Let us replace b_2 by b_1. Then after applying $B \to C$:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a_3	b	c	d

Next, we observe that the first and third rows agree in both columns C and D. Thus, we may apply the FD $CD \to A$ to deduce that these rows also have the same A-value; that is, $a = a_3$. We replace a_3 by a, giving us:

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

At this point, we see that the last row has become equal to t, that is, (a, b, c, d). We have proved that if R satisfies the FD's $A \to B$, $B \to C$, and $CD \to A$, then whenever we project onto $\{A, D\}$, $\{A, C\}$, and $\{B, C, D\}$ and rejoin, what we get must have been in R. In particular, what we get is the same as the tuple of R that we projected onto $\{B, C, D\}$.

Example 1.7 (Chase Test for Lossless-join Decomposition)

Consider relation $R(A, B, C, D)$ with functional dependencies:

$$A \to B, \quad C \to D, \quad AD \to C, \quad BC \to A$$

We will perform the chase test on decomposition $\{AB, ACD, BC, BD\}$ to prove it is lossless.

1. Initial chase table with subscript 'a' for known values and variables for unknown:

A	B	C	D	Source
a	b	c	d	Original
a	b	b_1	b_2	From AB
b_3	b_4	c	d	From ACD
b_5	b_4	c	b_6	From BC
b_7	b	b_8	d	From BD

2. Apply functional dependencies sequentially:
   - Apply $A \to B$:
     - Where A values match, B values must match
     - Rows 1 and 2: since $A = a$, B must be same
     - No change needed as $B = b$ in both
   - Apply $C \to D$:
     - Where C values match, D values must match
     - Rows 1, 3, 4: $C = c$, so D must be same
     - b_6 becomes d

     Updated table after $C \to D$:

A	B	C	D
a	b	c	d
a	b	b_1	b_2
b_3	b_4	c	d
b_5	b_4	c	d
b_7	b	b_8	d

   - Apply $AD \to C$:
     - Where A and D match, C must match
     - Row 1 and 2: $A = a$ but different D, no change
   - Apply $BC \to A$:
     - Where B and C match, A must match
     - Row 3 and 4: same B (b_4) and C (c), so A must match
     - b_3 becomes b_5

3. Final chase table:

A	B	C	D
a	b	c	d
a	b	b_1	b_2
b_5	b_4	c	d
b_5	b_4	c	d
b_7	b	b_8	d

4. Conclusion:
   - The chase process shows variable equating ($b_3 = b_5$)
   - The decomposition is lossless because:
     - AB and BC share B, enforcing $BC \to A$
     - ACD preserves both $C \to D$ and $AD \to C$
     - Common attributes ensure lossless reconstruction
     - Each subrelation preserves relevant FDs
   - The natural join: $(AB \bowtie BC) \bowtie ACD \bowtie BD$ recovers R without loss

Therefore, $\{AB, ACD, BC, BD\}$ is a valid lossless-join decomposition in BCNF.

1.2 The Entity-Relationship Model

We have now learned to decompose relations effectively without anomalies, but it is not clear what these relations may represent. When designing a database, it isn't really ideal to just dump all data into a relation and then try to iteratively decompose it until you get a bunch of relations in BCNF. It's better to have a starting plan of what relations you should need and try and construct interpretable relationships between a pair or set of relations. To do this, *Entity-Relationship diagrams* are of great help.

Let's briefly talk about keys again. 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 1.4 (Entity-Relationship Model)

We can think of every relation as modeling some *data* or a *relationships between data.*[a] This is shown in an **E/R diagram**.

1. An **entity set** is a relation that contains data, represented as a *rectangle*. Its tuples are called *entities.*[b]
2. A **relationship set** is a relation that contains relationships (sometimes stored as the key pairs of two entity sets), represented as a *diamond*. Its tuples are called **relationships**.[c]
3. **Attributes** are properties of entities or relationships, like attributes of tuples or objects, represented as *ovals*. Key attributes are underlined.

Don't confuse entity sets with entities and relationship sets with relationships.

Example 1.8 (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_date)` recording whether a user is in a group and when they first joined.

[a] Sometimes, the line may be blurred, but the designer will have to make the choice.

[b] This is analogous to an entity set being a class and entities as objects.

[c] 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.

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.

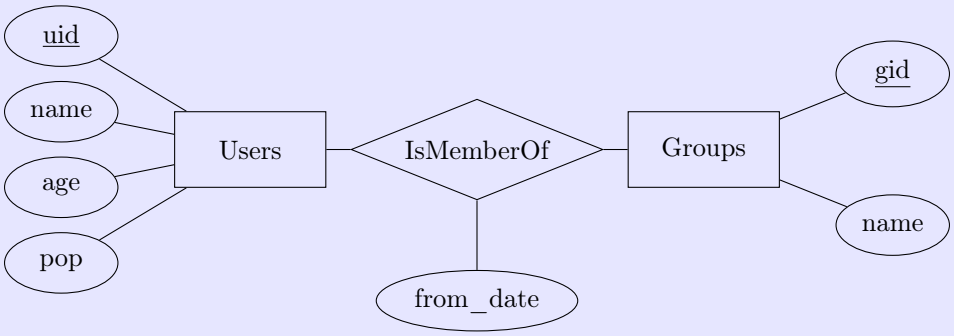

Figure 1: Social media database ER diagram.

Note that the `from_date` 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). If there is an instance that someone joins, leaves, and rejoins a group, then we can modify our design by either:

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

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`.

1.3 Relationships and Multiplicity

A relationship really stores minimal information in the sense that if you have you know the entities it connects, that determines everything about the relation.

Theorem 1.6 (Relationships are Functionally Dependent on Connecting Entities)

In a relationship set, each relationship is uniquely identified by the entities it connects. More formally, if a relationship R connects entities $e_1 \in E_1, \dots, e_n \in E_n$ with keys $\mathbf{k} = (k_1, \dots, k_n)$, then $\mathbf{k}$ is a superkey of R.

1.3.1 Multiplicity of Binary Relationships

The 4 categories representing the multiplicity just further categorizes how this functional dependency is realized.

Definition 1.5 (Multiplicity of 2-Way 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)`.[a]

Figure 2

2. *Many-One*: Each entity in E is related to 0 or 1 entities in F, but each entity in F is related to 0 or more in E. If E points to F with a straight arrow, then you can just think that this is a 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).

3. *One-One*: Each entity in E is related to 0 or 1 entity in F and vice versa. We can have either `IsLinkedTo(uid, twitter_uid)` or `IsLinkedTo(uid, twitter_uid)` and must choose a primary key from these two possible keys.

Figure 3: Bidirectional relationship between Group and User entities

Theorem 1.7 (Multiplicity with Functional Dependencies)

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

$$\mathbf{k}_R \mapsto \mathbf{r} \mapsto \mathbf{s} \mapsto \mathbf{k}_S \tag{8}$$

[a] Note that this is just a restating of the theorem before.

Example 1.9 (Movie Stars)

Given the relations
1. `Movies(title, year, length, name)`
2. `Stars(name, address)` of a movie star and their address.
3. `Studios(name, address)`
4. `StarsIn(star_name, movie_name, movie_year)`
5. `Owns(studio_name, movie_name, movie_year)`

We have the following ER diagram

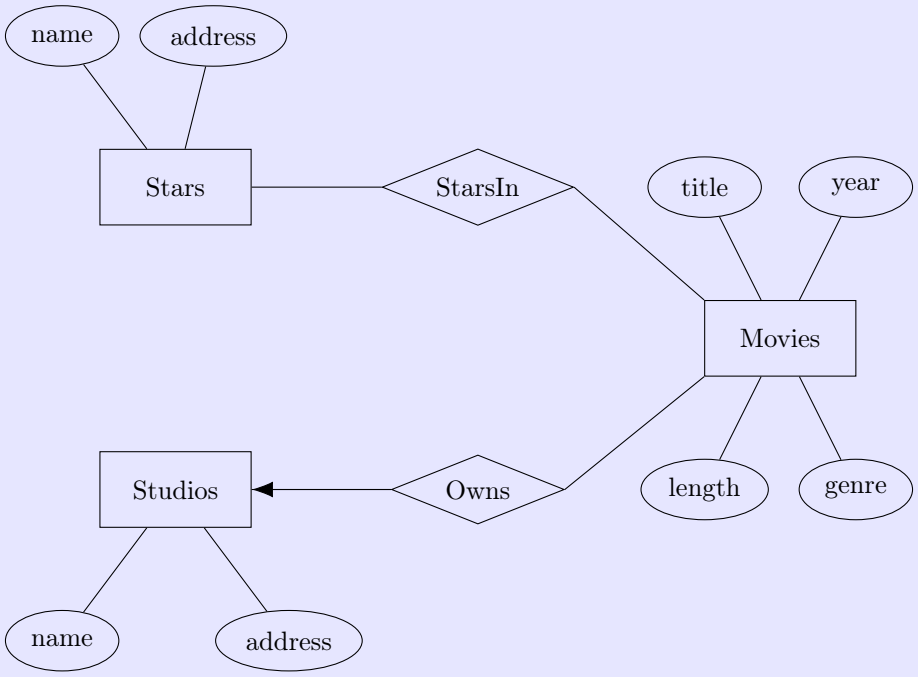

Figure 4: ER diagram showing relationships between Stars, Movies, and Studios entities

Example 1.10 (Relationship within Itself)

Sometimes, there is a relationship of an entity set with itself. This gives the relations
1. `Users(uid, ...)`
2. `IsFriendOf(uid1, uid2)`
3. `IsChildOf(child_uid, 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.

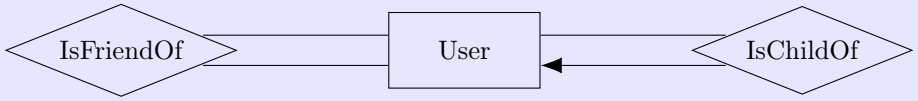

Figure 5: Self-referential relationships for User entity

1.3.2 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 1.11 (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(name, address)`
2. `Movies(title, year, length, name)`
3. `Studios(name, address)`
4. `Contracts(star_name, movie_name, studio_name)`

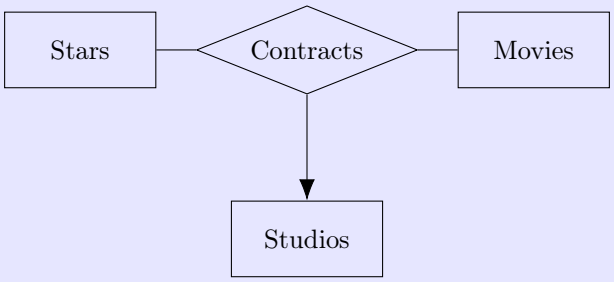

Figure 6

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)`

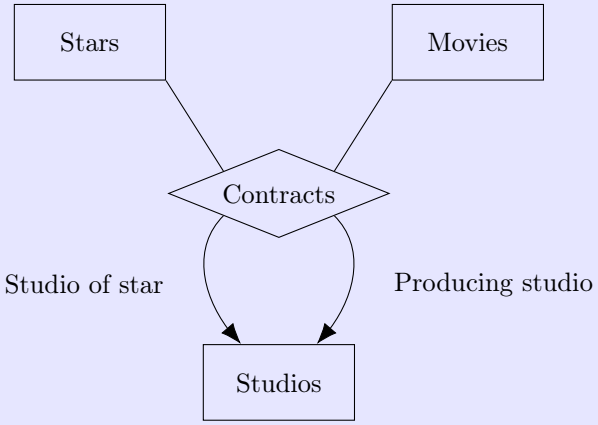

Figure 7: Note that contracts can also have attributes, e.g. salary or time period.

Example 1.12 (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.

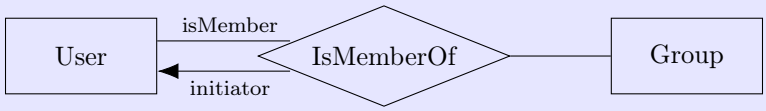

Figure 8: User and Group membership relationship with roles

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.

Example 1.13 (N-ary Relationships vs Multiple Binary Relationships)

N-ary relationships in general cannot be decomposed into multiple binary relationships. Consider the following diagram.

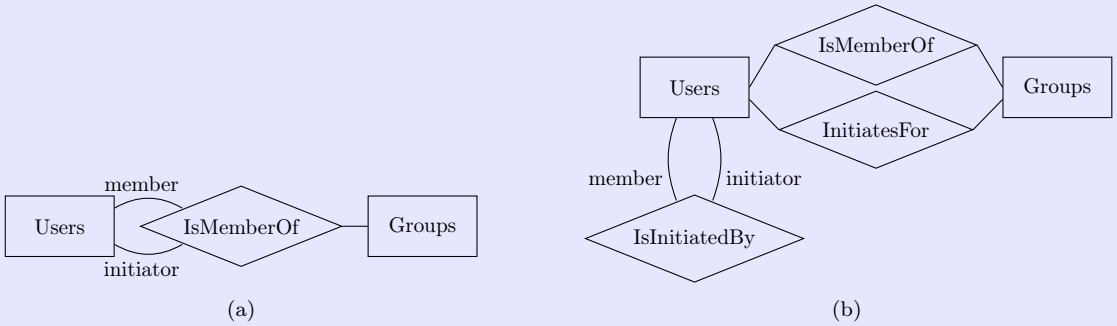

Figure 9: Attempt at reducing nary to binary ER relationships. Are they equivalent?

