CS121 MIDTERM REVIEW

CS121: Relational Databases

Fall 2018 – Lecture 13

Before We Start…

Midterm Overview

- ? hours, multiple sittings
- Open book, open notes, open lecture slides
- No collaboration
- Possible Topics:
  - Basically, everything you've seen on homework assignments to this point
  - Relational model
    - relations, keys, relational algebra operations (queries, modifications)
  - SQL DDL commands
    - **`CREATE TABLE`**, **`CREATE VIEW`**, integrity constraints, etc.
    - Altering existing database schemas
    - Indexes

Midterm Overview (2)

- Possible Topics (cont):
  - SQL DML commands
    - **`SELECT`, `INSERT`, `UPDATE`, `DELETE`**
    - Grouping and aggregation, subqueries, etc.
    - Aggregates of aggregates ☺
    - Translation to relational algebra, performance considerations, etc.
  - Procedural SQL
    - User-defined functions (UDFs)
    - Stored procedures
    - Triggers
    - Cursors

Midterm Overview (2)

- You should use a MySQL database for the SQL parts of the exam
  - e.g. make sure your DDL and DML syntax is correct, check schema-alteration steps, verify that UDFs work
- WARNING: Don't let it become a time-sink!
  - I won't necessarily give you actual data for problems
  - Don't waste time making up data just to test your SQL

Midterm Overview (3)

- Midterm posted online around Friday, November 9
- Due Friday, November 16 at 5:00PM
  (the usual time)

- No homework to do next week

Assignments and Solution Sets

- Some assignments may not be graded in time for the midterm (e.g. HW3, HW4)
- HW1-HW4 solution sets will be on Moodle by the time of the midterm

Relational Model

- Be familiar with the relational model:
  - What's a relation? What's a relation schema? What's a tuple? etc.
- Remember, relations are different from SQL tables in a very important way:
  - Relations are <u>sets</u> of tuples. SQL tables are <u>multisets</u> of tuples.

Keys in the Relational Model

- Be familiar with the different kinds of keys
  - Keys uniquely identify tuples within a relation
- Superkey
  - Any set of attributes that uniquely identifies a tuple
  - If a set of attributes *K* is a superkey, then so is any superset of *K*
- Candidate key
  - A minimal superkey
  - If any attribute is removed, no longer a superkey
- Primary key
  - A particular candidate key, chosen as the primary means of referring to tuples

Keys and Constraints

- Keys constrain the set of tuples that can appear in a relation
  - In a relation *r* with a candidate key *K*, no two tuples can have the same values for *K*

- Can also have foreign keys
  - One relation contains the key attributes of another relation
  - Referencing relation has a foreign key
  - Referenced relation has a primary (or candidate) key
  - Referencing relation can only contain values of foreign key that also appear in referenced relation
  - Called <u>referential integrity</u>

Foreign Key Example

- Bank example:

  account(*account_number*, *branch_name*, *balance*)

  depositor(*customer_name*, *account_number*)

- *depositor* is the referencing relation
  - *account_number* is a foreign-key to *account*
- *account* is the referenced relation

A Note on Notation

- Depositor relation:
  - *depositor*(*customer_name*, *account_number*)
- In the relational model:
  - Every (*customer_name*, *account_number*) pair in *depositor* is unique
- When translating to SQL:
  - **`depositor`** table <u>could</u> be a multiset…
  - Need to ensure that SQL table is actually a <u>set</u>, not a multiset
  - **`PRIMARY KEY (customer_name, account_number)`** after all columns are declared

Referential Integrity in Relational Model

- In the relational model, you must pay attention to referential integrity constraints
  - Make sure to perform modifications in an order that maintains referential integrity
- Example: Remove customer "Jones" from bank
  - Customer name appears in *customer*, *depositor*, and *borrower* relations
  - Which relations reference which?
    - *depositor* references *customer*
    - *borrower* references *customer*
  - Remove Jones records from *depositor* and *borrower* first
  - Then remove Jones records from *customer*

Relational Algebra Operations

- Six fundamental operations:

