Contents

1 Database and DBMS

Introduction

Definition of a database and DBMS in Professor Notes.

2 Relational Data Model

Relational Data Model

Definition 1 *Relational data model is an approach to organizing collections of data*

- Relation
  - Relational Table $\longrightarrow$ **Name + Schema**
    - Schema: List of attribute name + attribute type pairs
- Relational Database $\longrightarrow$ **Collection of Relations tables**
- **Table Instance**: set of records with instantiated values of the attributes
  - Finite
  - Records, rows, tuples

One unit of data is called a **datum**.

Object, entity, event: description of one object, entity, event

- **Records** consist of attributes or fields (rows in the table).
- **Attributes** is a named container for a value of a specific type.

Database Table Constraint

Definition 2 *Limitations of table instances*

- **Candidate Key**: set or lists of attributes that uniquely define a record in a table, **minimal such set of attributes**, made up of multiple attributes sometimes.
  - **Every attribute is necessary.**

Examples

CSC 365 Example

Course Object:

- Prefix: CSC $\longrightarrow$ **String**
- Course #: 365 $\longrightarrow$ **Integer**
- Name: Introduction to Database Systems $\longrightarrow$ **String**
- Description: Basic Principles, ... $\longrightarrow$ **String**

- Units: 4 $\longrightarrow$ **Integer**

Department Object:

- Name: Computer Science and Software Engineering
- Abbreviation: CSSE
- Building: 14
- Room: 245
- College: CENG

Stringing these objects together based on relationship would make a **network model**.

Schema Example

```
Course(Prefix String, Course# Integer, Name String, Description
String, Units Integer)
```

Prefix	Course#	Name	Description	Units
CSC	365	Introduction to Database Systems	Basic Principles, ...	4
CSC	357	Systems Programming	...	4

```
Department(Name, College, Building, Room): Department would also have a table as well.
```

3 RDM Cont.

Relational Data Model

What makes a record unique?

- **Superkey**: any set of attributes that uniquely defines a record in a table
- **Primary Key**: candidate key chosen by you

4 SQL DDL and DML

MySQL Access

1. Server Address = host: **mysql.labthreesixfive.com**
2. Port: 3306
3. username
4. password

MySQL Database

- Namespace
- Collection of Tables
- Set of Permissions

Case Sensitivity

Case Sensitive

- Table Names
- Database Names

Not Case Sensitive

- Attribute Names
- SQL Keywords

Types

- **Numeric Types**
  - **Integer Types**
    - TINYINT
    - SMALLINT
    - MEDIUMINT
    - **INT**
    - BIGINT
  - **Floating Point Types**
    - **FLOAT**

        - **DOUBLE(P, D)**
        - **DECIMAL**

- **String Types**
    - **Character Types**
        - **CHAR(N) $\longrightarrow$ Fixed Length**
        - **VARCHAR(N) $\longrightarrow$ Variable Length**
        - TINYTEXT
        - **TEXT** $\longrightarrow$ for storing large amounts of text
        - MEDIUMTEXT
        - LONGTEXT
- **Date and Time Types**
    - **Date Types**
        - DATE
        - DATETIME
        - TIMESTAMP
        - TIME
        - YEAR

Data Definition Language (DDL)

Commands from DDL act upon the schema

- CREATE TABLE
- DROP TABLE
- ALTER TABLE

Define a Relational Table

Aspects needed to define a table:

- Table Name
- Attributes: Name + Type
- Constraints

```
CREATE TABLE <table_name> (
    <attribute_name> <sql_type> [<single_line_constraints>],
    ...,
    <attribute_name> <sql_type> [<single_line_constraints>] [,
    <constraints>[,
    <constraints>]
]);
```

Data Manipulation Language (DML)

Commands from DML act upon the instance.

- INSERT
- DELETE
- UPDATE

Inserting Data

```
INSERT INTO <table_name>(<attribute_name>, ...)
    VALUES (<value>, ...);
```

Supply values in order of attribute declarations in CREATE TABLE statement. Can omit the attribute names if values supplied are in the same order. If need to omit a value then omit that attribute name as well.

More on Constraints

- **[NOT] NULL** - attribute cannot be null
- **UNIQUE**
- **PRIMARY KEY**
- **FOREIGN KEY**
- **DEFAULT** <**exp**> - default value for attribute
- **AUTO_INCREMENT** - means that the attribute is an integer and is automatically incremented

Lab 2

MySQL Server

- LabThreeSixFive.com
- mysql command line client
- IDE (DatGrip)
- mysql connectivity from Python

Lab 2 uses Create Table, Drop Table, and Insert.

