Databases

Muchang Bahng

Fall 2024

Contents

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

Definition 0.1 (Data Model)

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

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

1 Relational Algebra

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

Definition 1.1 (Relational Data Model)

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

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

Definition 1.2 (Schema)

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

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

Definition 1.3 (Instance)

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

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

[a] The attribute type cannot be a nonprimitive type, such as a list or a set.
[b] Note that since this is a set, the ordering of the rows doesn't matter , even though the output is always in some order.

by convention.

Definition 1.4 (Primitive Types)

The primitive types are listed.

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

1.1 Tables, Attributes, and Keys

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

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

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

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

DROP TABLE Movies;
```

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

Theorem 1.2 (`ALTER TABLE`)

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

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

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

Theorem 1.3 (`DEFAULT`)

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

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

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

Definition 1.5 (Key)

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

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

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

Example 1.1 (Keys of User Relation)

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

1.2 Relational Algebra

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

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

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

Definition 1.6 (Relational Algebra)

A relational algebra consists of the following operands.
1. Relations R, with attributes A_i.
2. Operations.

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

[a] By minimal we do not mean that the number of attributes in K is minimal. It is minimal in the sense that no proper subset of $\mathcal{K}$ can be a key.

itself.

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

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

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

Definition 1.7 (Set Operations)

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

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

This is implemented as the `UNION`, `EXCEPT`, `INTERSECT` operations in SQL. Given that

1. Bag1 $= \{1, 2, 3\}$
2. Bag2 $= \{2, 3, 4\}$

We have

```
(SELECT * FROM Bag1)
UNION
(SELECT * FROM Bag2);     // {1, 2, 3, 4}

(SELECT * FROM Bag1)
EXCEPT
(SELECT * FROM Bag2);     // {1}

(SELECT * FROM Bag1)
INTERSECT
(SELECT * FROM Bag2);     // {2, 3}
```

Definition 1.8 (Selection)

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

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

This is analogous to the `WHERE` keyword.

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

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

This also allows us to define the in and not in keywords.

Definition 1.9 (IN and NOT IN)

The `IN` and `NOT IN` gives you filters that restrict the domain of a certain attribute.

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

Equals is denoted with `=`, and not equals is denoted with `<>`.

Definition 1.10 (Projection)

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

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

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

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

Now let's talk about operations between two relations.

Definition 1.11 (Cartesian Product)

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

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

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

```
SELECT *
FROM table1
CROSS JOIN table2;

