Week 1. Relational

Definitions

A relational database is a collection of relations (tables).

- **Attributes**: relation column names.
- **Relation Schema**: consists of a relation name and a set of attributes. The **meta-data** of a relation.
- **Tuples**: rows of a relation.
- **Tuple Components**: a tuple has one component for each attribute of the relation; a tuple component value must be from the domain of the attribute.
- **Relation instance**: a finite set of tuples. The **data** of a relation.
- **Relation**: a pair of a relation schema and a relation instance.

Consider the schema `Student(Sid, Sname, Major, Byear)`.
Relation name `Student` and attributes (the headers of table) `(Sid, Sname, Major, Byear)` will compose the **relation schema**. The body of the table is the **relation instance**, which is composed of a set of **tuples**.
In one **tuple** `t = (s1, John, CS, 1990)`, we have `t.Sid`, `t.Sname`, `t.Major` and `t.Byear` as the **tuple components** of `t`. The **domain values** of these tuple components are `s1`, `John`, `CS` and `1990` respectively.

Attribute Domains

Different types are possible. **(1) Basic Types**: boolean, integer, real, character, text. **(2) Composite Types**: monetary, date.**(3) Enumeration Types**: list of values, sets, arrays. **(4) Semi-structured Types**: XML, JSON. **(5)Arrays**. and **(6)** Others

Other Definitions

NULL represents a missing or unknown value of a tuple. Two NULL values may not be equal.
A **key** is a subset of attributes of the schema of a relation that can uniquely distinguish tuples. A key of a relation is a `constraint` of that relation. An insertion will be rejected if the primary key constraint is violated.
In the relation schemas (`Student, Course, Enroll`), the attribute `Sid` of `Enroll` is a **foreign key** referencing the primary key `Sid` of `Student`. **Foreign keys** impose an order on insertions: the tuple `(s1, c1, 'B')` can only be inserted into `Enroll` **AFTER** a tuple identified by the `Sid` value `s1` exists in `Student` **AND AFTER** a tuple identified by the `Cno` value `c1` exists in `Course`. Otherwise, rejection.
Foreign keys necessitate **cascading deletions**: deleting tuple `(s1, 'John', 'CS', 1990)` from `Student` requires deleting all tuples in `Enroll` referencing the `Sid` value `s1`. Two options in SQL: (1) allow cascading deletions. (2) disallow cascading deletions (and hence reject deletion).

Week 1. SQL Part 1

SQL Form

The simplest form of a SQL query is defined as follows.

```
SELECT S.sid, S.Sname --list of components of tuple variables
FROM Student S --list of tuple variables associated with
        realtions
WHERE S.Major='CS'; -- condition on components of tuple
        variables.
```

`S` is a **tuple variable** that ranges over all tuples in the `Student` relation. Specifically, the tuple variable is assigned, one-at-a-time, to each tuple in that relation. In addition, since the `Student` relation is a set, the order in which `S` is assigned to these tuples is **NOT** pre-determined.
If we only query `S.Major` instead of `(S.Sid, S.Sname)`, the result of this query can be a **bag** (**multiset**). We can use **DISTINCT** to coerce the result of the query into a **set**.

Time Complexity

If `Student` has m tuples and `Enroll` has n tuples, a naive query will run `IF S.Sid = E.Sid AND E.Grade='B'` $m \times n$ times. On the other hand, using the subquery to yield s tuples first reduces the total time to $n + m \times s$.

```
SELECT S.sid, C.Cno
FROM Student S, (SELECT E.Sid, E.Cno FROM Enroll E WHERE
        E.Grade='B') C
WHERE S.Sid = C.Sid;
```

Set Operations

The result of set operations **UNION**, **INTERSECT** and **EXCEPT** are all sets, even if the inputs are bags. To retain bag semantics, use **UNION ALL**, **INTERSECT ALL** and **EXCEPT ALL** instead.

Week 2. SQL Part 2

Predicates

We have **IN**, **NOT IN**, **= SOME**, **<= ALL**, **EXISTS**, **NOT EXISTS**.