Code from Lab

```
show tables

CREATE TABLE Departments (

    DeptId INT PRIMARY KEY,
    Abbr VARCHAR(20) UNIQUE, -- UNIQUE makes candidate key
    Name VARCHAR(128) UNIQUE,
    College CHAR(10),
    Building INT,
    Room CHAR(6),
    -- set multiple candidate keys at the bottom
    UNIQUE(Building, Room),
    -- foreign key always a separate line statement:
    -- FOREIGN KEY(College) REFERENCES colleges(abbr)

);

describe colleges;
SELECT * FROM colleges;

show CREATE TABLE colleges;

show CREATE TABLE Departments;

INSERT INTO Departments
    VALUES(1, 'CSSE', 'Computer Science and Software Engineering', 'CENG', 14, '245');

INSERT INTO Departments(DeptId, Abbr, Name, College, Building, Room)
    VALUES(1, 'CSSE', 'Computer Science and Software Engineering', 'CENG', 14, '245');
```

5 DDL and DML Continued

DML

Updating Data

```
UPDATE <table_name>
    SET <attribute_name> = <value>
    WHERE <condition>;
```

Example

```
UPDATE colleges
    SET abbr = 'COSAM'
    WHERE abbr = 'COASM'
```

WHERE clause is a filter that determines which rows are updated.

Deleting Data

```
DELETE FROM <table_name> -- just this is a valid command to delete all rows
    WHERE <condition>;
```

DDL

Altering Tables

```
ALTER TABLE <table_name>
    <Command> <parameters>;
```

Commands

- ADD - add a column/attribute/key
- DROP
- MODIFY
- RENAME

Parameters

- COLUMN
- CONSTRAINT

- FOREIGN KEY
- PRIMARY KEY
- UNIQUE

Adding an attribute, dropping/adding a constraint, renaming a table, disable/enabling keys, and modifying attributes examples are in this professor notes: 4-SQLDDLDML.pdf

6 DML/DDL Cont., WHERE Clause, and MySQL Conn.

Announcements

Running Scripts for Lab 2

Can run from command line using mysql command or using mysql client. For running using mysql command need to specify the database if not using default database.

```
source script.sql
```

DML/DDL

Data Manipulation works on instance and Data Definition works on schema.

Altering a Table

Modifying the schema. ALTER examples in class.

For CREATE TABLE, you can name constraints:

```
CREATE TABLE Example (
    Id int PRIMARY KEY,
    X INT,
    Y INT,
    CONSTRAINT Point UNIQUE (X, Y)
);
```

Updating and Deleting from Table: WHERE Clause

Ex. Deleting in Table

```
DELETE FROM test02
    WHERE b > c
```

This deletes rows where b is greater than c.

Ex. Deleting with Scope

```
DELETE FROM test02
    FOR EACH ROW in test01
    DO
        DELETE FROM test02 -- delete(row, condition)
        WHERE b > c
```

SQL Boolean Expressions

- 0, 1
- Builtin: IN(...) $\longrightarrow$ returns bool
- $<$ *Expression* $>$ $<$ *op*1 $>$ $<$ *op*2 $>$ // can also use IN or LIKE
- $<$ *Expression* $>$ AND $<$ *Expression* $>$
- $<$ *Expression* $>$ OR $<$ *Expression* $>$
- NOT $<$ *Expression* $>$

MySQL Connectivity

Breifly went over the Python examples on Course webpage that connect to MySQL server.

7 Python Connectivity and Relational Algebra

Python MySQL Connectivity

Relational Database is sitting on a server. It is listening for connections, and our program is a client that connects to the server via the port. Essentially, there is a pipe and a exchange of messages that is happening. Generates a connection object that stores info about how to properly access the database.

Package

```
import mysql.connector
```

Connection

5 Things Needed: Host, Port, Username, Password, Database (sometimes not necessarily)

These get passed to mysql.connector.connect() function. This returns a connection object. is_connected() returns a cursor object.

Cursor object that is returned from the connection object. Cursor object is used to execute queries.

Relational Algebra

Relational $\longrightarrow$ Database Model $\longrightarrow$ Relational Model. Algebra: set of elements & operations on elements

Relational Algebra is operations on relational tables.

Boolean Algebra introduces operations on truth values

- T, F
- $\tilde{}$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$

Notation

Upper case letters like R, S, T, R_1, S_7, ... are relational table names. Letters from first half of alphabet like A, B, C, ... are attributes names. R(A_1,..., A_n) are to represent schema. t, s, r $\in$ R are tuples. a_1, a_2, ... are values. Ex. t = (a_1,..., a_n) and it could be referred to as t.A_1 = a_1.