σ	select operation
Π	project operation
$\cup$	set-union operation
$-$	set-difference operation
$\times$	Cartesian product operation
ρ	rename operation

  - Operations take one or two relations as input
  - Each produces another relation as output

Additional Relational Operations

- Several additional operations, defined in terms of fundamental operations:
  - $\cap$ set-intersection
  - $\bowtie$ natural join (also theta-join $\bowtie_\theta$)
  - $\div$ division
  - $\leftarrow$ assignment
- Extended relational operations:
  - Π *generalized* project operation
  - $\mathcal{G}$ grouping and aggregation
  - ⟕ ⟖ ⟗ left outer join, right outer join, full outer join

Join Operations

- Be familiar with different join operations in relational algebra
- Cartesian product $r \times s$ generates every possible pair of rows from *r* and *s*
- Summary of other join operations:

r =

attr1	attr2
a	r1
b	r2
c	r3

s =

attr1	attr3
b	s2
c	s3
d	s4

$r \bowtie s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3

$r ⟕ s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

$r ⟖ s$

attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

$r ⟗ s$

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

Rename Operation

- Mainly used when joining a relation to itself
  - Need to rename one instance of the relation to avoid ambiguities
- Remember you can specify names with both Π and $\mathcal{G}$
  - Can rename attributes
  - Can assign a name to computed results
  - Naming computed results in Π or $\mathcal{G}$ is shorter than including an extra ρ operation
- Use ρ when you are <u>only</u> renaming things
  - Don't use Π or $\mathcal{G}$ just to rename something
  - Also, ρ doesn't create a new relation-variable! Assignment $\leftarrow$ does this.

Examples

- Schema for an auto insurance database:

  car(*license*, *vin*, *make*, *model*, *year*)

  - *vin* is also a candidate key, but not the primary key

  customer(*driver_id*, *name*, *street*, *city*)

  owner(*license*, *driver_id*)

  claim(*driver_id*, *license*, *date*, *description*, *amount*)

- Find names of all customers living in Los Angeles or New York.

  $\Pi_{name}(\sigma_{city=\text{"Los Angeles"} \lor city=\text{"New York"}}(customer))$

  - Select predicate can refer to attributes, constants, or arithmetic expressions using attributes
  - Conditions combined with $\land$ and $\lor$

Examples (2)

- Schema:

  car(*license*, *vin*, *make*, *model*, *year*)

  customer(*driver_id*, *name*, *street*, *city*)

  owner(*license*, *driver_id*)

  claim(*driver_id*, *license*, *date*, *description*, *amount*)

- Find customer name, street, and city of all Toyota owners
  - Need to join *customer*, *owner*, *car* relations
  - Could use Cartesian product, select, etc.
  - Or, use natural join operation:

    $\Pi_{name,street,city}(\sigma_{make=\text{“Toyota”}}(customer \bowtie owner \bowtie car))$

Examples (3)

- Schema:

  car(*license*, *vin*, *make*, *model*, *year*)

  customer(*driver_id*, *name*, *street*, *city*)

  owner(*license*, *driver_id*)

  claim(*driver_id*, *license*, *date*, *description*, *amount*)

- Find how many claims each customer has
  - Don't include customers with no claims…
  - Simple grouping and aggregation operation

    $_{driver\_id}\mathcal{G}_{\textbf{count}(license)\ \textbf{as}\ num\_claims}(claim)$

    - The specific attribute that is counted is irrelevant here…
  - Aggregate operations work on multisets by default
  - Schema of result?

    (*driver_id*, *num_claims*)

Examples (4)

- Now, include customers with no claims
  - They should have 0 in their values
  - Requires outer join between *customer*, *claim*
  - "Outer" part of join symbol is towards relation whose rows should be null-padded
  - Want all customers, and claim records if they are there, so "outer" part is towards *customer*

    $_{driver\_id}\mathcal{G}_{\textbf{count}(license)\ \textbf{as}\ num\_claims}(customer ⟕ claim)$
  - Aggregate functions ignore *null* values

Selecting on Aggregate Values

- Grouping/aggregation op produces a relation, not an individual scalar value

  You cannot use aggregate functions in select predicates!!!