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.

1.4 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.

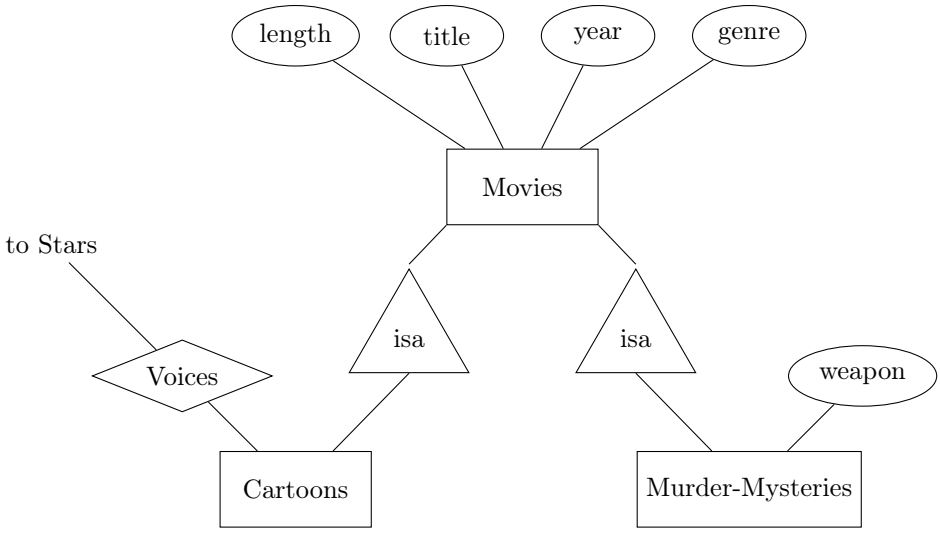

Figure 10: There are two types of movies: cartoons and murder-mysteries, which can have their own sub-attributes and their own relationships.

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.

1.5 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 1.6 (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.

[1] Think of university rooms in different buildings.

Example 1.14 ()

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.

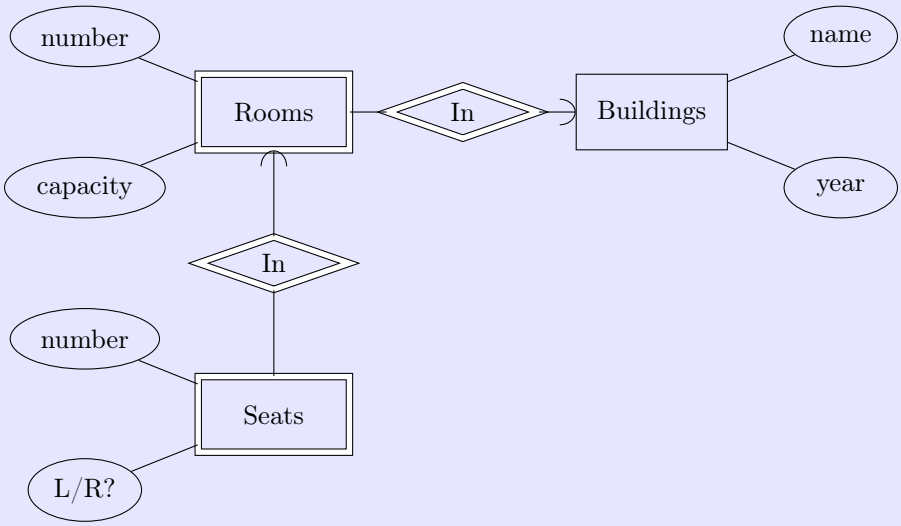

Figure 11: Specifying a seat is not enough to determine the exact location in a university. We must know the room number and the building to fully identify it. Note that we must keep linking until we get to a regular, non-weak entity.

We generally want to use a weak entity set if an entity does not have attributes to define itself.

Example 1.15 ()

Say that we want to make a database with the constraints.

1. For states, record the name and capital city.
2. For counties, record the name, area, and location (state)
3. For cities, record the name, population, and location (county and state)
4. Names of states should be unique.
5. Names of counties are unique within a state.
6. Names of cities are unique within a county.
7. A city is always located in a single county.
8. A county is always located in a single state.

Then, our ER diagram may look like

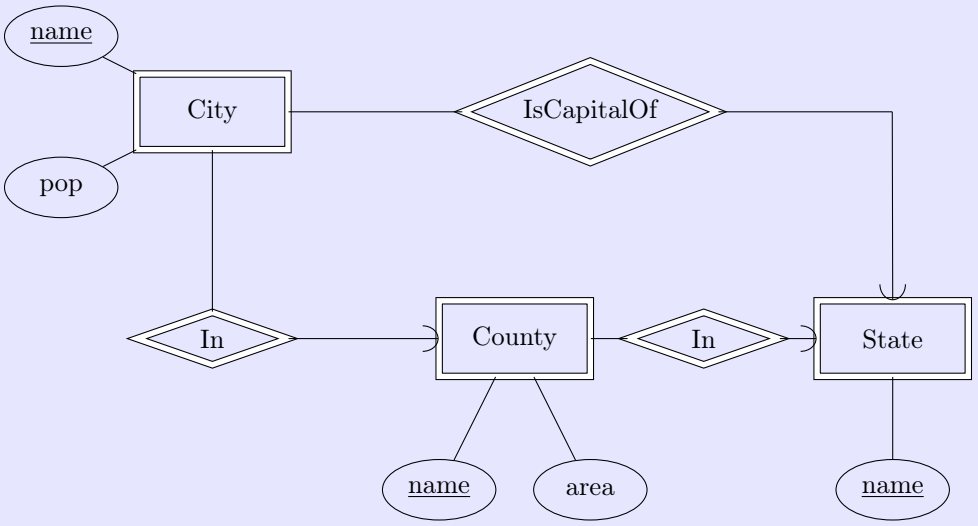

Figure 12: A weakness is that this doesn't prevent a city in state X from being the capital of another state Y.

Example 1.16 ()

Design a database with the following.

1. A station has a unique name and address, and is either an express station or a local station.
2. A train has a unique number and engineer, and is either an express or local train.
3. A local train can stop at any station.
4. An express train only stops at express stations.
5. A train can stop at a station for any number of times during a train.
6. Train schedules are the same every day.

Then, our ER diagram may look like

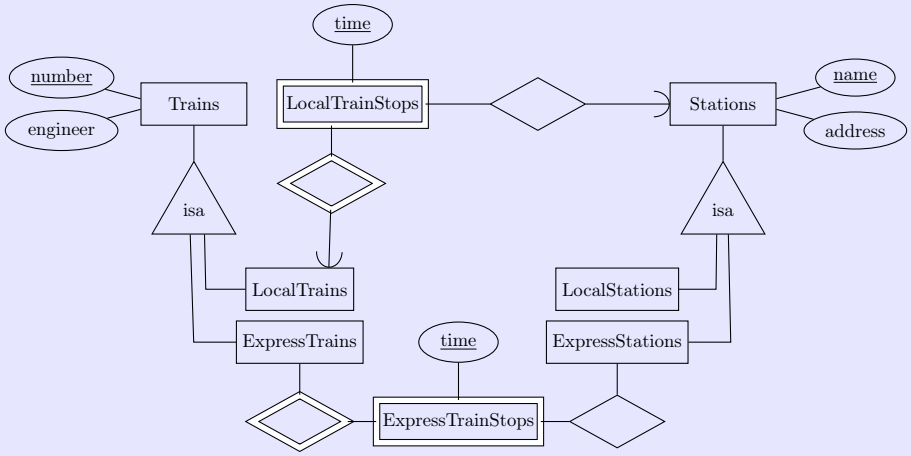

Figure 13

1.6 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 1.8 (Converting Entity Sets)

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

Theorem 1.9 (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 1.10 (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 1.11 (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.

Example 1.17 ()

To translate the seat, rooms, and buildings diagram,

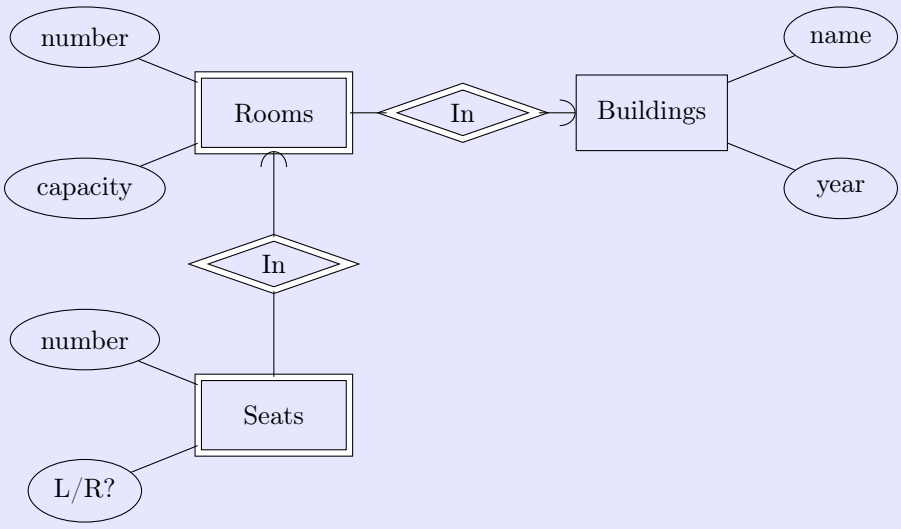

Figure 14

we have
1. `Building(name, year)`
2. `Room(building_name, room_num, capacity)`
3. `Seat(building_name, room_num, seat_num, left_or_right)`

Note that we do not need to convert the relationships since they are contained within the entity set relations. So ignore double diamonds.

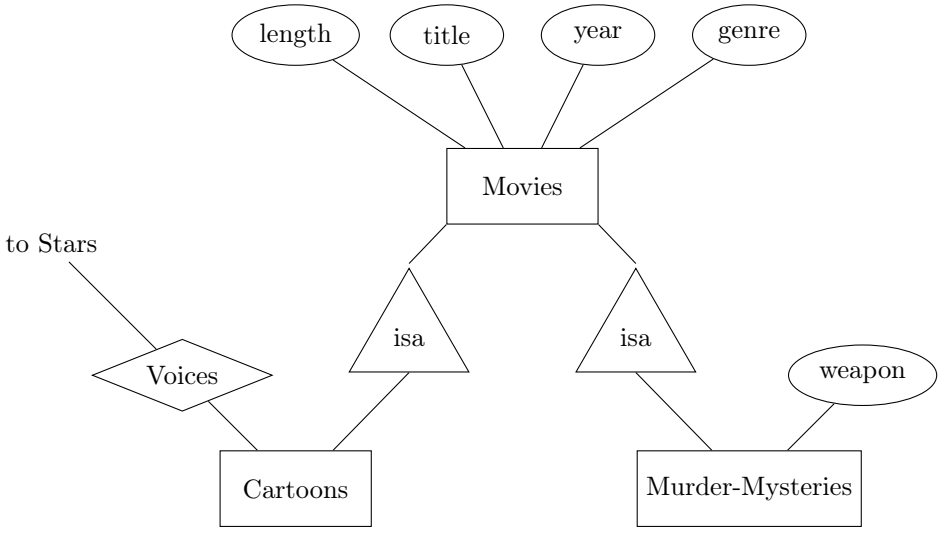

Figure 15: A figure of the movie hierarchy for convenience.

Theorem 1.12 (Converting Subclass Structures)

To convert subclass structure with a *isa* hierarchy, there are multiple ways we can convert them.
1. *E/R Standard.* An entity is in all superclasses and only contains the attributes its own subclass. 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)`

Note that the difference between the first two is that in ER, `MurderMysteries` does not contain the attributes of its superclass, while in OO, it does.

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.

Example 1.18 ()

Let's put this all together to revisit the train station example.

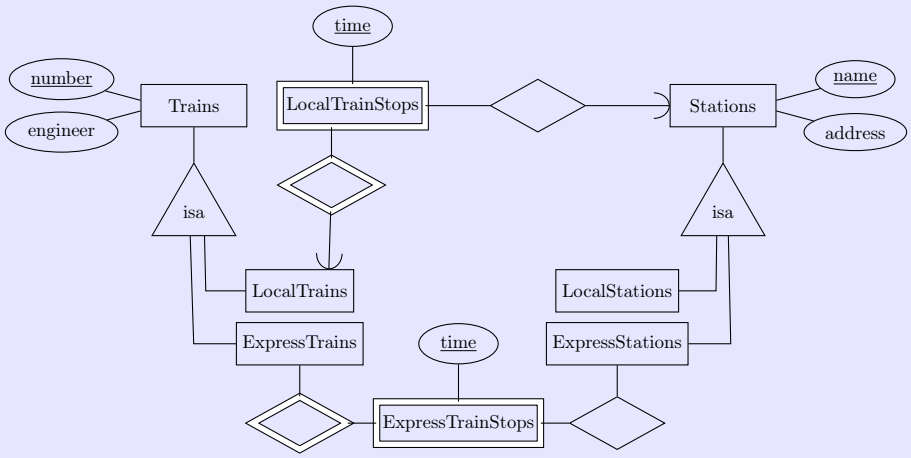

Figure 16: For convenience

We can use the ER standard to define the first 6 regular relations in single rectangles.