Week 2. Tuple Relational Calculus (TRC)

TRC Introduction

TRC is a non-procedural query language to retrieve data from relational tables. **TRC query (simplest form)** is denoted as $\{t \mid P(t)\}$, where `P(t)` denotes a formula in which tuple variable `t` appears. **Answers** are a set of all tuples `T` for which the formula `P(t)` evaluates to true.
Formula is recursively defined as (a) a collection of **atom formulas** connected via logical connectives, and (b) is described by a language based on **first-order logic** (variables, predicates, quantifiers).
Atomic formulas inclue $s \in Student$, $s.sid <> 'Eric'$. **Formulas** include an atomic formula p; or $(\neg p)$; $p \vee q$; $p \wedge q$; $p \rightarrow q$; $\exists t,\ P(t)$; $\forall t,\ P(t)$.

TRC Free variables

The use of existential or universal quantifiers for variable t in a formula is to **bound** t in the formula. A tuple variable that is not bound is **free**. A formula P is a **constraint** if it has no free variables; such a constraint is a statement that is either true or false. `Primary keys` and `foreign keys` are examples of constraints.
There are two examples of constraints; both of them implies "There is a student whose name is Eric". Especially, the second is a TRC constraint that uses <>∅ set predicate (∃ is implicitly used).

- $\exists s(\text{Student}(s) \wedge s.\text{sname} = \text{Eric})$.
- $\{1 \mid \text{Student}(s) \wedge s.\text{sname} = \text{Eric}\} <> \emptyset$

TRC Examples

There is a student who only bought books with bookno <> 1000 (two solutions).
$\exists s(\text{Student}(s) \wedge \forall t((\text{Buys}(t) \wedge t.\text{sid} = s.\text{sid}) \rightarrow t.\text{bookno} <> 1000))$
$\exists s(\text{Student}(s) \wedge \neg\exists t(\text{Buys}(t) \wedge t.\text{sid} = s.\text{sid} \wedge t.\text{bookno} <> 1000))$

The corresponding SQL.

```
SELECT EXISTS (
    SELECT 1 FROM Student S WHERE
        NOT EXISTS (
            SELECT 1 FROM Buys T
            WHERE T.sid = S.sid AND T.bookno = 1000))
```

TRC Constraints

The variables that appears to the left of | must be the only free variables in the formula $P(\cdot)$. All other tuple variables must be bound using a quantifier. In SQL, the free variables are exclusively those that are introduced in the **outmost** FROM clause of the query.

Week 3. Relational Algebra

RA Operations

- **Select** (the filter): $\sigma_{\text{dept\_name='Physics'} \wedge \text{salary} > 90000}(\text{instructor})$. Selection is **commutative** $(\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R)))$ and **cascading** $(\sigma_{c_1 \wedge c_2}(R) = \sigma_{c_2 \wedge c_1}(R))$.
- **Project** (a subset of attributes): $\pi_{\text{ID, name, salary}}(\text{instructor})$. Not projection returns a set.
- **Cartesian-Product** (cross join): $\sigma_{\text{instructor.id=teaches.id}}(\text{instructor} \times \text{teaches})$. After cartesian-product, we may need to do the selection.
- **Join** (join with condition): $r \bowtie_\theta s = \sigma_\theta(r \times s)$. **Commutative** $(r \bowtie s = s \bowtie r)$ and **cascading** $(r \bowtie (s \bowtie t) = (r \bowtie s) \bowtie t)$.
- **Union** ∪; **Set Difference** −; **Set-Intersection** ∩.
- **Rename**: $\rho_x(E)$ or $\rho_{x(A_1, A_2, \cdots A_n)}(E)$ will return the result of expression E under the name x.
- **Assignment**: Physics $\leftarrow \sigma_{\text{dept\_name='Music'}}(\text{instructor})$.

Week 4. Joins And Semi-Joins

Operations in query languages that can aid in discovering relationships between objects in the database are called **joins**. Operations in query languages that can aid in discovering objects in the database that satisfy certain properties are called **semi-joins**.

Two types of Joins

The first one is **Regular Joins**, joins that compute relationships between between objects (o_1, o_2) based on **tuple-component comparisons**, like "$t_1.A \theta t_2.B$". It contains **JOIN** (or called **INNER JOIN**); **NATURAL JOIN** (a special case of **JOIN**); **CROSS JOIN** (cartesian product, a special case of **NATURAL JOIN**).