Operations

Binary

Unary

- Selection σ - filter rows

- Projection π - filter columns

Selection Operation

- $\sigma_{<selectioncondition>}(R)$ - returns rows that satisfy condition
- Selection Condition denoted by C
- Ex. C = $A_2 =' Riley' \wedge A_3 =' Hicks'$
- **Formal Notation:** $\sigma_C(R) = \{t \in R | tsatisfiesC\}$

Projection Operation

- $\pi_{<attributelist>}(R)$ - returns columns that are in attribute list
- F is the projection list which is a list of attributes
- Ex. F = $(B_1, ..., B_m)$ where $B_i \in A_1, ..., A_n$
- **Formal Notation:** $\pi_F(R) = \{t' | \exists, t \in R, s.t. \forall B \in F, t'.B = t.B\}$
- **Projection squeezes out duplicates**

8 Relational Algebra

Relational Algebra

R, S, T are relational tables and these are sets. Set Operations include Union, Intersection, Difference, Symmetric Difference...

Example:

Table 1: R

A	B	C
1	2	a
2	4	b
3	1	d
4	4	d
4	5	a

Table 2: S

B	D	E
2	a	1
3	a	2
7	b	3
5	b	1
1	c	2

Selection Review

Duplicate Elimination: $\sigma_{A=4 \vee B=4}(R)$

A	B	C
2	4	b
4	4	d
4	5	a

Projection Review

• Subset and Columns: $\pi_{B,D}(R)$. The schema here is $R(B, D)$.

• Can make composition of operations: $\pi_{B,E}(\sigma_{D=b}(R))$ because an result of an operation is a relational table. Result:

B	E
7	3
5	1

• Can also reorder columns: $\pi_{EBD}(R)$

• Duplicate Columns: $\pi_{ABA}(R)$. Issue with this is that there are 2 columns with the same name. Disambiguate by renaming: $\pi_{A_1 B A_2}(R)$

• Can also introduce new columns: $\pi_{A,B,2 \cdot A}(R)$

Cartesian Product

• $R \times S$ is the cartesian product of R and S. Result is a table with all possible combinations of rows from R and S.

• Notation: $R \times S = \{(t, t`)|t \in R, t \in S\}$

Ex. $\sigma_{A<3}(R) \times \sigma_{B<3}(S)$

A	B	C	B	D	E
1	2	a	2	a	1
1	2	a	3	a	2
1	2	a	5	b	1
2	4	b	2	a	1
2	4	b	3	a	2
2	4	b	5	b	1

Join

Table 3: R

Id	Customer	C
1	2	a
2	4	b
3	1	d
4	4	d
4	5	a

Table 4: S

Id	Name	E
2	a	1
3	a	2
7	b	3
5	b	1
1	c	2

• Who purchased Receipt 3 $\longrightarrow$ Find a person with Id 1: $\pi_{Name}(\pi_{R.Cust=S.Id}(\sigma_{Id=3}(R)) \times S)$

• Notation: $R \bowtie_C S = \sigma_{R.x operation S.Y}(R \times S)$

9 Joins

Announcements

- Double-sided cheat sheet allowed for Midterm.
- Homework assigned on Thursday as prep for Midterm.

Example

Table 5: T

Id	Name	Course	Grade
1	Joe	CSC 365	A
2	Mary	CSC 365	A
3	Lisa	CSC 101	B
4	Luis	STAT 301	C
5	Steve	STAT 301	A
1	Joe	CSC 357	B
2	Mary	CSC 357	C

Table 6: C

CourseId	Instructor
CSC 357	Nico
CSC 365	Migler
STAT 301	Bodwin
CSC 101	Rivera
STAT 302	Frame

Transcript Table T has primary key $(Id, Course)$ and foreign key $(Course)$ referencing Course Table C.

- **Find all Joe's instructors.**

$$\sigma_{Name=Joe}(T)$$
$$\pi_{Course}(\sigma_{Name=Joe}(T))$$
$$C \times \pi_{Course}(\sigma_{Name=Joe}(T))$$
$$\sigma_{C.CourseNo=T.Course}(C \times \pi_{Course}(\sigma_{Name=Joe}(T)))$$
$$\pi_{Instructor}(\sigma_{C.CourseNo=T.Course}(C \times \pi_{Course}(\sigma_{Name=Joe}(T))))$$

Cartesian Product in line 3 makes table with Course No., Instructor, and Course combination for Joe. So it would be every combination with Course and CourseNo.:

C.CourseNo	C.Instructor	T.Course
CSC 357	Nico	CSC 365
CSC 365	Migler	CSC 365
...	...	...
CSC 357	Nico	CSC 357
CSC 365	Migler	CSC 357
...	...	...

In Line 4, we are selecting the rows where the CourseNo. in C is equal to the Courses in T.