1. `Train(number, engineer)`
2. `LocalTrain(number`
3. `ExpressTrain(number`
4. `Station(name, address)`
5. `LocalStation(name)`
6. `ExpressStation(name)`

Then we can construct the weak entity sets.

1. `LocalTrainStops(local_train_num, time)`
2. `ExpressTrainStops(express_train_num, time)`

Then we can construct the relationships (marked with the arrows).

1. `LocalTrainStopsAtStation(local_train_number, time, station_name)`
2. `ExpressTrainStopsAtStation(express_train_number, time, express_station_name)`

Note that we can simplify these 10 relations to 8. For example, the `LocalTrain` and `LocalStation` relations are redundant since it can be computed as

$$LocalTrain = \pi_{number}(Train) - ExpressTrain \tag{9}$$

$$LocalStation = \pi_{number}(Station) - ExpressStation \tag{10}$$

There is a tradeoff since it's an extra computation when checking. However, if we had used the Null Value strategy, this would be a lot simpler, and we can use value constraints on the train and station type, which can be implemented in the DBMS (though not directly in the ER diagram).

2 SQL

SQL (Structured Query Language) is the standard query language supported by most DBMS and is a syntactically sugared form of relational algebra. 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. 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.

Just like how we approached relational algebra with structure, operations, and constraints, we will do the same with SQL, although we will introduce structure with constraints first (it's natural to talk about constraints with structure, since you define the constraints when you create the SQL relations). While this wasn't talked about much before, SQL actually uses *multisets*, or *bags*, by default. It's operations support both set and bag operations, and we will cover both of them here. Working with sets or bags is really context dependent, but it does have a few advantages.

1. To take the union of two bags, we can just add everything into the other without going through to check for duplicates.
2. When we project relations as bags, we also don't need to search through all pairs to find duplicates.

This allows for efficiency at the cost of memory.

2.1 Structure and Constraints

Definition 2.1 (Primitive Types)

The primitive types are listed.[a]

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.
7. *Null.* `NULL` represents a null value.

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

Theorem 2.1 (`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
```

[a] One thing to note is that keywords are usually written in uppercase by convention.

```
);

DROP TABLE Movies;
```

Theorem 2.2 (DEFAULT)

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

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

2.1.1 Nulls

We are not guaranteed that we will have all data. What if there are some null values? We need some way to handle unknown or missing attribute values. One way is to use a default value (like -1 for age), but this can mess with other operations, such as getting average values of certain groups of users, or can make computations harder since we have to first filter out users with `age=-1` before querying. Another way is to use a valid bit for every attribute. For example, `User(uid, name, age)` could map to `User(uid, name, name_valid, age, age_valid)`, but this is not efficient as well.

A better solution is to decompose the table into multiple relations such that a missing value indicates a missing row in one of the subrelations. For example, we can decompose `User(uid, name, age, pop)` to

1. `UserID(uid)`
2. `UserName(uid, name)`
3. `UserAge(uid, age)`
4. `UserPop(uid, pop)`

This is conceptually the cleanest solution but also complicates things. Firstly, the natural join wouldn't work, since compared to a single table with null values, the natural join of these tables would exclude all tuples that have at least one null value in them.

Definition 2.2 (NULL)

SQL's solution is to have a special value `NULL` indicating an unknown but not empty value. It has the following properties.

1. It holds for every domain (null is the same for booleans, strings, ints, etc.).
2. Any operations like $+, -, \times, >$... leads to a `NULL` value.
3. All aggregate functions except `COUNT` return a `NULL`. `COUNT` also counts null values.

Theorem 2.3 (Three-Valued Logic)

Here is another way to implement the unknown logic with *three-valued logic*. Suppose we set True=1, False=0. Then we can see that given statements x, y which evaluate to $0, 1$,

1. x and y is equivalent to $\min(x, y)$
2. x or y is equivalent to $\max(x, y)$
3. not x is equivalent to $1 - x$

It turns out that if we set unknown=0.5, then this logic also works out very nicely. Check it yourself.

The way operations treat null values is extremely important, and we will add a sentence on how it treats nulls for each operation.

Example 2.1 (Warnings)

Note that null breaks a lot of equivalences, leading to unintended consequences.

1. The two are not equivalent since if we have nulls, the average ignores all nulls, while the second query will sum up all non-nulls and divide by the count including the nulls.

```
SELECT AVG(pop) FROM User;
SELECT SUM(pop) / COUNT(*) FROM User;
```

2. The two are also not equivalent since `pop = pop` is not True, but Unknown, for nulls, so it would not return nulls. The first query would return all tuples, even nulls.

```
SELECT * from User;
SELECT * from User WHERE pop = pop;
```

3. Never compare equality with null, since this never outputs True. Rather, you should use the special keywords `IS NULL` and `IS NOT NULL`.

```
SELECT * FROM User WHERE pop = NULL; // never returns anything
SELECT * FROM User WHERE pop IS NULL; // correct
```

2.1.2 Modifying Tables

This was not an operation we defined in relational algebra, but it is so essential that we should state it now. What if we want to add or delete another attribute? This is quite a major change.

Theorem 2.4 (`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;
```

We have briefly saw how to create and drop tables. To update a table, we can do the following.

Definition 2.3 (`INSERT`)

You can either

1. insert one row

```
INSERT INTO Member VALUES (789, "Muchang")
```

2. or you can insert the output of a query.

```
INSERT INTO Member
(SELECT uid, name FROM User);
```

Definition 2.4 (DELETE)

You can either

1. delete everything from a table (but not the schema, unlike `DROP TABLE`).

```
DELETE FROM Member;
```

2. Delete according to a `WHERE` condition

```
DELETE FROM Member
WHERE age < 18;
```

3. Delete according to a `WHERE` condition extracted from another query.

```
DELETE FROM Member
WHERE uid IN (SELECT uid FROM User WHERE age < 18);
```

Definition 2.5 (UPDATE)

You can either

1. Update a value of an attribute for all tuples.

```
UPDATE User
SET pop = (SELECT AVG(pop) from User);
```

2. Update a value of an attribute for all tuples satisfying a `WHERE` condition.[a]

```
UPDATE User
SET name = 'Barney'
WHERE uid = 182;
```

2.1.3 Constraints

We mainly use constraints to protect the data integrity and relieve the coder from responsibility. It also tells the DBMS about the data so it can optimize better.

Definition 2.6 (NOT NULL Constraints)

This tells that an attribute cannot be entered as null (this is already enforced for keys).

```
CREATE TABLE User
(uid INTEGER NOT NULL,
name VARCHAR(30) NOT NULL,
twitterid VARCHAR(15) NOT NULL,
```

[a] Note that this does not incrementally update the values. It updates all at once from the average of the old table from the subquery.

```
  age INTEGER,
  pop FLOAT);
```

Definition 2.7 (`PRIMARY KEY`, `UNIQUE` Key Constraints)

There are multiple ways to identify keys.

1. Use the `PRIMARY KEY` keyword to make `name` the key. It can be substituted with `UNIQUE`. You can include at most one primary key, but any number of `UNIQUE`. This means that either name or id can be used as a key, but we must choose one primary key, so we are restricted to at most one.

```
CREATE TABLE Movies(
  name CHAR(30) NOT NULL PRIMARY KEY,
  id CHAR(30) NOT NULL UNIQUE,
  year INT NOT NULL,
  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`.

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

Definition 2.8 (Referential Integrity)

Like we said before, a referential integrity means that if an attribute a appears in R, then a must appear in some other T. That is, there are no **dangling pointers**, where $R.a$ does exists but $T.a$ does not. There are names for these:

1. **Foreign keys** is like $R.a$, where it must point to a valid primary key, e.g. `User.uid` or `Group.gid`, which are like entity sets.
2. **Primary keys** are like $T.a$, where it must exist if $R.a$ exists, e.g. `Member.uid` or `Member.gid`, which are relationships.

In SQl, we must make sure that the referenced columns must be the primary key and the referencing columns form a foreign key. There are two ways to do it.

```
CREATE TABLE Member (
uid INT NOT NULL
  REFERNECES User(uid),  // 1. put the references as you define it
gid CHAR(10) NOT NULL,   // 2. define the attribute first
PRIMARY KEY(uid, gid),
FOREIGN KEY (gid) REFERENCES Group(gid)); // 2. then reference it
```

If you have multi-attribute referential integrity, the second method is better.

```
...
```

```
  FOREIGN KEY (gid, time) REFERENCES Group(gid, time));
```

Example 2.2 (Handling Referential Integrity Violations)

Say that you have a referential integrity constraint as above. Then there are two scenarios. If we insert or update a Member row so it refers to a non-existent User.uid, the DBMS will not allow this. If we delete or update a User row whose uid is referenced by some Member row, then there are certain scenarios.

1. *Reject.* The DBMS will reject this.
2. *Cascade.* It will ripple changes (on an update) to all referring rows.
3. *Set null.* It will set all references to NULL.

These options can be specified in SQL.

Definition 2.9 (Tuple/Attribute-Based Checks)

These checks are only associated with a single table and are only checked when a tuple/attribute is inserted/updated. It rejects if the condition evaluates to False, but *True/Unknown are fine* (unlike only True in `WHERE` conditions!). There are two ways to write this in SQL.

```
CREATE TABLE User(
  ... // 1. Directly put the check constraint in definition
  age INTEGER CHECK(age IS NULL OR age > 0),
  ...
);

CREATE TABLE Member(
  uid INTEGER NOT NULL,           // 2. First define attribute
  CHECK(uid IN (SELECT uid FROM User)),  // 2. Then check it
...
)
```

Note that in the second example, this is sort of like a referential integrity constraint. However, this is weaker since it only checks for changes in the Member relation, while the referential integrity constraint is checked for every change in both the Member and User relations. If a check evaluates to False, then the DBMS rejects the insertions/updates.

2.2 Basic Relational Algebra Operations

We won't repeat the definitions of the operations as they are mentioned in the first chapter. We will cover SQL syntax for both set and bag operations. Note that the way set operations work in SQL is that they remove duplicates in the operands and then remove duplicates in the result. In bag operations, we can think of each row a having an implicit count of times c_a it appears in the table.

2.2.1 Renaming

Counterintuitively, the first thing we should know how to do is rename. This is because when we do the rest of these operations, due to naming conflicts, the need to copy a relation, or reduce confusion, we often rename relations and attributes within these queries.

Definition 2.10 (Renaming)

There are two ways you can rename an attribute or a relation.

1. Explicit, using the `AS` keyword.

```
old_name AS new_name
```

2. Implicit, using a space.

```
old_name new_name
```

Example 2.3 ()

Here's a more informative example, using syntax that we will learn later. Note that by renaming the relation, all attributes are renamed as well.

```
SELECT
    s.id,
    s.name AS student_name,
    c.name AS course_name
FROM
    Students s          -- rename relation
    JOIN Courses c      -- rename relation
    ON s.course_id = c.id;
```

2.2.2 Set/Bag Operations

Definition 2.11 (Union)

The **UNION** and `UNION ALL` keywords calculate the set and bag union, respectively. Bag union sums up the counts from the 2 tables.

```
(SELECT * FROM Relation1)  // {1, 2, 3}
UNION
(SELECT * FROM Relation2); // {2, 3, 4}
// {1, 2, 3, 4}
```

```
(SELECT * FROM Relation1)  // {1, 1, 2}
UNION ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1, 1, 1, 2, 2, 2}
```

`UNION` and `UNION ALL` both treat NULLs as equal to each other.

Definition 2.12 (Intersection)

The **INTERSECT** and `INTERSECT ALL` keywords calculate the set and bag intersection, respectively. Bag intersection does a proper-subtract[a]

```
(SELECT * FROM Relation1)  // {1, 2, 3}
INTERSECT
(SELECT * FROM Relation2); // {2, 3, 4}
```

```
(SELECT * FROM Relation1)  // {1, 1, 2}
INTERSECT ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1, 2}
```

Both treat nulls as equal to each other and will return NULL if it appears on both sides.

[a] Subtracts the counts and truncates counts to 0 if negative. So $\{a, a\} - \{a, a, a\} = \{\}$.

Definition 2.13 (Difference)

The **EXCEPT** and `EXCEPT ALL` keywords calculate the set and bag difference, respectively. Bag difference takes the minimum of the two counts.

```
(SELECT * FROM Relation1)  // {1, 2, 3}
EXCEPT
(SELECT * FROM Relation2); // {2, 3, 4}
```

```
(SELECT * FROM Relation1)  // {1, 1, 2}
EXCEPT ALL
(SELECT * FROM Relation2); // {1, 2, 2}
// {1}
```

Both treat NULLs as equal to each other and will return NULL if it appears more times in the first table than the second.

Example 2.4 (Interpretation of Except All)

Look at these two operations on the schema `Poke(uid1, uid2, timestamp)`.

