CSC 370 Notes

lshatzel

January 2023

Contents

1 ER Diagrams

1.1 Database Design

- Specify what information the db must hold.
- What relationships exist between components of that info.
- We use Entity Relationship (E/R) model to express this design.

General process: Ideas → E/R Design → Relational schema → Relational DBMS.

1.2 Elements of E/R Model

- Attributes: Oval
- Relations: Diamonds
- Entity Set: Rectangles

1.2.1 Attributes

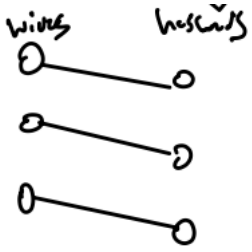
- Can only be atomic types
- int, float, varchar, char
- NOT lists, arrays
- We rename attributes to avoid name clashes. (rel alg $name \rightarrow newname$)

1.3 Keys

- Primary Key (PK): Unique identifier for each tuple
- Foreign Key (FK): References primary key of another tuple (allows us to reference other tuples from a different table [defines relationships]).
- Surrogate Keys: Introducing a unique identifier when there isn't a preexisting one (i.e. student number).

1.4 Relationships

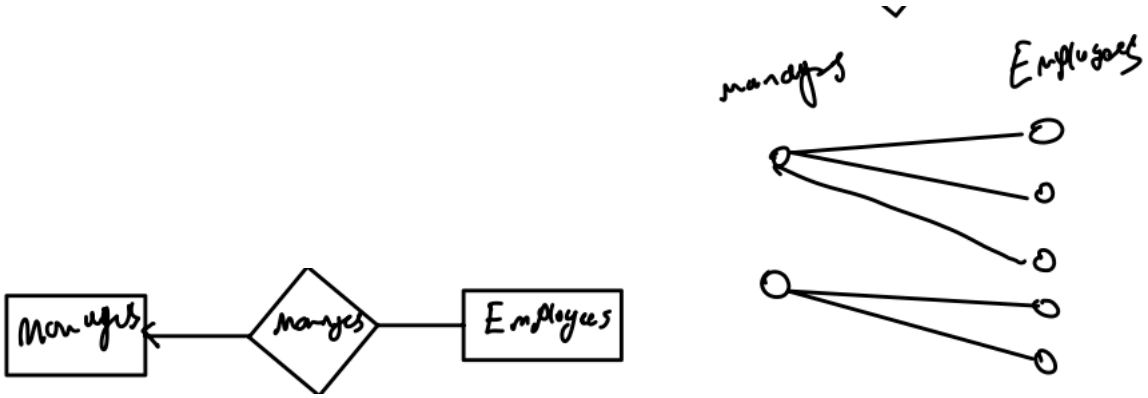
1.4.1 One-One



- Given a wife she is married to exactly one husband
- Given a husband he is married to exactly one wife

ER→Relation
When creating the relations, one of the tables takes the PK of the other as a FK

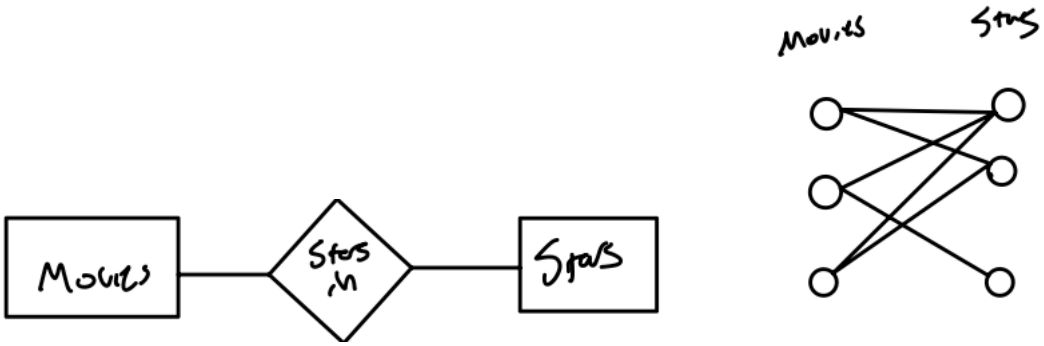
1.4.2 Many-One



- Given a manager they manage many employees
- Given an employee they have one manager

ER→Relation
The many side of the relationship takes the key of the one side as a FK in its table

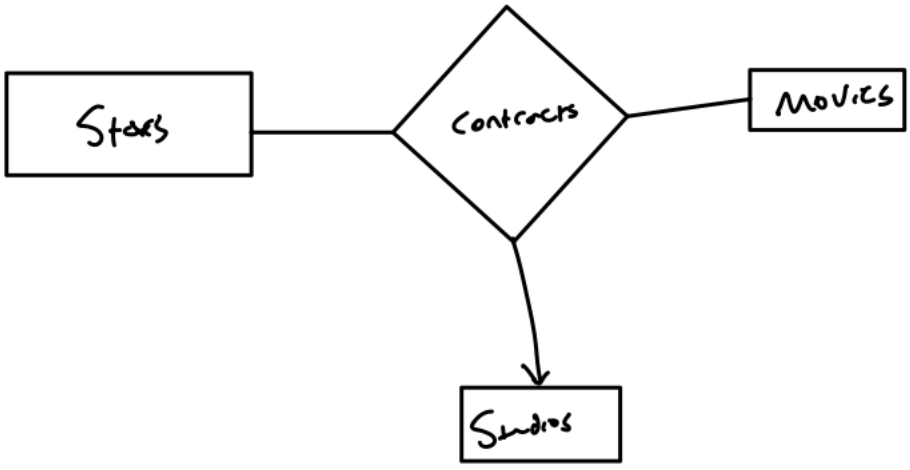
1.4.3 Many-Many



- Given a star they may star in many movies
- Given a movie they may have many stars, star in them.

ER→Rel
Stars in becomes its own table with PK as combination of PKs (as FKs) from all participating ESs

1.5 Multiway Relationships



- they always get their own table.
- Given a star and a movie they are related to one studio
- More specifically, the arrow within the multiway relationships means if we select one entity from each of the other entity sets in the relationship (stars, movies) those entities are related to at most one entity in E (studios).
- Selecting a star and a movie
- i.e. selecting a star and a movie will only have a contract with one studio.
- Given a studio they may contract with many stars for a movie.
- Given a studio they may contract with many movies with a star.

ER→Rel
Turn into table, take all keys from many side and put into table as combined PK, one side keys also go in but not as PK

1.6 Roles

- For labelling entities when they appear twice or more in a single relationship.
- Each line represents a role to the relationship as a label.
- Given a movie there may be only one orignal
- Given a movie there may be many sequels

ER→Rel
Follows rules from standard relationships

1.7 Attributes on Relationships

- We may put attributes on relationships
- The attribute goes into the relationship table as attribute name
- On a one to many/one to one relationship the attribute on the relationship follows the foreign key into the table (i.e. going into the many side)

1.8 Converting Multiway to Binary

- we can think of this as introducing a new ES in place of the previous relationship while maintaining the same schema structure (through many-one rels).
- Introduce a new ES (entities of which are tuples of relationship set)
- Called connecting entity set (ES).
- Introduce many-one relationships from the connecting ES to each participating ES in original relationship.
- Many-one allows connecting ES to act as a relationship tuple.

1.9 Subclasses

- General class as root
- More specialized classes as leaves
- is-a relationship
- keys from a parent class are taken into subclasses, subclass tables are constructed normally

1.10 ES vs Attribute

1.10.1 Using ES over Attribute

- Rule 1: ES when it is more than the name of something
- Rule 2: ES when it is the "many" side of a relationship

Using attribute over ES

- All relationships with E must have arrows pointing at E. (it is the one side in many-one relationships)
- If E has more than one attribute then no attribute relies on the other attributes (i.e. the only key for E is all of its attributes combined)
- No relationships involves E more than once

1.10.2 Replacement

- If E is connected in a binary relationship to another ES, F then F takes the attributes of E
- If E is in a multiway relationship then the relationship takes the attributes of E

1.11 Design Principles

1. Faithfulness: Design should reflect reality
2. Avoid Redundancy: remove redundant relationships
3. Keep it Simple: use attributes when possible but dont make everything an attribute

1.12 Keys

- Unique over all attributes
- Every ES must have a key (except "isa" and weak ES)
- There can be more than one key but usually pick one
- Root of isa always needs to have the attributes for keys of subclasses

1.13 Subclasses

- General class as root
- More specialized classes as leaves
- Forms an is-a relationship
- Keys from parent class are taken into subclasses, which are constructed normally.
- In other words the key of the root is the key for all

1.14 Weak Entity Sets

Note: better to just introduce a surrogate key. Requirements:

1. Zero or more of its own attributes
2. Keys are reached through supporting relationships
3. Key attributes must come from keys of other ES
4. If a weak-weak rel is present, keys come from a strong ES further down the line
5. You can use many supporting rels to generate a key for a weak ES.