SELECT *
FROM table1, table2;
```

Definition 1.12 (Theta-Join)

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

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

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

to be

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

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

There are other types of joins that we will use.

Example 1.2 (Simple Filter)

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

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

Table 1: Bar Information

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

Table 2: Frequents Information

We do the following.

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

Finally, we look at renaming.

Definition 1.13 (Renaming)

Given a relation R,

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

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

1.3 More on Operators

Definition 1.14 (Monotone Operators)

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

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

Example 1.3 (Monotone Operators)

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

1. *Selection is monotone.*

2. *Projection is monotone.*
3. *Cross Product is monotone.*
4. *Join is monotone.*
5. *Natural Join is monotone.*
6. *Union is monotone.*
7. *Intersection is monotone.*
8. *Difference $R - S$ is monotone* w.r.t. R but not monotone w.r.t. S.

Example 1.4 (Getting maximum of an attribute)

Given a schema $R(a, b, c)$, how do we find the maximum of a? This is hard to come up in the first time since we are not allowed to compare across lines. However, we can do the following:

1. Take the cross product of $R_1(a_1, b_1, c_1) \times R_2(a_2, b_2, c_2)$.
2. Select all tuples that are not maxes by selecting all rows where $a_1 < a_2$. The resulting rows will contain only those that are not maximums.

Therefore, we do the following

$$\max_a(R) = [\pi_a(R) \times \pi_a(R)] - \sigma_{a_1 < a_2}[\pi_{a \mapsto a_1}(R) \times \pi_{a \mapsto a_2}(R)] \tag{8}$$

The minimum can be solved analogously.

Notice that the $\max_{att}$ operator is *not* monotone, since the old answer is overwritten.

$$\{\text{oldmax}\} \not\subset \{\text{newmax}\} \tag{9}$$

Generally, whenever we want to construct a non-monotone operator, we want to use the set difference since the composition of monotones is monotone.

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

1.4 Constraints

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

Definition 1.15 (Set Constraints)

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

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

Definition 1.16 (Referential Integrity Constraints)

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

$$\pi_r(R) \subset \pi_s(S) \tag{10}$$

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

Definition 1.17 (Key Constraints)

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

$$\sigma_{R_1.\mathbf{k}=R_2.\mathbf{k} \text{ and } R_1.\mathbf{k}'\neq R_2.\mathbf{k}'}(R_1 \times R_2) = \emptyset \tag{11}$$

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

Definition 1.18 (Domain Constraints)

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

$$\sigma_{\text{not } C(r)}(R) = \emptyset \tag{12}$$

2 Design Theory for Relational Databases

2.1 Functional Dependencies

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

Definition 2.1 (Functional Dependency)

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

$$\mathbf{a} \mapsto \mathbf{b} \tag{13}$$

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

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

Lemma 2.1 (FDs as Key Constraints)

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

$$\sigma_{R_1.\mathbf{a}=R_2.\mathbf{a} \text{ and } R_1.\mathbf{b}\neq R_2.\mathbf{b}'}(R_1 \times R_2) = \emptyset \tag{14}$$

Definition 2.2 (Superkey)

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

$$\mathbf{k} \mapsto \mathbf{r} - \mathbf{k} \tag{15}$$

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

Example 2.1 (Warning!)

Given a relation $R(A, B, C, D, E, F)$ with functional dependencies

$$AEF \mapsto C, BF \mapsto C, EF \mapsto D, ACDE \mapsto F \tag{16}$$

We can see that every attribute not on the right hand side (C, D, F) must be a key. If we do a bit of

testing out, we can see that
1. $ABEF$ is a key.
2. $ABCDE$ is also a key since even though it has 5 attributes, it is minimal in the sense that no proper subset functionally determines every other attribute.

2.1.1 Structure on Spaces of Functional Dependencies

To introduce additional structure, we will introduce two spaces.

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

Theorem 2.1 (Armstrong Axioms)

Let's prove a few properties of FDs, which have nice structure.
1. *Splitting and Combining.* The two sets of FDs are equal.
$$\{\mathbf{a} \mapsto \mathbf{b}\} \iff \{\mathbf{a} \mapsto b_i \mid i = 1, \dots, m\} \tag{17}$$
2. *Trivial FDs.* Clearly elements of $\mathbf{a}$ uniquely determines its own attributes.
$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a} \mapsto \mathbf{b} - \mathbf{a} \tag{18}$$
or can also be written as
$$\mathbf{b} \subset \mathbf{a} \implies \mathbf{a} \mapsto \mathbf{b} \tag{19}$$
3. *Augmentation.*
$$\mathbf{a} \mapsto \mathbf{b} \implies \mathbf{a}, \mathbf{c} \mapsto \mathbf{b}, \mathbf{c} \tag{20}$$
4. *Transitivity.* If $\mathbf{a} \mapsto \mathbf{b}, \mathbf{b} \mapsto \mathbf{c}$, then
$$\mathbf{a} \mapsto \mathbf{c} \tag{21}$$

Proof.

Trivial.

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

Definition 2.3 (Partial Order)

Given two FDs f and g, consider the set of all relations R satisfying f and g, denoted as R_f and R_g.
1. Then $f \implies g$ iff $R_f \subset R_g$.
2. $f \iff g$ iff $R_f = R_g$.

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

Definition 2.4 (Closure of Attributes)

The **closure** of $\mathbf{r}$ under a set of FDs F is the set of attributes $\mathbf{b}$ s.t.
$$R_F = R_{\mathbf{b}} \tag{22}$$
We denote this as $\mathbf{b} = \mathbf{r}^+$. To actually compute the closure, we take a greedy approach by starting with $\mathbf{r}$ and incrementally adding attributes satisfying F until we cannot add any more.

Theorem 2.2 ()

If we want to know where one FD $f : \mathbf{a} \mapsto \mathbf{b}$ follows from a set F of functional dependencies,
1. We compute the closure $\mathbf{a}^+$ w.r.t. F.
2. If $\mathbf{b} \subset \mathbf{a}^+$, then f follows from F.

Alternatively, we can also use the *Armstrong axioms* above to derive all implications.

2.2 Projections of Functional Dependencies

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

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

2.3 Anomalies and Decomposition

Definition 2.5 (Anomaly)

Beginners often try to cram too much into a relation, resulting in **anomalies** of three forms.
1. *Redundancies.* Information repeated unnecessarily in several tuples.
2. *Updates.* Updating information in one tuple can leave the same information unchanged in another.
3. *Deletion.* If a set of values becomes empty, we may lose other information as a side effect.

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 $\mathbf{x}$. 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.

a	**b**	**c**
x	y	z_1
x	y	z_2

Table 3: Redundant information in **a** and **b** attributes.

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

Definition 2.6 (Decomposition)

Given relation $R(\mathbf{r})$, we can decompose R into two relations $R_1(\mathbf{r}_1)$ and $R_2(\mathbf{r}_2)$ such that
1. $\mathbf{r} = \mathbf{r}_1 \cup \mathbf{r}_2$
2. $R_1 = \pi_{r_1}(R)$
3. $R_2 = \pi_{r_2}(R)$

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 2.3 (Decomposition)

Any decomposition R_1, R_2 of R satisfies

$$R \subset R_1 \bowtie R_2 \tag{23}$$

Example 2.2 (Lossy)

Consider the table

uid	gid	name
857	abc	Bob
857	gov	Brent

Suppose we decompose this into the schemas.

uid	gid
857	abc
857	gov

Table 4: First Table

uid	name
857	Alice
857	Anna

Table 5: Second Table

Then, if we natural joined based on the uid, then we would have the following table, which is not the original one.

uid	gid	name
857	abc	Bob
857	abc	Brent
857	gov	Bob
857	gov	Brent

Example 2.3 (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	Y
a	c_1
a	c_2
a_1	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	Y
b	c_1
b	c_2

Generally, the intuition behind trying to find a lossless decomposition is 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, 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 k b's.

Example 2.4 ()

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

2.4 Boyce-Codd Normal Form

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

Definition 2.7 (BNCF)

A relation R is in **BNCF** iff whenever there is a nontrivial FD $\mathbf{a} \mapsto \mathbf{b}$, it is the case that $\mathbf{a}$ is a superkey for R. Note that since there are multiple keys, $\mathbf{a}$ does not always have to include the same key. Given a relation R with a set of 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.

Example 2.5 (Non-BNCF Form)

The table below is not in BCNF form since

$$(\texttt{title}, \texttt{year}) \mapsto (\texttt{length}, \texttt{genre}, \texttt{studioName}) \tag{24}$$

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 6: Movie Data

Example 2.6 (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 7: 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 8: Movie Titles, Years, and Stars

Algorithm 2.1 (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
$$R_1 = \pi_{\mathbf{a},\mathbf{a}^+}(R) \tag{25}$$
2. We don't want any loss in data, so we take the rest of the attributes not in the closure and define
$$R_2 = \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}^+}(R) \tag{26}$$

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 2.4 (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

$$R \subset \pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}}(R) \tag{27}$$

and anything that comes back in the join must be in the original relation

$$\pi_{\mathbf{a}^+}(R) \bowtie \pi_{\mathbf{a},\mathbf{r}-\mathbf{a}}(R) \subset R \tag{28}$$

using the fact that $\mathbf{a} \mapsto \mathbf{b}$.

Theorem 2.5 (Any 2-Attribute Relation Satisfies BNCF)

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

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

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

3 Design Models

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

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

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

3.1 The Entity-Relationship Model and Cons

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

Definition 3.1 (Entity-Relationship Model)

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

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

Example 3.1 (E/R Diagram)

Let us model a social media database with the relations

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

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

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

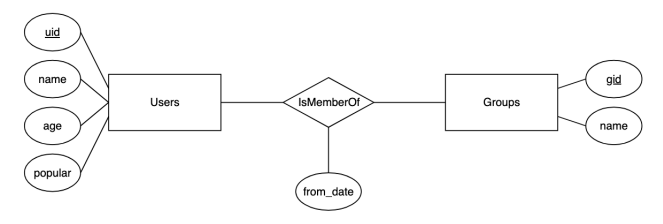

Figure 2: Social media database ER diagram.

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

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

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

Theorem 3.1 ()

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

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

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

3.1.1 Multiplicity of Binary Relationships

Definition 3.2 (Multiplicity of Relationships)

Given that E and F are entity sets,

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

Figure 3

2. *Many-One*: Each entity in E is related to 0 or 1 entity in F, but each entity in F is related to

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

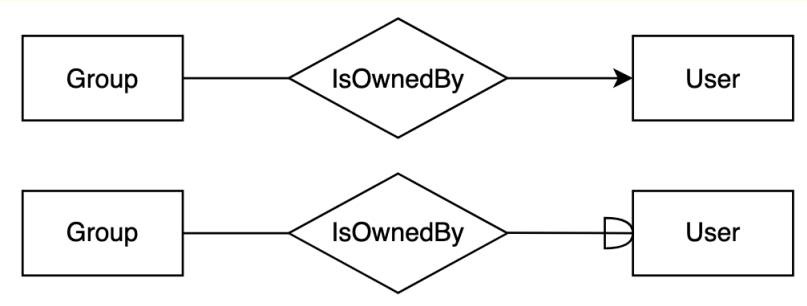

Figure 4

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

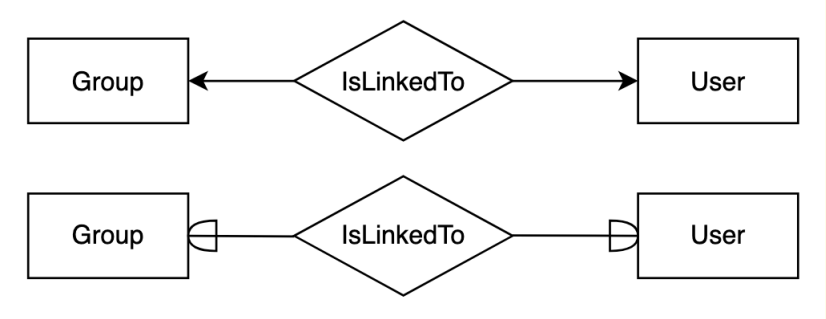

Figure 5

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

$$\mathbf{k}_R \mapsto \mathbf{r} \mapsto \mathbf{s} \mapsto \mathbf{k}_S \tag{29}$$

Example 3.2 (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

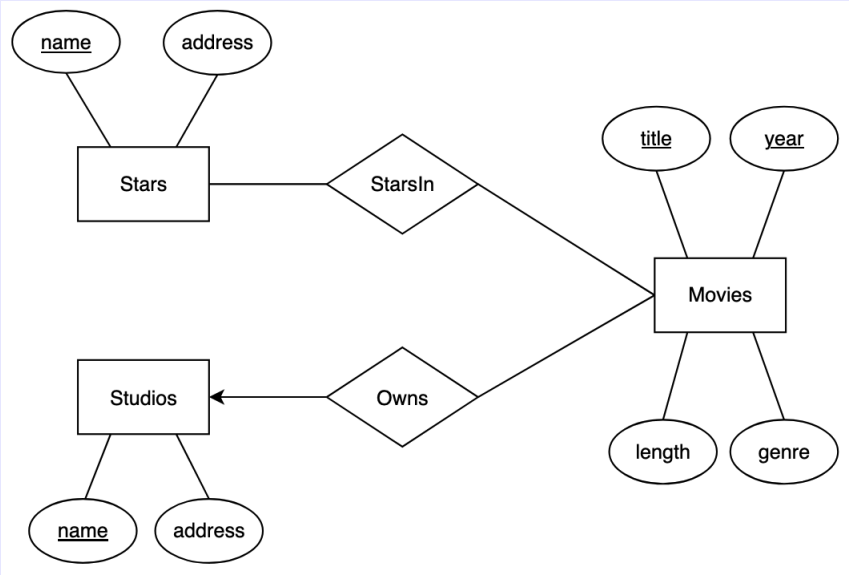

Figure 6: Movie stars.

Example 3.3 (Relationship within Itself)

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

1. *Users(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.

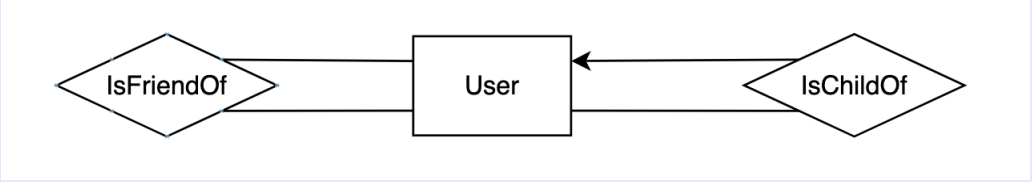

Figure 7

3.1.2 Multiplicity of Multiway Relationships

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

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

Example 3.4 (Movie Stars)

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

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

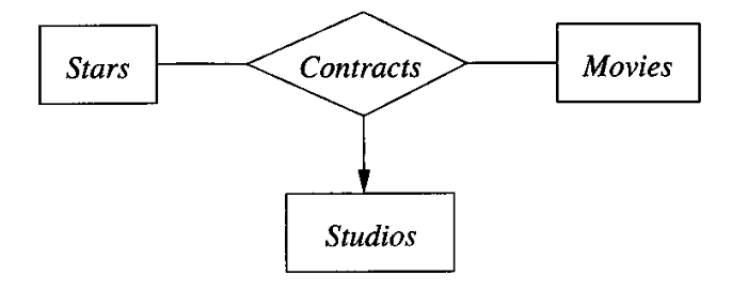

Figure 8

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

1. *Contracts(star_name, movie_name, produce_studio_name, star_studio_name)*

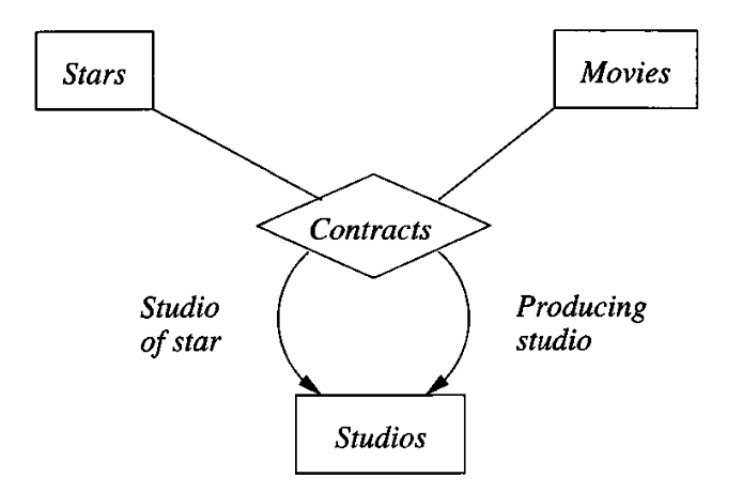

Figure 9

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

Example 3.5 (Social Media)

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

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

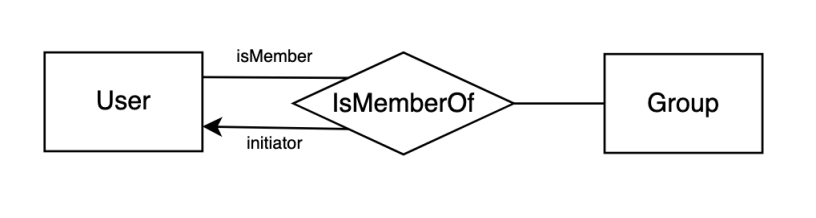

Figure 10

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

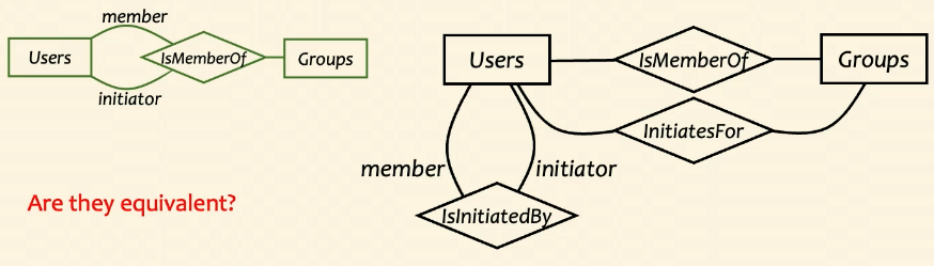

Figure 11: Attempt at reducing nary to binary ER relationships.

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

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

3.1.3 Subclasses of Entity Sets

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

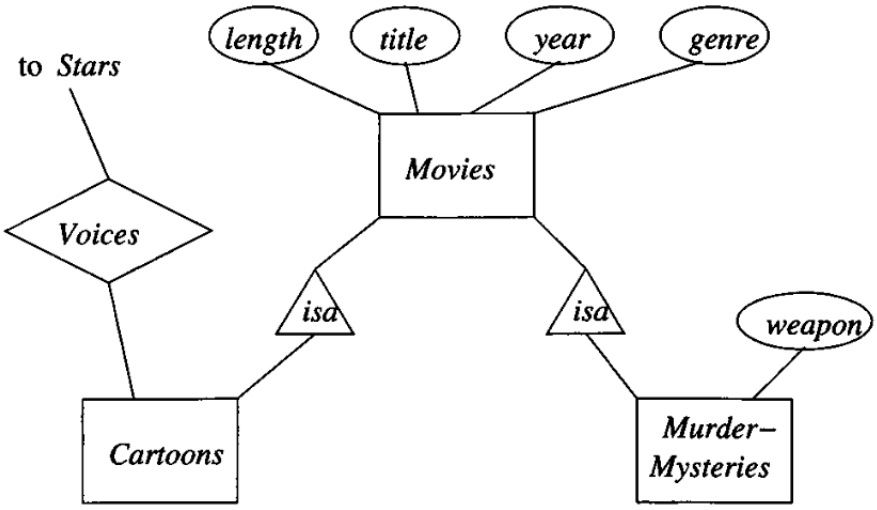

Figure 12: There are two types of movies: cartoons and murder-mysteries, which can have their own sub-attributes and their own relationships.

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

3.2 Design Principles

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

3.3 Weak Entity Sets

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

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

Definition 3.3 (Weak Entity Set)

A **weak entity set** R (double rectangles) depends on other sets. It is an entity set that
1. has a key consisting of 0 or more of its own attributes, and
2. has key attributes from **supporting entity sets** that are reached by many-one **supporting relationships** (double diamonds) from it to other sets S.

It must satisfy the following.
1. The relationship T must be binary and many-one from R to S.
2. T must have referential integrity from R to S (since these are keys and therefore must exist in supporting sets), which is why we have a rounded arrow.
3. The attributes that S supplies for the key of R must be key attributes of S, unless S is also

[1] Think of university rooms in different buildings.

weak, and it will get keys from its supporting entity set.
4. If there are several different supporting relationships from R to the same S, then each relationship is used to supply a copy of the key attributes of S to help form the key of R.

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

Example 3.6 ()

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

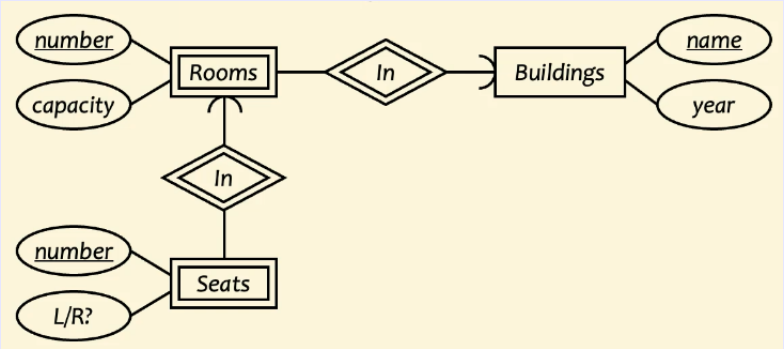

Figure 13: Specifying a seat is not enough to determine the exact location in a university. We must know the room number and the building to fully identify it. Note that we must keep linking until we get to a regular, non-weak entity.

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

Example 3.7 ()

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

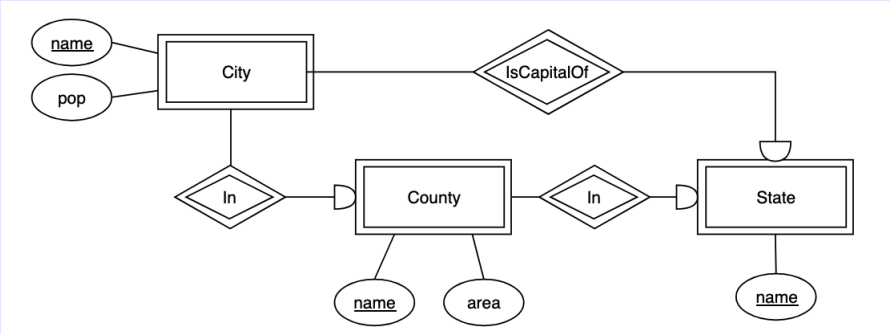

Figure 14: A weakness is that this doesn't prevent a city in state X from being the capital of another state Y.

Example 3.8 ()

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

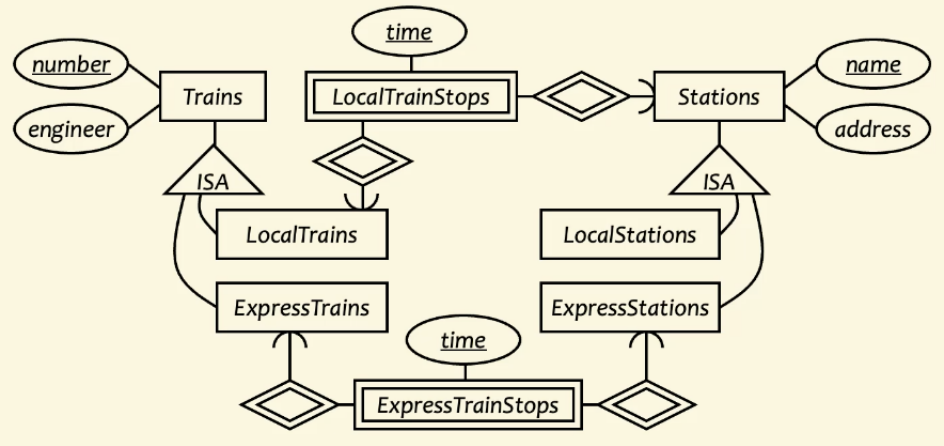

Figure 15

3.4 Translating ER Diagrams to Relational Designs

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

Theorem 3.2 (Converting Entity Sets)

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

Theorem 3.3 (Converting Relationships)

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

Theorem 3.4 (Reduce Repetition for Many-One Relationships)

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

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

Theorem 3.5 (Handling Weak Entity Sets)

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

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

Example 3.9 ()

To translate the seat, rooms, and buildings diagram,

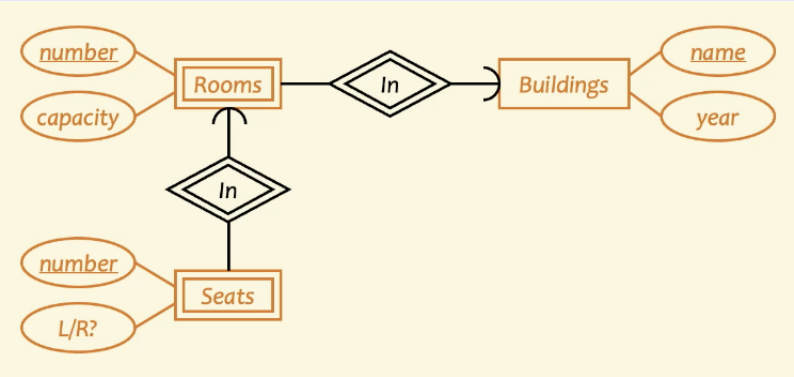

Figure 16

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.

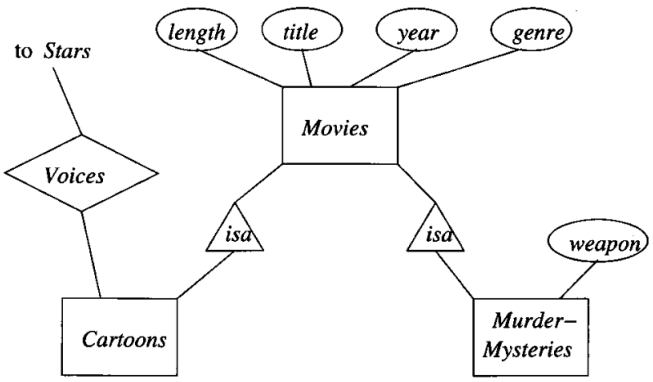

Figure 17: A figure of the movie hierarchy for convenience.

Theorem 3.6 (Converting Subclass Structures)

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

1. *E/R Standard.* 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 3.10 ()

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

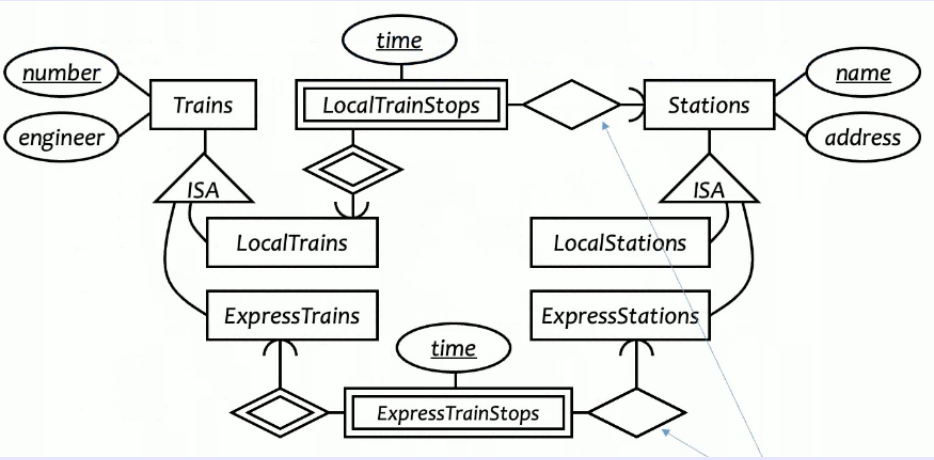

Figure 18: For convenience

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

$$LocalTrain = \pi_{number}(Train) - ExpressTrain \tag{30}$$

$$LocalStation = \pi_{number}(Station) - ExpressStation \tag{31}$$

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).

4 Intermediate SQL

We have went over the basic SQL queries that were directly translations of relational algebra.

4.1 Bags

We were used to working with sets, which don't allow duplicate elements, but let's talk about *multisets*, or **bags**, which do. Why is this advantageous?

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.

Recall that the `UNION`, `EXCEPT`, and `INTERSECT` are set semantics that removes duplicates in the input tables first and then removes duplicates in the result.

Definition 4.1 (Bag Operations)

However, we can also use bag semantics. We can think of each row a having an implicit count of times c_a it appears in the table.

1. **Bag union** sums up the counts from two tables.
2. **Bag difference** does a proper-subtract[a]
3. **Bag intersection** takes the minimum of the two counts.

This is analogous to the `UNION ALL`, `EXCEPT ALL`, and `INTERSECT ALL` operations in SQL. Given that

1. Bag1 $= \{1, 1, 2\}$
2. Bag2 $= \{1, 2, 2\}$

We have