```
(SELECT uid1 FROM Poke)
EXCEPT
(SELECT uid2 FROM Poke);
```

```
(SELECT uid1 FROM Poke)
EXCEPT ALL
(SELECT uid2 FROM Poke);
```

The first operation returns all users who poked others but were never poked, while the second returns all users who poked others more than they were poked.

2.2.3 Predicates, Projection, Selection

Definition 2.14 (Condition/Predicate)

The condition p used in in selection and joins is implemented with the `WHERE` keyword. The operations used in predicates are listed below.

1. Equality is denoted with `=`, and not equals is denoted with `<>`. Equality evaluates to true if both attribute values are not NULL and match.
2. Comparisons is done with `>`, `<`, `>=`, `<=`.
3. The `IN` and `NOT IN` gives you filters that restrict the domain of a certain attribute.

```
SELECT *
FROM relation
WHERE sex in ['male', 'female'];
```

`WHERE` will only select rows for which the condition is True (not False or unknown).

Both selection and projection, known as **queries**, is done with the same keyword `SELECT`, and we can do both of them at the same time (the DBMS will determine which operation to do first). Unlike set/bag operations before, `SELECT` is by default a bag operator.

Definition 2.15 (Selection)

The `SELECT` and `SELECT DISTINCT` keywords apply the selection bag and set operator, respectively. The $*$ species all attributes in the relation.

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

Definition 2.16 (Projection)

The `SELECT` and `SELECT DISTINCT` keywords apply the projection bag and set operator, respectively. Rather than putting an $*$, we should now specify our desired attributes.

```
SELECT date, age, uid
FROM relation
```

Example 2.5 (Comparing Null values always returns Null)

The two are also not equivalent since `pop = pop` is not True, but Unknown, for nulls, so it would not return nulls. The first query would return all tuples, even nulls.

```
SELECT * from User;
SELECT * from User WHERE pop = pop;
```

Example 2.6 (Never Compare Equality with Null)

Never compare equality with null, since this never outputs True. Rather, you should use the special keywords `IS NULL` and `IS NOT NULL`.

```
SELECT * FROM User WHERE pop = NULL; // never returns anything
SELECT * FROM User WHERE pop IS NULL; // correct
```

2.2.4 Products and Joins

All products and joins use bag semantics and to use set semantics just use a `SELECT DISTINCT`.

Definition 2.17 (Cartesian Product)

The `CROSS JOIN` keyword or the `,` operator is used to calculate the cartesian product of two relations using bag semantics.

```
SELECT *
FROM table1
CROSS JOIN table2;

SELECT *
FROM table1, table2;
```

Definition 2.18 (Theta Join)

A theta join is performed by simply adding a `WHERE` clause after a cross join, using bag semantics.

```
SELECT *
FROM table1, table2
WHERE table1.A = table2.B;
```

Definition 2.19 (Natural/Inner Join)

The `JOIN` keyword returns records that have matching values in both tables using bag semantics.

```
SELECT * FROM Students s
JOIN Courses c ON s.id = c.id;
```

For inner join, note that NULL attributes do not match each other (as stated in equality predicate) so they are not included.

Definition 2.20 (Left Outer Join)

`LEFT (OUTER) JOIN`: Returns all records from the left table R, and the matched records from the right table S. For a given $r \in R$, if there are no $s \in S$ that matches it, then the S attributes will all be NULL.

```
SELECT * FROM Students s
LEFT JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the left table.

Definition 2.21 (Right Outer Join)

`RIGHT (OUTER) JOIN`: Returns all records from the right table S, and the matched records from the left table R with potential NULLs. For a given $s \in S$, if there are no $r \in R$ that matches it, then the R attributes will all be NULL.

```
SELECT * FROM Students s
RIGHT JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the right table.

Definition 2.22 (Full Outer Join)

`FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table with potential NULLs.

```
SELECT * FROM Students s
FULL OUTER JOIN Courses c ON s.id = c.id;
```

This keeps all NULLs on the both tables.

Example 2.7 (Motivation for Outer Join)

Take a look at the following motivating example. Suppose we want to find all members and their respective groups from `Group(gid, name)`, `Member(uid, gid)`, `User(uid, name)`. Then we can write the query

```
SELECT g.gid, g.name AS gname,
    u.uid, u.name AS uname
FROM Group g, Member m, User u
WHERE g.gid = m.gid AND m.uid = u.uid;
```

This looks fine, but what happens if *Group* is empty? That is, there is a group in the *Group* relation

but does not appear in the *Member* relation. Then, `m.gid` will evaluate to False and would not appear in the joined table, which is fine, but what if we wanted to make sure all groups appeared in this master membership table? If a group is empty, we may want it to just have null values for `uid` and `uname`. In this case, we want to use outer join.

Example 2.8 (Complex Example with Beers)

Given the schemas
1. `Frequents(drinker, bar, times_a_week)`,
2. `Serves(bar, beer, price)`,
3. `Likes(drinker, beer)`,

say that we want to select drinkers and bars that visit the bars at least 2 times a week and the bars serves at least 2 beers liked by the drinker and count the number of beers served by the bars that are liked by the drinker. The query is shown below.

```
SELECT F.drinker, F.bar, COUNT(L.beer)
FROM Frequents F, Serves S, Likes L
WHERE F.drinker = L.drinker
  AND F.bar = S.bar
  AND L.beer = S.beer
  AND F.times_a_week >= 2
GROUP BY F.drinker, F.bar
HAVING COUNT(L.beer) >= 2
```

2.3 Arithmetic and Sorting

We can also do arithmetic on columns.

Definition 2.23 (Arithmetic Operations)

SQL supports
1. *Addition* using `+`. It returns NULL if at least 1 of the operands are NULL.
2. *Subtraction* using `-`. It returns NULL if at least 1 of the operands are NULL.
3. *Multiplication* using . It returns NULL if at least 1 of the operands are NULL.
4. *Division* using `/`. It returns NULL if at least 1 of the operands are NULL or there is division by 0 (or may result in error depending on DBMS). For integers it will truncate.
5. *Modulo* using `%`. It returns NULL if at least 1 of the operands are NULL.

You can use parentheses to compose operations together.

Example 2.9 (Arithmetic)

Here is a comphrensive example.

```
SELECT
  price + tax as total_price,
  salary - deductions as net_salary,
  price * quantity as total_value,
  total / count as average,
  id % 2 as is_odd
FROM Orders;
```

Definition 2.24 (Sorting)

The `ORDER BY` keyword sorts the relation. In fact, it is more of a display operation since the tuples in the relation are unsorted by definition.

1. `ORDER BY attribute ASC` orders in ascending order. This the default.
2. `ORDER BY attribute DESC` orders in descending order.

2.4 Aggregate Functions

Definition 2.25 (Standard SQL Aggregate Functions)

Aggregate functions compute something from a collection of tuples, rather than a single row.

1. `COUNT` counts the number of rows, including NULLS, in a query. `COUNT(DISTINCT att)` counts the distinct count of an attribute in a query and ignores NULLs.
2. `SUM` counts the sum of the values of an attribute in a query. It ignores NULL values.
3. `AVG` is the average. It ignores NULL values.
4. `MIN` is the minimum of an attribute. It ignores NULL values and only returns NULL if all values are NULL.
5. `MAX` is the maximum of an attribute. It ignores NULL values and only returns NULL if all values are NULL.

Example 2.10 (Calculate Popularity within Age Group)

If we want to find the number of users under 18 and their average popularity, then we can write

```
SELECT COUNT(*), AVG(pop)
FROM User
WHERE age < 18;
```

Example 2.11 (Average is Not Sum/Count)

The two are not equivalent since if we have nulls, the average ignores all nulls, while the second query will sum up all non-nulls and divide by the count including the nulls.

```
SELECT AVG(pop) FROM User;
SELECT SUM(pop) / COUNT(*) FROM User;
```

2.4.1 Group By

By default, the aggregate function operates on the entire relation. If we want more fine grained control by looking at a subset of tuples, then we can use the `GROUP BY` clause.

Definition 2.26 (`GROUP BY`)

`GROUP BY` is used when you want to group the query by equal values of the attributes. The syntax is

```
SELECT ... FROM ... WHERE ...
GROUP BY age;
```

To parse this, first form the groups based on the same values of all attributes in the group by clause. Then, output only one row in the select clause per group. We can look at the following order

1. `FROM`. Look at where we are querying from.
2. `WHERE`. Find out if this condition is satisfied to filter the main query.
3. `GROUP BY`. Group rows according to the values of the GROUP BY columns.
4. `SELECT`. Compute the select query for each group. The number of groups should be equal to the number of rows in the final output.

Note that if a query uses aggregation/group by, every column referenced in select must be either aggregated or a group by column.

Example 2.12 ()

If we want to find the number of users in a certain age and their average popularity, for all ages, then we can write

```
SELECT age, AVG(pop)
FROM Users
GROUP BY age;
```

You don't necessarily have to report the group by attribute in the select. The two following examples are perfectly fine, though in the right query, `age` may not functionally determine `AVG(pop)`.

```
SELECT AVG(pop)
FROM User
GROUP BY age;
```

```
SELECT age, AVG(pop)
FROM User
GROUP BY age, name;
```

In the example below, this left query is not syntactically correct since `name` is not in the group by clause or aggregated. This is true even if `age` functionally determines `name`. Neither is the right since the lack of a group by clause means that the aggregate query is over the entire relation, which has multiple uid values.

```
SELECT age, name, AVG(pop)
FROM User
GROUP BY age;
```

```
SELECT uid, MAX(pop)
FROM User;
.
```

As you can see, this is great to use for aggregate functions. If there is no group by clause, this is equivalent to grouping everything together.

Example 2.13 ()

Given the relation

A	B	C
1	1	10
2	1	8
1	1	10
2	3	8
2	1	6
2	2	2

Table 5: Original relation.

Running the query

```
SELECT A, B, SUM(C) AS S
FROM R
```

```
GROUP BY A, B;
```

gives

A	B	S
1	1	20
2	1	14
2	3	8
2	2	2

Table 6: Our query.

2.4.2 Having

Definition 2.27 (HAVING)

If you want to filter out groups having certain conditions, you must use the `HAVING` keyword rather than `WHERE`. The syntax is

```
SELECT A, B, SUM(C) FROM ... WHERE ...
GROUP BY ...
HAVING SUM(C) < 10;
```

You should look at the `HAVING` clause after you look at the `GROUP BY` but before `SELECT`. `HAVING` will only select rows for which the condition is True (not False or null).

Example 2.14 ()

Given the relation

A	B	C
1	1	10
2	1	8
1	1	10
2	3	8
2	1	6
2	2	2

Table 7: Original relation.

Running the query

```
SELECT A, B, SUM(C) AS S
FROM R
GROUP BY A, B
HAVING SUM(C) > 8;
```

gives

A	B	S
1	1	20
2	1	14

Table 8: Our query.

Example 2.15 ()

Given the schema `Sailor(sid, name, age, rating)`, to find the age of the youngest sailor with age at least 18, for each rating with at least 2 sailors, we can run the query.

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUPBY S.rating
HAVING COUNT(*) > 1;
```

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

(a) Original table.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5
3	25.5

(b) Filter out ages less than 18.

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

(c) Group by rating.

Figure 17: The thought process. Data tables showing the transformation of rating and age values.

2.5 Nested Queries and Views

2.5.1 Subqueries

We have so far worked with a single query consisting of a single select statement. However, we can extend this by including queries that have other queries in them. Conceptually, this is easy to understand since we are just composing more operators in relational algebra.

Definition 2.28 (Subquery)

A **subquery** is a query contained inside another query.

Definition 2.29 (Correlated Subqueries)

A **correlated subquery** is a subquery that is an input to a parent query. It is a query that needs a parameter from the main query and are generally slower to run. They are hard to parse, but generally you should look in this order (though this is not how the DBMS actually implements it).

1. `FROM`. Look at where we are querying from and for loop over each row.

2. `SELECT`. For each row in the outer query, take whatever values/parameters are needed for this row to input into the subquery.
3. `WHERE`. Find out if the condition resulting from the subquery is satisfied. If so, keep this.

Definition 2.30 (`EXISTS`)

The `EXISTS`(subquery) keyword checks if a subquery is empty or not, and `NOT EXISTS` checks the negation.

Example 2.16 (Age)

Given `User(name, age)`, say that we want to get all users whose age is equal to a person named Bart. Then, we want to select all users from the relation. For each user, we perform the subquery where this original tuple age coincides the others age and the others name is Bart. The outer query only returns those rows for which the `EXISTS` subquery returned true. Then we can write the two equivalent queries.

```
SELECT *
FROM User as u
WHERE EXISTS(SELECT * FROM User
               WHERE name = "Bart"
               AND age = u.age);
```

```
SELECT *
FROM User
WHERE age IN(SELECT age
               FROM User
               WHERE name = 'Bart');
```

To understand this, we use the rules from above.

1. `FROM`. We take each row in the outer query.
2. `SELECT`. For each row, we take the age from that row.
3. `WHERE`. The subquery looks for any user named `Bart` with that same age. If such a user exists, the row is kept.

Here is a very useful keyword that simplifies complex nested queries, one example of a **common table expression (CTEs)**.