- **For each course grade, report the instructor who assigned it.**

$$\pi_{T.Course,T.Grade,C.Instructor}(\sigma_{T.Course=C.CourseId}(T \times C))$$

First, we are taking the Cartesian product which gives us 35 rows. Then we are making the table smaller by only selecting the rows that are equal in Courses.

$\Theta-$Join

$$R \bowtie_\Theta S = \sigma_\Theta(R \times S)$$

Θ - selection condition where each comparison uses attributes from R and S

Equi-Join

$$R \bowtie_\Theta S$$

$$\Theta = (\text{R.A} = \text{S.B}) \wedge (\text{R.C} = \text{S.D}) \wedge (\text{R.E} = \text{S.F})$$

Natural Join

$$R(A_1, ...A_k, B_1, ...B_k)$$
$$S(B_1, ...B_k, C_1, ...C_k)$$

$$R \bowtie S = \pi_{R.*,S.C_1,...S.C_k}(R \bowtie_{Condition} S)$$

Condition is where $R.B_1 = S.B_1 \wedge ... \wedge R.B_k = S.B_k$.

Natural join looks at all **common attributes** of two relations and joins on them, removing one set of common attributes from the final relation.

Semi-Join

Special case of natural join where only attributes of one relation are kept. Essentially, it is a projection on all elements of 1 relation.

Left Semi Join:

$$R \bowtie_\Theta S = \pi_{R.*}(R \bowtie_\Theta S)$$

Right Semi Join:

$$R \rtimes_\Theta S = \pi_{S.*}(R \rtimes_\Theta S)$$

Example:

$$(\sigma_{Name=Joe}(R) \rtimes C)$$

10 Relational Algebra Operations and Expression Trees

Examples

Table 7: Patients (P)

Id	Name	DoB
1	Mary	1/1/2000
2	Jack	3/7/1982
3	Amy	4/12/1976
4	Tom	12/17/1995

Table 8: Prescriptions (R)

PId	MId	Date	Refills
1	1	3/30/23	2
1	3	12/20/22	12
2	4	3/15/23	0
3	5	3/15/23	7
3	2	1/1/23	3

Table 9: Medications (M)

Id	Name	Dosage
1	Ibuprofen	200
2	Ibuprofen	500
3	Metformin	100
4	Aspirin	10
5	Romapril	20

• **Find Ibuprofen dosages for each patient (Name, Dosage).**

$$R \bowtie_{R.MId=M.Id} \sigma_{Name=Ibuprofen}(M)$$
$$P \bowtie_{P.Id=R.Id} (R \bowtie_{R.MId=M.Id} \sigma_{Name=Ibuprofen}(M))$$
$$\pi_{P.Name,Dosage}(P \bowtie_{P.Id=R.Id} (R \bowtie_{R.MId=M.Id} \sigma_{Name=Ibuprofen}(M)))$$

First line we add the columns of M to R but only the Ibuprofen rows. At the same only adding these columns to the rows in R which have the MIds match:

PId	MId	Date	Refills	Id	Name	Dosage
1	1	3/30/23	2	1	Ibuprofen	200
3	2	1/1/23	3	2	Ibuprofen	500

Finally, in the second line we join this new schema with P only referring to the Ids:

Id	Name	DoB	PId	MId	Date	Refills	Id	Name	Dosage
1	Mary	1/1/2000	1	1	3/30/23	2	1	Ibuprofen	200
3	Amy	4/12/1976	3	2	1/1/23	3	2	Ibuprofen	500

Then we just use projection to get the needed columns:

Name	Dosage
Mary	200
Amy	500

• **Find all patients who are older than Jack.**

$$\sigma_{DoB<3/7/82}(P)$$

This above equation is **wrong** because you are searching through all rows of P.

$$P \bowtie_{DoB<DoB} (\sigma_{Name=Jack}(P))$$

This above is called a **self-join** because you are joining a table with itself. But you need to disambiguate so you need to use the renaming operator ρ.

$$P \bowtie_{P.DoB<J.DoB} \rho_J(\sigma_{Name=Jack}(P))$$
$$\pi_{P.*}(P \bowtie_{P.DoB<J.DoB} \rho_J(\sigma_{Name=Jack}(P)))$$

right another way using left semi join.

Renaming Operator ρ

$$\rho_x(R)$$
$$\rho_{(A_1 \longrightarrow B_1,\dots)}(R)$$
$$\rho_{x(A_1 \longrightarrow B_1,\dots)}(R)$$

First line renames the table name to x. Second line renames the attributes to the new names. This is useful for self-joins. Can also combine first 2 lines as in the third line.