```
(SELECT * FROM Bag1)
UNION ALL
(SELECT * FROM Bag2);     // {1, 1, 1, 2, 2, 2}

(SELECT * FROM Bag1)
EXCEPT ALL
(SELECT * FROM Bag2);     // {1}

(SELECT * FROM Bag1)
INTERSECT ALL
(SELECT * FROM Bag2);     // {1, 2}
```

Example 4.1 ()

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.

4.2 Nested Queries and Subqueries

We have so far worked with a single query consisiting of a single select statement. However, we can extend this.

Definition 4.2 (`EXISTS`)

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

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

Example 4.2 (Ages)

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');
```

The left is a **correlated subquery**, which is a query that needs a parameter from the main query and are generally slower. To understand this, you should always look in the following order.

1. `FROM`. Look at where we are querying from.
2. `WHERE`. Find out if this condition is satisfied.
3. `SELECT`. Return all tuples that satisfies this condition.

However, this is not actually how the database system will do this. It will do it in an equivalent but more efficient way.

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

Definition 4.3 (`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 4.3 ()

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;
```

4.3 Aggregate Functions

Definition 4.4 (Standard SQL Aggregate Functions)

The aggregate functions offered are

1. `COUNT` counts the number of rows in a query. `COUNT(DISTINCT att)` counts the distinct count of an attribute in a query.
2. `SUM` counts the sum of the values of an attribute in a query.
3. `AVG` is the average.
4. `MIN` is the minimum of an attribute.
5. `MAX` is the maximum of an attribute.

Example 4.4 ()

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;
```

4.4 Group By

Definition 4.5 (`GROUP BY`)

`GROUP BY att` 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 4.5 ()

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 Usere
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)`.

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

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

However, 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.

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

```sql
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 4.6 ()

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 9: Original relation.

Running the query

```sql
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 10: Our query.

4.5 Having

Definition 4.6 (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`.

Example 4.7 ()

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 11: 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 12: Our query.

Example 4.8 ()

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;
```

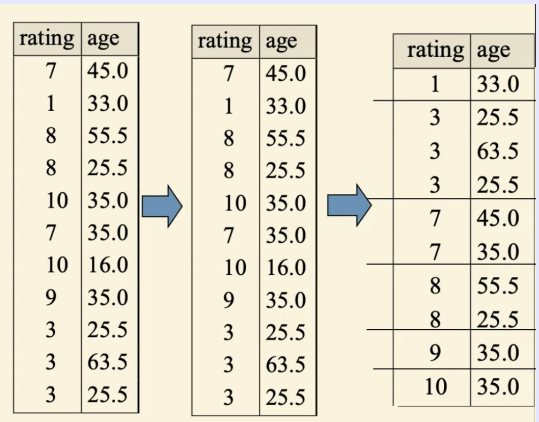

Figure 19: The thought process.

Definition 4.7 (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 4.9 ()

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.

4.6 Quantified Subqueries

Definition 4.8 (ALL, ANY)

We have the following **quantified subqueries**, which performs a broadcasting condition and checks if all (`ALL`) or any (`ANY`) are true.

Example 4.10 (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.

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

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

4.7 Incomplete Information

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 4.9 (`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 4.1 (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. Therefore, `WHERE` and `HAVING` clauses only select rows for which the condition is True, not False or Unknown.

Example 4.11 (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
```

4.8 Joins

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.

Definition 4.10 (Outer Joins)

An outer join guarantees that we have all elements in one or more tables.

1. `(INNER) JOIN`: Returns records that have matching values in both tables, with the notation $R \bowtie S$.
2. `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table with potential NULLs, with notation $R \bowtie\!\!\!\!\!\!\;\; S$.
3. `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table with potential NULLs, with notation $R \;\; \bowtie\!\!\!\!\!\!\; S$.
4. `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table with potential NULLs, with notation $R \bowtie\!\!\!\!\!\!\; S$.

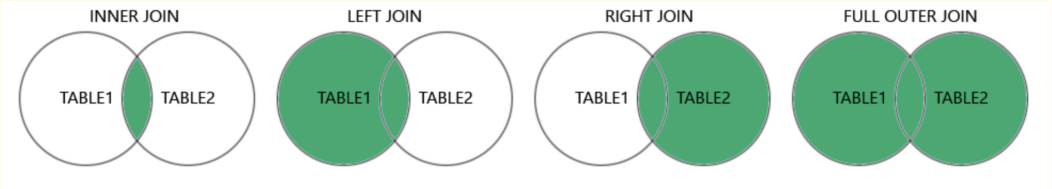

Figure 20: Nice diagram from W3Schools.

The SQL syntax is below.

```
// inner joins, you can use ON rather than WHERE
SELECT * FROM Group JOIN Member
  ON Group.gid = Member.gid;

SELECT * FROM Group LEFT OUTER JOIN Member
  ON Group.gid = Member.gid;

SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;

SELECT * FROM Group RIGHT OUTER JOIN Member
  ON Group.gid = Member.gid;
```

Example 4.12 (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
```

4.9 Inserting, Deleting, and Updating Tuples

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

Definition 4.11 (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 4.12 (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 4.13 (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;
```

4.10 Views

Definition 4.14 (View)

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

[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.

Example 4.13 (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;
```

4.11 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 4.15 (`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,
age INTEGER,
pop FLOAT);
```

Definition 4.16 (`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 4.17 (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 4.14 (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 4.18 (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.

5 Index and B+ Trees

Recall the memory hierarchy, which starts with CPU registers, followed by caches, memory, and a disk. The speed of read/write, called I/O, determines how fast you can retrieve this data.

Definition 5.1 (Hard Disk)

In SQL queries, the (disk) I/O dominates the execution time, so this must be analyzed first. A typical hard drive consists of a bunch of **disks/platters** that are spun by a **spindle** and read by a **disk head** on a **disk arm**. Each disk is pizza sliced into **sectors** and donut sliced into **tracks** (and a collection of tracks over all disks is a **cylinder**), and the arm will read one **block** of information, which is a logical unit of transfer consisting of one or more blocks (i.e. like a word, which is usually 4 bytes).[a]

Definition 5.2 (Access Time)

The access time is the sum of
1. the *seek time*: time for disk heads to move to the correct cylinder
2. the *rotational delay*: time for the desired block to rotate under the disk head
3. the *transfer time*: time to read/write data on the block

This spindle rotation and moving arm is slow, so the times are dominated by the first two.

Note that this is heavily dependent on the data being accessed. Sequential data is extremely fast while random access is slow. Therefore, we should try to store data that should be accessed together next to each other in the disk.

Definition 5.3 (Memory, Buffer Pool)

As we expect, the DBMS stores a cache of blocks, called the **memory/buffer pool** containing some fixed size of M maximum blocks. We read/write to this pool of blocks (which costs some time per blocks) and then these dirty (which are written/updated) are flushed back to the disk. It essentially acts as a middleman between the disk and the programmers, and every piece of data that goes between the two must go through the memory.

[a] You cannot just write 1 byte. You must rewrite the entire block with the 1 byte updated. If we want to write 2 bytes which are on separate blocks, we must do 2 block writes.

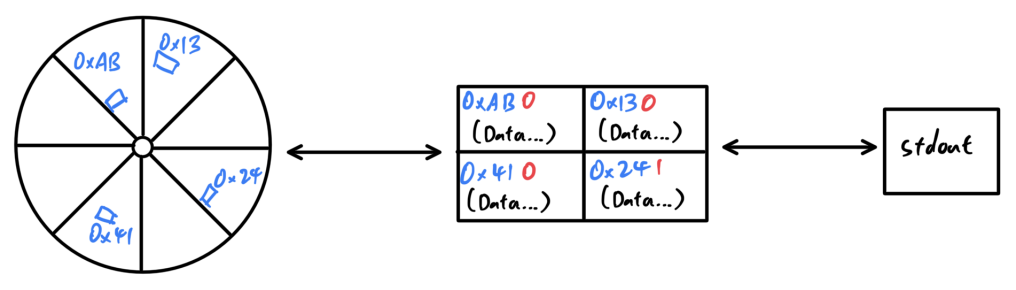

Figure 21: We store the memory address of the block (blue) and a bit indicating whether it is dirty or not (red).

Therefore, if we want to read $N > M$ blocks, then the first M blocks must be loaded into memory, outputted into stdout, and then the memory must be refilled with the rest of the blocks. If we have updated a block in memory, then we should flush it before overwriting this block in memory.[2] Replacement strategies won't be covered here. Note that unlike algorithms, which focus on the cost of the algorithm after it is read into memory, we focus on the cost of loading data from the disk into memory.

So how should we increase performance?

1. *Disk Layout*: Keep related things close together, e.g. in the same sector/block, or same track, or same cylinder, or adjacent cylinders.
2. *Prefetching*: When fetching a block from the disk, fetch the next block as well since it's pretty likely to access data from the next block. This is basically locality.
3. *Parallel I/O*: We can have more heads working at the same time.
4. *Disk Scheduling Algorithm*: e.g. the elevator algorithm sorts the cylinders so that you don't go back and forth between cylinders when fetching.
5. *Track Buffer*: Read/write one entire track at a time.

Now let's talk about how the actual bytes are stored in memory.

Definition 5.4 (Row Major Order, NSM)

We group rows together contiguously in the disk block.[a] If we have a relation with schema `R(INT(4), CHAR(24), INT(4), DOUBLE(8))`, we first store the rows together, with extra space in between since we might append new attributes, and have a tuple of pointers to the start of each row (orange lines).

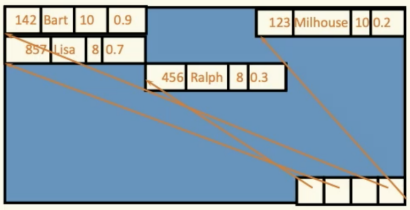

Figure 22

Note that `VARCHAR` still allocates the same number of bytes, but adds padding.

[2] idk perhaps we can have a overhead bit indicating whether a block is updated, along with the memory address of this block so it can find it back on the disk.

[a] This is the most standard storage policy.

Definition 5.5 (Column Major, PAX)

We group columns together contiguously in the disk block, which allows for optimization since all types are the same and we can just use pointer arithmetic to get every attribute in $O(1)$ time. We split the block into chunks and have metadata of pointers that point to the start of the array for each attribute.

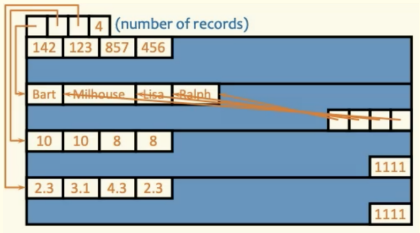

Figure 23

5.1 Dense/Sparse and Primary/Secondary Index

Definition 5.6 (Index vs Search Key)

Indexes and search keys should be distinguished. They both refer to an attribute of a relation, but are slightly different.

1. An **index** is a set of attribute values (e.g. `uid` of values `123, 456, 789`) that is a part of our query.
2. A **search key** refers to the attribute on which the index is defined.

```
SELECT * FROM Users WHERE Age = 50;
```

`Age` is the search key, and its search key values may be `20, 30, 12, 34, 50`.

Definition 5.7 (Dense vs Sparse Index)

Sparse and dense just refers to how much an index "covers" a disk block.

1. A **dense index** means that there is one index entry for each search key value. One entry may point to multiple records
2. A **sparse index** means that there is one index entry for each block. Records must be clustered according to the search key as shown below, and this can optimize searching.

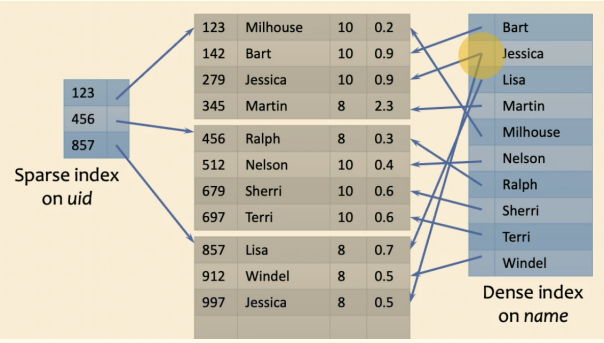

Figure 24: The dense index has 10 values, but there are two Jessicas, so it is indeed dense. The sparse index is on the uid, and since the relation is sorted (a more specific form of *clustering*) in the disk, we can use sparse keys.

Note that

1. The sparse index must contain at least the number of blocks, while the dense index must contain at least the number of unique search key values. Since the sparse index is much smaller, we may be able to fit it into main memory and not have to pay the I/O cost.
2. A dense index does not require anything on the records, while the sparse requires the data to be clustered.
3. Lookup is easy on dense since we can directly see if it exists. For sparse, we must first follow the pointer and scan the block. (e.g. if we wanted to look for 279, we want to look at the address pointed to by 123, and scan down until we hit it or reach a number greater than 279).
4. Update is usually easier on sparse indices since we don't need to update the index unless we add a new block. For dense, if we added a new person Muchang, then we would have to add `Muchang` to the dense index.