- To select rows based on an aggregate value:
  - Create a grouping/aggregation query to generate the aggregate results
    - This is a relation, so…
  - Use Cartesian product (or another appropriate join operation) to combine rows with the relation containing aggregated results
  - Select out the rows that satisfy the desired constraints

Selecting on Aggregate Values (2)

- General form of grouping/aggregation:
  - $_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1), F(A_2), \ldots}(\ldots)$
- Results of aggregate functions are unnamed!
- This query is <u>wrong</u>:
  - $\sigma_{F(A_1) = \ldots}(_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1), F(A_2), \ldots}(\ldots))$
  - Attribute in result does <u>not</u> have name $F(A_1)$!
- Must *assign* a name to the aggregate result
  - $_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1) \textbf{ as } V_1, F(A_2) \textbf{ as } V_2, \ldots}(\ldots)$
- Then, can properly select against the result:
  - $\sigma_{V_1 = \ldots}(_{G_1, G_2, \ldots}\mathcal{G}_{F(A_1) \textbf{ as } V_1, F(A_2) \textbf{ as } V_2, \ldots}(\ldots))$

An Aggregate Example

- Schema: *car*(<u>*license*</u>, *vin*, *make*, *model*, *year*)
  customer(<u>*driver_id*</u>, *name*, *street*, *city*)
  owner(<u>*license*</u>, <u>*driver_id*</u>)
  claim(<u>*driver_id*</u>, <u>*license*</u>, <u>*date*</u>, *description*, *amount*)
- Find the claim(s) with the largest amount
  - Claims are identified by (*driver_id*, *license*, *date*), so just return all attributes of the claim
  - Use aggregation to find the maximum claim amount:
    $\mathcal{G}_{\mathbf{max}(amount)\ \mathbf{as}\ max\_amt}(claim)$
  - This generates a relation! Use Cartesian product to select the row(s) with this value.

$$\Pi_{driver\_id, license, date, description, amount}(\sigma_{amount = max\_amt}(claim \times \mathcal{G}_{\mathbf{max}(amount)\ \mathbf{as}\ max\_amt}(claim)))$$

Another Aggregate Example

- Schema: *car*(*license*, *vin*, *make*, *model*, *year*)
  customer(*driver_id*, *name*, *street*, *city*)
  owner(*license*, *driver_id*)
  claim(*driver_id*, *license*, *date*, *description*, *amount*)
- Find the customer with the most insurance claims, along with the number of claims
- This involves two levels of aggregation
  - Step 1: generate a count of each customer's claims
  - Step 2: compute the maximum count from this set of results
- Once you have result of step 2, can reuse the result of step 1 to find the final result
- Common subquery: computation of how many claims each customer has

Another Aggregate Example (2)

- Use assignment operation to store temporary result

  $claim\_counts \leftarrow {}_{driver\_id}\mathcal{G}_{\mathbf{count}(license)\ \mathbf{as}\ num\_claims}(claim)$

  $max\_count \leftarrow \mathcal{G}_{\mathbf{max}(num\_claims)\ \mathbf{as}\ max\_claims}(claim\_counts)$

- Schemas of *claim_counts* and *max_count* ?

  claim_counts(*driver_id*, *num_claims*)

  max_count(*max_claims*)

- Finally, select row from *claim_counts* with the maximum count value
  - Obvious here that a Cartesian product is necessary

    $$\Pi_{driver\_id,num\_claims}(\sigma_{num\_claims=max\_claims}(claim\_counts \times max\_count))$$

Modifying Relations

- Can add rows to a relation

  $r \leftarrow r \cup \{ (\ldots), (\ldots) \}$

  - $\{ (\ldots), (\ldots) \}$ is called a constant relation
  - Individual tuple literals enclosed by parentheses ( )
  - Set of tuples enclosed with curly braces { }
- Can delete rows from a relation

  $r \leftarrow r - \sigma_P(r)$
- Can modify rows in a relation

  $r \leftarrow \Pi(r)$

  - Uses generalized project operation

Modifying Relations (2)

- **Remember to include unmodified rows!**

  $r \leftarrow \Pi(\sigma_P(r)) \;\cup\; \sigma_{\neg P}(r)$

- Relational algebra is <u>not</u> like SQL for updates!
  - Must <u>explicitly</u> include unaffected rows
- Example:

  Transfer \$10,000 in assets to all Horseneck branches.