- **JOIN**: $E_1 \bowtie_C E_2$ is called the join between E_1 and E_2 on condition C.
- **CROSS JOIN**: $E_1 \bowtie_{\text{true}} E_2 = E_1 \times E_2$.
- **NATURAL JOIN**: $\text{Enroll} \bowtie \text{TaughtBy} = \pi_{E.\text{sid}, E.\text{cno}, E.\text{grade}, E.\text{tid}}(\text{E} \bowtie_{E.\text{cno}=T.\text{cno}} \text{T})$. A **natural join** between two relations is a series of "equality" joins on the common attributes of the relations, followed by a projection to remove redundant columns.

Multiple natural joins are possible. If we have `Enroll(sid, cno, grade)`, `Student(sid, sname, age)` and `Course(cno, cname, dept)`, the join condition will be `E.sid = S.sid` ∧ `E.cno = C.cno`.
The second one is **Set joins**, joins that compute relations between objects based on comparisons between sets of tuples associated with these objects (e.g., a comparison between the set of courses taken by students and the set of courses taught by a teacher). SQL has limited support for set joins. We can simulate them using predicates **[NOT] EXISTS** and **[NOT] IN**.

Two types of Semi-Joins

The first one is **Regular Semi-Joins**. SQL doesn't have special operations for semi-joins, but they can be simulated by **NATURAL JOIN** operations, or with the **IN** set predicate.
The second one is **Set Semi-Joins**. SQL doesn't have set semijoins. They can be shown to be special cases of set joins and as such can be simulated.

SQL and RA

SQL is a **declarative** language. However, with the addition of **JOIN**, **NATURAL JOIN** and **CROSS JOIN**, SQL can be viewed as an algebra that can faithfully simulate RA.

Semi-Join

A **Semi-Join** will only retain attributes from the first relation (i.e. Student). Please note the rhs of the following formula is natural join. Semi-join can be simulated by **NATURAL JOIN** or **IN**.

$$\begin{aligned}\text{Student} \ltimes \text{Enroll} &= \pi_{\text{S.sid, S.sname, S.age}}(\text{Student} \bowtie \text{Enroll}) \\ &= \text{Student} \bowtie \pi_{E.\text{sid}}(\text{Enroll})\end{aligned}$$

Semi-Join is **not associative**. In general $E_1 \ltimes E_2$ can be implemented using hash in linear time. In contrast, $E_1 \bowtie E_2$ runs in $O(|E_1||E_2|)$ time.
We can use **anti-semijoin** to find students who have not enrolled in any course, which can be simulated with **IN**. The time complexity is linear with hashing.

$$\text{Student} \bar{\ltimes} \text{Enroll} = \text{Student} - \text{Student} \ltimes \text{Enroll}$$

Let E_1 and E_2 be expressions from the same schema, we have

$$E_1 \ltimes E_2 = E_1 \bowtie E_2 = E_1 \cap E_2$$

$$E_1 \bar{\ltimes} E_2 = E_1 - (E_1 \cap E_2) = E_1 - E_2$$

Week 4. Views

A **view** is a virtual relation defined by a query. It works like a **Macro**.

```
CREATE VIEW CS_Course AS
    SELECT C.Cno, C.Cname FROM Course C WHERE C.Dept = 'CS'

CREATE VIEW Student_enrolled_in_CS_course AS
    SELECT S.Sid, S.Sname, S.Major, S.Byear FROM StudentS
    WHERE S.Sid IN (
        SELECT E.Sid FROM Enroll E
        WHERE E.Cno IN (
            SELECT C.Cno FROM CS_Course C));
```

Views can be used to **modularize** a complex query into simpler units. The previous examples illustrate this. Views provide **independence** from changes in the underlying database due to **growth and restructuring** of the base relation schemas. This is called **logical data independence**.

Properties of Views

Both **growth** and **reconstruction** shall not affect views. (A) We can expand the base table by adding new attributes and constraints. No need to redefine views because of these alterations. (B) If we divide the `Student` table into `Student_Info` and `Student_Major`, we can create a view called `Student`.
DROP TABLE table_name will only succeed if there are no views with **table_name** as base relation. **DROP TABLE table_name CASCADE** will succeed and drop all the views that were defined in terms of **table_name**. The same applies to **DROP VIEW view_name**.
View expansion refers to the process of rewriting queries by expanding the views by their definitions in these queries, which precedes query evaluation.