Definition 5.8 (Primary vs Secondary Index)

Primary and secondary refers to what the index is over.

1. A **primary index** is created for the primary key of the relation. Records are usually clustered by the primary key, so this can be sparse.
2. A **secondary index** is any index that is not over a primary key and is usually dense.

In SQL, the `PRIMARY KEY` declaration automatically creates a primary index, and the `UNIQUE` declaration creates a secondary index.

Example 5.1 ()

You can also create an additional secondary index on non-key attributes. For example, if you think that you will query based on popularity often, you can do

```
CREATE INDEX UserPopIndex ON User(pop);
```

which will create a dense index to speed up lookup at the cost of memory.

5.2 B+ Trees

This leads to the problem of the index being too big to fit into a block. In this case, we can just create a sparse index on top of the dense index. This allows us to store a large index across blocks.

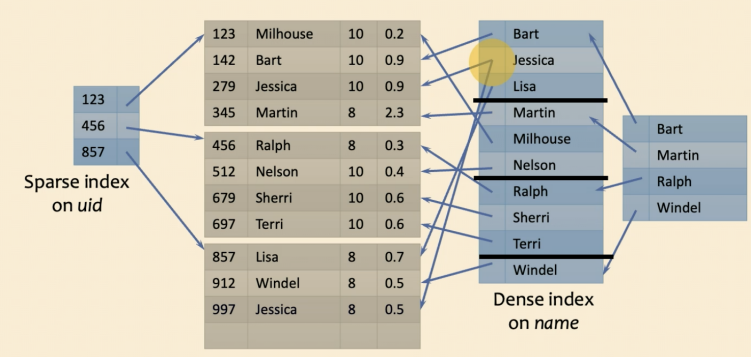

Figure 25

If the index is still too big, we store another index on top of that, and we have pretty much a tree. This is called the Index Sequential Access Method (ISAM).

Example 5.2 ()

If we want to look up 197 in this tree, we traverse down.

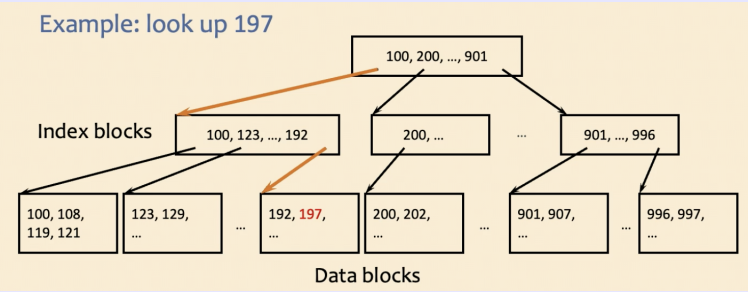

Figure 26

A problem with this method is also a problem with BSTs. If we want to delete 123 and add 107 ten times, then we have an unbalanced binary search tree and this reduces to a linear search.

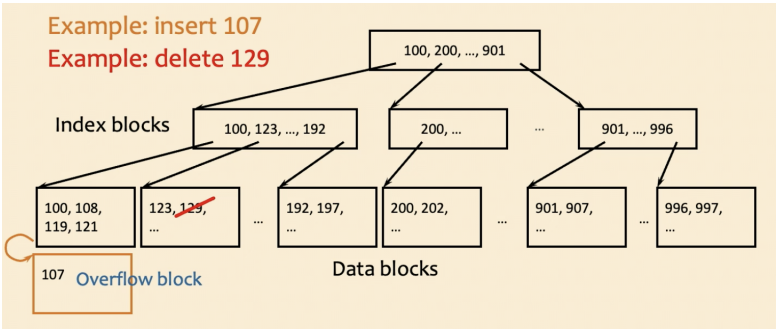

Figure 27: If this block overflows, then we want to expand this, leading to an unbalanced BST and reducing our search to linear time.

Therefore, this static data structure is not good, and we must use a more flexible one.

Definition 5.9 (B Tree)

We look at a generalization of the BST to a B-tree.

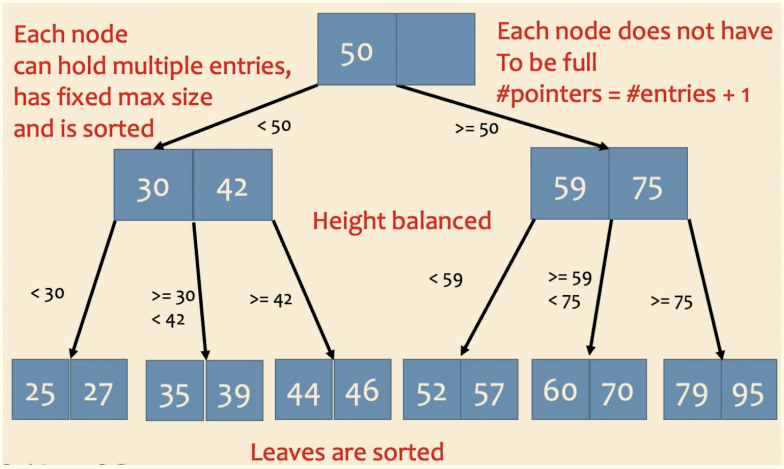

Figure 28

The actual structure that DBMS uses is the B+ tree.

Definition 5.10 (B+ Tree)

While in B trees, the non-leaf nodes are also data, the B+ trees divide the nodes into leaf nodes, which represent the data in this data structure, and the non-leaf nodes, which do not represent data but index nodes containing index entries.

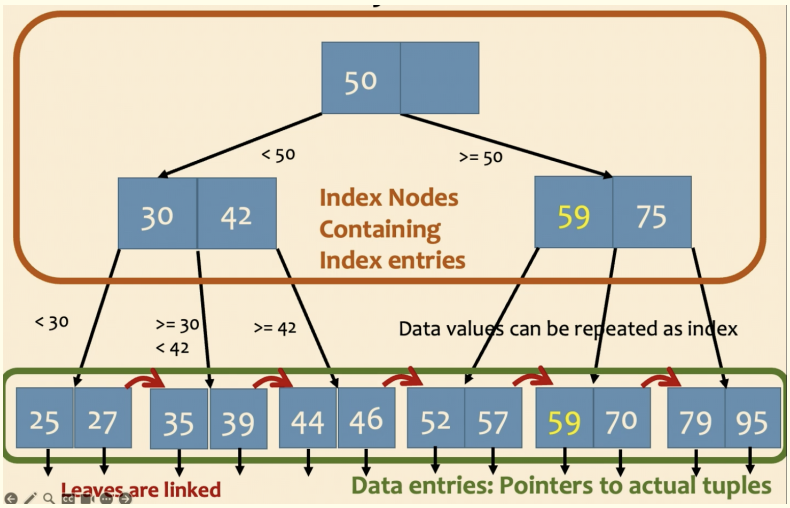

Figure 29: Note that the 59, which represents both the index and the data value, are repeated. In here, we assume a block size of 2. The leaf nodes are the indices that point to the tuples in the disk/memory. Sometimes, we may store the tuple directly in the nodes, which saves us another level of indirection, but this may cause memory problems if the tuples have too many attributes.

It has the following constraints: the **fanout** refers the to maximum number of pointers that can come out of each node.

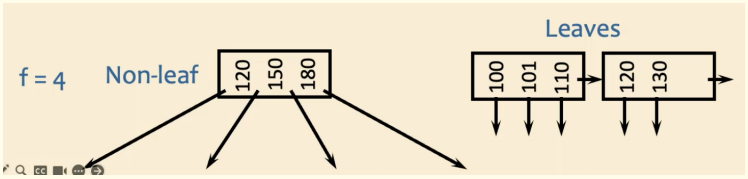

Figure 30: With fanout constraint 4, for index nodes, we have 3 values with 4 pointers representing each range. For the leaves, we have 3 pointers to the disk addresses plus 1 pointer to the next node.

Therefore, if we have a fanout of f, the table shows the constraints. Note that there is a minimum constraint to ensure that the tree is balanced.

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

Figure 31

Note that the root is the only node that can store one value. It is special in this way.

Definition 5.11 (Lookup)

If we query `SELECT * FROM R WHERE k = 179;`, we go through the B+ tree's indices and reach the leaf node. From here, we can use the pointer to go to the memory address holding this tuple in memory/disk. If we query `k = 32`, then we will not find it after reaching `35`. If we want to query a range `32 < k < 179`, we start at 32 and use the leaf pointers to go to the next leaf until we hit 179.

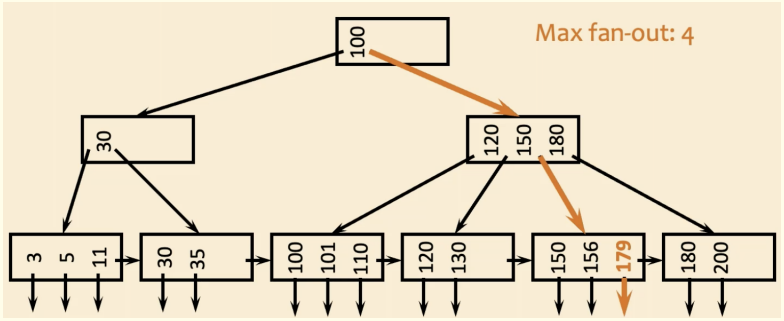

Figure 32

In practice, there are much more pointers, so we could start at 179 and go back to 32 if we had backwards pointers.

Let's see how it is dynamic, i.e. how it changes as we add/delete/modify data.

Definition 5.12 (Insertion)

Insertion onto a leaf node having space is easy.

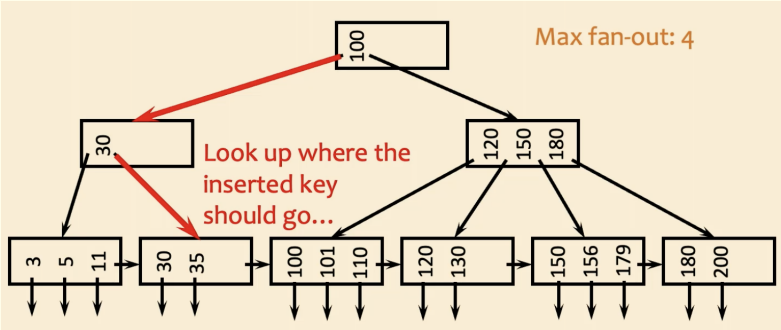

Figure 33: Note that to traverse this tree, we had to access 3 blocks of memory since for each block, we had to go to the disk, look up its contents, and retrieve the index of the next block (retrieve the blocks containing (100), (30), and (30, 35)) Then we have to update this block of (30, 35) and flush it.

When we have a full node, it is more complicated.

1. Say that we have a full node. We can try to push the rest of the leaf nodes to next node having space, but we are not guaranteed that the neighbors will have space.

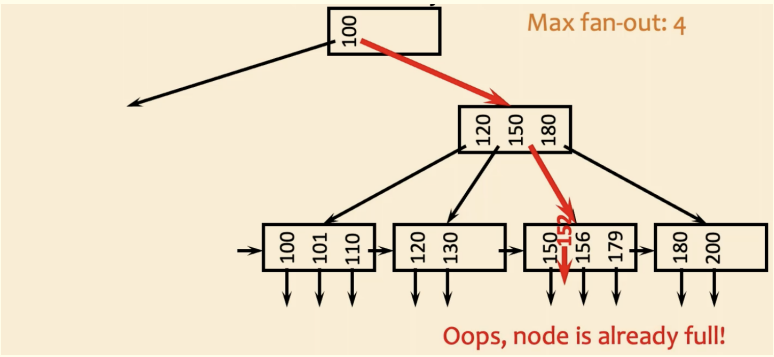

Figure 34

2. We can split the node, called a **copy-up**. However, this new node now has no pointer. If the parent node is not full, we can just add the pointer and update its value.

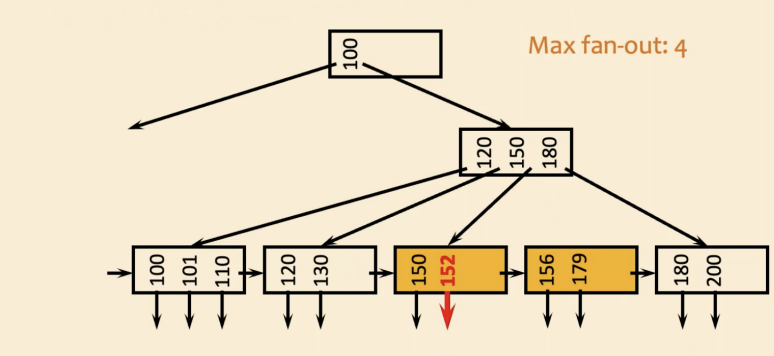

Figure 35

3. If it is full, then we should split the parent node as well. This is called a **push-up**.

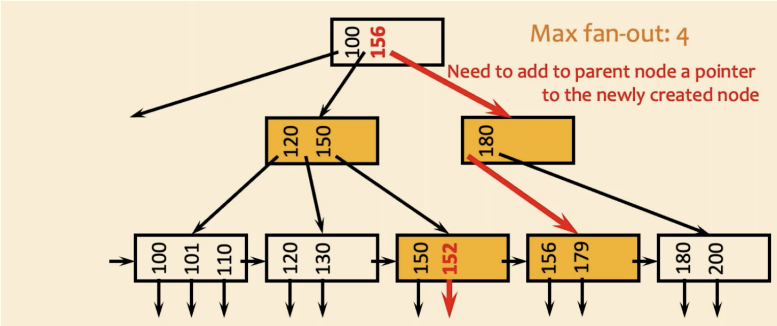

Figure 36

4. This means that we have to update the parent of the parent. If the parent is not full, then we simply add it, and if it is full, then we split the parent of the parent, and so on. If we reach the root node, then we just split the root and increase the height of the tree.[a]

Definition 5.13 (Deletion)

Deleting a value is simple if after deletion, the leaf node has at least $f/2$ pointers ($f/2 - 1$ values).

1. If the node has less than $f/2$ pointers, then it is too empty. We can adjust by taking adjacent nodes and moving them from the full nodes to the empty nodes, but this may steal too much from siblings.

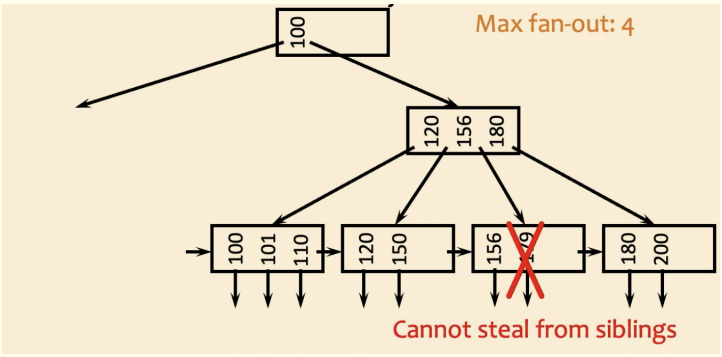

Figure 37

2. The adjacent nodes may be empty as well, and in this case we want to **coalesce** or merge the nodes together. This results in a dangling pointer, so we delete the pointer and remove a value from the parent node.[a]

[a] This is rare in practice since the fanout is much greater than 3.