  $branch \leftarrow \Pi_{branch\_name,branch\_city,assets+10000}(\sigma_{branch\_city=\text{“Horseneck”}}(branch))$

  Wrong: This version *throws out* all branches not in Horseneck!

  $branch \leftarrow \Pi_{branch\_name,branch\_city,assets+10000}(\sigma_{branch\_city=\text{“Horseneck”}}(branch)) \cup \sigma_{branch\_city \neq \text{“Horseneck”}}(branch)$

  Correct. Non-Horseneck branches are included, unmodified.

Structured Query Language

- Some major differences between SQL and relational algebra!
- Tables are like relations, but are multisets
- Most queries generate multisets
  - **SELECT** queries produce multisets, unless they specify **SELECT DISTINCT** …
- Some operations do eliminate duplicates!
  - Set operations: **UNION**, **INTERSECT**, **EXCEPT**
    - Duplicates are eliminated automatically, unless you specify **UNION ALL**, **INTERSECT ALL**, **EXCEPT ALL**

SQL Statements

- **SELECT** is most ubiquitous

```
SELECT A1, A2, ... FROM r1, r2, ...
  WHERE P;
```

  - Equivalent to: $\Pi_{A_1,\, A_2,\, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$
- **INSERT**, **UPDATE**, **DELETE** all have common aspects of **SELECT**
  - All support **WHERE** clause, subqueries, etc.
  - Also **INSERT** … **SELECT** statement

Join Alternatives

- **`FROM r1, r2`**
  - Cartesian product
  - Can specify join conditions in **`WHERE`** clause
- **`FROM r1 JOIN r2 ON (r1.a = r2.a)`**
  - Most like theta-join operator: $r \bowtie_\theta s = \sigma_\theta(r \times s)$
  - Doesn't eliminate any columns!
- **`FROM r1 JOIN r2 USING (a)`**
  - Eliminates duplicate column **`a`**
- **`FROM r1 NATURAL JOIN r2`**
  - Uses <u>all</u> common attributes to join **`r1`** and **`r2`**
  - Also eliminates <u>all</u> duplicate columns in result

Join Alternatives (2)

- Can specify inner/outer joins with **JOIN** syntax
  - **r INNER JOIN s ...**
  - **r LEFT OUTER JOIN s ...**
  - **r RIGHT OUTER JOIN s ...**
  - **r FULL OUTER JOIN s ...**
- Can also specify **r CROSS JOIN s**
  - Cartesian product of *r* with *s*
  - Can't specify **ON** condition, **USING**, or **NATURAL**
- Can actually leave out **INNER** or **OUTER**
  - **OUTER** is implied by **LEFT**/**RIGHT**/**FULL**
  - If you just say **JOIN**, this is an **INNER** join

Self-Joins

- Sometimes helpful to do a self-join
  - A join of a table with itself
- Example: employees

  employee(*emp_id*, *emp_name*, *salary*, *manager_id*)
- Tables can contain foreign-key references to themselves
  - *manager_id* is a foreign-key reference to *employee* table's *emp_id* attribute
- Example:
  - Write a query to retrieve the name of each employee, and the name of each employee's boss.

```
SELECT e.emp_name, b.emp_name AS boss_name
  FROM employee AS e JOIN employee AS b
       ON (e.manager_id = b.emp_id);
```

Subqueries

- Can include subqueries in **FROM** clause
  - Called a derived relation
  - Nested **SELECT** statement in **FROM** clause, given a name and a set of attribute names
- Can also use subqueries in **WHERE** clause
  - Can compare an attribute to a scalar subquery
    - This is different from the relational algebra!
  - Can also use set-comparison operations to test against a subquery
    - **IN**, **NOT IN** – set membership tests
    - **EXISTS**, **NOT EXISTS** – empty-set tests
    - **ANY**, **SOME**, **ALL** – comparison against a set of values