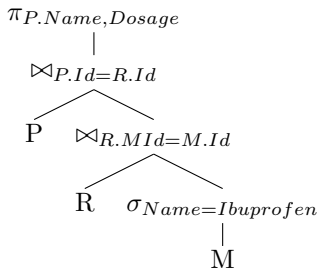
Once M and R are renamed then they are forgotten for the duration of the operation.

Expression Trees

Example: Ibuprofen Relational Tree

Begin writing relational tree from the bottom up. Start with the inner most operation and work your way up. For the above example the selection would be the inner most operation and the outer join would be the topmost operation.

There can be multiple ways to represent the relational tree, but some trees work better as they generate less data.



11 Relational Algebra Continued and SQL Select Statement

Sort: τ

τ_F where F is a list of attributes. Sorts the relation by the attributes. $F = A, B, C$ which is a project list. Can also put operations on the attributes. $F = Desc(A\ MOD\ B), B, Desc(C)$.

Example

A	B	C
1	2	3
4	5	6
7	8	9

$$\pi_B(\tau_{Desc(A)}(R))$$

This above sorts the table by A in descending order and then projects the B column. The result is:

B
8
5
2

Set Operations

- **Only apply these when R, S have the same schema.**
- **These are bag operations.**

Union: $\cup$

All elements that are in either set.

Definition:

$$R \cup S = \{t | t \in R \vee t \in S\}$$

Set Difference: $-$

All elements that are in R but not in S.

Definition:

$$R - S = \{t | t \in R \wedge t \notin S\}$$

Intersection: $\cap$

All elements that are in both sets.

Definition:

$$R \cap S = \{t | t \in R \wedge t \in S\}$$

Example

Table 10: R

A	B
1	2
4	5
7	8

Table 11: S

A	B
1	2
7	8
5	11
6	3

Table 12: $R \cup S$

A	B
1	2
4	5
7	8
5	11
6	3

Table 13: $R - S$

A	B
4	5

Table 14: $R \cap S$

A	B
1	2
7	8

Duplicate Elimination: δ

Removes duplicate tuples from a relation. Definition: $\delta(R)$. This is a set representation of bag R.

This operation allows us to go between bag and set representations. Example: $R \cap S$ would allow duplicates, but $\delta(R - S)$ would not.

SQL Select Statement

Technical Debt: Grouping and Outer Joins

```
SELECT <select-list>
```

```
FROM <table1>, <table2>, ...
WHERE <condition>
ORDER BY <sort-list>
```

- FROM gets evaluated first. Here cartesian products and renaming operations are done.
- WHERE gets evaluated second. Responsible for the conversion between cartesian product to joins.
- ORDER BY gets evaluated next. Responsible for sorting.
- SELECT gets evaluated last. Responsible for projection and some renaming to be done.

Patterns:

$\sigma_C(R)$ translates to the following SQL statement:

```
SELECT *
FROM R
WHERE C
```

$\pi_F(R)$ translates to the following SQL statement:

```
SELECT DISTINCT F --removes duplicates
FROM R
```

$\tau_F(R)$ translates to the following SQL statement:

```
SELECT DISTINCT F
FROM R
ORDER BY F
```

$R \times S$ translates to the following SQL statement:

```
SELECT * -- start because there is no projection
FROM R, S
```

12 SQL Select Statement

SQL Select

$$\pi(\tau(\sigma(T_1 \times \cdots \times T_n)))$$
is equivalent to
$$\pi_L(\tau_F(\sigma_C((T_1 \bowtie_\theta T_2) \ldots \bowtie_\theta T_k))$$

Translations

$$\sigma_c(R) \rightarrow \text{SELECT * FROM R WHERE condition}$$
$$\pi_L(R) \rightarrow \text{SELECT L FROM R}$$
$$\pi_L(R) \rightarrow \text{SELECT DISTINCT L F FROM R}$$
$$\tau_L(R) \rightarrow \text{SELECT * FROM R ORDER BY L}$$
$$R \times S \rightarrow \text{SELECT * FROM R, S}$$
$$R \bowtie_C S \rightarrow \text{SELECT * FROM R, S WHERE condition}$$

First π projection is a bag projection (lazy), while the second is a set projection (eager).