ER $\rightarrow$ Rel
Take the key (and all attributes) from the one side and put it in the many-side schema. Key of the weak relation becomes a compound key with the key of the one side and many side.

1.15 Crows Foot Notation

- Multiplicity one: $--|----$
- Multiplicity one or none: $--|o----$
- Many: $->|----$
- Many or none: $->o----$
- Exactly one: $--|-|----$

1.16 Terminology

Every attribute has atomic types

- Relation Schema (atomic): Relation name + attribute names + attribute types [i.e. Stars(name $<$ *type* $>$, DOB $<$ *type* $>$)
- Relation Instance: Set of tuples (therefore only one copy of any tuple)
- Database Schema (atomic): Set of relation schemas
- Database instance: A relation instance for every relation in the schema (the actual data connected to the table headers for many tables)

2 Relational Algebra

2.1 Operations

Set operations (schemas need to be of the same shape):

- Union - $\cup$ - `UNION`
- Intersection - $\cap$ `INTERSECT`
- Difference - $\setminus$ `EXCEPT`

Operations that remove parts of a relation:

- Selection - σ_C - `WHERE C` *eliminates tuples*
- Projection - Π_L - `SELECT` *eliminates columns*

Operations that combine tuples of two relations:

- Cartesian Product - $\times$ - `FROM R1, R2`
- Join - $\bowtie$ - `R1 JOIN R2`

2.2 Set Operations

Conditions:

- we may apply set operations ($\cup$, $\cap$, $\setminus$) to R and S if:
- R and S have the same attributes
- R and S have the same order of the same attributes (must be reordered if not)

2.3 Projection

- $\Pi_{A_1,\ldots,A_n}(R)$
- Projection produces a new relation from R.
- Where only the $A_1, \ldots, A_n$ columns are preserved.

2.4 Selection

- $\sigma_C(R)$
- Produces a new relation with tuples of R satisfying condition C.

2.5 Cartesian Product

- $R \times S$
- Set of tuples rs that are formed by choosing first part to be any tuple of R and second part s to be any tuple of S
- Resulting schema is the union of schemas for R and S
- If R and S have some attributes in common then they are prefixed by the relation name (to avoid confusion)

R:

A	B
1	2
3	4

S:

B	C	D
2	5	6
4	7	8

$R \times S$:

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
3	4	2	5	6
3	4	4	7	8

2.6 Theta-Join

- $R \bowtie_C S$
- Cartesian product with selection specified (equivalent to $\sigma_C(R \times S)$)
- When the condition is equality, it is called equijoin
- Any attributes with the same name are also prefixed with original table name after joining just like cartesian product.

2.7 Natural Join

- $R \bowtie S$
- Special case of equijoin where we want the attributes with the same name to be joined into one column in the new table
- We join all of the attributes that both R and S agree upon.

$R \bowtie S$

A	B	C	D
1	2	5	6
3	4	7	8

2.8 Renaming

- $\rho_S(A1, A2, \ldots An)(R)$
- creates a new relation
- Renaming R to S
- Renames the original tuples to $A1, A2, \ldots An$

2.9 Bags

- A bag is like a set but elements may appear more than once
- order doesnt matter
- SQL is a bag language (typically everything is done in bags)
- exception: UNION, INTERSECTION, DIFFERENCE

2.10 Bag Union

- A element appears in the union of two bags the sum of the number of times it appears in each bag
- $\{1, 2, 1\} \cup \{1, 1, 2, 3, 1\} = \{1, 1, 1, 1, 1, 2, 2, 3\}$

2.11 Bag Intersection

- An element appears in the intersection of two bags the minimum number of times it appears in either bag.
- $\{1, 2, 1\} \cap \{1, 2, 3\} = \{1, 2\}$
- Notice: 3 goes away because it appears a minimum of 0 times in first bag

2.12 Bag Difference

- An element appears in bag difference (A - B) as many times it appears in A minus the number of times it appears in B (never less than 0 times)
- $\{1, 2, 1\} - \{1, 2, 3\} = 1$
- Notice: In this case 3 never even comes into play.

2.13 Extended Relational Algebra

- δ : Eliminates duplicates from bags
- τ: Sorts tuples (we can add subscript to denote which attribute to sort)
- Π: We may now do arithmetic and duplicate columns
- γ: Grouping and aggregation
- $\overset{\circ}{\bowtie}$: Outerjoin (superset of join)

2.14 Extended Projection

- Using Π_L we can now have expressions involving attributes
- $\Pi_{A+B \to C}$
- The above with sum the columns of A and B and rename it to C

2.15 Aggregation

- SUM, AVG, COUNT, MIN, MAX
- usually used in conjuction with the grouping operator.
- IMPORTANT NOTE: THEY MAY BE USED ALONE IN A PROJECTION

2.16 Grouping Operator

- γ_L
- Where L is a list of elements that are either individual (grouping) attributes
- Or AGG(A), aggregation operators and A is an attribute
- How it works:
- Group R according to attributes on L
- Form one group for each distinct list of values for those attributes in R
- Within each group compute AGG(A) for each aggregation on L
- The resulting relation has

Consider:
R =

A	B	C
1	2	3
4	5	6
1	2	5

Then (after grouping) $\gamma_{A,B,AVG(C)}(R) =$

A	B	C
1	2	3
1	2	5
4	5	6

Final Product:

A	B	C
1	2	4
4	5	6

2.17 Outer Join

- If we ant to join R and S
- A tuple of R which doesnt join with a tuple of S is dangling
- the same thing happens with tuples of S
- We lose dangling tuples
- Outerjoin allows us to retain dangling tuples by padding them with nulls

R=

A	B
1	2
4	5

S=

B	C
2	3
6	7

$R \overset{\circ}{\bowtie} S$

A	B	C
1	2	3
4	5	NULL
NULL	6	7

- Both (4,5) and (6,7) are dangling tuples when joining using B because they have no match.
- Therefore outer join pads them with null
- We also have left and right outer join which allows us to pad onlt the table on the left or the right

2.17.1 Left Outerjoin

- Left outer join pads the only the table on the left with null values
- (since A is the column that came from the table on the left it is the one that gets padded)

$R \overset{\circ}{\bowtie}_L S$

A	B	C
1	2	3
NULL	6	7

2.17.2 Right Outerjoin

A	B	C
1	2	3
4	5	NULL

- Right outer join pads only the table on the right with null values
- (since C came from the table on the right it is the one that gets padded

3 SQL

3.1 Creating Tables

```
CREATE TABLE Students(
    sid INT PRIMARY KEY,
    name VARCHAR(20)
    ...
);

--alternative primary key
- allows the primary key to be more than one attribute
CREATE TABLE Students(
    sid INT,
    name VARCHAR(20)
    ...
    PRIMARY KEY(sid, name)
);

--Foreign keys
CREATE TABLE Takes(
    sid INT REFERENCES Students(sid),
    ...
);

--alternatively (when foreign key has more than one attr)
CREATE TABLE Takes(
    sid INT,
    name VARCHAR(20),
   FOREIGN KEY(sid, name) REFERENCES Students(sid, name)
);

--Dropping the table
DROP TABLE Students;
```

- Types:
- INT
- FLOAT

- VARCHAR
- CHAR
- DATE
- NOTE: VARCHAR is used with strings which vary in length, whereas CHAR is used with strings which are always fixed in length (phone numbers)