Scalar Subqueries

- Find name and city of branch with the least assets
  - Need to generate the "least assets" value, then use this to select the specific branch records
- Query:

```
SELECT branch_name, branch_city FROM branch
WHERE assets = (SELECT MIN(assets) FROM branch);
```

  - This is a scalar subquery: one row, one column
  - Don't need to name **MIN(assets)** since it doesn't appear in final result, and we don't refer to it
- Don't do this:

```
WHERE assets=ALL (SELECT MIN(assets) FROM branch)
```

  - **ANY**, **SOME**, **ALL** are for comparing a value to a set of values
  - Don't need these when comparing to a scalar subquery

Subqueries vs. Views

- Don't create views unnecessarily
  - Views are part of a database's schema
  - Every database user sees the views that are defined
- Views should generally expose "final results," not intermediate results in a larger computation
  - Don't use views to compute intermediate results!
- If you *really* want functionality like this, read about the **`WITH`** clause (Book, 6th ed: §3.8.6, pg. 97)
  - MariaDB 10.2 now supports **`WITH`** clause! Use it to simplify complicated queries! ☺

WHERE Clause

- **WHERE** clause specifies selection predicate
  - Can use **AND**, **OR**, **NOT** to combine conditions
  - **NULL** values affect comparisons!
    - Can't use **=** **NULL** or **<>** **NULL**
      - <u>Never</u> evaluates to true, regardless of other value
    - Must use **IS NULL** or **IS NOT NULL**
  - Can use **BETWEEN** to simplify range checks
    - **a >= v1 AND a <= v2**
    - **a BETWEEN v1 AND v2**

Grouping and Aggregation

- SQL supports grouping and aggregation
- **GROUP BY** specifies attributes to group on
  - Apply aggregate functions to non-grouping columns in **SELECT** clause
  - Can filter results of grouping operation using **HAVING** clause
    - **HAVING** clause can refer to aggregate values too
- Difference between **WHERE** and **HAVING** ?
  - **WHERE** is applied <u>before</u> grouping;
    HAVING is applied <u>after</u> grouping
  - **HAVING** can refer to aggregate results, too
    - Unlike relational algebra, can use aggregate functions in **HAVING** clause

Grouping: SQL, Relational Algebra

- Another difference between relational algebra notation and SQL syntax
- Relational algebra syntax:

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),F_2(A_2),\ldots,F_m(A_m)}(E)$$

  - Grouping attributes appear only on <u>left</u> of $\mathcal{G}$
  - Schema of result: $(G_1, G_2, \ldots, F_1, F_2, \ldots)$
    - (Remember, F_i generate <u>unnamed</u> results.)
- SQL syntax:

```
SELECT G1,G2,...,  F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

  - To include group-by values in result, specify grouping attributes in **SELECT** clause <u>and</u> in **GROUP BY** clause

SQL Query Example

- Schema:

  car(<u>*license*</u>, *vin*, *make*, *model*, *year*)

  customer(<u>*driver_id*</u>, *name*, *street*, *city*)

  owner(<u>*license*</u>, *driver_id*)

  claim(<u>*driver_id*</u>, <u>*license*</u>, <u>*date*</u>, *description*, *amount*)

- Find customers with more claims than the average number of claims per customer
- This is an aggregate of another aggregate
- Each **SELECT** can only compute <u>one level</u> of aggregation
  - **AVG(COUNT(*))** is **not allowed** in SQL
    (or in relational algebra, so no big surprise)

Aggregates of Aggregates

- <u>Two steps</u> to find average number of claims
- Step 1:
  - Must compute a count of claims for each customer

```
SELECT COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
```

  - Then, compute the average in a second **SELECT**:

```
SELECT AVG(num_claims)
  FROM (SELECT COUNT(*) AS num_claims
        FROM claim GROUP BY driver_id) AS c
```

- This generates a single result
  - Can use it as a scalar subquery if we want.

Aggregates of Aggregates (2)

- Finally, can compute the full result:

```
SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
HAVING num_claims >=
    (SELECT AVG(num_claims)
     FROM (SELECT COUNT(*) AS num_claims
           FROM claim GROUP BY driver_id) AS c);
```

  - Comparison must be in **HAVING** clause
- This won't work:

```
SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id
HAVING num_claims >= AVG(num_claims);
```

  - Tries to do two levels of aggregation in one **SELECT**

Alternative 1: Make a View

- Knowing each customer's total number of claims *could* be generally useful…
- Define a view for it:

```
CREATE VIEW claim_counts AS
  SELECT driver_id, COUNT(*) AS num_claims
  FROM claim GROUP BY driver_id;
```

  - Then the query becomes:

```
SELECT * FROM claim_counts
WHERE num_claims >
      (SELECT AVG(num_claims) FROM claim_counts)
```

  - View hides one level of aggregation

Alternative 2: Use **WITH** Clause

- **WITH** is like defining a view for a single statement
- Using **WITH**:

```
WITH claim_counts AS (
     SELECT driver_id, COUNT(*) AS num_claims
     FROM claim GROUP BY name)
SELECT * FROM claim_counts
WHERE num_claims > (SELECT AVG(num_claims)
                    FROM claim_counts);
```

  - **WITH** doesn't pollute the database schema with a bunch of random views
  - Can specify multiple subqueries in the **WITH** clause, too (see documentation for details)

SQL Data Definition

- Specify table schemas using **CREATE TABLE**
  - Specify each column's name and domain
  - Can specify domain constraint: **NOT NULL**
  - Can specify key constraints
    - **PRIMARY KEY**
    - **UNIQUE** (candidate keys)
    - **REFERENCES table (column)** (foreign keys)
  - Key constraints can go in column declaration
  - Can also specify keys after all column decls.
- Be familiar with common SQL data types
  - **INTEGER**, **CHAR**, **VARCHAR**, date/time types, etc.

DDL Example

- Relation schema:

  car(*<u>license</u>*, *vin*, *make*, *model*, *year*)

  - *vin* is also a candidate key

- **CREATE TABLE** statement:

```
CREATE TABLE car (
  license CHAR(10)     PRIMARY KEY,
  vin     CHAR(30)     NOT NULL UNIQUE,
  make    VARCHAR(20) NOT NULL,
  model   VARCHAR(20) NOT NULL,
  year    INTEGER      NOT NULL
);
```

DDL Example (2)

- Relation schema:

  claim(*<u>driver_id</u>*, *<u>license</u>*, *<u>date</u>*, *description*, *amount*)

- **CREATE TABLE** statement:

```
CREATE TABLE claim (
  driver_id   CHAR(12),
  license     CHAR(10),
  date        TIMESTAMP,
  description VARCHAR(4000) NOT NULL,
  amount      NUMERIC(8,2),

  PRIMARY KEY (driver_id, license, date),
  FOREIGN KEY driver_id REFERENCES customer,
  FOREIGN KEY license REFERENCES car
);
```

Key Constraints and **NULL**

- Some key constraints automatically include **NOT NULL** constraints, but not all do.
- **PRIMARY KEY** constraints
  - Disallows **NULL** values
- **UNIQUE** constraints
  - Allows **NULL** values, unless you specify **NOT NULL**
- **FOREIGN KEY** constraints
  - Allows **NULL** values , unless you specify **NOT NULL**
- Understand how **NULL** values affect **UNIQUE** and **FOREIGN KEY** constraints that allow **NULL**s

Referential Integrity Constraints

- Unlike relational algebra, SQL DBs automatically enforce referential integrity constraints for you
  - You still need to perform operations in the correct order, though
- Same example as before:
  - Remove customer "Jones" from the bank database
  - DBMS will ensure that referential integrity is enforced, but you still have to delete rows from **`depositor`** and **`borrower`** tables first!

```
DELETE FROM depositor WHERE customer_name = 'Jones'
DELETE FROM borrower WHERE customer_name = 'Jones'
DELETE FROM customer WHERE customer_name = 'Jones'
```

Midterm Details

- No homework to do next week
- Good luck! ☺