Materialized View

In many cases, especially if the database state is not changing, it is useful to precompute the view and then store it in the database for future use. A view that is precomputed is called a **materialized view**.
The major challenge with materialized views is that they need to be updated when updates to the base relations (i.e., the state of the basis) occur. However, when views are defined by queries that involve negation with expressions involving **EXCEPT**, **ALL**, **NOT IN**, and **NOT EXISTS**, incremental view maintenance may not be possible. In such cases, the entire process to materialize the views need to be redone.

```
REFRESH MATERIALIZED VIEW CS_Course AS -- CREATE/REFRESH:
        initialize / recalculate
    SELECT C.cno FROM Course C WHERE C.dept = âĂŸCSâĂŹ;
```

The **WITH** statement of SQL permits us to define **temporary views** which are only local to the query we wish to solve. These temporary views are not made persistent outside the query.

Recursive Views

A recursive definition of Path is as follows: (A) Base rule: If Graph(s,t) then Path(s,t); (B) Inductive rule: If Graph(s,u) and Path(u,t) then Path(s,t).

```
WITH RECURSIVE Path(source, target) AS (
    SELECT E.source, E.target FROM Graph E
    UNION
    SELECT E.source, P.target
        FROM Graph E, Path P
        WHERE E.target = P.source
)
SELECT * FROM Path;
```

Parameterized Views

In standard SQL, subqueries such as on the previous slide cannot be used to define parameterized views. PostgreSQL, however, permits the use of user-defined functions that return tables. Using this feature, we can specify **parameterized views**.

```
CREATE FUNCTION coursesOfferedByDept(deptname TEXT) RETURNS
        TABLE(Cno TEXT) AS
$$
SELECT C.Cno FROM CourseC WHERE C.Dept = deptname;
$$ LANGUAGE SQL;

SELECT C.Cno FROM coursesOfferedByDept(âĂŸMathâĂŹ) C;
```

Week 6. Aggregate Functions and Data Partitioning

A **collection** is a grouping of some variable number of data items (possibly zero). Usually the data items in a collection are of **the same type**. **Aggregate functions** are functions that apply to collections, i.e., they consider all these data items in these collections. Applied to a collection, an aggregate function returns **a single value**.
Collections include: **(a)** sets, multisets, dictionaries (maps), relations; **(b)** vectors, lists, arrays, series; **(c)** data structures: stacks, queues, hash tables, trees, graphs.

Aggregate Functions

COUNT applied to gives 0; **SUM** applied to gives NULL.

```
SELECT E.Sid, COUNT(*) AS No_Courses FROM Enroll E -- Map COUNT
        phease
GROUP BY(E.Sid) -- Partition Phase
```

Partition phase: the **GROUP BY** operator places each tuple E into the cell identified by its `E.Sid` value.
Map COUNT phase: the **COUNT** function is mapped over the cells identified by the different possible `E.sid` values.

Example of GROUP BY

Find student ids as well as the number of courses they take.

```
(SELECT E.Sid, COUNT(E.Cno) AS No_Courses FROM Enroll E GROUP BY
        (E.Sid))
UNION
(SELECT S.Sid, 0 AS No_Courses FROM Student S
 WHERE S.Sid NOT IN (SELECT E.Sid FROM Enroll E))
```

The following query will raise an error since s.x is not necessarily unique in a cell defined by s.x+s.y values S.

```
SELECT s.x FROM Ss GROUP BY (s.x+s.y)
```

HAVING Aggregate Function

The **HAVING** clause in a **GROUP BY** selects those cells from the partition induced by the **GROUP BY** clause that satisfy an **Aggregate Condition**. Only those cells are passed onto the **SELECT** clause.
For each student who majors in CS determine the number of courses taken by that student, provided that this number is at least 2.

```
SELECT E.Sid, COUNT(E.Cno) FROM Enroll E, Student S
WHERE E.Sid = S.Sid AND S.Major = 'CS'
GROUP BY (E.Sid) HAVING COUNT(E.Cno) âĹé 2;
```