3.2 Insertions

```
INSERT INTO <Table> VALUES(<tuple>);

INSERT INTO Students VALUES (12345, 'bobby');

--Deletions
DELETE FROM <table> WHERE <condition>

DELETE FROM Students WHERE name='Bobby';
```

3.3 Updates

```
UPDATE <Table>
SET <attribute>
WHERE <condition>

UPDATE Students
SET name='Johnny
WHERE name='Bobby'
```

3.4 Altering Table Structure

```
ALTER TABLE <table> ADD <attribute> <type>;
--other arguments: MODIFY, DROP

--Examples:

ALTER TABLE Stars ADD phone CHAR(7);

ALTER TABLE Starys MODIFY phone CHAR(10);

ALTER TABLE Stars DROP COLUMN phone;
```

3.5 General Query Syntax

```
--Typically queries have form: SELECT, FROM, WHERE
SELECT <attributes>
FROM <table(s)>
WHERE <condition>
```

3.5.1 WHERE

Uses the following tools:

- Operators: $=, <>$ $(!=), <, >, <=, >=$
- strings use single quotes: ”
- Boolean Operators: AND, OR, NOT

- NOTE: precedence of OR is less than AND.
- Patterns:
  - <attribute> LIKE <pattern>
  - <attribute> NOT LIKE <pattern>
- Where pattern may contain % (any string) _ (any character)

3.6 Ordering

```
SELECT *
FROM movies
WHERE studioName='Disney'
ORDER BY length;

--This will order the movies by length
--Default is ASC (shortest first)
```

3.7 Products and Joins

3.7.1 Cartesian Product

```
--In SQL the cartesian product is denoted by the comma operator

SELECT website
FROM Movies, Studios
WHERE title = 'Pretty Woman' AND studioName=name;

--When we join tables with attributes that have the same name
-- We disambiguate by adding the table name and then a dot
--to tell which table that attribute is referencing
SEELCT Stars.name, Stars.birthdate, Stars.birthplace
FROM Movies, Studios, StarsIn
WHERE StarsIn.starname = Stars.name
      AND studio='Paramount'
      AND Movies.title = StarsIn.title
      AND Movies.year = StarsIn.year
```

3.7.2 Natural Join

```
SELECT *
FROM movies NATURAL JOIN starsin;

--the difference between natural join and the cartesian product is that
--when we take a cartesian product the attribute being joined on has two colums
--one from table A and one from table B
--On the other hand with a natural join, all like values from table A and B get joined into
--one column

--We can also natural join with using
SELECT *
FROM Movies JOIN Starsin USING(title, year);
```

3.7.3 Joins with On

```
SELECT *
FROM movies JOIN StarsIn ON(movies.title =starsin.title AND movies.year = starsin.year);

--it is the same as
```

```
SELECT *
FROM Movies, Starsin
WHERE movies.title=starsin.title AND movies.year = starsin.year;
```

3.7.4 Outer Joins

```
--See rel alg outer join section for visualization

--Full outer join
SELECT *
FROM R NATURAL FULL OUTER JOIN S

-- left outer join
SELECT *
FROM R NATURAL LEFT OUTER JOIN S

--right outer join
SELECT *
FROM R NATURAL RIGHT OUTER JOIN S

--note USING, ON also work with OUTER JOIN
```

3.8 Aliases

- When combining a table with itself we need a way to refer to the attributes of each table individually
- We can use aliases to rename the tables in the from clause
- These are called tuple variables

```
SELECT s1.starname, s2.starname
FROM StarsIn s1, StarsIn s2
WHERE s1.title = s2.title AND s1.year = s2.year
AND s1.starname < s2.starname

--notice the use of < rather than <>
--this is because upon joining the two tables we will have repetitions
--of s1, s2 name pairs
--using the < makes sure each name pair only appears once
```

3.9 Aggregations

```
SUM
AVG
COUNT
MIN
MAX
SELECT

--We can apply all of these on a column in select to produce desired output
SELECT AVG(length)
FROM movies
WHERE studioname='Disney'
```

3.10 Group By

See rel alg for visualization

- Groups values according to listed attributes in group by

- any aggregation is applied within each group

```
SELECT studionName, AVG(length)
FROM movies
GROUP BY studioName;
```

3.11 Having

- Having is in conjunction with a group by clause
- The condition applies to each group and groups not satisfying the condition are eliminated
- Note difference between HAVING and WHERE:
- HAVING: is for conditions on tuples
- WHERE: is for conditions on groups

```
SELECT starname, SUM(length)
FROM starsin JOIN movies USING(title, year)
GROUP BY starname
HAVING MIN(starsin.year) < 2000;
```

Requirements on Having:

- Conditions may refer to attributes of relations in FROM clause
- As long as they are either:
  - A grouping attribute
  - An aggregated attribute

3.12 Views

- Virtual Views:
- Views which are not stored in DB but can be queried (kind of expensive because we have to requery every time we want something from the view)

```
--Creating a view:
CREATE VIEW <view-name> AS(
    SELECT *
    FROM ...
    ...
);

--Dropping a view:
DROP VIEW <view-name>
```

3.12.1 Updatable Views

Within the query defining the view:

- WHERE clause must not involve view itself in a subquery
- FROM only consists of one table
- SELECT → attributes in select must be enough to fill out remaining attributes with NULL or default values.

Check Option

- CHECK OPTION prevents insertion of "invisible tuples" into base table of view
- CHECK OPTION can only be used when:

1. Only one table in FROM
2. SELECT attributes include enough attributes to fill out other attributes in tuple with NULL values
3. Also have the above tuple, which is inserted into the base table, appear in the view

```
--Example:
CREATE VIEW ParamountMovies AS (
    SELECT title, year
    FROM movies
    WHERE studioname='Paramount'
    WITH CHECK OPTION
);

INSERT INTO ParamountMovies VALUES('Star Trek' 1980);
--This insertion wont work because it wont show up in the view
--check option prevents it
```

3.13 Subqueries

3.13.1 Correlated Subqueries

- Queries that are evaluated multiple times, for each assignment of a value to some term, which comes from a tuple outside the subquery.

```
--Example:
SELECT title
FROM Movies Old
WHERE year < ANY
         (SELECT year
          FROM Movies
          WHERE title = Old.title
         );
 --This query will find movie titles that appear more than once.
 --Easier to think about if we replace Old.title with a real movie name
 --i.e. 'King Kong'
```

- Scoping rules for names:
- Attributes are assumed to come from the FROM clause in the subquery
- If they are not found there, the outer query is then looked at. Continues as such

3.14 Nulls

- Tuples can have NULL as a value for one or more components
- Meaning depends on context
- Usually will be either: missing value or inapplicable (value of spouse for an unmarried person)

3.14.1 Three Valued Logic

- Comparing NULLs to values
- SQL is 3 value logic TRUE, FALSE, UNKNOWN
- When any value is compared to NULL the truth value is UNKNOWN
- A query only produces a tuple if its truth value is TRUE (not FALSE or UNKNOWN)
- We can think of three valued logic as TRUE=1, FALSE=0, UNKNOWN = 0.5
- AND = MIN, OR = MAX, NOT(X) = 1- x

- For example: TRUE AND (FALSE OR NOT (UNKNOWN)) =
- MIN(1, MAX(0, (1 - 0.5)) = MIN(1, 0.5) = 0.5

```
SELECT *
FROM movies
WHERE length <= 120 OR length > 120

--if any tuple has NULL length then they will not be in the result
--even though we would expect this query to contain all tuples
--This is because comparison with NULL results in UNKNOWN
```

3.14.2 Checking for NULLs

```
--cant use = or <>
 --use
 IS NULL
 IS NOT NULL
```

3.14.3 NULLs are Ignored in Aggregation

- Nulls never contribute to a sum, average, count, and can never be min or max of a column
- But if ther eare no-non Null values in a column then the result of aggregation is null

```
 -- Notice the difference between:

 --(1)
 SELECT COUNT(*)
 FROM movies
 WHERE studioName = 'Disney'

 --(2)
 SELECT COUNT(length)
 FROM movies
 WHERE studioname = 'Disney'

 --(1)
 --Counts all movies made by disney

 --While (2)
 --counts all disney movies where the length is known
```