Definition 2.31 (`WITH`)

The `WITH` clause aliases many relations returned from queries.

```
WITH Temp1 AS (SELECT ...),
     Temp2 AS (SELECT ...)
SELECT X, Y
FROM Temp1, Temp2
WHERE ...
```

Example 2.17 ()

To extend the Bart age example, we can think of temporarily storing a query of only Bart's ages, and then comparing it when doing the main query over `User`.

```
WITH BartAge AS
  (SELECT age
  FROM User
  WHERE name = 'Bart')
SELECT U.uid, U.name, U.pop,
FROM User U, BartAge B
```

```
WHERE U.age = B.age;
```

2.5.2 Scalar Subqueries

Definition 2.32 (Scalar Subqueries)

Sometimes, if a query returns 1 scalar, then you can use it in your `WHERE` clause. You must be sure that a query will return exactly 1 scalar (not 0 and not more than 1), or there will be a runtime error.

Example 2.18 ()

If we want to compute users with the same age as Bart, we can write

```
SELECT * FROM User
WHERE age = (SELECT age from User WHERE name = 'Bart');
```

However, we may not know if Bart functionally determines age, so we must be careful.

2.5.3 Quantified Subqueries

Definition 2.33 (`ALL`, `ANY`)

We have the following **quantified subqueries**, which performs a broadcasting condition.

1. `ALL` checks if all conditions are true. If any are NULL, then the whole result is NULL.
2. `ANY` checks if any condition is true. If any are true, then the whole result is true. If all are either NULL or false, then `ANY` will return NULL.

Example 2.19 (Popular Users)

Which users are the most popular? We can write this in two ways.

```
SELECT *
FROM User
WHERE pop >= ALL(SELECT pop from User);
.
```

```
SELECT *
FROM User
WHERE NOT
  (pop < ANY(SELECT pop from User));
```

To review, here are more ways you can do the same query.

```
SELECT *
FROM User AS u
WHERE NOT EXISTS
  (SELECT * FROM User
  WHERE pop > u.pop);
```

```
SELECT * FROM User
WHERE uid NOT IN
  (SELECT u1.uid
  FROM User as u1, User as u2
  WHERE u1.pop < u2.pop);
```

2.5.4 Views

Definition 2.34 (View)

A **view** is a virtual table that can be used across other queries. Tables used in defining a view are called **base tables**.

Example 2.20 (Jessica's Circle)

You can create a temporary table that can be used for future queries.

```
CREATE VIEW JessicaCircle AS
SELECT * FROM User
WHERE uid in (SELECT uid FROM Member WHERE gid = 'jes');
```

Once you are done, you can drop this view with

```
DROP VIEW JessicaCircle;
```

2.6 Recursion

Recursion seems like a foreign concept in relational databases and was not supported until SQL3 , but we can consider the following motivating example.

Example 2.21 (Ancestors)

Given a schema `Parent(parent, child)`, we can find Bart's grandparents with the following query.

```
SELECT p1.parent as grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent
AND p2.child = 'Bart';
```

We can do the same for parents, great grandparents, and so forth, but there is no clean way to get all of Bart's ancestors with a single query.

Definition 2.35 (Recursion)

You can implement a recursive query using the `WITH RECURSIVE` keyword.

Example 2.22 (Recursive Query for Ancestors)

Lines 2-7 defines the relation recursively, and then lines 8-10 is the query using the relation defined in the WITH clause.

```
1 WITH RECURSIVE
2 Ancestor(anc, desc) AS
3 ((SELECT parent, child FROM Parent) -- base case
4 UNION
5 (SELECT a1.anc, a2.desc             -- recursion step
6 FROM Ancestors a1, Ancestors a2     -- recursion step
7 WHERE a1.desc = a2.anc))            -- recursion step
8 SELECT anc
```

```
9  FROM Ancestor
10 WHERE desc = 'Bart';
```

3 Query Processing and Optimization

Definition 3.1 (Disk/Memory Blocks)

To set up some notation, let $B(R)$ represent the number of disk blocks that relation R takes up, and say M is the number of blocks available in our memory. We start off with some simple operations.

Example 3.1 (Basic Block Storage)

Suppose we have relation R with $|R| = 1000$ and each block can hold 30 tuples. Then $B(R) = \lceil 1000/30 \rceil = 34$, or 35 if there is overlap.

Definition 3.2 (Buffer Blocks)

It turns out that whenever we output data to stdout, the blocks need to be stored in a **buffer block**. If the buffer block is full, then it is flushed to stdout.

Example 3.2 (Cost of Querying Everything)

If we do `SELECT * FROM R`, then

1. we are at most retrieving $B(R)$ pages from disk to memory, so our IO cost is $B(R)$.
2. Our memory cost is 2 pages since we take each page, load it to memory, and put the answer in the output page. The next page (if any) can overwrite the previously loaded page.

We can stop early if we lookup by key. Remember that this is not counting the cost of writing the result out.

It turns out that the efficiency of most operations defined on modern DBMS depends on two things: sorting and hashing. We'll divide up our analysis this way, focusing on their applications in joining, which tends to be the mostly used and expensive operation. Note that there are many ways to process the same query. We can in the most basic sense just scan the entire relation in disk. We can sort it. We can use a tree or hash index, and so on. All have different performance characteristics and make different assumptions about the data, so the choice is really problem-dependent. What a DBMS does is implements all alternatives and let the *query optimizer* choose at runtime. We'll talk about the algorithms now and talk about the query optimizer later.

3.1 Brute-Force Algorithms

We start off with the most brute-force algorithms of theta joins. In here, we describe how to implement it, its IO cost, and its memory cost. In the following, when we compute $R \bowtie_p S$, R is called the **outer table** and S the **inner table**. Furthermore, another convention is that when we calculate IO cost, we *do not* factor in the cost of writing our result back to disk.

3.1.1 Nested Loop Joins

Our first algorithm simply takes in every block from R, and for every tuple $r \in R$, we take in every tuple in S to calculate the predicate $p(r, s)$.

Algorithm 3.1 (Nested Loop Join)

Nested-loop join just uses a brute-force nested loop when computing $R \bowtie_p S$.

Algorithm 1 Nested loop Join

```
Require: Outer table R, Inner table S
  function NestedLoop(R, S)
      for each block B_R ∈ R do
          load B_R into memory block M_1
          for each tuple r ∈ B_R do
              for each block B_S ∈ S do
                  load B_S into memory block M_2
                  for each tuple s ∈ B_S do
                      if p(r, s) is true then
                          Write r ⋈ s into buffer block M_3
                          Flush M_3 to stdout when it's full.
                      end if
                  end for
              end for
          end for
      end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R
2. For every tuple $r \in R$, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + |R| \cdot B(S) \tag{11}$$

with memory cost 3 (since we use M_1, M_2, M_3 blocks).

This is clearly not efficient since it requires a lot of IOs. We can make a slight improvement by trying to do as much as we can with the loaded blocks in memory.

Algorithm 3.2 (Block-Based Nested-Loop Join)

Block-based nested-loop join loops over the blocks rather than the tuples in the outer loops.