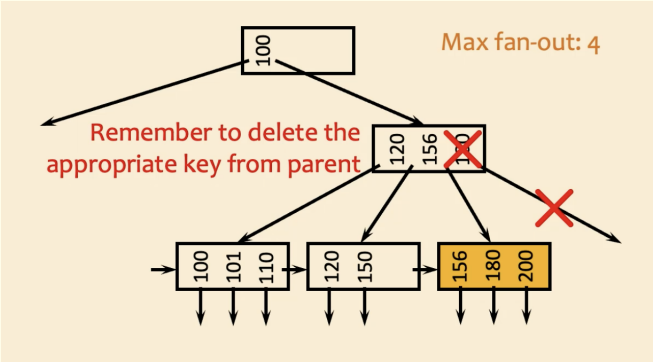

Figure 38

3. We keep doing this until the B+ tree requirements are satisfied or we reach the root, at which point we delete the root node and our B+ tree height decreases by 1.

In general, all these operations have similar runtimes:

1. They require us to go to the bottom of the tree, so it is h operations, where h is the height.
2. We also maybe have $+1$ or $+2$ to manipulate the actual records, plus $O(h)$ for reorganization like we saw before (which is rare if f is large).
3. Minus 1 if we cache the root in memory, which can be decreased even further if we cache more blocks.

The actual size of h is roughly $\log_{f/2} N$, where N is the number of records. f is the fanout, but the B+ tree properties guarantee that there are at least $f/2$ pointers, so it is $\log_{f/2} N$ at worst and $\log_f N$ at best. f is very large usually so this is quite good.

The reason we use B+ trees rather than B trees is that if we store data in non-leaf nodes, this decreases the fanout f and increases h. Therefore, records in leaves require more I/Os to access.

5.3 Clustered vs Unclustered Index

Definition 5.14 (Clustered vs Unclustered Index)

If the order of the data records in a file is the same as or close to the order of data entries in an index (basically means sorted with respect to the order on the relation), then it is **clustered**, and otherwise **unclustered**.

[a] In practice, this is not done every deletion. This reorganizing process can be deferred and the B+ tree property may temporarily not hold.

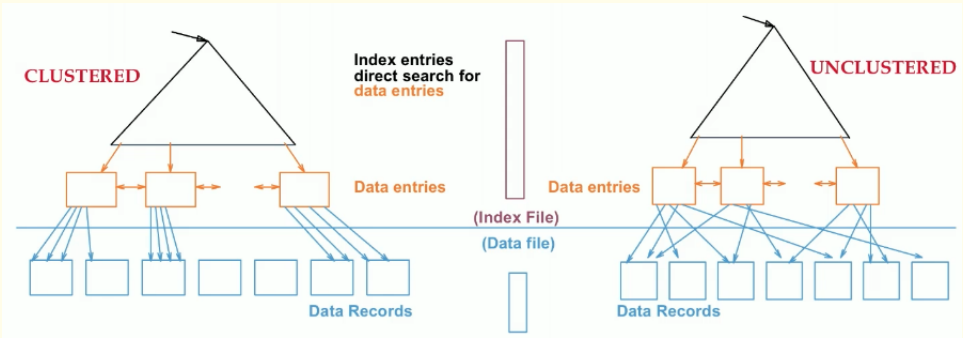

Figure 39: Even if the data entries (leaf nodes) are sorted, the memory addresses of the blocks on disk that they point to may not be sorted.

Note that the B+ tree is still a search tree! It is sorted. The clustered is a property of the data (blue squares).

The performance can really depend on whether the index is clustered or unclustered.

Example 5.3 (Possible Final Exam Question)

Assume that we have the query `SELECT  FROM User WHERE age = 50;` with the following assumptions:

1. Assume 12 Users with `age = 50`.
2. Assume one data page (block) can hold 4 User tuples (so $f = 5$).
3. Suppose searching for data entry requires 3 IOs in a B+ tree, which contain pointers to the data records (so $h = 3$).

If the index is clustered, then we can just traverse down the B+ tree to get to the leaf node containing the value 50.

1. We have a cost of 3 to traverse down the tree to access the index pages which show the memory addresses of the tuples on disk.
2. Then, we will find entries and we want to find the cost to access the data pages. There are 12 Users, and for every address, we load the entire block in memory, which will retrieve the other users of age 50 (since this is clustered). At best, we will retrieve 3 blocks (of 4 tuples each) and at worst, due to block overlap, we will retrieve 4 blocks and read the rest from memory. This gives us a cost of +3 or +4.

The total cost is 6. If the index is unclustered, then we are not guaranteed that the data with values 50 will be contiguous, so we will in the worst case have to look at 12 different blocks, leading to a total cost of $3 + 12 = 15$.

5.4 Hash and Composite Index

Definition 5.15 (Hash Index)

So far, we have used tree indices. However, an alternative is to use a **hash index** which hashes the search keys for comparison. Hash indices can only handle equality queries.

1. `SELECT * FROM R WHERE age = 5;` (requires hash index on `(age)`)
2. `SELECT * FROM R, S WHERE R.A = S.A;` (requires hash index on `R.A` or `S.A`)
3. `SELECT * FROM R WHERE age = 5 AND name = 'Bob'` (requires hash index on `(age, name)`)

They are more amenable to parallel processing but *cannot handle range queries or prefixes*, e.g. `SELECT * FROM R WHERE age >= 5;`. Its performance depends on how good the hash function is

(whether it distributes data uniformly and whether data has skew).

Definition 5.16 (Composite Index)

We've looked at queries in the form `SELECT  FROM User WHERE age = 50;`, but what if there are multiple conditions. For example, if we have

```
SELECT * FROM User WHERE age >= 25 AND name = 'B';
```

then we can do a couple things:

1. We have the index on `(age)`, at which point the leaf nodes will look something like

$$25, 25, 25, 26, 26, 28, 29, 29 \tag{32}$$

   We traverse through all the addresses in these leaf nodes and find the ones with name `B`.
2. If we have a **composite index** on `(age, name)`, then our leaf nodes will sort them as

$$(25, A), (25, A), (25, B), (26, A), (26, C), (28, B), (29, A), (29, C) \tag{33}$$

3. If we index on `(name, age)`, then our leaf nodes will sort them as

$$(A, 25), (A, 25), (A, 26), (A, 29), (B, 25), (B, 28), (C, 26), (C, 29) \tag{34}$$

   Note that if we had the query `SELECT  FROM R WHERE age >= 25`, then this sorting would not help since we do not prioritize ordering by age. So we cannot use tree indexing over name, age for this query.

Example 5.4 ()

Therefore, given a query, there are certain indices that we can use or cannot use for it.

1. If we have a query `A >= 5`,
   (a) Can use hash index in general.
   (b) Can use tree index in general.
2. If we have query `A >= 5`,
   (a) Can use tree with index (A).
   (b) Can use tree with index (A, B).
   (c) Cannot use tree with index (B, A) since A is not prefix.
3. If we have query `A = 5`,
   (a) Can use hash with index (A).
   (b) Can use tree with index (A).
   (c) Cannot use hash with index (A, B) since hashing this tuple does not allow us to compare to A or retrieve it. It is one-way and pseudo-random.
   (d) Can use tree with index (A, B).
4. If we have `A = 5 AND B = 7`,
   (a) Can use hash with index (A, B).

Each index has its pros and cons, so why not just use both tree and hash indices? The problem is that when we modify a relation on the disk, we need to update the index as well. Therefore, having too many indices requires us to update more and takes more disk space.

Okay, so we can't use too many indices, but are indices *always* better than table scans? Not exactly.

Example 5.5 (Table Scans Wins)

Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$ with around 20% of R satisfying $A > v$ (could happen even for equality predicates). We need to follow pointers to get the actual result tuples.

1. IOs for scan-based selection is simply $B(R)$ (where we can retrieve multiple tuples in this block), while
2. IOs for index-based selection is the lookup-cost (to traverse down the tree) plus $0.2|R|$ (since for each tuple, we do a IO lookup, retrieve it, and then have to retrieve the next tuple which is likely not in the same block)

So table scan wins if a block contains more than 5 tuples since we might as well grab everything rather than look them up one by one.

Thankfully, the query optimizer will make these decisions for you.

5.5 Index Only Plans

Definition 5.17 (Index-Only Plans)

There are queries that can be answered only by accessing the index pages and not the data pages, known as **index-only plans**. For index-only plans, clustering is not important since we are looking only at the leaf nodes at most. Therefore, we only need to compute the I/O cost of traversing the tree and not to access data.
For equality, we can just compute the number of tuples in the index pages where the equality condition is satisfied. For ranges, we may need to traverse the leaf nodes, which will lead to additional I/O cost to retrieve the leaf index pages.

Example 5.6 (Index Only Queries)

If we look at the following query

$$\pi_A(\sigma_{A>v}(R)) \tag{35}$$

we see that we only care about the value of A and not the rest of the tuples, so we only need to look at the index pages and not the data itself.

Example 5.7 (Primary Index Clustered According to Search Key)

If we have a primary index, in most cases the actual records are also stored in the index pages/leaf nodes. If they are clustered according to attribute A, then one lookup can lead to all result tuples in their entirety. You can just hit a leaf and grab your records as you walk along the leaves.

Example 5.8 (Other Index-Only Queries)

For example, if we just wanted to look at the count of users with age 50, then we don't need the data. We can just look at the number of values in the leaf nodes of the B+ tree with this value.

```
SELECT E.do COUNT(*)
FROM Emp E
GROUP BY E.dno;
```

If we have an index on `(E.dno, E.sal)`, then the two queries are also index-only plans. However, if we index on `(E.dno)`, then we need to retrieve `E.sal` on the data page, incurring more cost.

```sql
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno;
```

```sql
SELECT AVG(E.sal)
FROM Emp E
GROUP BY E.dno;
```

Example 5.9 (Halloween Problem)