3.15 DB Modifications

3.15.1 Insert

```
--To insert a single tuple into a table
INSERT INTO <relation> VALUES(<list of values>);

--We can also specify attributes in the insert statement
INSERT INTO movieexec(name, address, cert, netWorth)
VALUES('Melanie Griffith', NULL, 800, 300000);

--we do this for two main reasons
--1.) we forget the order
--2.) We dont have values for all attributes

--inserting many tuples
INSERT INTO <relation>
```

```
<query>;

--Ex:

CREATE TABLE DisneyMovies(
     name VARCHAR(25),
     year INT
);

INSERT INTO DisneyMovies (SELECT title, year FROM movie WHERE studioName='Disney');
```

3.15.2 Delete

```
DELETE FROM <relation> WHERE <condition>;


--will delete all tuples from relation
DELETE FROM <relation>
```

3.15.3 Update

```
UPDATE <relation>
SET <list of attribute assignments>
WHERE <condition on tuples>

UPDATE movies
SET length = 200
WHERE title = 'Godzilla'
```

3.16 Constraints

3.16.1 Primary Key Constraints

```
--As we've seen before, primary keys enforce uniqueness and non null of an attribute
--(or combination of attributes)

--Ex:

CREATE TABLE Movies(
     title CHAR(40) PRIMARY KEY,
     year INT,
     length INT,
     type CHAR(2)
)

CREATE TABLE Movies(
     title CHAR(40),
     year INT,
     length INT,
     type CHAR(2),
     PRIMARY KEY(title, year)
)
```

3.16.2 Not Null Constraints

```
--prevents values from being null

CREATE TABLE ABC(
     A INT NOT NULL,
     B INT NULL,  --allows null
```

```
    C INT  --also allows null by default
);

INSERT INTO ABC VALUES(1, NULL, NULL); --goes through
INSERT INTO ABC VALUES(2, 3, 4);       --goes through
INSERT INTO ABC VALUES(NULL, 5, 6);    --throws error (inserting null into NOT NULL constraint)
```

3.16.3 Unique Constraints

```
--Doesnt allow duplicate values in a column
--The only difference between this and PRIMARY KEY is that NULL can be inserted multiple times
--We also can declare UNIQUE on multiple attributes separately

--example:
CREATE TABLE AB(
    A INT UNIQUE,
    B INT
);

INSERT INTO AB VALUES(2,1);      --goes through
INSERT INTO AB VALUES(NULL, 9);  --goes through
INSERT INTO AB VALUES(NULL, 9);  --Allowed because UNIQUE allows this
INSERT INTO AB VALUES(2,7);     --Fails because of UNIQUE constraint
```

3.16.4 Constraint Names

Default Constraint names (Postgres):

- Primary Key: <table>_pkey
- Foreign Key: <curtable>_<reftable>_<attr>_pkey

```
--Renaming constraints for easier handling

CREATE TABLE ABC(
     A INT,
     B INT,
     C INT
     CONSTRAINT my_unique_constraint UNIQUE (A,B)
);
```

3.16.5 Adding and Dropping Constraints

```
 ALTER TABLE <table> ADD/DROP CONSTRAINT <constraint_name>;
ALTER TABLE AB DROP CONSTRAINT ab_a_key;

ALTER TABLE AB ADD CONSTRAINT unique_a UNIQUE (A);
```

3.16.6 Foreign Key Constraints

```
--shorthand NOTE: Table AB needs to exist before referencing it.
CREATE TABLE ABC(
    A INT REFERENCES AB(B),
    ...
)

CREATE TABLE ABC(
    A INT,
    B INT,
    ...
```

```
    FOREIGN KEY (A, B) REFERENCES AB(B, C)
)

--can also name them as constraints
CONSTRAINT fk_movies FOREIGN KEY (title, year) REFERENCES Movies(title, year)
```

Rules for Foreign Key Constraints:
Foreign Keys Must:

1. Appear as a primary key in the parent table
2. If not, be null

Note: We can force foreign keys to not be null using `NOT NULL`

3.16.7 Check Constraints

```
[CONSTRAINT <name>] CHECK(<condition>)
--(optional)

--example:
--column constraints
CREATE TABLE Emp(
    empno INT,
    ename VARCHAR(30) CHECK(ename = UPPER(ename)),  --these are column constraints
    sal INT CHECK(sal >= 500),                      --because they only constrain columns
    deptno INT CHECK(deptno BETWEEN 10 AND 100)
);

--table constraints (condition may be on any column of the table)
--ex: having a proj start date before proj end date

CREATE TABLE Project(
    pstart DATE,
    pend DATE,
    ...
    CHECK (pend > pstart)
);
```

3.17 Security and Authorization

3 main parts of DB security

- Secrecy
- Integrity
- Availability

3.17.1 Grant and Revoke

```
--Grant allows us to give certain permissions to different users

GRANT <privilege> ON <object> FROM users [WITH GRANT OPTION]
--Where privilege may be:
SELECT
INSERT
DELETE
REFERENCES
UPDATE
--Or some combination of the above
```

- Grant option allows the grantee to also grant these privileges to other users
  - Grant option is necessary on all participating tables in a view if a user wants to further grant privileges on that view to other users (even if they are the creator [superuser] for that view).
- SELECT: Allows users to select from tables (i.e. make queries)
- INSERT/INSERT(column-name): Allows users to insert into a table, or insert into a specified column
- DELETE: Allows users to delete tuples in a table
- REFERENCES: Allows users to create tables referencing attributes in other tables (powerful because it can prevent the original creator from dropping tables, or deleting tuples, in other words constrains the table creator if other users have created tables with foreign keys referencing tuples in base table).
- UPDATE: Allows users to update tuples. Note: doing something like a reassignment (+=, -=, ++, –) wont work if a user only has UPDATE (as they need SELECT to view the tuple values).

```
REVOKE privileges ON object FROM users CASCADE
```

- Revoking privileges with cascade will cause sub-users to also have privileges revoked (called abandoned privileges) if they were not given permissions from another user.
- The other option is RESTRICT; this prevents privilege revocation if any privileges will be abandoned.
- This will also cause any created views to be dropped as they are no longer accessible by their creators.

3.17.2 DAC vs MAC

- DAC: Discretionary Access Control
  - Privileges on objects
  - Used by all commercial and free DBs
  - Susceptible to trojan horse problem
  - Possible to allow creator to see secret info of users
- MAC: Mandatory Access Control
  - Used by military
  - System wide policies that cant be changed by users
  - Each DB object is assigned a security class
  - Users are given clearance to objects
- Special note: Statistical DBs
  - Have data in aggregate, to be more secure
  - still suffer from inferred data
  - users can infer secret info from the db by certain queries
  - this can be prevented by only allowing queries on a subset of the database, with conditions imposed on it.

3.17.3 Authorization Graphs

- Nodes: Users
- Arcs (edges): Privileges passed, in the form of (Grantor, Grantee, Privileges, With grant option (bool))
- When a privilege is revoked with CASCADE the edge directly between the node and the grantee node is removed (any unreachable subgraphs are then also removed).
- Also, any unreachable views are also dropped
- If references is granted to a user then when it is revoked the constraints they created will be dropped.