```
Require: Outer table R, Inner table S
  function BlockNestedLoopJoin(R, S)
      for each block B_R ∈ R do
          load B_R into memory block M_1
          for each block B_S ∈ S do
              load B_S into memory block M_2
              for each r ∈ B_R, s ∈ B_S do
                  if p(r, s) is true then
                      Write r ⋈ s into buffer block M_3
                      Flush M_3 to stdout when full.
                  end if
              end for
          end for
      end for
  end function
```

The IO cost of this is

1. $B(R)$ to load R.

2. For every block B_R, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$\text{IO} = B(R) + B(R) \cdot B(S) \tag{12}$$

The memory cost is 3 (since we use M_1, M_2, M_3 blocks).

Algorithm 3.3 (Saturated Block-Based Nested-Loop Join)

The next optimization is to use more memory by basically stuffing the memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory.

Require: Outer table R, Inner table S
  function SatBlockNestedLoopJoin(R, S)
    for each set of blocks $\mathbf{B} = \{B_{i+1}, \dots, B_{i+(M-2)}\} \subset R$ **do**
      load $\mathbf{B}$ into memory blocks $M_1, \dots, M_{M-2}$.
      for each $B_S \in S$ **do**
        load B_S into memory block M_{M-1}.
        for each $r \in \mathbf{B}, s \in B_S$ **do**
          if $p(r, s)$ is true **then**
            Write $r \bowtie s$ into buffer block M_M.
            Flush M_M to stdout when it's full.
          end if
        end for
      end for
    end for
  end function

The total IO cost of this is

1. $B(R)$ to load R.
2. For every set of $M-2$ blocks, we run through all of $s \in S$, requiring $B(S)$ IOs.

Therefore

$$B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S) \approx B(R) \cdot B(S)/M \tag{13}$$

You want to pick the bigger table as R since you want the smaller table S to be loaded/streamed in multiple times.

Algorithm 3.4 (Index Nested Loop Join)

If we want to compute $R \bowtie_{R.A=S.B} S$, the idea is to use the value of $R.A$ to probe the index on $S(B)$. That is, for each block of R, we load it into memory, and for each r in the block, we use the index on $S(B)$ to retrieve s with $s.B = r.A$, and output rs.

The IO runtime, assuming that S is unclustered and secondary, is

$$B(R) + |R| \cdot (\text{indexlookup}) \tag{14}$$

Since typically the cost of an index lookup is 2-4 IOs, it beats other join methods mentioned later if $|R|$ is not too big. Since this does not scale at all with S, it is better to pick R to be the smaller relation. The memory requirement as with other operations is 3 blocks.

3.2 Sort-Based Algorithms

3.2.1 External Merge Sort

Now let's talk about processing of queries, namely how sorting works in a database system, called **external merge sort**. In an algorithm course, we know that the runtime is $O(n \log n)$, but this is for CPU comparisons where the entire list is loaded in memory. This is extremely trivial in comparison to the IO commands we use in databases, so we will compute the runtime of sorting a relation by an attribute in terms of IO executions.

The problem is that we want to sort R but R does not fit in memory. We divide this algorithm into **passes** which deals with intermediate sequences of sorted blocks called **runs**.

1. Pass 0: Read M blocks of R at a time, sort them in memory, and write out the sorted blocks, which are called *level-0 runs*.
2. Pass 1: Read $M-1$ blocks of the level 0 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-1 runs*.[2]
3. Pass 2: Read $M-1$ blocks of the level 1 runs at a time, sort/merge them in memory, and write out the sorted blocks, which are called *level-2 runs*.
4. ...
5. Final pass produces one sorted run.

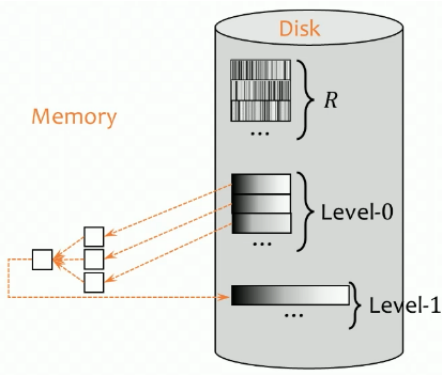

Figure 18

Algorithm 3.5 (External Merge Sort)

The implementation has a lot of details. Not finished yet.

[2]The reason we need $M-1$ rather than M is that now we are merging. We are not merging in-place so we need this extra buffer to store as we traverse our pointers down each of the $M-1$ blocks.

Require: Relation R
  function ExternalMergeSort(R)
    $L = [0]$ ▷ Array storing number of level-i runs
    while while there are blocks to read from R **do** ▷ First pass
      read the next M blocks $B = \{B_1, \dots, B_M\}$ at a time and store it in memory.
      sort B to generate a level-0 run $B^{(0)} = \{B'_1, \dots, B'_M\}$.
      write $B^{(0)}$ to disk.
      $L[0] += 1$
    end while
    while $L[-1] \geq M$ **do**
      append 0 to L
      let $\mathbf{B} = \{B\}$ be the set of previous runs
      while there exists blocks to be read in $\mathbf{B}$ **do**
        read $M-1$ blocks starting from the beginning of each run into memory.
        sort them to produce the ith run $B^{(i)}$.
        write $B^{(i)}$ to disk.
      end while
    end while
  end function

To compute the cost, we know that

1. in pass 0, we read M blocks of R at a time, sort them, and write out a level 0 run, so there are
$$\lceil B(R)/M \rceil \tag{15}$$
level 0 sorted runs, or passes.
2. in pass i, we merge $M-1$ level $(i-1)$ runs at a time, and write out a level i run. We have $M-1$ memory blocks for input and 1 to buffer output, so
$$\text{Num. of level i runs} = \left\lceil \frac{\text{Num. of level (i-1) runs}}{M-1} \right\rceil \tag{16}$$
3. The final pass produces 1 sorted run.

Therefore, the number of passes is approximately

$$\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1 \tag{17}$$

and the number of IOs is $2B(R)$ since each pass reads the entire relation once and write it once. The memory requirement is M (as much as possible).

Example 3.3 (Baby Merge)

Assume $M = 3$, with each block able to hold at most 1 number. Assume that we have an input (relation)

$$1, 7, 4, 5, 2, 8, 3, 6, 9 \tag{18}$$

Then we go through multiple passes.

1. Pass 0 will consist of 3 runs. You load each of the 3 numbers in memory and sort them.
$$1, 7, 4 \mapsto 1, 4, 7 \tag{19}$$
$$5, 2, 8 \mapsto 2, 5, 8 \tag{20}$$
$$9, 6, 3 \mapsto 3, 6, 9 \tag{21}$$

2. Pass 1. You merge them together by first taking 1 and 2, loading them in memory, and then comparing which one should go first. Once 1 is outputted, then the next number 4 overwrites 1 in memory, and then 2 is outputted, and so on.

$$1, 4, 7 + 2, 5, 8 \mapsto 1, 2, 4, 5, 7, 8 \tag{22}$$
$$3, 6, 9 \tag{23}$$

3. Pass 2. Merges the final two relations.

$$1, 2, 3, 4, 5, 7, 8 + 3, 6, 9 \mapsto 1, 2, 3, 4, 5, 6, 7, 8, 9 \tag{24}$$

Therefore, pass 0 uses all M pages to sort, and after that, when we merge, we only use $M - 1$ pages to merge the inputs together and 1 page for the output.

Some performance improvements include:

1. *Double Buffering*. You allocate an additional block for each run, and while you are processing (merging the relations in memory), you run the IO concurrently and store it in the new block to save some time.
2. *Blocked IO*. Instead of reading/writing one disk block at a time, we can read/write a bunch of them in clusters. This is sort of like parallelization where you don't output just one block, but multiple blocks done from multiple processing.

The problem with both of these is that we have smaller fan-in, i.e. more passes. Since we are using more blocks per run than we have, we can look at fewer runs at once.

3.2.2 Sort Merge Joins

Now that we know how to sort, let's exploit this to optimize joins beyond nested-loops. We introduce a naive version of sort-merge join.

Algorithm 3.6 (Naive Sort-Merge Join)

A clever way is to first sort R and S by their join attributes, and then merge. Given that the first tuples in sorted R, S is r, s, we do repeat until one of the R or S is exhausted.

1. If $r.A > s.B$, then $s =$ next tuple in S
2. Else if $r.A < s.B$, then $r =$ next tuple in R.
3. Else output all matching tuples and $r, s =$ next in R, S, which is basically a nested loop.

Therefore, given that it takes $\text{sort}(R), \text{sort}(S)$ to sort R, S (the equations above is too cluttered to write), the IO cost consists of

1. Sorting R and S.
2. We then must write R and S to disk in order to prepare for merging, so $B(R) + B(S)$.
3. We then must write R and S back into memory, one at a time, to merge them, so $B(R) + B(S)$.

Therefore, the IO is really just $2B(R) + 2B(S)$ more than it takes to sort both R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) + 2B(R) + 2B(S) \tag{25}$$

which is worst case $B(R) \cdot B(S)$ when everything joins.

Algorithm 2

Require:

function FUNC(x)
end function

Example 3.4 (Worst Case)

To see the worst case when the IO cost is $B(R) \cdot B(S)$, look at the following example. By the time we got to the first 3, we can't just increment the pointers for both relations. We must scan through all of the tuples of A with value 3 and all those in B with value 3 and do a nested loop to join them.

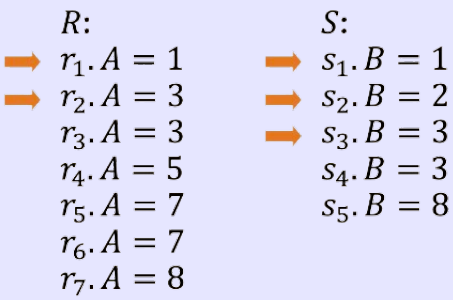

Figure 19

We have completely isolated the sorting phase and the merging phase, but we don't have to do this. Just like regular merge-sort, we can integrate them together to save IO in the last merge phase.

Algorithm 3.7 (Optimized Sort-Merge Join)

The algorithm is just slightly modified from the naive implementation. After the final pass from sorting both R and S, say that we have W_R and W_S final runs such that

$$M > W_R + W_S \tag{26}$$

We can assume that this is true since if it wasn't, we can just add another pass of external sort to reduce one of the W's. Then, we can do these 3 things simultaneously.

1. We can load in the next smallest block from R from its runs, merge them together.
2. We can load in the next smallest block from S form its runs, merge them together.
3. We can join the merged blocks *in memory* and output to the buffer to be flushed.

This saves us the IO cost of writing the sorted runs back into memory and then loading them again to write, giving us a total IO cost that is equal to that of simply sorting R and S.

$$\text{IO} = \text{sort}(R) + \text{sort}(S) \tag{27}$$

The memory varies depending on how many passes, but if R and S are moderately big in that we need full memory to sort them, then the memory cost is M (we use everything).

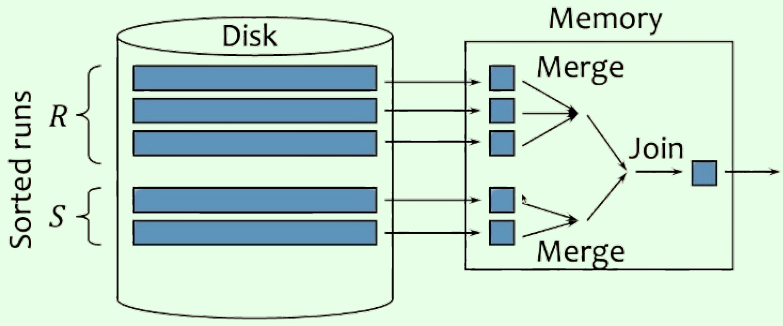

Figure 20: Sorting: produce sorted runs for R and S such that there are fewer than M of them total. Then in merge and join, we merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!

Example 3.5 (Two-Pass SMJ)

If SMJ completes in two passes, then the IOs is really cheap since we are basically getting a $2B(R) + 2B(S)$ cost of a level-0 pass, plus the final merge-join step which takes another $B(R) + B(S)$.

$$\text{IO} = 3(B(R) + B(S)) \tag{28}$$

If SMJ cannot complete in 2 passes, then we repeatedly merge to reduce the number of runs as necessary before the final merge and join.

3.2.3 Zig-Zag Join

Definition 3.3 (Zig Zag Join using Ordered Indices)

To compute $R \bowtie_{R.A=S.B} S$, the idea is to use the ordering provided by the indices on $R(A)$ and $S(B)$ to eliminate the sorting step of merge-join. The idea is similar to sort-merge join. We start at the leftmost leaf node of both indices of R and S, and traverse (right) through the leaves, querying both of the data at leaf a in R and b in S if the leaf values are equal.
Note that we don't even have to traverse through all leaves. If we find that a key is large, we can just start from the root node to traverse, possibly skipping many keys that don't match and not incurring all those IO costs. This can be helpful if the matching keys are distributed sparsely.

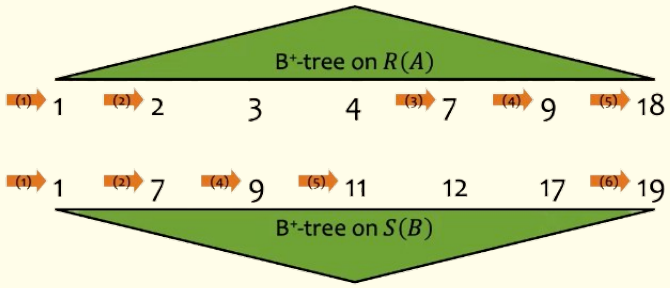

Figure 21: We see that the B+ tree of B has value 7 while that of A has value 2. Rather than traversing $2 \mapsto 3 \mapsto 4 \mapsto 7$, we can just traverse to 7 from the root node of A. This can give us a stronger bound.

3.2.4 Other Sort Based Algorithms

The set union, intersection, difference is pretty much just like SMJ.

For duplicate elimination, you simply modify it so that during both the sort and merge steps, you eliminate duplicates if you find any.

For grouping and aggregation, you do external merge sort by the group-by columns. The trick is you produce "partial" aggregate values in each run and combine them using merge.

3.3 Hash-Based Algorithms

3.3.1 Hash Join

Hash joining is useful when dealing with equality predicates: $R \bowtie_{R.A=S.B} S$.

Definition 3.4 (Hash Join)

The main idea of **hash join** is that we want to partition R and S by hashing (using a hash function that maps to $M-1$ values) their join attributes and then consider corresponding partitions (that get mapped to the same hash value) of R and S. If $r.A$ and $s.B$ get hashed to the same number, they might join, and if they don't, then they definitely won't join. The figure below nicely visualizes this.

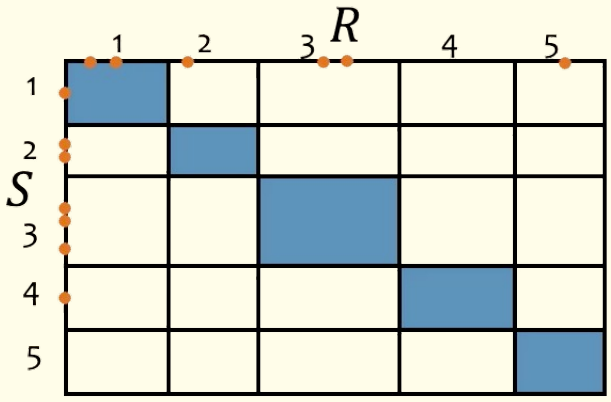

Figure 22: Say that the orange points represent the hashed values of $r.A$ and $s.B$ for $r, s \in R, S$. The nested loop considers all slots (all pairs of orange points between R and S), but hash join considers only those along the diagonal.

Then, in the **probing phase**, we simply read in each partition (the set of tuples that map to the same hash) of R into memory, stream in the corresponding partition of S, compare their *values* (not hashes since they are equal), and join if they are equal. If we cannot fit in a partition into memory, we just take a second hash function and hash it again to divide it into even smaller partitions (parallels merge-sort join).