The Halloween problem refers to a phenomenon in databases where an update operation causes a change in the physical location of a row, potentially allowing the row to be visited again later in the same update operation. Look at the update.

```sql
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

This caused everyone to have a salary of 25000+. This is because when we updated someone with salary of say 1000, it went to 1100 and is moved further right in the B+ tree. Therefore, this is revisited again is increased again in the same update. To fix this, we could update the values in reverse, from the rightmost leaf node to the leftmost one so that increasing values are visited once. Or we can just create a to-do/done list that keeps track of which ones have been updated.

6 Query Processing

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 6.1 ()

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.

Example 6.2 (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.

6.1 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.

Definition 6.1 (External Merge Sort)

The problem is that we want to sort R but R does not fit in memory. We divide this algorithm into passes.

1. Pass 0: Read M blocks of R at a time, sort them, and write out a level 0 run.
2. Pass 1: Merge $M-1$ level 0 runs at a time, and write out a level 1 run.
3. Pass 2: Merge $M-1$ level 1 runs at a time, and write out a level 2 run.
4. Final pass produces one sorted run.

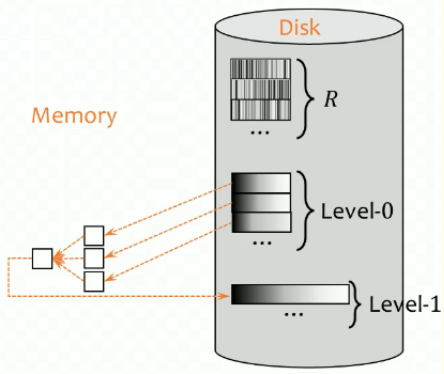

Figure 40

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
$$\lceil B(R)/M \rceil \tag{36}$$
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
$$\text{Num. of level i runs} = \left\lceil \frac{\text{Num. of level (i-1) runs}}{M-1} \right\rceil \tag{37}$$
3. The final pass produces 1 sorted run.

Therefore, the number of passes is

$$\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1 \tag{38}$$

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 6.3 ()

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

$$1, 7, 4, 5, 2, 8, 3, 6, 9 \tag{39}$$

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.

$$1, 7, 4 \mapsto 1, 4, 7 \tag{40}$$
$$5, 2, 8 \mapsto 2, 5, 8 \tag{41}$$
$$9, 6, 3 \mapsto 3, 6, 9 \tag{42}$$

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.

$$1, 4, 7 + 2, 5, 8 \mapsto 1, 2, 4, 5, 7, 8 \tag{43}$$
$$3, 6, 9 \tag{44}$$

3. Pass 2. Merges the final two relations.

$$1, 2, 3, 4, 5, 7, 8 + 3, 6, 9 \mapsto 1, 2, 3, 4, 5, 6, 7, 8, 9 \tag{45}$$

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.

6.2 Nested Loop Joins

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.

Definition 6.2 (Nested Loop Join)

Let's look at a brute force method of a theta join, called **nested-loop** join. If we run $R \bowtie_p S$, we do a nested for loop:

1. For each block of R and for each r in the block, we look at each block of S
2. For each s in the block, we output rs is p evaluates to true over r and s.

In here, R is called the outer table and S the inner table. The number of IOs is

$$B(R) + |R| \cdot B(S) \tag{46}$$

since we need $B(R)$ to load R, and for every tuple, we run through all of S. The memory requirement is 3 since we need 1 block to buffer R, 1 to buffer S, and 1 output block which acts as a buffer before we write to disk or stdout.[a]

This is clearly not efficient since it requires a lot of IOs. We can make a slight improvement.

[a] This is basically a buffer block where if it is filled, it is automatically flushed to disk.

Definition 6.3 (Block Based Nested Loop Join)

For each block in R and for each block in S, we try to basically do as much as we can with these two blocks. That is, we take a look at all $r, s \in R, S$ and see if p evaluates to true. The IO is therefore

$$B(R) + B(R) \cdot B(S) \tag{47}$$

This is because for each block of R, we go through all blocks of S and look through all pairwise tuples. The memory requirement is still 3.

We can 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. Therefore, the IO cost can be decreased to

$$B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S) \approx B(R) \cdot B(S)/M \tag{48}$$

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

Definition 6.4 (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

$$B(R) + |R| \cdot (\text{indexlookup}) \tag{49}$$

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.

6.3 Merge Sort Joins

Definition 6.5 (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.

This means that the IOs is really

$$\text{sorting} + 2B(R) + 2B(S) \tag{50}$$

which is worst case $B(R) \cdot B(S)$ when everything joins. Note that after sorting them, we must write them back to disk (which is usually the case if we can't fit the sorted relation in memory), which is $B(R) + B(S)$. Then we must load these sorted blocks back into memory, which takes an additional $B(R) + B(S)$.
To further optimize this, we can integrate the join with the merging step. We basically want to combine join with the (last) merge phase of merge sort. The runs are merging together, and then in memory we join them.

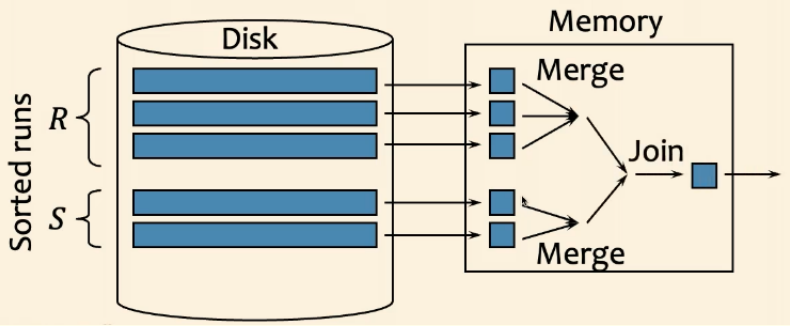

Figure 41: Sorting: produce sorted runs for R and S such that there are fewer than M of them total. Then in merge and join, we merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!

To analyze the runtime, if SMJ completes in two passes, then the IOs is really cheap.

$$3(B(R) + B(S)) \tag{51}$$

As for memory, we must have enough memory to accomodate one block from each run, so

$$M > \left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(R)}{M} \right\rceil \text{ or roughly } M > \sqrt{B(R) + B(S)} \tag{52}$$

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.

Example 6.4 ()

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.

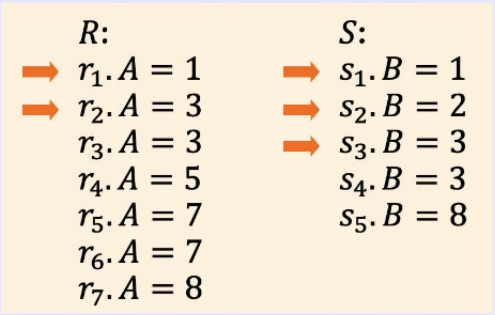

Figure 42

Definition 6.6 (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.

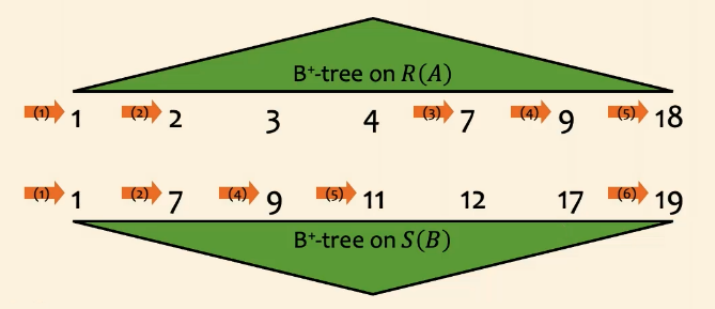

Figure 43: We see that the B+ tree of B has value 7 while that of A has value 2. Rather than traversing $2 \mapsto 3 \mapsto 4 \mapsto 7$, we can just traverse to 7 from the root node of A. This can give us a stronger bound.

6.4 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.

6.5 Hashing Based Algorithms

Definition 6.7 (Hash Join)

Let's start off with **hash joining**: $R \bowtie_{R.A=S.B} S$. The main idea 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.

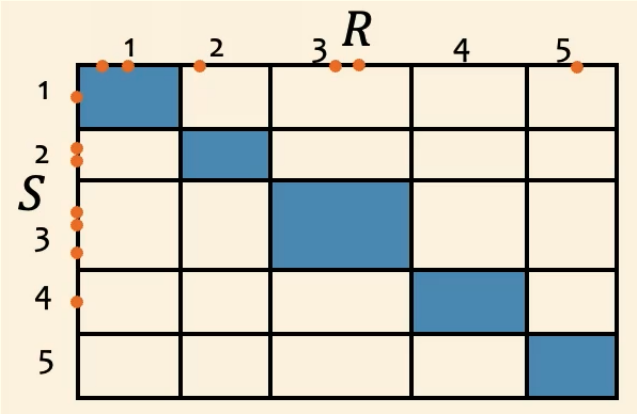

Figure 44: Say that the orange points represent the hashed values of $r.A$ and $s.B$ for $r, s \in R, S$. The nested loop considers all slots (all pairs of orange points between R and S), but hash join considers only those along the diagonal.

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).

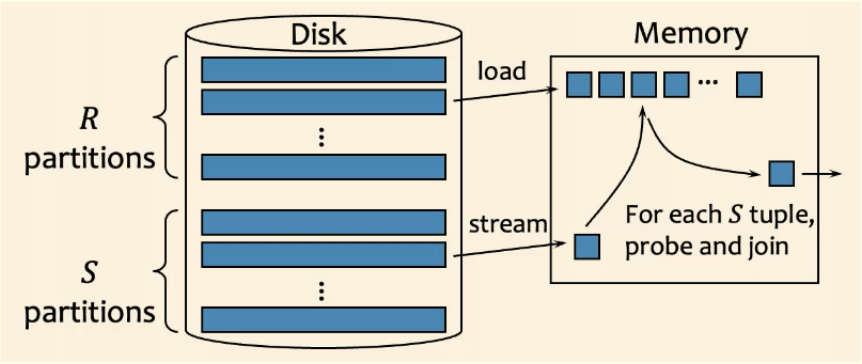

Figure 45

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

$$3(B(R) + B(S)) \tag{53}$$

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

$$M > \sqrt{\min\{B(R), B(S)\}} + 1 \tag{54}$$

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.

$$\sqrt{\min\{B(R), B(S)\}} + 1 < \sqrt{B(R) + B(S)} \tag{55}$$

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

6.6 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)

7 XML

So far, we have talked about relational data, which is a type of **structured data** with a schema, attributes, tuples, etc along with their types and constraints. For example, just trying to add a new attribute to a relation may require you to add the attribute for each tuple if it cannot be null. Some **unstructured data** is just plaintext, which doesn't confirm to any schema, and is on the other extreme end. Some types of data in between is HTML, XML, or JSON, which we call **semi-structured**, which may contain sections, subsections, etc.

Definition 7.1 (XML)

The **extensible markup language (XML)** is a semi-structured data that is similar to HTML, where the data self-describes the structure.[a]

1. There are **tags** marked by a start (`<tag>`) and end (`</tag>`) tags.
2. An **element** is enclosed by a pair of start and end tags(`<t>...</t>`), and elements can be nested (`<t><r></r></t>`or empty(`<t></t>`.
3. Elements can also have **attributes** (`<book ISBN="...", price="80.00">`).[b] Attributes must be unique, i.e. there cannot be duplicate attributes in a tag.

Note that there are some conventions that we must follow to get a **well-formed** XML document. First, the tags should be closed appropriately like parantheses matching, and we should not have the characters < or > as a part of the element. Rather, we should use `&lt;` and `&gt;`.

Example 7.1 ()

An example of XML data regarding a book is

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <title>The Great Gatsby</title>
    <author>
        <firstName>F. Scott</firstName>
        <lastName>Fitzgerald</lastName>
    </author>
    <published>
        <year>1925</year>
        <publisher>Charles Scribner's Sons</publisher>
    </published>
    <genre>Literary Fiction</genre>
    <price currency="USD">14.99</price>
```

[a] The names and nesting of its tags describe its structure.
[b] This is not the same attributes in relational databases.

```
    <inStock>true</inStock>
    <rating>4.5</rating>
</book>
```

This can be represented by a tree.

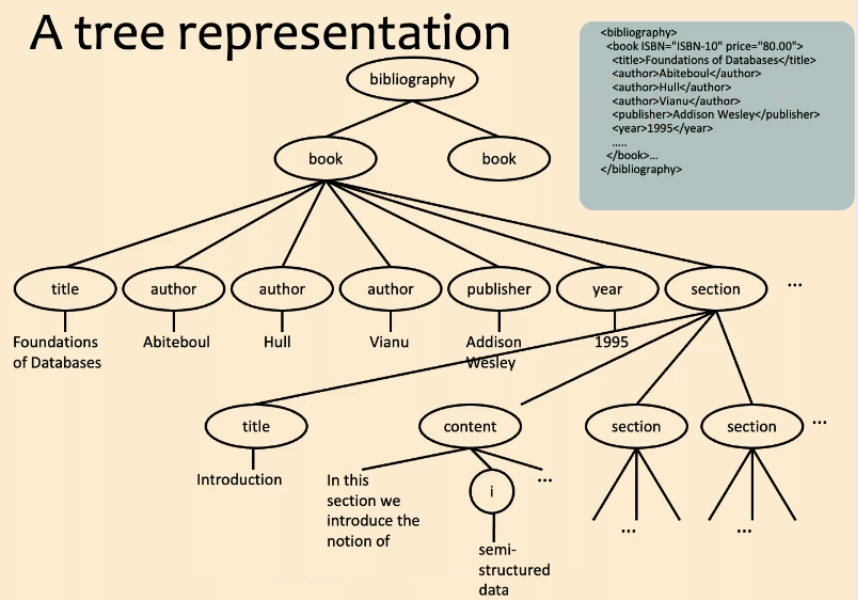

Figure 46: Tree representation of an XML file.

Let's do a quick comparison of relations and XML.

1. One advantage of XML is portability/exchangeability, since it self-describes itself, so all you really need is this text, unlike relations where you should also send the schema and other metadata. On the other hand, exchanging is problematic.
2. Second, its has flexibility to represent any information (e.g. structured, semi-structured, documents, etc.). Unlike a relation where each attribute of each tuple is atomic (meaning you can't place another relation in there), you can make a new XML tree.
3. Finally, since the data describes itself, you can change the schema easily. In contrast, the schema of a relational database is always fixed in advance and difficult to change.
4. Almost all major DBMS supports relations, and XML is often implemented as an add-on on top of relations.
5. Unlike the relational database where the order of the rows don't matter, the order does matter for XML.

Definition 7.2 ()

DTD Just because we don't need metadata does not mean that we cannot define one. This is done with the **document type definitions (DTD)**, which species the schema and constraints for XML just like relational databases. It has the following syntax, which should be written in the beginning of the XML file.

```
<!DOCTYPE root-element [
   <!ELEMENT element-name (content-specification)>
```

```
]>
```

Here are the common element specifications.

1. `EMPTY`: Element has no content
2. `ANY`: Element can contain any content
3. `(#PCDATA)`: Element contains parsed character data
4. `Child elements`: Listed in parentheses

with an example of a simple DTD being

```
<!DOCTYPE library [
   <!ELEMENT library (book+)>
   <!ELEMENT book (title, author, year)>
   <!ELEMENT title (#PCDATA)>
   <!ELEMENT author (#PCDATA)>
   <!ELEMENT year (#PCDATA)>
]>
```

Really, this is like a tree directory structure, so to query data, it makes sense to try and traverse the paths. There are three major query languages for XML, which are

1. **XPath**, which uses path expressions with conditions and are the building blocks of the rest of the standards.
2. **XQuery**, which is XPath plus a full-fledged SQL-like query language.
3. **XSLT**, or eXtensible Stylesheet Language Transformations and is the recommended style sheet language for XML.

7.1 XPath

Definition 7.3 (Basic XPath Constructs)

The syntax for building a XPath is as follows:

1. `/`: used as a separator between steps in a path (traversing down).
2. `name`: matches any child element with this tag name.
3. `*`: matches any child element
4. `name`: matches the attribute with this name.
5. `*`: matches any attribute
6. `//`: matches any descendent element of the current element itself.
7. `.` : matches current element.
8. `.` : matches parent element.[a]

Note that these names are all case sensitive, and another important fact is that NULL evaluates to NOT (so if there are no elements/attributes), then a condition on that element/attribute will evaluate to NOT.

Example 7.2 ()

Let's look at this example, which shows bibliographies for different books.

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliography>
    <book ISBN="ISBN-10" price="70">
```

[a] This is just like when we are navigating directories in a shell.

```
        <title>Foundations of Databases</title>
        <author>Abiteboul</author>
        <author>Hull</author>
        <author>Vianu</author>
        <publisher>Addison Wesley</publisher>
        <year>1995</year>
        <section>abc</section>
    </book>
    <book ISBN="ISBN-11" price="20">
        <title>DBSTS</title>
        <author>Ramakrishnan</author>
        <author>Gehrke</author>
        <publisher>Addison Wesley</publisher>
        <year>1999</year>
        <section>
          <title>bruh</title>
        </section>
    </book>
    <report>
      <author>Muchang</author>
      <addon>
        <author>Jon</author>
      </addon>
    </report>
</bibliography>
```

Let's look at some XPaths.

1. `/bibliography/book/author` gets all author element reachable via this path.

```
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
<author>Ramakrishnan</author>
<author>Gehrke</author>
```

2. `/bibliography/book/title` gets all book titles reachable via this path.

```
<title>Foundations of Databases</title>
<title>DBSTS</title>
```

3. `/bibliography/book/ISBN` gets all book ISBN attributes (we need the to search the *attribute*, not the element)

```
ISBN="70"
ISBN="ISBN:11"
```

4. `//title` returns all title elements, anywhere in the document. Note that the title of the section is also added.

```
<title>Foundations of Databases</title>
<title>DBSTS</title>
<title>bruh</title>
```

5. `//section/title` returns all section titles, anywhere in the document.

```
<title>bruh</title>
```

6. `/bibliography/*/author` returns all authors in bibliography that is a grandchild of bibliography. Note that Muchang is retrieved but not Jon.

```
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
<author>Ramakrishnan</author>
<author>Gehrke</author>
<author>Muchang</author>
```

7. `/bibliography/book[@price<50]` returns all books that are children of bibliography with attribute price less than 50.

```
<book ISBN="ISBN-11" price="20">
    <title>DBSTS</title>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <publisher>Addison Wesley</publisher>
    <year>1999</year>
    <section>
      <title>bruh</title>
    </section>
</book>
```

8. `/bibliography/book[author='Abiteboul']` returns all books that are children of bibliography that has an author tag children with element Abiteboul.
9. `/bibliography/book[author]` returns all books that are children of bibliography with some child author tag.
10. `book[40 <= @price and @price <= 50]` returns books with price attribute between 40 and 50.
11. `/bibliography/book[author='Abiteboul' or @price >=50]` returns books either authored by Abiteboul or price at least 50. Note that the first condition is on an element and the second is on an attribute.
12. `/bibliography/book[author='Abiteboul' or not (@price < 50)]` returns books authored by Abiteboul or price not below 50. Note that this is different from the query above since if a book does not have a price, then `price < 50` is null (which is NOT in XPath), so it returns the book.

Hopefully you get a good feel for how XPaths work. Here's a trickier example.

Example 7.3 (Conditions on Any vs All Attributes/Elements)

Say you want to get all books with some price in the range $[20, 50]$. Then you would think of writing

```
/bibliography/book/[price>=20 and price <=50]
```

This may not work if we have a XML element of this form, with multiple prices.