3.18 SQL Analytics

Extracting specifics from DATE:

```
--allows us to extract specific values from the DATE type

SELECT EXTRACT(year FROM date_of_birth)
--The argument can be of the following forms:
month
day
doy --day of year
dow --day of week (numeric)
quarter --3 month periods
week -- Week number in year

--CASE WHEN expression
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE result
END;

--Example:
SUM(CASE WHEN EXTRACT(dow from orderdate)=0 THEN 1 ELSE 0 END) AS Sun,
SUM(CASE WHEN EXTRACT(dow from orderdate)=1 THEN 1 ELSE 0 END) AS Mon
...
```

3.19 Window Functions

- Window functions allow us to perform aggregation without grouping rows into a single output row.
- This allows us to select more than just the aggregated attribute, without using group by.
- General Strategy: Mix detail and aggregate information over a window of tuples, then extract what you want with an enclosing query.

```
--consider the following illegal query
SELECT title, MIN(length)
FROM movies
WHERE studioname='Disney';

--this doesnt work because the tuples have been collapsed
--we cant go back to retrieve the studios where the name is disney

--instead we can use window functions
SELECT title, studioname, length, --this part is called the detail
     MIN(Length) OVER (PARTITION BY studioname) AS min_length
     --The partition by part produces a set of tuples
 FROM movies;


--Using order by in the window function
--Simply orders whatever we are aggregating over
--Need this to match up the running total with our query
--Also creates the window (SQL wont let us run without it)

SELECT started_at, duration_seconds, SUM(duration_seconds) OVER (ORDER BY started_at) running_total
FROM biketrips
```

LAG and LEAD:

- LAG: Allows us to move our window backward over a certain number of tuples.
- LEAD: Allows us to look forward a certain number of tuples.

```
--the following query selects months revenues from a category drop below
--those of the same month one year ago without increasing
--the next year

SELECT *
FROM (
    SELECT year, month, cat, revenue,
        LAG(revenue, 12) OVER (PARTITION BY cat ORDER BY year, month) AS prev_year_rev,
        LEAD(revenue, 12) OVER (PARTITION BY cat ORDER BY year, month) AS next_year_rev
        FROM T) X
WHERE revenue < prev_year AND next_year_rev <= revenue
ORDER BY year, month;
```

3.19.1 Ranking Results

Three main ranking operations:

- ROW_NUMBER: Returns the rank of the detail in the ordered window (each row has a distinct number and ties are broken arbitrarily)
- RANK: Numbers each column but ties are not broken, ranks produced have gaps
- DENSE_RANK: Ties not broken and no gaps.

Examples:

```
--ROW NUMBER
SELECT *
FROM (
    SELECT cat, year, month, ROUND(revenue, -3) --rounds up to the nearest thousand
    ROW_NUMBER() OVER (PARTITION BY cat ORDER BY ROUND(revenue, -3) DESC) AS rank
    FROM T) X
WHERE rank <= 10
ORDER BY cat, rank;
/*Looks Like:
    1
    2
    3
    4
    5
    ...
*/

--RANK
SELECT *
FROM (
    SELECT cat, year, month, ROUND(revenue, -3) --rounds up to the nearest thousand
    RANK() OVER (PARTITION BY cat ORDER BY ROUND(revenue, -3) DESC) AS rank
    FROM T) X
WHERE rank <= 10
ORDER BY cat, rank;

/*Looks Like:
    1
    2
```

```
    3
    4
    4
    6
*/

SELECT *
FROM (
    SELECT cat, year, month, ROUND(revenue, -3) --rounds up to the nearest thousand
    DENSE_RANK() OVER (PARTITION BY cat ORDER BY ROUND(revenue, -3) DESC) AS rank
    FROM T) X
WHERE rank <= 10
ORDER BY cat, rank;

/*Looks Like:
    1
    2
    3
    4
    4
    5
*/
```

4 Storage Mechanics

How many bytes in each:

K	Kilo	2^{10}	10^3
M	Mega	2^{20}	10^6
G	Giga	2^{30}	10^9
T	Tera	2^{40}	10^{12}
P	Peta	2^{50}	10^{15}

4.1 HDD vs SSD

- HDD: Hard disk drive
- SSD: Solid state drive
- HDD more cost effective while SSD is faster by a magnitude of 2.
- Writing an algorithm that accesses HDD memory sequentially is actually faster than an algorithm accessing SSD randomly.

4.2 Memory Vocab:

- Block: Smallest unit of read/write to/from external storage (disk) (usually 16KB in size).
- Platter: disk within HDD, usually multiple (in our model 8 platters)
- Surface: Read/write surface of platter, two per platter (top and bottom) (two per platter = 16 surfaces)
- Head: One per surface (two per platter) read and write to platters (all heads move in unison)
- Track: One specific ring on the platter of a specific radius (heads can read and write tracks of same radius concurrently. (after one track is full the one on the next surface is used next).
- Cylinder: All tracks at a common radius. (we only fill the next track when the cylinder of the current track is full)
- Sectors: Tracks are divided into sectors of equal size
- Gap: The sectors are divided by gaps.

4.3 The Numbers

Theoretical Model: Megatron 747

- Sectors per block: block = 16kb, sector = 4kb, therefore n*sector = block = 4sector. 4 sectors per block.
- 8 platters = 16 surfaces = 2^4
- $2^8 = 256$ sectors per track
- $2^{16} = 65536$ cylinders/tracks per surface
- $2^{12} = 4096$ bytes per sector approx 4kb
- Multiplying all together gives space of megatron = $2^{40} = 1T$
- gaps make up about 10% of a track
- blocks are made up of 4 sectors

4.3.1 Time to Read and Write a block of 16k

- Seek time: Heads need to be positioned on the right track
- Rotational delay: wait for right sector to come under head
- Transfer time: time to actually read data

4.3.2 Megatron Timing

- 1 ms = heads to stop/start moving
- 1 ms = traverse 4000 cylinders
- 7200 RPM = 8.33 ms per rotation
- Time for innermost track to outermost track:
- $\frac{65536}{4000} = 17.38ms$

4.3.3 Minimum Time to Read a Block

Suppose:

- Seek time = 0
- Rotational Delay = 0
- Transfer time $\neq 0$
- 1 block = 4 sectors, 3 gaps
- 265 sectors per track (same as gaps)

Since we know gaps are 10% of the track, the total amount they take up in degress is 36

$$36^o = 360^o \times 0.1$$

The amount of degrees one gap takes is the number of degrees of the total gaps divided by the number of gaps

$$\frac{36^o}{256} = 0.14^o$$

The total amount of degrees that sectors take up is then the total degrees minus the total gaps:

$$360 - 36 = 324^o$$

Therefore the degrees for one sector is

$$\frac{324}{256} = 1.256^o$$

Recall a block has 4 sectors and 3 gaps, therefore the total degrees for a block is:

$$4 * 1.265^o + 3 * 0.14^o = 5.48^o$$

Transfer time is then the portion of the track covered by one block (have to move head across whole block to get all of that information)

$$\frac{5.48}{360} * 8.33ms = 0.13ms$$

Where 8.33 is the rotation speed

4.3.4 Max Time to Read a Block

$$0.13ms + 17.38ms + 8.33ms = 25.84ms$$

17.38 is the time for innermost track to outermost track
0.13ms is the transfer time from above
8.33ms rotation delay (might have to do a full rotation if we are on bit two of a block)

4.3.5 Average Time to Read a Block

0.13ms Transfer time
4.17ms rotation delay(from $\frac{8.33}{2}$)
6.46ms Seek time (need to travel 1/3 cylinders on average [see slides for proof])

$$0.13ms + 4.17ms + 6.46ms = 10.76ms \approx 11ms$$

4.3.6 Times to Know:

Min time to Read/Write a block: $0.13ms$

Max time to read/write a block: $25.84ms$

Avg Time to read/write a block: $11ms$

4.3.7 Writing and Modifying Blocks

Similar to RAM model but we count one operation as one I/O which is a block read or write.
Block Modification Requires:

- Read the block into main memory
- Modify the block there
- Write the block back to the disk

4.4 Sorting in External Storage

- Algorithms are different in DBMS model, rather than RAM model (everything fits in main memory so basic operations cost the same) We assume that data does not fit in main memory.
- Algorithms that do few random disk accesses are the fastest in this case
- Consider classic (two-way) merge sort, the number of passes we have to do on the data is $\log_2(n)$, therefore if we have 10 million records to sort, that would be 23 passes.

Instead we use two-phase multiway merge sort, does a small number of passes, where every record is read into main memory once and written out to disk once

4.4.1 2PMMS

Two-phase multiway merge sort:

Phase 1:

- Fill main memory with records
- Sort using favorite main memory sort
- Write sorted sublist to disk
- Repeat until all records have been put into one of the sorted lists.
- We sort sections of the disk into sublists using main memory because it is faster.