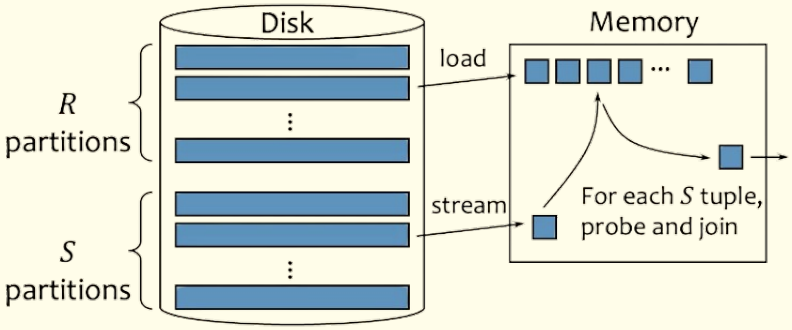

Figure 23

Therefore, if the hash join completes in two passes, the IO runtime is

$$3(B(R) + B(S)) \tag{29}$$

which is similar to merge-sort join. As for the memory requirement, let's first assume that in the probing phase, we should have enough memory to fit one partition of R, i.e. $M-1 > \lceil B(R)/(M-1) \rceil$, so solving for it roughly gives $M > \sqrt{B(R)} + 1$. We can always pick R to be the smaller relation, so roughly

$$M > \sqrt{\min\{B(R), B(S)\}} + 1 \tag{30}$$

Theorem 3.1 (Hash vs Sort-Merge Join)

To compare hash join and SMU, note that their IOs are the same, but for memory requirements, hash join is lower, especially when the two relations have very different sizes.

$$\sqrt{\min\{B(R), B(S)\}} + 1 < \sqrt{B(R) + B(S)} \tag{31}$$

Some other factors include the quality of the hash (may not generate evenly sized partitions). Furthermore, hash join does not support inequality joins unlike SMJ, and SMJ wins if either R and/or S is already sorted. SMJ also wins if the result needs to be in sorted order.

Sometimes, even block nested loop join may win in the following cases.

1. if many tuples join (as in the size of the join is $|S| \cdot |R|$) since we are doing unnecessary processing in hash/merge-sort joins.
2. if we have black-box predicates where we may not know the truth/false values of the θ

3.3.2 Other Hash Based Algorithms

The union, difference, and intersection are more or less like hash join.

For duplicate elimination, we can check for duplicates within each partition/bucket.

For grouping/aggregation, we can apply the hash functions to the group-by columns. Tuples in the same group must end up in the same partition/bucket. Or we may keep a running aggregate value for each group.

To compare the duality of sort and hash, note that

1. in sorting, we have a physical division and logical combination
2. in hashing, we have a logical division and physical combination

When handling large inputs,

1. in sorting we have multi-level merge
2. in hashing we have recursive partitioning

For IO patterns,

1. in sorting we have sequential write and random read (merge)
2. in hashing we have random write and sequential read (partition)

3.4 Exercises

Here are some exercises for calculating IO costs of these joins.

Example 3.6 (Join Operations)

Consider the two tables
1. `Orders(OrderID, CustomerID, OrderDate, TotalAmount)`
2. `Customers(CustomerID, Address)`

`Orders.CustomerID` is a foreign key referring to `Customers.CustomerID`.
The rows are not sorted on any particular attribute. We want to join the two tables using the attribute CustomerID:

$$\text{Orders} \bowtie_{\text{Orders.CustomerID} = \text{Customers.CustomerID}} \text{Customers}$$

The inner and outer tables can be swapped to reduce I/O costs in the following questions.
Assume the following:

- The cost metric is the number of page/block I/Os unless otherwise noted.
- DO NOT count the I/O cost of writing out the final result unless otherwise noted.
- M = 19 blocks (= pages) available in memory unless otherwise noted.
- Table Orders contains 50,000 records (rows/tuples) on disk. One block can contain 20 Orders-tuples.
- Table Customers contains 20,000 records (rows/tuples) on disk. One block can contain 10 Customers-tuples.
- Assume uniform distribution for Orders.CustomerID and Customers.CustomerID – i.e. the same number of Orders-tuples join with a Customers-tuple.
- Ignore page boundary wherever applicable.
- Assume uniform distributions wherever applicable.

Question 2.1

(a) For the Orders and Customers tables, we want to perform a nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost? (Choose the table that will reduce the I/O cost as the outer table.)

(b) For the Orders and Customers tables, we want to perform a block-based nested-loop join (**using 3 memory blocks**). What is the **minimum** total I/O cost?

For (a), we choose $R =$ Customers as the outer table since it is smaller. Therefore, our IO cost is

$$B(R) + |R| \cdot B(S) = 2000 + 20,000 \cdot 2500 = 50,002,000 \tag{32}$$

For (b), we also choose $R =$ Customers as outer since it's smaller. The IO cost is

$$B(R) + B(R) \cdot B(S) = 2000 + 2000 * 2500 = 5,002,000 \tag{33}$$

Question 2.2

(a) If we want to perform an external merge sort on the Orders table, how many level-0 runs does the external merge sort produce for the Orders table (M=19)?

(b) Continuing with the question (a), how many passes in total does the external merge sort take (including the first sorting pass)? Show the calculations for each pass (including the number of runs and size of each run).

(c) What is the total I/O cost of the external merge sort for the Orders table? Remember, do not count the cost of final write.

(d) Do an improved sort-merge-join, i.e., do merge and join in the same pass when the total number of sorted runs from Orders and Customers will fit in memory including an output block. Compute the minimum total I/O cost of sort-merge-join of table Orders and Customers.

For (a), we read M blocks of R at a time, so we need to have $\lceil B(R)/M \rceil = \lceil 2500/19 \rceil =$ **132 passes**.

For (b), we saw that

1. The level 0 pass takes 132 runs.
2. The level 1 pass takes $\lceil 132/(19-1) \rceil = 8$ runs.
3. At this point we can just run once more since $8 < 18$, so our level 2 pass takes 1 run.

So we have **3 passes** with **141 runs**.

For (c), we have $B(R) = 2500$ and just compute the following.

1. For level 0, we read all blocks, and write them all out: $2B(R)$.
2. For level 1, it's the same thing since we again read all blocks and write them back to disk: $2B(R)$.
3. For level 2, we just read through and do not include the final write, so $B(R)$.

This is a total of $5B(R) = 5 \cdot 2500 =$ **12,500 IOs**.

For (d),

1. For the first pass, you load all blocks of both relations in memory for the first iteration of sort merge and then write them to disk, giving us $2(B(R) + B(S))$ IOs.
2. The second pass is the same, giving us $2(B(R) + B(S))$ IOs.

3. By the third pass, we complete the sort-merge join giving us $B(R) + B(S)$.

This is a total of $5(B(R) + B(S)) = 5 \cdot (2000 + 2500) =$ **22,500 IOs**.

Question 2.3 Assume uniform distribution for the hash function(s).

(a) For the Orders and Customers tables, we want to perform a multi-pass hash join. How many passes do we need (including partitioning phase and join phase)?

(b) What is the minimum I/O cost of joining Orders and Customers using a hash join?

(c) What is the minimum number of memory blocks required if we want to perform a 2-pass hash join? Note that M should be an integer.

For (a), we must partition both relations.

1. In the first partition pass, we
   (a) take Customers and have $\lceil 2000/(19-1) \rceil = 112 > 18$. Each partition contains 112 blocks, which is too big for our memory, so we must partition again.
   (b) take Orders and have $\lceil 2500/(19-1) \rceil = 139$. Each partition contains 139 blocks, which is too big for our memory, so we must partition again.
2. In the second partition pass, we
   (a) take Customers and have $\lceil 112/18 \rceil = 7 \leq 18$, so we are done since each partition (of a partition) of 7 blocks can fit in memory.
   (b) take Orders and have $\lceil 139/18 \rceil = 8 \leq 18$, so we are done since each partition (of a partition) of 8 blocks can fit in memory.
3. Finally, we load each partition of Customers in memory (7 blocks), and for each partition, we just iterate through the corresponding partition of Customers. This is one join pass.

There are a total of **3 passes**, or 5 passes if we consider each partition step of each relation as an individual pass.

For (b), we see that since `Orders.CustomerID` is a foreign key, we won't have a case where every row in `Orders` will trivially join with every row in `Customers`. We can compute the steps as such.

1. The 1st partition pass.
   (a) For Customers, it requires you to read all blocks (2000 IOs) and then write $112 \cdot 18$ blocks back onto disk (2016 IOs),[a] or we are just approximating, this is about 2000 IOs as well.
   (b) For Orders, it requires you to read all blocks (2500 IOs) and then write $139 \cdot 18$ blocks back onto disk (2502 IOs), or just 2500 IOs as an approximation. This is flushed out as well.

   This gives us a total of $2000 + 2016 + 2500 + 2502 = 9018$ IOs or with the approximations 9000 IOs.
2. The 2nd partition pass.
   (a) For Customers, now you have 18 partitions with each 112 blocks. For each partition, you load it again (112 IOs) and then partition it again to write back $18 \cdot 7$ blocks (126 IOs) for a total of 238 IOs. You do this 18 times for each partition giving us $238 \cdot 18 = 4284$ IOs. Again, we can just approximate it by saying that loading all partitions is 2000 and writing all is 2000, giving us 4000 IOs.
   (b) For Orders, now you have 18 partitions with each 139 blocks. For each partition, you load it again (139 IOs) and then partition it again to write back $18 \cdot 8$ blocks (144 IOs) for a total of 283 IOs. You do this 18 times for each partition giving us $283 \cdot 18 = 5904$ IOs. Again, we can just approximate it by saying that loading all partitions is 2500 and writing all is 2500, giving us 5000 IOs.

   This gives us a total of $4284 + 5904 = 10188$ IOs, or with the approximations 9000 IOs.
3. In the join phase, you load the partitions of R and S with the matching hashes once each to compare, so we have (not including write)
   (a) the partitions of R, which is $18 \cdot 18 \cdot 7 = 2268$ IOs, or 2000 IOs approximately.
   (b) the partitions of S, which is $18 \cdot 18 \cdot 8 = 2592$ IOs, or 2500 IOs approximately.

   giving us a total of 4860 IOs.

Therefore, we have a total of $9018 + 10188 + 4860 =$ **24066 IOs**, or if we use our approximations, it's simply $5(B(R) + B(S)) = 5 \cdot 4500 =$ **22,500 IOs**.

For (c), M must satisfy

$$M \geq \lceil \sqrt{\min\{2500, 2000\}} \rceil + 1 = 46 \tag{34}$$

giving us $M = 46$. Checking this is indeed the case, since now the image of our hash function is $\{1, \dots, 45\}$, and assuming uniformity the number of blocks in each partition is $\lceil 2000/45 \rceil = 45$, which is just enough to fit in memory and then use the last block as an input buffer for the bigger relation when joining.

3.5 Logical Plans

When the DBMS chooses the best way to sort or merge two relations, it needs to choose it immediately. This can be done crudely by looking at the statistics of the relations that it is working with, but it doesn't guarantee that you will get the optimal plan. Therefore, it goes by the principal that you shouldn't try and waste time choosing the optimal one, but rather avoid the horrible ones. As a user of this DBMS, we should also take care in writing queries that are not too computationally or IO heavy. Two general heuristics that we should follow are:

1. You want to *push down*, i.e. use as early as possible, selections and projections.
2. You want to join smaller relations first and avoid using cross product, which can be devastating in memory.

Definition 3.5 (Logical Plan)

To have an approximate sense of how computationally heavy a query is, we can construct a high-level **logical plan**, which shows the computation DAG of the relational operators that we will perform for a query. We can optimize the logical plan by modifying our intermediate steps, such as optimizing our relational algebra logic or our implementation of SQL.

3.5.1 Query Rewrite

Using what we know, we can get a bit more theoretical and use the following identities in relational algebra. However, this has already been optimized and only does so much in practice.

Theorem 3.2 (Identities)

The following hold:

1. Selection-Join Conversion: $\sigma_p(R \times S) = R \bowtie_p S$
2. Selection Merge/Split: $\sigma_{p_1}(\sigma_{p_2} R) = \sigma_{p_1 \wedge p_2} R$
3. Projection Merge/Split: $\pi_{L_1}(\pi_{L_2} R) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
4. Selection Push Down/Pull Up: $\sigma_{p \wedge p_r \wedge p_s}(R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \wedge p'} (\sigma_{p_s} S)$, where:
   (a) p_r is a predicate involving only R columns
   (b) p_s is a predicate involving only S columns
   (c) p and p' are predicates involving both R and S columns
5. Projection Push Down: $\pi_L(\sigma_p R) = \pi_L(\sigma_p(\pi_{LL'} R))$, where L' is the set of columns referenced by p that are not in L

[a] This is not exactly 2000 because $2000/18 = 111.11$, which means that after repeatedly flushing out the filled partitions to disk 111 times, at the end we will have a partially filled buffer block in memory for each of the 18 partitions. This will need to be flushed out as well.

Definition 3.6 (SQL Query Rewrite)

We can rewrite SQL queries directly, though this is more complicated and requires knowledge of the nuances of the DBMS.

1. Subqueries and views may not be efficient, as they divide a query into nested blocks. Processing each block separately forces the DBMS to use join methods, which may not be optimal for the entire query though it may be optimal for each block.
2. Unnest queries convert subqueries/views to joins.

Therefore, it is usually easier to deal with select-project-join queries, where the rules of relational algebra can be cleanly applied.

Example 3.7 (Query Rewrite)

Given the query, we wish to rewrite it.