```
<book>
  <title>newbooktitle</title>
  <price>10</price>
  <price>70</price>
</book>
```

since it compares is any element satisfies the conditions. Therefore, we can think of these conditions

all as the `any` condition over child elements. If we want to use the `all` condition, we can write

```
/bibliography/book/[price[.>=20 and .<=50]]
```

Similarly, if we have the query `/bibliography/book[author='A' and author!='A']`, this is an any clause, so it will return all books with author A and another author not A.

Definition 7.4 (XPath Operators and Functions)

In conditions, we can use the following operations.

1. Arithmetic: `x + y`, `x - y`, `x * y`, `x div y`, `x mod y`
2. `contains(x, y)` returns true if string `x` contains string `y`
3. `count(node-set)` counts the number of nodes in `node-set`
4. `position()` returns the *context position* (i.e the position of the node amongst its siblings)
5. `last()` returns the *context size* (roughly the size of the node-set)
6. `name()` returns the tag name of the current element

Example 7.4 ()

Some more queries is

1. `/bibliography/book[count(section)<10]` returns books with fewer than 10 sections.
2. `//*[contains(name(), 'Ab')]` returns all elements whose tag names contain `Ab` (or `section`?)
3. `/bibliography/book/section[position()=1]/title` returns the title of the first section in each book.
4. `/bibliography/book/section[position()=last()]/title` returns the title of the last section in each book.

7.2 XQuery

XQuery is a superset of the XPath, but it also encompasses FLWOR expressions, quantified expressions, aggregation, sorting, etc. An XQuery expression can even return a new result XML document. Let's put the XML example file again for convenience.

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliography>
    <book ISBN="ISBN-10" price="70">
        <title>Foundations of Databases</title>
        <author>Abiteboul</author>
        <author>Hull</author>
        <author>Vianu</author>
        <publisher>Addison Wesley</publisher>
        <year>1995</year>
        <section>abc</section>
    </book>
    <book ISBN="ISBN-11" price="20">
        <title>DBSTS</title>
        <author>Ramakrishnan</author>
        <author>Gehrke</author>
        <publisher>Addison Wesley</publisher>
        <year>1999</year>
        <section>
          <title>bruh</title>
        </section>
    </book>
</bibliography>
```

Example 7.5 ()

For now, let's start with a simple XQuery based on XPath. To find all books, we can write the following, where the first `doc` refers to the specific document to query.

```
<result>{
  doc("bib.xml")/bibliography/book
}</result>
```

Note that like Python string interpolation, text outside of `{}` are copied to output verbatim, while those inside are evaluated and replaced by the result. We can use conditionals the same way.

```
<result>{
  doc("bib.xml")/bibliography/book[@price<50]
}</result>
```

which will return

```
<book ISBN="ISBN-11" price="20">
    <title>DBSTS</title>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <publisher>Addison Wesley</publisher>
    <year>1999</year>
    <section>
      <title>bruh</title>
    </section>
</book>
```

Definition 7.5 (FLWOR Expressions)

The fundamental building block of XQuery is the FLWOR expression, which stands for:

1. `FOR`: Iterates over sequences
2. `LET`: Binds variables to values
3. `WHERE`: Filters the tuples
4. `ORDER BY`: Sorts the results
5. `RETURN`: Constructs the result

Note that only the `RETURN` clause is mandatory; all others are optional.

Example 7.6 (FLWOR with Loops and Conditionals)

To retreive the titles of books published before 2000, together with their publisher, we can write either of the following. The logic is pretty self explanatory.

```
<result>{
  for $b in /bibliography/book
  let $p := $b/publisher
  where $b/year < 2000
  return
    <book>
    { $b/title }
    { $p }
    </book>
}</result>
```

```
<result>{
  for $b in /bibliography/book[year<2000]
  return
    <book>
    { $b/title }
    { $p/publisher }
    </book>
}</result>
.
.
```

which will return

```
<result>
  <book>
    <title>Foundations of Databases</title>
    <publisher>Addison Wesley</publisher>
  </book>
  <book>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
```

Note that `$p` may be assigned to a set of elements. It does not have to be one element.

Just like XPath, a conditional expression (with where) only checks the `any` condition.

Example 7.7 (Nested Loops and List Comprehension)

We can also use nested loops to solve the query above, but note the logic. If a book of price < 2000 has 2 publishers, then the book may be returned 2 times. On the other hand, if a book has a price < 2000 but has no publishers, then a null is really a false, so that book will not be returned.

```
<result>
{
  for $b in /bibliography/book,
  $p in $b/publisher
  where $b/year < 2000
  return
    <book>{ $b/title }{ $p }</book>
}
</result>
.
.
.
.
.
```

```
<result>
  <book>
    <title>Foundations of
    Databases</title>
    <publisher>Addison Wesley</publisher>
  </book>
  <book>
    <title>Foundations of
    Databases</title>
    <publisher>Springer</publisher>
  </book>
  <book>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
```

If we use a comprehension statement, we can let `$b` be all the books and try to parse them element by element where its year is < 2000. This is also wrong since the where clause is an any expression, so if at least 2 book has year < 2000, then it will return all the books. In fact, it doesn't even return it in the right format, since it returns all the book titles first and then all the publishers.

```
<result>
{
  let $b := /bibliography/book
  where $b/year < 2000
  return
    <book>
      { $b/title }
      { $b/publisher }
    </book>
}
</result>
```

```
<result>
  <book>
    <title>Found of Databases</title>
    <title>DBSTS</title>
    <publisher>Addison Wesley</publisher>
  </book>
</result>
.
.
.
.
```

Now that we went over FLWOR, let's talk about how we can do joins.

Example 7.8 (Explicit Join)

To find all pairs of books that have common authors, we can write

```
<result>
{
  for $b1 in //book
  for $b2 in //book
  where $b1/author = $b2/author
  and $b1/title > $b2/title
  return
    <pair>
      { $b1/title }
      { $b2/title }
    </pair>
}
</result>
```

Remember that since the where is an any condition, this works to our advantage now. We also use a string comparison of the authors and titles to remove duplicates. This gives us the result.

```
<result>
  <pair>
    <title>Foundations of Databases</title>
    <title>DBSTS</title>
  </pair>
</result>
```

We will stop here, but there are more features such as subqueries, existential (some) vs universal (all) queries, aggregation, conditional, etc.

7.3 Converstion of XML to Relational Data

Relational to XML is trivial, but XML to relational isn't really since there can be different data types that are siblings (e.g. books vs articles, vs papers all under bibliography), there may be repeats of the same tag, and there are both child elements and attributes.

The most trivial thing to do is to store the entire XML in a column (called a CLOB, character large Object type), but this isn't useful since it first does not satisfy the condition that each element must be atomic in a relation. There are two alternatives.

1. **Schema-Oblivious Mapping** takes a well-formed XML and converts it to a generic relational schema. In here, there are more subtypes.
   (a) **Node/edge based mapping** for graphs.
   (b) **Interval based mapping** for trees.
   (c) **Path based mapping** for trees.
2. **Schema-Aware Mapping** takes a valid XML and converts it to a special relational schema based on DTD.

Let's go over the node/edge based mapping. This is quite intuitive since we can visualize a XML structure as a directed graph.

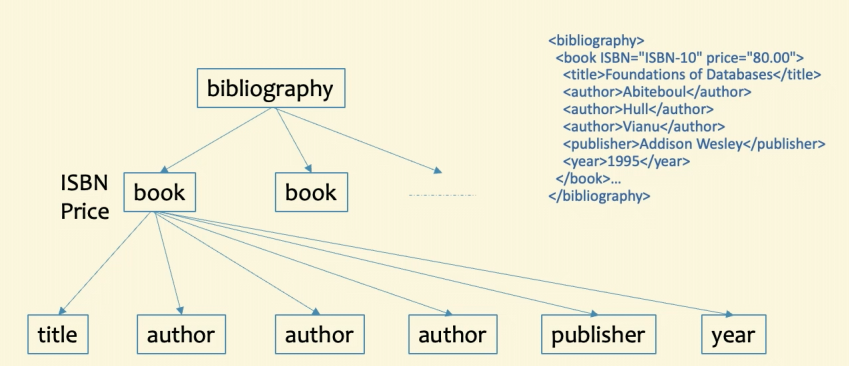

Figure 47

We can convert the XML to the following 4 relations. We also put the bibliography example to make it easier to follow along.

1. **Element(<u>eid</u>, tag)**. All elements will have a certain id with their tag type. Note that elements can have text inside of them, but we will consider this a "child" of the element, stored in the `Text` relation.

Table 13: Element Table

eid	tag
e0	bibliography
e1	book
e2	title
e3	author
e4	author
e5	author
e6	publisher
e7	year

Table 14

2. **Attribute(eid, attrName, attrValue)**. All attributes will need to have their name and type, along with which eid that they are a part of. Note that there is only one functional dependency `(eid, attrName) -> attrValue`.

Table 15: Attribute Table

eid	attrName	attrValue
e1	ISBN	ISBN-10
e1	price	80

3. **ElementChild(eid, pos, child)**. Each element will have a children, with pos referring to the position of the children. Child references either `Element(eid)` or `Text(tid)`.

Table 16: ElementChild Table

eid	pos	child
e0	1	e1
e1	1	e2
e1	2	e3
e1	3	e4
e1	4	e5
e1	5	e6
e1	6	e7
e2	1	t0
e3	1	t1
e4	1	t2
e5	1	t3
e6	1	t4
e7	1	t5

4. **Text(tid, value)**. All text in an element, with an id value (which cannot be the same as any eid) and the actual text in the value.

Table 17: Text Table

tid	value
t0	Foundations of Databases
t1	Abiteboul
t2	Hull
t3	Vianu
t4	Addison Wesley
t5	1995

Note that we need to invent a lot of ids and need indices for efficiency.

Given this, we can write the equivalent SQL queries from these XPath queries.

1. `//title`

```
SELECT eid FROM Element WHERE tag='title';
```

2. `//section/title`

```
SELECT e2.eid
FROM Element e1, ElemtnChild c, Element e2
WHERE e1.tag = 'section'
AND e2.tag = 'title'
AND e1.eid = c.eid
```

```
6  AND c.child = e2.eid;
```

Therefore, path expression becomes joins. The number of joins is proportional to the length of the path expression.

8 Query Processing and Optimization

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.

We can get a bit more theoretical and use the following identities in relational algebra.

Theorem 8.1 (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: - p_r is a predicate involving only R columns - p_s is a predicate involving only S columns - 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

8.1 Rewriting SQL Queries

Definition 8.1 (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 8.1 (Query Rewrite)

Given the query, we wish to rewrite it.

```
1  SELECT name
2  FROM User
3  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 8.2 (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);
```

8.2 Logical and Physical Plans

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 the intermediate steps with the following identities. They may or may not help in the runtime, since they are dependent on the size of the intermediate inputs. On the implementation side, we want to show the **physical plan**, which is the actual implementation including even more operations in between each node of the logical plan. This may include scanning or sorting.

The difference between the logical and physical plan is that the logical plan represents *what* needs to be done and not *how*. Consider the two approaches.

Example 8.3 (Query Plans)

Consider the following SQL query:

```
SELECT Group.title
```

[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.

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

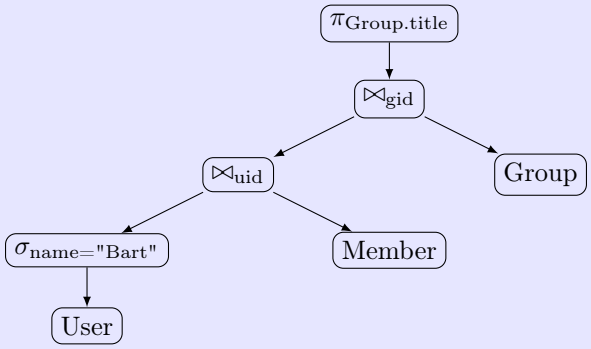

Figure 48: Logical Plan. We first take User, select Bart, and join it to Member over uid. Then we join it with Group on gid, and finally project the title attribute.

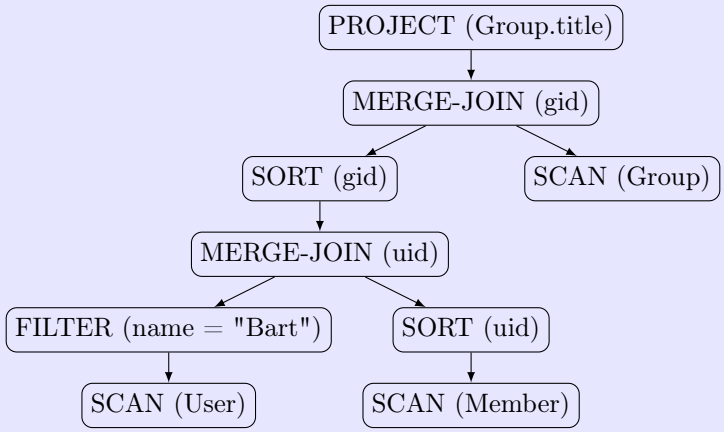

Figure 49: Physical Plan. We first take User and do a scan before filtering/selecting tuples with Bart. We also scan Member and sort it by uid in order to prepare for merge-join. Once we merge-join over uid, we sort it again to prepare a second merge-join with Group (which we scan first). Once we do this, we finally project the title attribute.

8.3 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 8.4 (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

$$|\sigma_{A=a}(R)| = \frac{|R|}{|\pi_A(R)|} = 2000 \tag{56}$$

tuples.

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

Example 8.5 (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,

$$|\sigma_{A=a,B=b}(R) = \frac{|R|}{|\pi_A(R)| \cdot |\pi_B(R)|} = \frac{100,000}{50 \cdot 10} = 200 \tag{57}$$

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

Example 8.6 (Negated, Disjunctive Predicates)

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

$$|\sigma_{A\neq a}(R)| = |R| \cdot \left(1 - \frac{1}{|\pi_A(R)|}\right) \tag{58}$$

and using I/E principle, we have

$$|\sigma_{A=a \vee B=b}(R)| = |R| \cdot \left(\frac{1}{|\pi_A(R)|} + \frac{1}{|\pi_B(R)|} - \frac{1}{|\pi_A(R)| \cdot |\pi_B(R)|}\right) \tag{59}$$

Example 8.7 (Range Predicates)

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

$$|\sigma_{A>a}(R)| = |R| \cdot \frac{\max(R.A) - a}{\max(R.A) - \min(R.A)} \tag{60}$$

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,

$$|\pi_A(R)| \leq |\pi_A(S)| \implies \pi_A(R) \subset \pi_A(S) \tag{61}$$

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

Example 8.8 (Two Way Equi-Join)

With the containment assumption, we have

$$|R \bowtie_A S| = \frac{|R| \cdot |S|}{\max(|\pi_A(R)|, |\pi_A(S)|)} \tag{62}$$

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.