Phase 2:

- We take each sorted sublist and put it into input buffers in the main memory
- We make a competition between the first (smallest) element in sorted sublists, take the smallest one and put it into an output buffer (we can do this using a PQ in main memory)
- We also have the output buffer which is in main memory.
- We have certain outcomes:
    1. If an input buffer is emptied, we refill it using data from same sorted sublist (if it exists).
    2. If the output buffer becomes full, we empty it to memory.

In total for phase 1 we have 1 read and 1 write (read from disk, write back to disk, for each block)
For phase 2 we have another read and write to disk for each block (1 read from sorted sublists on disk, write back to disk)
In total, we have 2 reads and 2 writes = 4 I/Os per block.

4.4.2 Analysis of 2PMMS with 16K Blocks

- 10,000,000 tuples of 160 bytes = 1.6 GB file.
- Stored on megatron 747 disk, with blocks of 16K each holding 100 tuples
- 100MB of main memory
- Number of blocks that can fit in 100MB = 6400 blocks (1/16th of the total file)

Using the above stats we can analyze phase 1 as follows:
6400 of the 100,000 blocks fill main memory on one pass, therefore main memory needs to be refilled 16 times. Therefore to read and write 100,000 blocks there are 200,000 disk I/Os with an average time of 11 ms

$$200,000 * 11ms = 2200\,\text{sec}\,/60 \approx 37\,\text{min}$$

Phase 2 has a similar analysis:

- Each block as a sorted sublist is read from the disk once. Total number of reads is 100,000
- Each record is placed on an output block once and therefore written to disk once
- The second phase also takes 37 mins

In total the overall operation takes 74 minutes.

4.4.3 Optimization by Scaling Blocks

We as the programmer have control over the block size.

- Doubling the block size halves the number of disk I/Os (we count operation on a whole block as one I/O)
- Only increase is in transfer time = 0.13*2 = 0.26ms (small price to pay related to rotational delay and seek time.)
- Therefore we have 32K blocks
- We only have to read and write 200,000 now (in total)
- 200,000*11ms = 2200 seconds = 37 minutes.
- If we 32x the block size to 512K
- Transfer time is 0.13*32 = 4.16 ms
- Total = 10.63 + 4.16 = 15ms
- Table will be 100,000/32 = 3125 blocks instead of 100,000
- Total time = 3125 * 4 * 15ms = 187500ms = 3.12mins
- 23X speedup

There are benefits to limiting block size:

1. Small tables occupy a fraction of a block so large blocks waste space on the disk
2. Larger the blocks are the fewer records we can sort by 2PMMS.

4.4.4 Number of Sortable Records

- Block size is B bytes
- Main memory is M bytes
- record (equiv to tuples) is R bytes
- Number of buffers = M/B - 1 blocks (we assume a buffer is one block) The -1 is to account for the output buffer.
- How many sorted sublists can we merge?
- (M/B) - 1 (at once, we need a buffer for each)
- What is the total number of records we can sort?
- Each time we fill memory with M/R records
- (M/R)*[(M/B)-1] = M^2/RB
- Bigger block size = fewer records we can sort.
- We can also create sorted sublists of M^2/RB records and merge $(M/B) - 1$ in a third pass
- The third pass lets us sort:
- $[(M/B) - 1] * [M^2/RB] \approx M^3/RB^2$

5 BTree Indexes

- Allow for the fast lookup of attributes to help with queries
- An index is a data structure that:
  - Takes the value of one or more attributes and finds the records with that value quickly.
  - Only has to look at a small fraction of possible records.

They help us answer queries like:

```
SELECT *
FROM R
WHERE a=10;
```

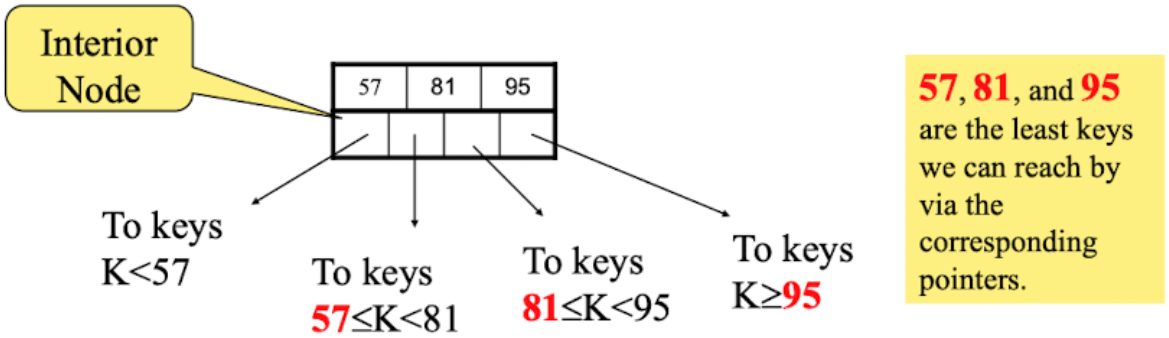
5.1 Types of BTrees

1. Clustered
2. Unclustered

Node stats: 1 block each, therefore 1 I/O. Pointers points to an address which is about 20 bytes.

5.2 Unclustered BTrees

- Leaf Nodes have:
  1. min $= \lfloor (n+1)/2 \rfloor$
  2. Pointers directly to records and one pointer to the next leaf in the sequence
  3. The keys of the leaf notes correspond to the key of the record
  4. Pointers also facilitate range queries, we simply collect all of the pointers
- Interior Nodes have:
  1. min $= \lceil (n+1)/2 \rceil$
  2. Pointers to subtrees with keys in certain ranges
  3. The leftmost pointer is associated with values strictly less than the smallest key (points us toward the right subtree)
  4. The middle pointers are associated with a range of values between their right and left associated keys (weak inequality)
  5. Rightmost pointer is greater than or equal to the associated key value.



5.3 Clustered BTrees

- Leafs have data stored in indices rather than following a pointer to the record. Still have pointer to next leaf in sequence.
- Interior nodes stay the same shape.
- Can only have one clustered index per table.
- Clustered indices are always sorted, range queries are also facilitated.

5.4 Lookup

- Keys always indicate smallest values found in subtrees while following that pointer
- We can follow the pointer to the right of a certain key to find it in its subtree
- Recursive procedure
  - Base Case: If we are at the leaf we look among the keys there and if the ith key is K then the ith pointer will take us to the desired record (unclustered)
  - Recurisve Step: If we are at an interior node with keys $K_1 \dots K_n$ we look at the pointer ranges to decide which pointer to follow next.

5.5 Insertion

- To insert, we first do a lookup to find an insertion location
- if the node is full we have to split the node into two nodes
- The keys and pointers are redistributed so that each node is about half full
- If the parent node is also full we have to also split it (to add new parent to previously split nodes)
- Any new keys to represent smallest key in a given subtree are updated in nodes.

5.6 Deletion

- If the node that we've deleted from is less than half full we can borrow a key from siblings to fill it
- If we have no siblings to borrow from we place remaining non deleted keys back into other sibling node and remove whole node.
- In practice we will usually not delete things, and if we do decide to have deletions we will use tombstones rather than actually deleting and restructuring.

5.7 Structure with Blocks

- Degree n (number of keys in a node) means that all nodes have space for
  1. n search keys
  2. $n + 1$ pointers
- Since a node=block let:
  1. Block size = 16,384 bytes. 16KB
  2. key = 20 bytes
  3. pointer = 20 bytes

Solving for n:

$$20n + 20(n + 1) \leq 16,384$$

$$n \leq 409$$

This means that each node (which is a block) will have 409 keys and 410 pointers.

- Typically nodes will only be about 70% full this means there will be about 300 keys
- Therefore at level 3 there are 300^2 nodes (because at level 2 there are 300 nodes each which have 300 pointers) and each of these has 300 pointers to records $300^3 \approx 27,000,000$.
- If each record is 1024 bytes, this means we can index a file of size 27 GB
- This also means that to reach any index requires only 3 disk I/Os. Since the root is kept in main memory (cached). 33ms to do a look up.

5.8 Pointer Intersection

- We can index certain attributes in order to get pointers to certain records, but not follow them right away
- do the same on another attribute
- intersecting these two pointer pools gives us a much smaller number of pointers to search through.