Example

• Find all pokemon whose genus is 'Worm'

$$\pi_{Name}(Pokemon \bowtie_{Pokemon.PokedexId=Species.PokedexId} \sigma_{genus='Worm'}(Species))$$

Can use renaming to make it easier to read (P for Pokemon and S for Species).

```
SELECT Name
FROM Pokemon P, Species S
WHERE P.PokedexId = S.PokedexId
AND S.Genus = 'Worm'
```

• Find the base experience for all Cactus pokemon. Output name and base experience.

```
SELECT *
FROM Species s, Pokemon p
WHERE p.PokedexId = s.PokedexId and s.Genus = 'Cactus'
```

$$\pi_{Pokemon,A.BaseExperience}((\sigma_{gGenus='Cactus'}(S) \bowtie_{S.PokedexId=P.PokedexId} (P)) \bowtie_{P.PokedexId=A.PokedexId} A)$$

• Find all the names of fire pokemon.

```
SELECT p.Name
FROM Species s, Types t, Pokemon p
WHERE s.TypeId = t.TypeId and
    s.PokedexId = p.PokedexId and
    t.Type = 'Fire'
```

$$\pi_{Name}((\sigma_{Type='Fire'}(T) \bowtie_{T.TypeId=S.TypeId} (S)) \bowtie_{S.PokedexId=P.PokedexId} (P))$$

• Find any fire pokemon who is heavier than Gurdurr.

```
SELECT p.Name, a.weight
FROM Species s, Types t, Pokemon p, Attributes a, Pokemon g, Attributes gw
WHERE s.TypeId = t.TypeId and
    s.PokedexId = p.PokedexId and
    a.PokedexId = p.PokedexId and
        t.Type = 'Fire' and
    g.name = 'Gurdurr' and
    g.PokedexId = gw.PokedexId and
    gw.weight < a.weight
ORDER BY a.weight DESC
```

$$\pi_{Name}((\sigma_{Type='Fire'}(T) \bowtie_{T.TypeId=S.TypeId} (S)) \bowtie_{S.PokedexId=P.PokedexId} (P))$$

13 SQL Select Statement Continued and Aggregate Operations

SQL Select Cont.

• Find all the BMI of all fire pokemon.

```
SELECT p.PokedexId, p.Name
FROM Pokemon p, Species s, Types T
WHERE s.TypeId = t.TypeId AND
    p.PokedexId = s.PokedexId AND
    t.Type = 'Fire'

SELECT PokedexId, weight/(height*height) AS BMI
FROM Attributes
WHERE weight/(height*height) < 10 -- cannot use BMI here
ORDER BY BMI DESC

-- Combine above two queries

SELECT p.PokedexId, p.Name, a.weight/(a.height*a.height) AS BMI
FROM Pokemon p, Species s, Types T, Attributes a
WHERE s.TypeId = t.TypeId AND
    p.PokedexId = s.PokedexId AND
    t.Type = 'Fire' AND
    p.PokedexId = a.PokedexId
ORDER BY BMI DESC
```

Can use "AS" to rename columns. Can also use in "ORDER BY" to sort by that renamed attribute. It **cannot be used** in "WHERE" because it is not a real attribute.

Aggregate Operations

LIKE Operator

• What are all Pokemon whose name starts with 'B'?

```
SELECT Name
FROM Pokemon
WHERE Name LIKE 'B%'
ORDER BY Name
```

"LIKE" is a string operator. % is a wildcard. _ is a single character wildcard. Wildcard means any number of characters. **LIKE is case sensitive**.

Expressions

1. **COUNT** - number of rows

2. **SUM** - sum of values
3. **AVG** - average of values
4. **MIN** - minimum value
5. **MAX** - maximum value
6. **STD** - population standard deviation
7. **VAR** - population variance
8. **GROUP_CONCAT** - concatenates strings

Expressions 1-7 take $< Expression >$ as an argument and return a single value. Expression 8 takes $< Expression >$ and $< Separator >$ as arguments and returns a single value. **COUNT only counts non-null values.** Can use DISTINCT to count.

• Example using multiple expressions.

```
SELECT count(*) AS Num, AVG(G3) AS 'Mean Grade', MAX(G3) AS 'Best', MIN(G2)
as 'Worst Midterm', AVG(dalc+walc)
FROM Math
WHERE school = 'GP' AND age = 15 AND sex = 'F'
```

• Examples using GROUP_CONCAT: Write out names of all Fox Pokemon in one line.

```
SELECT GROUP_CONCAT(p.Name ORDER BY p.Name)
FROM Pokemon p, Species s
WHERE p.PokedexId = s.PokedexId AND
    s.Genus = 'Fox'

SELECT GROUP_CONCAT(p.Name ORDER BY p.Name),
        GROUP_CONCAT(DISTINCT s.Genus)
    FROM Pokemon p, Species s
    WHERE p.PokedexId = s.PokedexId AND
        s.Genus = 'Fox'

SELECT GROUP_CONCAT(p.Name ORDER BY p.Name SEPARATOR ' --> '),
        GROUP_CONCAT(DISTINCT s.Genus)
    FROM Pokemon p, Species s
    WHERE p.PokedexId = s.PokedexId AND
        s.Genus = 'Fox'
```