```
SELECT name
FROM User
WhERE uid = ANY(SELECT uid FROM Member);
```

The following is wrong since there may be one user in two groups, so it will be duplicated.[a]

```
SELECT name
FROM User, Member
WHERE User.uid = Member.uid;
```

The following is correct assuming `User.uid` is a key.

```
SELECT name
FROM (SELECT DISTINCT User.uid, name)
FROM User, Member
WHERE User.uid = Member.uid);
```

Example 3.8 (Correlated Subqueries)

Look at this query where we want to select all group ids with name like Springfield and having less than some number of members.

```
SELECT gid
FROM Group, (SELECT gid, COUNT(*) AS cnt FROM Member GROUP BY gid) t
WHERE t.gid = Group.gid AND min_size > t.cnt
AND name LIKE 'Springfield%';
```

This is inefficient since for every `gid`, we are making an entire extra query to select the counts. This is called a **non-correlated** query since this subquery is being run independently for every run. It ends up computing the size of *every* group, unlike the following one, where it filters out groups named Springfield first and then computes their size.

```
SELECT gid FROM Group
WHERE name LIKE 'Springfield%'
AND min_size > (SELECT COUNT(*) FROM Member WHERE Member.gid = Group.gid);
```

[a] A bit of review: when testing whether two queries are equal, think about if the two queries treat duplicates, null values, and empty relations in the same way.

3.5.2 Search Strategies

Given a set of operations we have to do, the number of permutations that we can apply these operations grows super-exponentially. The problem of finding the best permutation is called a **search strategy**.

Example 3.9 (Left-Deep Plans)

Say that we have relations $R_1, \dots, R_n$ that we want to join. The set of all sequences in which we can join them is bijective to the set of all binary trees with leaves R_i. This grows super-exponentially, reading $30,240$ for $n = 6$. There are too many logical plans to choose from, so we must reduce this search space. Here are some heuristics.

1. We consider only **left-deep** plans, in which case every time we join two relations, it is the outer relation in the next join.[a]

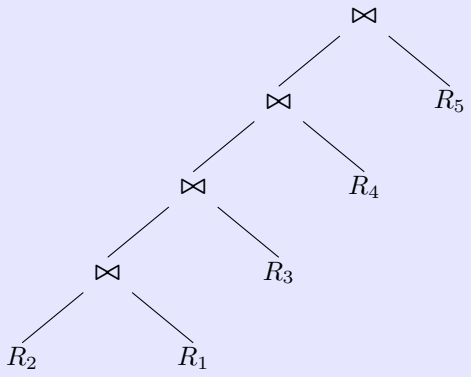

Figure 24: Left deep plans have a search space of only $n!$, which is better than before.

2. We can consider a balanced binary tree, which can be parallel processed, but this causes more runtime on the CPU in sort-merge joins, you must materialize the result in the disk, and finally the search space of binary trees may be larger than that of the left-deep tree.

Even left-deep plans are still pretty bad, and so optimizing this requires a bit of DP (dynamic programming), using *Selinger's algorithm*.

Algorithm 3.8 (Selinger's Algorithm)

Given $R_1, \dots, R_n$, we must choose a permutation from the $n!$ permutations. Say that the cost of the optimal join of a set $\mathbb{R}$ is $f(\mathbb{R})$. Note the recursive formula for some $S \subset [n]$.

$$f(\{R_i\}_{i \in S}) = \min_i f(\{R_j\}_{j \in S, j \neq i}) + f(R_i, \bowtie_{j \in S, j \neq i} R_j) \tag{35}$$

Where we sum up the cost of getting the accumulated relation and add it to the additional cost of joining once more with R_i. Therefore, given the R_i's,

1. We compute all $f(\{R_i, R_j\})$ for $i < j$ (since $j > i$ requires us the larger one to be inner).
2. Then we apply the recursive formula for all 3-combinations and so on, until we get to n-combinations.

Given a certain logical plan, the DBMS tries to choose an optimal physical plan as we will see later. However, the globally optimal plan may not be achieved with a greedy approach of first choosing the optimal logical

[a]Note that since the right/inner relation is the one that is being scanned, we want the right one to be smaller since for each block of the left relation, we are looping over all blocks of the right relation. Therefore, left-deep plans are much more efficient since we don't have to scan the huge relation from the disk multiple times. We can just send it directly to the next join.

plan and then its optimal physical plan. Due to the sheer size of the search space, we tend to go for "good" plans rather than optimal ones.

3.6 Physical Plan

The logical plan gives us an abstract view of the operations that we need to perform. It is mainly defined at the relational algebra or language level. But simply sorting or joining two relations is not done in just one way (e.g. we can scan, hash, or sort in different ways). Optimizing the logical plan may or may not help in the runtime, since operations are dependent on the size of the intermediate inputs.

Definition 3.7 (Physical Plan)

The actual sub-decisions needed to execute these operations constitute the **physical plan**, which is the actual implementation including even more operations in between each node of the logical plan. Here are a few terms to know.

1. *On the fly.* The computations are done in memory and are not written back to disk.
2. *(Index) Scan.* We scan for index value using a clustered/unclustered index on a B+ tree.
3. *Sort.* Usually means external merge sort.
4. *Filter.* Means the same as selection.

The difference between the logical and physical plan is that the logical plan represents *what* needs to be done (which we write SQL) and not *how* (which the DBMS chooses). Consider the two approaches.

Example 3.10 (Query Plans)

Consider the following SQL query:

```
SELECT Group.title
FROM User
JOIN Member ON User.uid = Member.uid
JOIN Group ON Member.gid = Group.gid
WHERE User.name = 'Bart';
```

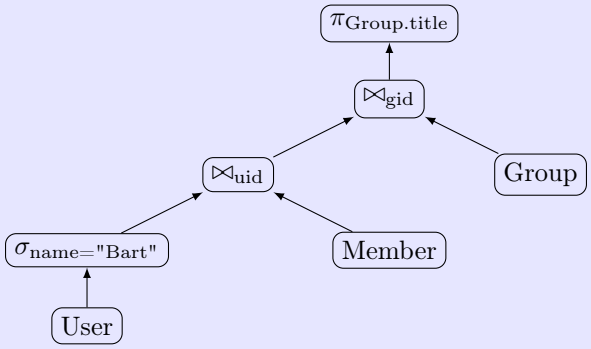

Figure 25: Logical Plan. We first take User, select Bart, and join it to Member over uid. Then we join it with Group on gid, and finally project the title attribute.

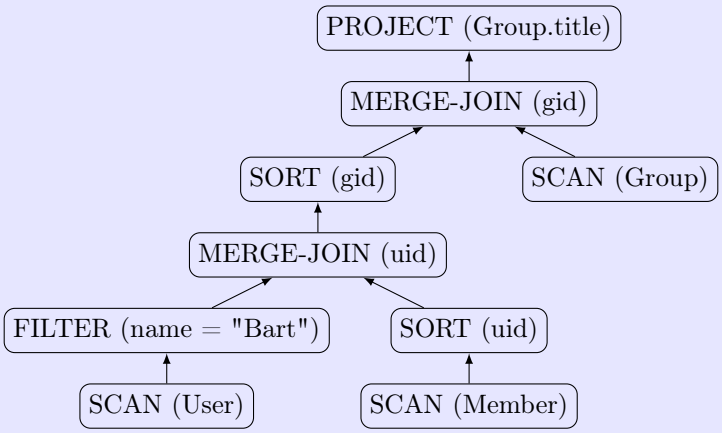

Figure 26: Physical Plan. We first take User and do a scan before filtering/selecting tuples with Bart. We also scan Member and sort it by uid in order to prepare for merge-join. Once we merge-join over uid, we sort it again to prepare a second merge-join with Group (which we scan first). Once we do this, we finally project the title attribute.

3.6.1 SQL Rewrite

At the language level, SQL provides some APIs to force the DBMS to use a certain physical plan if desired. This requires expertise and should not be done by beginners, however.

3.6.2 Cardinality Estimation

In the physical plan, we need to have a cost estimation for each operator. For example, we know that `SORT(gid)` takes $O(B(\text{input}) \cdot \log_M B(\text{input}))$, but we should find out what B, the number of blocks needed to store our input relation, is. To do this, we need the size of intermediate results through cardinality estimation.

Usually we cannot do quick and accurate cardinality estimation without strong assumptions, the first of which is uniformity of data.

Example 3.11 (Selection with Equality Predicates)

Suppose you have a relation R with $|R| = 100,000$ tuples. Assume that it has an attribute A taking integer values in $[50, 100)$ *distributed uniformly*. Then, there are 50 distinct values, and when we want to do $\sigma_{A=a}(R)$, then we would expect it to return

$$|\sigma_{A=a}(R)| = \frac{|R|}{|\pi_A(R)|} = 2000 \tag{36}$$

tuples.

The second assumption is *independence* of the distributions over each attribute.

Example 3.12 (Selection with Conjunctive Predicates)

If we have the same relation R with integer attributes $A \in [50, 100), B \in [10, 20)$ independently and uniformly distributed. Then,

$$|\sigma_{A=a,B=b}(R) = \frac{|R|}{|\pi_A(R)| \cdot |\pi_B(R)|} = \frac{100,000}{50 \cdot 10} = 200 \tag{37}$$

At this point, we are just using inclusion-exclusion principle and this becomes a counting problem.

Example 3.13 (Negated, Disjunctive Predicates)

We list these identities for brevity. The math is pretty simple.

$$|\sigma_{A\neq a}(R)| = |R| \cdot \left(1 - \frac{1}{|\pi_A(R)|}\right) \tag{38}$$

and using I/E principle, we have

$$|\sigma_{A=a\vee B=b}(R)| = |R| \cdot \left(\frac{1}{|\pi_A(R)|} + \frac{1}{|\pi_B(R)|} - \frac{1}{|\pi_A(R)| \cdot |\pi_B(R)|}\right) \tag{39}$$

Example 3.14 (Range Predicates)

Range also works similarly, but only if we know the actual bounds of the attribute values.

$$|\sigma_{A>a}(R)| = |R| \cdot \frac{\max(R.A) - a}{\max(R.A) - \min(R.A)} \tag{40}$$

Clearly, if we know that an attribute follows, say a Gaussian or a Poisson distribution, we can just calculate the difference in the CDFs and scale up by the relation size to get the approximate cardinality. I think this is what the professor refers to as *histogram estimation*.

For joins, we need yet another assumption, called *containment of value sets*. This means that if we are natural joining $R(A, B) \bowtie S(A, C)$, every tuple in the smaller (as in fewer distinct values for the join attribute A) joins with some tuple in the other relation. In other words,

$$|\pi_A(R)| \leq |\pi_A(S)| \implies \pi_A(R) \subset \pi_A(S) \tag{41}$$

which again is a very strong assumption in general but holds in the case of foreign key joins.

Example 3.15 (Two Way Equi-Join)

With the containment assumption, we have

$$|R \bowtie_A S| = \frac{|R| \cdot |S|}{\max(|\pi_A(R)|, |\pi_A(S)|)} \tag{42}$$

Think of this as looking at the cross product between the two relations, and then filtering out the actual tuples that don't belong there.

3.7 Exercises

Let's do a more comprehensive exercise.

Example 3.16 (Cost Estimation of Physical Query Plan)

Say we have three relations

1. `Student(sid, name, age, addr)`. $T(S) = 10,000$ tuples, $B(S) = 1,000$ pages.
2. `Book(bid, title, author)`. $T(B) = 50,000$ tuples, $B(S) = 5,000$ pages.
3. `Checkout(sid, bid, date)`. $T(C) = 300,000$ tuples, $B(C) = 15,000$ pages.

And say that the number of `author` attribute values in `Book` with $7 \leq$ `age` ≤ 24 is 500 tuples. There is an unclustered B+ tree index on `B.author`, a clustered B+ tree index on `C.bid`, and all index pages are in memory. Also we assume unlimited memory to simplify things.

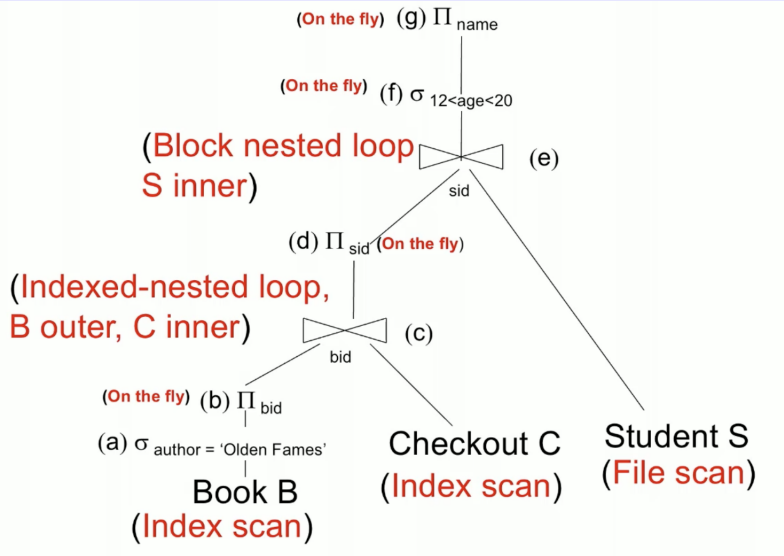

Figure 27: We have the following physical plan.

Okay, so let's do this. Note that since all index pages are in memory, we are storing the entire B+ tree in memory and don't need extra IOs to traverse it.

a) For the selection, the author is an unclustered B+ tree. There are 50,000 tuples and 500 distinct authors. Since we're querying 1 author, by uniformity we would need to access 100 book which may all be in their own disk page, so we need 100 IOs.[a] We end up with an output relation of 100 tuples in memory, so we also have a cardinality of 100.

$$\text{IO}(a) = \frac{T(B)}{500} = \frac{50,000}{500} = 100, \ \text{Card}(a) = 100 \tag{43}$$

b) For the projection on `bid`, we have already loaded in our relation in memory, so the IO cost is 0. We are still working with 100 pages, so our cardinality is still 100.

$$\text{IO} = 0, \ \text{Card}(b) = 100 \tag{44}$$

c) Now we do a join with an index-nested loop join on `bid`. Recall that we want to use the value of the outer table `R.bid` to probe the index on the inner table `C.bid`, which is clustered. We already have our outer table in memory, and we use the index `bid` to probe our inner table `C`. For each of the 50,000 book tuples in `B`, there are 300,000 checkout tuples in `C`, meaning that there are about $300,000/50,000 = 6$ checkout per book. For each of the 100 book tuples in (a), we expect to get 6 checkouts per book. There are $300,000/15,000 = 20$ checkout tuples per page, so counting for page boundaries we assume that 6 tuples will fit in at most 2 pages (or maybe 1). Therefore, we have

$$\text{IO}(c) = 100 \cdot 2 = 200, \ \text{Card}(c) = 100 \cdot 6 = 600 \tag{45}$$

d) This is done in memory so IO is 0. Note that we have a total of 600 checkout/book tuples. Since this is a projection, the cardinality also remains the same.

$$\text{IO}(d) = 0, \ \text{Card}(d) = 600 \tag{46}$$

e) Now we have a block nested loop join. Since (d) is already in memory (on the fly), all we have to do is load all of `S` into memory (unlimited), which means our IO cost is $B(S) = 1000$. We are joining with the student relation, and since there is 1 student per checkout, our output relation is still 600 tuples long.

$$\text{IO}(e) = 1000, \ \text{Card}(e) = 600 \tag{47}$$

f) Finally we select, and assuming that the ages are uniformly distributed, we expect $(20-12-1)/(24-7+1) = 7/18$ of the relations to remain after selection. IO is 0 since this is on the fly.

$$\text{IO}(f) = 0,\ \text{Card}(f) = 600 \cdot \frac{7}{18} \approx 234 \tag{48}$$

g) Finally, we project onto name. IO is 0 since on the fly. We are projecting on names and assuming we don't remove duplicates[b] our output relation is still the same length.

$$\text{IO}(g) = 0,\ \text{Card}(g) = 234 \tag{49}$$

The total cost is $1000 + 200 + 100 = 1300$ and the final cardinality is 234.

[a] If this was clustered, then each page can store 10 tuples, so we actually need 10 IOs.
[b] Is this really a realistic assumption?