6 Query Evaluation

- Rearrange query so that it runs faster.
- We expand SQL queries into relational algebra.
- A query evaluation plan is represented as tree of relational operators WITH attached algorithms (identifying algorithm to use at each node)
- Initial trees can be transformed into better trees by finding good evaluation plans (called query optimization)

6.1 Example Data:

Tables:

Employees(SIN INT, ename VARCHAR(20), rating INT, addresss VARCHAR(90));
Maintenances(SIN INT, planeID INT, day DATE, desc CHAR(120))

Maintenances Stats (big table):

- A tuple is 160 bytes
- A block can hold 100 tuples (16K block)
- We have 1000 blocks of such tuples

Employees Stats:

- A tuple is 130 bytes.
- A block can hold 120 tuples.
- We have 50 blocks of such tuples.

6.2 Algorithms for Selection

- Notes on runtimes for selection:
- On average we have to scan half of an allocated amount of blocks in order to find the value for a unique attribute (i.e. PK). Otherwise, we have to scan all blocks if the attribute is not unique.
- If we are performing an exact selection $\sigma_{R.attr} = value(R)$ (assuming R.attr is unique)
  - If there is **no index** on R.attr just scan R (there is no B-Tree already). (On average half the number of blocks R is occupying will need to be scanned (recall R is the table))
  - If there already is an **index** on R.attr use the index. Typically this will cost 3 disk accesses, assuming non-clustering B-Tree with 3 levels where the root is in main memory.

- If we have a range query $\sigma_{R.attr} < value(R)$
  - If we have a **non-clustering index** it might be better to scan the table and ignore the index. Because with a B-Tree we will not only have to gather the pointers but also follow them, this could look more like random accesses to memory. Whereas a scan is sequential (we have better odds).
  - If we have a **clustering index** we use it.

6.2.1 When to use Indexes for Unordered Attributes:

$B(R)$ = Number of blocks for table R
$T(R)$ = Number of tuples in R
$V(R, a)$ = number of unique values of a in R
$\frac{T(R)}{V(R,a)}$ = Nuber of tuples containing a particular value
If $3 * \frac{T(R)}{V(R,a} \leq B(R)$ we use indexes, otherwise we scan blocks.

Example

$$T(R) = 100,000$$
$$B(R) = 1000$$
$$V(R, a) = 100$$
$$3 * \frac{100,000}{100} \leq 1000$$
$$3000 \not\leq 1000$$

Therefore we don't use indexes.

6.3 Algorithms for Projection:

- Given a projection we simply scan the relation and drop certain attributes we dont need (of each tuple).

6.4 Algorithms for Joins

6.4.1 Index Nested Loops Join

- Suppose employees has unclustered index (B-Tree) on the SIN column
- To natural join maintenance and employees
- Index nested loop joins scan maintenances and for each tuple probe employees for matching tuples
- 100,000 Maintenance tuples we try to access the corresponding employee with 3 I/Os via the index
- So 100,000* 3 = 300,000 I/Os (this is a lot)

6.4.2 Merge Join

- Sorts both tables on the join column (2PMMS)
- Once sorted scans them to find matches
- Sorting maintenance's $2 * 2 * 1000 = 4000 I/Os$
- Sorting employees $= 2 * 2 * 50 = 200 I/Os$
- (because we count operations on a block as one I/O, standard cost for 2PMMS is 4 I/Os (2 for phase 1, 2 for phase 2))
- Merge join requires an additional scan of both tables
- The total cost is $400 + 200 + 1000 + 50 = 5250 I/Os$
- We can throw out all non matching tuples right away

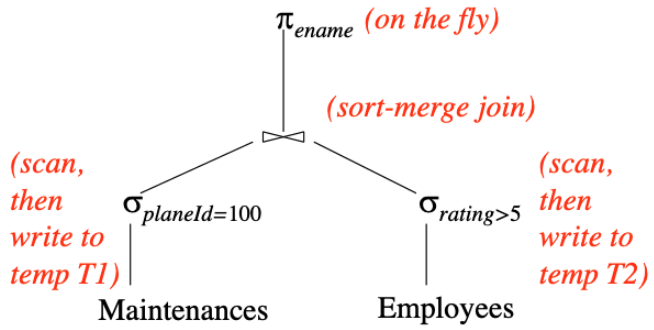
- this is a giant speed up compared to index nested loop joins because rather than comparing against each tuple we just have to look to see if there is a matching one in the next block.
- Why bother with nested loop joins then?
- They are incremental → if we take a small subset of tuples then we avoid computing the full join of maintenances and employees
- If we want a join with say two tuples, then we just have to probe employess twice and we are done (rather than taking both tables, sorting them, merging, and then joining).
- This means we have to look at query as a whole rather than just optimizing the join operation

6.5 Query Optimization

- Query evaluation plans consist of RA tree with indication of each algorithm used at each step.

6.5.1 Pushing Selections

- We can push selections down the plan to before joins in order to have to join less tuples.



Analysis:

- We have to first calculate the cost of applying selection to maintenance's
- Need to also account for cost of writing to temp table T1
- Since maintenance is 1000 blocks and (assume planeID is not indexed or unique) we do a full scan for selection and write to T1
- To estimate the size of T1 (to count writing) we reason as follows:
  - If we know that there are 100 planes we can assume that maintenance's are spread out uniformly across all planes and estimate the number of blocks in T1 to be $1000/100 = 10$

Total cost: 1000+10 = 1010 I/Os
On the other hand for Employees:

- Scanning employees (50 blocks)
- Writing to T2 where we assume that ratings are in range [1, 10] and uniformely distributed over all employees = $50/2 = 25$
- Total cost = 75 I/Os

Then we sort merge join for T1 and T2

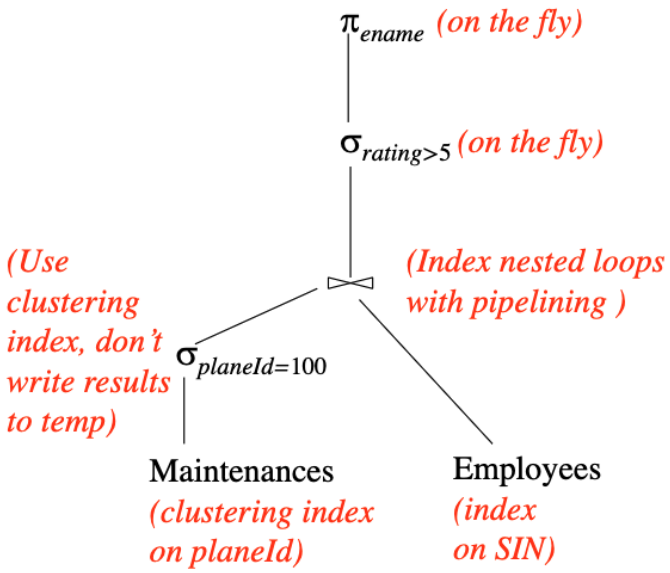
- $2 * 2 * 10 + 2 * 2 * 25 = 140 I/Os$ 2PMMS
- For the join we need to scan = 10+25=35 I/Os
- Final projection is done on the fly and by convention we ignore the cost of writing the final result.

The total cost of the plan is: (1010+75) + (40+100+35) = 1260 I/Os.

6.5.2 Pushing Projections

- Only the SIN attr of T1 and the SIN and ename attrs of T2 are required.
- When we scan maintenances and employees to do selectoins we can also eliminate unwanted columns
- This reduces the sizes of temporary tables T1 and T2
- the reduction is substantial because we now only have integer field
- This means we have all of T1 in three blocks in MM and can be joined with nested loops to T2 in a single pass.