14 Aggregate Operations

Grouping and Aggregation in Relational Algebra: γ

Grouping operate creates a transformation from a set of tuples to a set of groups. Each group is a set of tuples that have the same value for the attributes in the group by clause.

$$\gamma_{A_1,...A_k,Agg(C_1),...Agg(C_m)}(< Relation >)$$
$$\text{where } \{A_1, \dots A_k\} \in B_1, \dots B_n$$
$$\text{where } \{C_1, \dots C_m\} \in B_1, \dots B_n$$
$$A_1 \dots A_k \cap C_1 \dots C_m = \emptyset$$
$$\text{Agg} \in COUNT, SUM, AVG, MIN, MAX, STD, VAR, GROUP\_CONCAT$$

$\gamma_{<GroupingAttributes>,<AggregationAttributes>}(< Relation >) = \{t | t = (a_1, ..., a_k, c_1, ..., c_m)\}$ $a_1, ...a_m$ are all combinations of values of $A_1, \dots A_k$ found in R. $c_1, ..., c_m$ are all combinations of values of $C_1, \dots C_m$ found in R.

• Aggregate functions that show up in other clauses should still be included in the γ clause. Projection would then remove them from showing up in the final result.

Examples

$\gamma_{x,max(Y)}(R)$

$\gamma_{A,x,max(Y),count(*)}R$ results in:

A	x	max(Y)	count(*)
A	1	9	2
B	1	10	2
A	2	18	3
A	3	-2	1
B	3	7	1
A	4	7	1

Group By in SQL

Order of SQL clauses:

```
6 - SELECT
1 - FROM
2 - WHERE
3 - GROUP BY
4 - HAVING
5 - ORDER BY
```

Group By Clause in SQL

```
SELECT <Grouping Attributes>, <Aggregation Attributes>
FROM <Relation>
WHERE <Condition>
GROUP BY <Grouping Attributes>
HAVING <Condition>
ORDER BY <Attribute>
```

If using Group By in SQL, then SELECT clause must contain only:

- Attributes that are listed in the GROUP BY clause.
- Aggregate operations on attributes not listed in the GROUP BY clause.
- COUNT(*)

15 HAVING Clause, GROUP_CONCAT Example, and Nested Queries

Grouping and Aggreation Continued

This can be done:

```
SELECT ...
FROM ...
WHERE A > 1
GROUP BY
```

This cannot be done:

```
SELECT A, Sum(B)
FROM R
WHERE SUM(B) > 10
GROUP BY A
```

Sum(B) is not an attribute of R, so it cannot be used in the WHERE clause.

HAVING Clause

It is allowed to access any aggregation attribute such as from the GROUP BY list and ones not in that list. It is similar to the WHERE clause, but it is applied after the GROUP BY clause.

Nested Queries

Demonstrated examples. Notes in professor's document.

16 Nested Queries

IN

```
SELECT ...
FROM ...
WHERE A IN (SELECT ...)
```

IN can be used to check if a value is in a set of values. The subquery must return a set of values.

Example: IN

• Find the max and min weight of a ground pokemon

```
SELECT Pokemon.Name, weight
FROM Attributes, Pokemon
WHERE Attributes.PokedexID IN
    (SELECT PokedexId FROM Species
    WHERE typeID = (SELECT typeId
                    FROM Types
                    WHERE type = 'ground')) AND
    Pokemon.PokedexId = Attributes.PokedexId
```

Example: ANY

• Find the weights of fire pokemon that are heavier than some ground pokemon

```
SELECT Attributes.PokedexId, weight
FROM Attributes, Pokemon
WHERE Attributes.PokedexID IN
    (SELECT PokedexId FROM Species
    WHERE typeID = (SELECT typeId
                    FROM Types
                    WHERE type = 'fire')) AND
    Pokemon.PokedexId = Attributes.PokedexId AND
    weight > ANY
        (SELECT weight
        FROM Attributes, Pokemon
        WHERE Attributes.PokedexID IN
            (SELECT PokedexId FROM Species
            WHERE typeID = (SELECT typeId
                            FROM Types
                            WHERE type = 'ground')) AND
            Pokemon.PokedexId = Attributes.PokedexId)
```

Example: ALL

• Find ground pokemon that are heavier than all fire pokemon

```
SELECT Attributes.PokedexId, weight
FROM Attributes, Pokemon
WHERE Attributes.PokedexID IN
    (SELECT PokedexId FROM Species
    WHERE typeID = (SELECT typeId
                    FROM Types
                    WHERE type = 'ground')) AND
    Pokemon.PokedexId = Attributes.PokedexId AND
    weight > ALL
        (SELECT weight
        FROM Attributes, Pokemon
        WHERE Attributes.PokedexID IN
            (SELECT PokedexId FROM Species
            WHERE typeID = (SELECT typeId
                            FROM Types
                            WHERE type = 'fire')) AND
            Pokemon.PokedexId = Attributes.PokedexId)
```

EXISTS

Correlated Nested Queries

• Find the pokemon with the highest weight for each type

```
SELECT *
FROM Stats a
WHERE weight = (SELECT MAX(weight) FROM Stats b
            WHERE a.type = b.type)
```

This is a correlated nested query because the sub-query depends on the outer query. It will compute joins for every row in the outer query.