8.4 Search Strategies

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.[3]

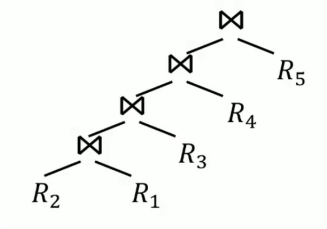

Figure 50: Left deep plans have a search space of only $n!$, which is better than before.

2. We can consider a balance 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*. 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 $S \subset [n]$.

$$f(\{R_i\}_{i \in S}) = \min_i f(\{R_j\}_{j \in S, j \neq i}) + f(R_i, \bowtie_{j \in S, j \neq i} R_j) \tag{63}$$

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.

9 JSON

NoSQL just stands for not SQL or not relational, like XML or **JSON**.[4] This relaxes some constraints and may be more flexible/efficient, with some popular data stores being MongoDB, CouchDB, Dynamo, etc.

[3] 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.

[4] I think it's called this because it's literally how JS objects are stored and printed.

They are designed to scale simple OLTP (online transaction processing) style application loads and provide good horizontal scalability. An example of where this is used is in pretty much all blockchains. Every transaction and block is accessible in a JSON format on say `etherscan.io`.

Definition 9.1 (JSON)

A **JSON**, short for **JavaScript Object Notation**, data model is an object with the following properties:

1. At the top level, it is an array of *objects*.
2. Each object contains a set of key-value pairs of form `{key : value}`, where key (attribute) names should be unique within an object.
3. It supports most primitive types (numbers, strings, double, booleans are stored in quotes e.g. `"true"`), along with arrays `[...]`, and finally **objects** `{...}` (which defines a recursive structure).
4. The order is unimportant.
5. You can't comment in JSON files.

Example 9.1 (Example JSON)

Here is an example with users, groups, and members.

```
{
  "users": [
    {"uid": 1, "name": "Bart"},
    {"uid": 2, "name": "Lisa"}
  ],
  "groups": [
    {"gid": 101, "title": "Skateboarding Club"},
    {"gid": 102, "title": "Chess Club"}
  ],
  "members": [
    {"uid": 1, "gid": 101},
    {"uid": 1, "gid": 102},
    {"uid": 2, "gid": 102}
  ]
}
```

Definition 9.2 (MongoDB Database)

A database has collections of similarly structured documents, similar to tables of records as opposed to one big XML document that contains all data.

1. **Database** is a list of collections. (analogous to a database)
2. **Collection** is a list a documents (analogous to a table)
3. **Document** is a JSON object (analogous to a row/tuple)

MongoDB actually stores this data with **BSON** (Binary JSON), which is the binary encoding of it.

MongoDB provides a rich set of operations for querying and manipulating data.

Definition 9.3 (find())

The `find()` operation is MongoDB's basic query mechanism. It returns a cursor to the matching documents.

1. Takes a query document that specifies the selection criteria

2. Can include projection to specify which fields to return
3. Supports comparison operators like `$eq`, `$gt`, `$lt`
4. Can query nested documents and arrays

Example 9.2 (Basic Query)

Find all users named "Bart":

```
db.users.find({"name": "Bart"})
```

Find users with uid greater than 1:

```
db.users.find({"uid": {$gt: 1}})
```

Definition 9.4 (sort())

The `sort()` method orders the documents in the result set.

1. Takes a document specifying the fields to sort by
2. Use 1 for ascending order, -1 for descending
3. Can sort by multiple fields
4. Applied after the query but before limiting results

Example 9.3 (Sorting)

Sort users by name in ascending order:

```
db.users.find().sort({"name": 1})
```

Definition 9.5 (Aggregation Pipeline)

An aggregation pipeline consists of stages that transform sequences of documents. Each stage performs an operation on the input documents and passes the results to the next stage. Despite its name, it handles more than just aggregation operations.

MongoDB supports several types of pipeline stages:

1. **Selection and Filtering** (`$match`)
   - Filters documents based on specified conditions
   - Similar to `find()` but within the pipeline context
2. **Projection** (`$project`)
   - Reshapes documents by including, excluding, or transforming fields
   - Can create computed fields
3. **Sorting** (`$sort`)
   - Orders documents based on specified fields
   - Equivalent to the `sort()` method
4. **Grouping** (`$group`)

- Groups documents by a specified expression
- Supports aggregation operators:
  - `$sum`: Calculates numeric totals
  - `$push`: Accumulates values into an array

5. **Document Transformation**
   - `$project`/`$addFields`: Adds computed fields
   - `$unwind`: Deconstructs arrays into individual documents
   - `$replaceRoot`: Promotes an embedded document to the top level
   - Array operators:
     - `$map`: Applies an expression to each array element
     - `$filter`: Selects array elements matching a condition
6. **Joining** (`$lookup`)
   - Performs left outer joins with other collections

Example 9.4 (Basic Pipeline)

A pipeline that finds users in the "Chess Club" and sorts them by name:

```
db.members.aggregate([
    {$lookup: {
        from: "users",
        localField: "uid",
        foreignField: "uid",
        as: "user"
    }},
    {$match: {"gid": 102}},
    {$sort: {"user.name": 1}}
])
```

Example 9.5 (Complex Pipeline)

A pipeline using multiple stages to group and transform data:

```
db.members.aggregate([
    {$group: {
        _id: "$gid",
        members: {$push: "$uid"},
        count: {$sum: 1}
    }},
    {$lookup: {
        from: "groups",
        localField: "_id",
        foreignField: "gid",
        as: "group_info"
    }},
    {$project: {
        _id: 0,
        group: {$arrayElemAt: ["$group_info.title", 0]},
```

```
        member_count: "$count",
        members: 1
    }}
])
```

The pipeline stages must be carefully ordered as each stage's output becomes the input for the next stage. For optimal performance:

1. Use `$match` early to reduce the number of documents
2. Place `$project` and `$unwind` before `$group` when possible
3. Consider memory limitations when using `$sort`

Definition 9.6 ($lookup)

The `$lookup` operation performs a left outer join to another collection.

1. Must be used within an aggregation pipeline
2. Specifies foreign collection to join with
3. Defines local and foreign fields to join on
4. Results stored in an array field

Example 9.6 (Join Operation)

Join members with users:

```
db.members.aggregate([
  {$lookup: {
    from: "users",
    localField: "uid",
    foreignField: "uid",
    as: "user_info"
  }}
])
```

Definition 9.7 (Aggregation Operators)

Special operators used within aggregation pipelines for computations.

1. `$sum`: Calculates sum of numeric values
2. `$push`: Adds value to an array
3. `$avg`: Calculates average
4. `$first`/`$last`: Returns first/last value in group

Example 9.7 (Aggregation Operators)

Calculate total members and collect group IDs:

```
db.members.aggregate([
  {$group: {
    _id: "$uid",
    total: {$sum: 1},
    groups: {$push: "$gid"}
```

```
  }}
])
```

10 Transactions

So far, we've had one query/update on one machine, but this is not the case in reality. In modern systems, you have users interacting with a database all the time and databases may be prone to failure. These two situations requires us to develop a safer way of interacting with the database.

Example 10.1 (Simultaneous Interaction)

In an airline, say that we have two parties A and B trying to book the same seat. A looks at the website, which queries the tuple representing the seat. B does the same. A and B then both book it at the same time, sending an update request to the database, causing an overbooking.
In a bank, say that we have some events that must be recorded in a database. Say that $A = 0$ and $B = 100$.

1. T1. B sends \$100 to A. So $A \mapsto A + 100, B \mapsto B - 100$.
2. T2. The bank sends all users an interest of 6%. $A \mapsto 1.06A, B \mapsto 1.06B$.

This is sensitive to order and perhaps at the end we expect $A = 100$. Consider the following cases.

1. $A \mapsto A + 100, B \mapsto B - 100, A \mapsto 1.06A, B \mapsto 1.06B$. This is fine.
2. $A \mapsto A + 100, A \mapsto 1.06A, B \mapsto 1.06B, B \mapsto B - 100$. This is not fine since both A and B got interest and the bank lost \$6.

Example 10.2 (Database Crash)

If $A \mapsto A + 100$ happened first and the bank crashed, then the system would find that A just gained \$100! This is clearly not good, so we must undo what we have done so far.

These two examples hint at some nice properties that we want in our database.

1. *Serializability.* The first example shows that we do not like parallelism and rather we want things to run *serially* in a way such that T1 and T2 occur separately, like a mutex lock. In practice, there are so many requests that parallelism is a must, so we must find ways to not run serially, but to modify our parallel computing to make it serializable, as if it is running serially.
2. *Atomicity.* The second example shows that we want these operations to be *atomic*, i.e. it is fully done or not done at all. That is, we do not want to update the disk (which preserves its state upon powering off) until T1 is completely finished. A simple solution is to do all the operations in memory (which is wiped upon crashing anyways) and then *commit* (announce that it is done) these changes. [5]

Definition 10.1 (Transaction)

The solution to both serializability and atomicity is to use **transactions**, which is a collection of one or more operations, called **actions**, on the database that must be executed atomically as a whole. At the end of every transaction, the DBMS always puts either a commit (indicating successful completion) or abort signal.

[5] Committing is not the same as disk writing, actually. It is not necessarily the case that one follows the other, so you can commit before writing and write before committing. There can also be some completed transactions with updated data still in memory and therefore lost in crash.

Definition 10.2 (ACID)

These properties are a subset of the **ACID** properties.

1. *Atomicity.* A user can think of a transaction as always executing all actions in one stop or not executing any at all. In an abort, this can be undone by storing recovery logs and going through the history to undo.
2. *Consistency.* Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database.[a]
3. *Isolation.* A user should be able to understand a transaction without considering the effect of any other concurrently running transaction. This allows us to have *concurrency control*, which handles multiple transactions by interweaving them in a serializable way (like the bank example we saw above), and sometimes they may ensure isolation with locks.
4. *Durability.* Once the DBMS informs the user that a transaction is completed, its effect should persist.[b]

Definition 10.3 (Schedule)

Now that we've cleared up on transactions, a schedule is simply a sequence of actions gotten from interweaving the transactions on the DBMS. It must obviously be consistent, and two actions from the same transaction T must appear in the schedule in the same order that they appear in T. Some examples of actions that can occur in a schedule is:

1. `READ`
2. `WRITE`
3. `COMMIT`
4. `ABORT`

Some terminology to compare schedules:

1. **Serial Schedule**. No interweaving of transactions.
2. **Equivalent Schedule**. Two schedules that have the same effect on the database state after completion.
3. **Serializable Schedule**. A schedule that is equivalent to a serial schedule.

Example 10.3 (Bank Schedule)

For the bank example above, we can construct a serial schedule S_1.

```
T1: R(A) W(A) R(B) W(B) C(T1)
T2:                          R(A) W(A) R(B) W(B) C(T2)
```

and a serializable schedule S_2.

```
T1: R(A) W(A)                          R(B) W(B) C(T1)
T2:           R(A) W(A) R(B) W(B) C(T2)
```

We can verify that $S_1 \equiv S_2$.

10.1 Isolation Levels

When we interweave actions, conflicts may arise if one transaction wants to write to a data that another transactions reads/writes. This causes schedules that are not equivalent to the serial schedule.

1. Write-Read (WR): reading uncommited or *dirty data*

[a] e.g. if you transfer money from one account to another, the total amount in circulation still remains the same. The constraints must also be maintained.

[b] Even if the system crashes before writing to disk, we can recover it by using the history logs and redoing them.

2. Read-Write (RW): unrepeatable reads.
3. Write-Write (WW): overwriting uncommitted data or lost updates, which can happen in the same or different transactions.
4. RR has no conflicts since there are no writes.

In SQL a transaction is automatically started when a user executes a SQL statement, and subsequent statements in the same session are executed as part of this transaction. Statements see changes made by earlier ones in the same transaction and statements in other concurrently running transactions do not. `COMMIT` and `ROLLBACK` commands are self explanatory.

SQL also supports isolation levels, which increases performance by eliminating overhead and allowing higher levels of concurrency, albeit at the cost of getting inconsistent answers if you don't know what you're doing. We start off with the most restrictive and go down.

1. `SERIALIZABLE` is the strongest isolation level, with complete isolation.
2. `REPEATABLE READ` allows repeated reads, and you are safe with updates, but you aren't safe with new insertions, which are called *phantoms*.

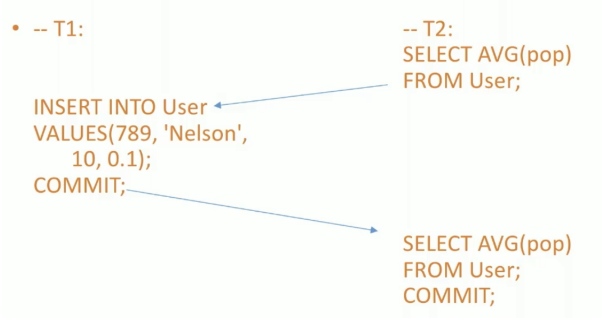

Figure 51: You still have a different average due to the insertion of a new element.

3. `READ COMMITTED` does not allow dirty reads, but non-repeatable reads are allowed (RW conflicts), which means that reading the same data twice can produce different results and is allowed.

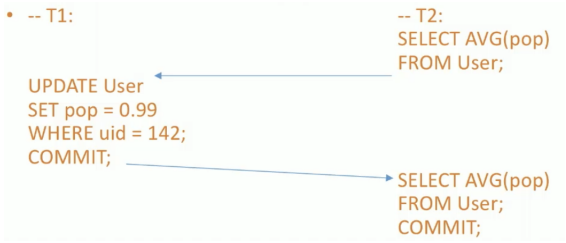

Figure 52: In T2, we first compute an average of say 0.6. Then in T1 we update the popularity and commit. Then in T2, we take the average again, getting 0.8, and finally commit. T2 ends up reading two different states of the database.

4. `READ UNCOMMITTED` allows you to read uncommitted/dirty data (WR conflict).

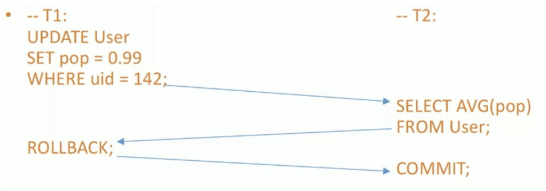

Figure 53: In T1, you are setting the popularity to 0.99. In T2, you read this uncommitted data to compute the average. However, T1 aborts this write, and in T2 you are left with the wrong average.

Here is a summary of the isolation levels.[6]

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

Figure 54

10.2 Serizability and Precedence Graph

Okay, so serial schedules are trivial to construct and ideally we want to construct serializable schedules. How do we detect them? We can make a **precedence graph**.

[6]Postgres defaults to read committed.