6.5.3 Plan using Indexes



- Suppose that planeID has a clustering index
- and Employees has a non-clustering index

Analysis:

- If we asssume there are 100 planes, uniformely distributed over maintenances we can estimate the number of selected tuples to be $\frac{100{,}000}{100} = 1000$
- Since the index on planeId is clustering these 1000 tuples appear consecutively and therefore the cost is: 10 blocks I/Os + 2 I/Os (finding beginning of blocks via index)
- For each selected tuple we retrieve matching employee tuples using the index on the SIN field.
- The join field SIN is a key for employees therefore at most one employee matches a given tuple in maintenances.
- The cost is 3 I/Os for 1000 maintenances
- Therefore 3000 I/Os overall
- In total = 3000+12 = 3012 I/Os
- Note: We didnt push the selection ahead of the join because we would have had to then scan employees (rating has no index), we lose the SIN index.

7 Concurrency

7.1 Transactions

- Transactions control concurrent behavior
- A process that reads or modifies the DB is called a transaction.
- Commit causes a transaction to complete
- ROLLBACK causes a transaction to end but by aborting
- Failures can cause rollbacks
- Production environments can have many users and therefore have many transactions submitted at the same time.
- We cant use first come first serve because some transactions are large and time consuming
- DBMS may run different transactions at the same time (in parallel (pseuodparallel))

7.2 Concurrent Transactions

- Several transactions at the same time can interact and turn a consistent state into an inconsistent state
- an inconsistent state is when the table no longer satisfies the constraints

7.3 ACID Transactions

Key Properties of a transaction (not sufficient to prevent anomalies)

- Atomic: Whole transaction or none is done
- Consistent: Database consistency is preserved
- Isolated: Appears to the user as if only their process executes (only using one system)
- Durable: Effects of a process survive a crash

7.4 Transactions and Schedules

- A schedule is a list of actions from a set of transactions
- there can be **anomalies** that happen with transactions
    1. Reading uncommitted data:
    2. Unrepeatable reads
    3. Overwriting uncommitted data

7.4.1 Reading Uncommitted Data

Put simply: One transaction is in the middle of altering data and the other reads the data in the middle of it being altered.

- Suppose we perform a read and a write in T1 and then another read and write in T2, before doing a final read and write in T1 again, then committing
- Example: T1 transfers $100 from A to B
- T2: Increments both A and B by 1% for interest payments
- But T1 transfers balance out of A, and bank pays interest on balance before transfer is in B, therefore bank isnt paying interest on the 100 being transferred.

7.4.2 Unrepeatable Reads

Put simply: One transaction does a read, and another subsequent read, but after the other transaction has already altered it.

- Suppose A is the number of copies available for a book
- If transactions T1 and T2 both place an order for the book they both have to first check for the availability for the book through a read
- Suppose that T1 checks for the availability for the book
- T2 also checks for the availability concurrently, since the book is available it makes the order
- T1 at this point also attempts to make the order but the book is now gone and it cannot.

7.4.3 Overwriting Uncommitted Data

A change and a subsequent change is out of order.

- Suppose Larry and Harry need to have equal salaries
- We want to first update larry and harrys salary to 2000 and then to 2100
- Suppose we do the 2000 in T1 and 2100 in T2
- Updating larry to 2000 in T1 we switch over to T2 and update harry
- We then switch over to T1 again where harry gets updated to 2000
- Finally, larry gets updated to 2100 in T2
- Conditions are no longer satisfied

7.5 Transaction Terminology

- Transaction: Sequence of r and w actions on database elements
- A schedule: sequence of read/write actions performed by a collection of transactions
- Serial Schedule: All actions for each transaction are consecutive
- Serializable Schedule: A schedule whose effect is equivalent to that of some serial schedule.

7.6 Conflict Serializable Schedules

Conflict serializable schedules can be converted into a serializable schedule with the same effect by a series of non-conflicting swaps of adjacent elements. (Sufficient condition conflict serializable $\rightarrow$ serializable, just because a schedule is serializable doesn't mean it is conflict serializable)

- We can perform non conflicting swaps
- There is a conflict when one of the following hold (between two different transactions):
  1. A read and a write of the same X,
  2. Two writes of the same X
- Cannot swap the above
- But we can swap any other read/writes

7.7 Serializibility Graphs

- Non-swappable pairs of actions represent potiential conflicts between transactions
- The nodes in the graph are the transactions T1, T2, ...
- Arcs (digraph): There is an arc from T_i to T_j if they have conflicting access to the same database element X and T_i is first, we write $T_i < T_j$
- If there is a cycle in the graph then there is no serial schedule which is conflict equivalent to S
- If there is no cycle: then any topological order of the graph suggests a conflict equivalent schedule.

Proof:

- IH: If the precedence graph is acyclic we can logically swap actions to form a serial schedule
- Basis: A single transaction is already serial
- Given that the precedence graph is acyclic there exists T_i in S such that there is no T_j in S
- We can swap all action sof T_i to the front of S (graph)
- (Actions of T_i)(ACTIONS OF THE OTHER N-1 TRANSACTIONS)
- THE TAIL IS A PRECEDENCE GRAPH THAT IS THE SAME AS THE ORIGinal without the T_i i.e. it has n-1 nodes
- Repeat for the tail

Inductively performing action at the "tail" of the topsorted graph, we know that a single transaction is serial?

- Schedulers take requests from transactions for reads and writes and decides if it is OK to allow them to operate on the DB or defer them
- A scheduler forwards requests if if cannot result in a violation of conflict serializability.

7.8 Locks

7.8.1 Lock Actions

- T_i requests a lock on X from the scheduler
- The scheduler can either grant the lock to T_i or make T_i wait for the lock
- If T_i is granted the lock it should eventually unlock

7.8.2 Validity of Locks

- Consistency of Transactions
  - Read or write X only when holding a lock on X
  - If T_i locks X T_i must eventually unlock X
- Legality of schedules
  - Two transactions may not lock the same element X without one having first released the lock

Legal schedule doesnt mean serializability We have to use two phase locking

7.8.3 Two Phase Locking

- In every transaction, every lock request precedes every unlock request
- Guarantees conflict serializability.

7.8.4 What should be locked?

- We are able to lock tuples and tables
- We can end up with the phantom problem if we lock only tuples within a transaction. Where within a transaction two reads to the same table can end up with different results (if someone else inserted into the table during our transaction)
- To prevent this we introduce transaction support

7.8.5 Transaction Support

```
SET TRANSACTION ISOLATION LEVEL <X>
where x is:

SERIALIZABLE --default
REPEATABLE READ
READ COMMITTED
READ UNCOMMITED
```

- Serializable: obtains locks before reading and writing objects including locks on sets (table) of object that it requires to be unchangeable and holds them until the end according to 2PL. Doesnt suffer from any anomalies
- Repeatable read: Transaction that sets the same locks as a serializable transaction except that it doesnt lock sets of objects but only individual objects (therefore can suffer from phantom problem)
- Repeatable Read: Sets the same locks as serializable but it doesnt lock sets of objects only individual objects (doesnt lock tables therefore suffers from the phantom problem)
- Read Committed: Obtains locks before writing objects and keeps them until the end, whereas for writing it obtains locks before reading and immediately releases them. May suffer from unrepeatable read or phantom read
- Read Uncommited: Doesnt obtain any lock, but we are only allowed to do read only transactions as such may suffer from all anomalies

7.9 Lock Types

7.9.1 Shared/Exlusive Locks

Simple locks doent allow two users to read a DB element at the same time
But having multiple reads is not a problem
To solve this they introduced shared locks $sl_i(x)$ and exclusive locks $xl_i(X)$j
If we have a shared lock we may impose another shared lock on it, we may never impose an exclusive lock on a shared lock or a shared lock on an exclusive lock.

- Instead of taking an exclusive lock immediately a transaction can take a shared lock when it is reading X and then upgrade to an exclusive lock so that it can write X (prevent unrepeatable reads)
- Upgrading locks allows form more concurrent operations
- the problem is that this also introduces the possibility for deadlocks:
- Suppose T1 and T2 each read X and later writes X
- they both request a shared lock at first and then try to upgrade to an exclusive lock which causes a deadlock

7.9.2 Update Locks

- Only update locks can be upgraded to exclusive locks (gets rid of shared locks)
- a transaction that will read and write an element A requests an update lock and then asks for an exclusive lock on A

- read actions are permitted when there is either a shared or an update lock
- Update locks can be granted while there is a shared lock but a scheduler will not grand a shared lock when there is an update lock.