Example: EXISTS

• Find all pokemon types that have a pokemon heavier than 3000

```
SELECT DISTINCT type
FROM Stats a
WHERE EXISTS (SELECT * FROM Stats b
            WHERE a.type = b.type AND weight > 3000)
```

17 Difference, Joins, and Additional SQL Syntax

Difference

In Mysql: Sub-queries + NOT IN

```
SELECT ...
FROM ...
WHERE A NOT IN (SELECT ...)
```

In Oracle and SQL: EXCEPT

```
SELECT ...
FROM ...
EXCEPT
SELECT ...
FROM ...
```

In ANSI SQL: MINUS

Example: Difference

```
SELECT Type, typeId
FROM Types
WHERE type NOT IN (
                    SELECT DISTINCT type
                    FROM Stats
                    WHERE weight > 1000
                    )
```

$$T - (T \ltimes_{T.typeId=S.typeId} (S \ltimes_{S.PokedexId=A.PokedexId} \sigma_{weight>1000}(A))) \tag{1}$$

Join Syntax and Outer Joins

JOIN Syntax

```
SELECT ...
FROM <table1> JOIN <table2> ON <condition>
```

Can also use USING instead of ON if the join condition is on the same attribute:

```
SELECT ...
FROM <table1> JOIN <table2> USING (<attribute>)
```

This is different from NATURAL JOIN because NATURAL JOIN will join on all attributes with the same name:

```
SELECT ...
FROM <table1> NATURAL JOIN <table2>
```

Example: JOIN

```
SELECT *
FROM Pokemon p JOIN Attributes a ON p.PokedexId = a.PokedexId
```

Equivalent to:

```
SELECT *
FROM Pokemon p, Attributes a
USING (PokedexId)
```

Equivalent to:

```
SELECT *
FROM Pokemon p, Attributes a
WHERE p.PokedexId = a.PokedexId
```

Outer Joins

Outer join is a join that includes tuples that do not have a match in the other table. There are three types of outer joins: left, right, and full.

SQL Syntax:

```
SELECT ...
FROM <table1> LEFT/RIGHT [OUTER] JOIN <table2> ON <condition>
```

Can omit OUTER keyword but it is good practice to include it for clarity.

Left Outer Join

$R \mathbin{⟕} S = (R \bowtie_\theta S) \cup \{(t, NULL, ..., NULL) | r \in R \text{ and no tuple } t' \in S \text{ can be joined with } t \text{ on condition } \theta\}$

Right Outer Join

$R \mathbin{⟖} S = (R \bowtie_\theta S) \cup \{(NULL, ..., NULL, t') | t' \in S \text{ and no tuple } t \in R \text{ can be joined with } t' \text{ on condition } \theta\}$

Full Outer Join

$$R \rtimes\ltimes S = (R \rtimes\bowtie S) \cup (R \ltimes S)$$

Additional SQL Syntax

In professor notes: WITH, IF, and CASE.

WITH Example:

```
WITH T1 AS (SELECT ...)
SELECT ...
FROM T1 JOIN ...
```

18 Midterm Information, Database Views, Query Plans, Indexes

Midterm Information

- 3 hours
- Data Description Pages Out Wednesday
- Cheat Sheet of 1 page both sides
- Comprehensive
- Topics: SQL DML/DDl, SQL Select, Relational Data Model (table, attributes, how to define there, schema vs instance, keys, role of candidate keys, primary keys get selected, candidate keys get used how), Relational Algebra (will include γ, $\rtimes$), Query Trees, Relational Algebra to SQL Select Statement or other way around

Database Views

Database View is a single table that is derived from other tables in the database, and whose content changes together with the content of the underlying tables.

View Maintenance is the process of keeping views up to date when the database is updated.

- Materialized Views $\longrightarrow$ create a table storing output, INSERT/UPDATE/DELETE; incremental view maintenance
- On Demand Views $\longrightarrow$ run SQL query defining the view each time
- MYSQL $\longrightarrow$ On Demand Views