For each student who majors in CS, determine the number of courses taken by that student, provided that this number is at least 3.âĹÌ ÂÛ The **HAVING** condition can be simulated in the WHERE clause with user-defined functions.

```
SELECT S.Sid AS Sid, NumberOfCourses(S.Sid) FROM Student S
WHERE S.major = âĂŸCSâĂŹ AND NumberOfCourses(S.Sid) âĹé 3
```

Rank

List the rank order of the price of each product among all the tuples of its type.

```
SELECT name, type, price, rank() OVER (PARTITION BY type ORDER
       BY price) FROM Product;
-- An equivalent query
SELECT p.name, p.type, p.price, (SELECT COUNT(1) FROM product p1
WHERE p1.type = p.type AND p1.price < p.price) + 1
FROM product p;
```

Week 6. SQL Functions and Expressions

The result of this expression is a relation with a single tuple.

```
SELECT 1 AS one;
```

We can therefore place such an expression in a FROM clause of another query.

```
SELECT q.one FROM (SELECT 1 AS one) q;
```

To use a SELECT expression statement as a value in another expression, it is required to place parentheses around that expression. The following is incorrect.

```
SELECT SELECT 1
```

The following statements are correct.

```
SELECT (SELECT 1); -- return 1
SELECT (SELECT 2)*(SELECT 3) -- return 6
```

Case Expression and SQRT

```
SELECT E.Eid,
    CASE WHEN E.Salary > 100000 THEN 'high';
         WHEN E.Salary < 10000 THEN 'low'
    ELSE 'medium' END AS SalaryRange
FROM Employee E;
```

Find the pairs of points that are within distance 3.

```
SELECT P1.Pid AS P1, P2.Pid AS P2
FROM Point P1, Point P2
WHERE sqrt(power(P1.X-P2.X,2)+power(P1.Y-P2.Y,2)) ãĺď 3
```

Raise the salary of an employee by 5% provided that the raise is less than $1000.

```
(SELECT E.Eid, E.Salary * 1.05 AS NewSalary FROM Employee E
 WHERE E.Salary * 0.05 < 1000)
UNION
(SELECT E.Eid, E.Salary AS NewSalary FROM Employee E
 WHERE E.Salary * 0.05 >= 1000)
```

Functions Returning Sets

A function can also return a set of tuples (relation). The return type of such as function is specified using the RETURN SET record clause. Let Pair(x int, y int) be a relation of pairs:

```
CREATE FUNCTION sum_and_product(OUT sum int, OUT product int)
RETURNS SETOF RECORD AS
$$
SELECT P.x+P.y, P.x*P.y FROM Pair P;
$$ LANGUAGE SQL;
```

Functions returning a record return a single record even if the body of the function computes a set of record. Leads to **non-deterministic** effects.

Quiz 1: Relational Model & Intro to SQL

(1) The simplest form of an SQL query is as follows: SELECT list of **components** of **tuples** FROM list of tuple variables associated with **relations** WHERE condition on **components**.

(2) Provided we have the following two schemas: Student(sid, sname, major, year) and Enroll(sid, cno, grade). How many tuple variables are required to execute the below query optimally: Find the name of each student who is enrolled in three courses? **2**.

(3) Which of the following is an invalid Relational model:

- Movies(title, year, length, genre)
- Movies(<u>title</u>, <u>year</u>, length, <u>genre</u>)
- **Movies(title: string, year:integer, length:integer, genre:Array[int])**
- Movies(title:string, year:integer, length:integer, genre:string)

Non-Atomic Attribute (genre): In a relational model, each attribute should contain atomic (indivisible) values. However, the genre attribute is defined as an array (or list) of integers. **This is not atomic and violates the first normal form (1NF), which requires that each field contains only a single value.** Instead, genres should be stored in a separate table with a many-to-many relationship to the movies.

(4) Which of the following is FALSE about the relational model?

- A tuple is a set of attributes.
- **A relation is a list of tuples.**
- The model uses a language that is consistent with predicate logic.
- **Duplicate tuples may occur in a relation.**

(5) cname is the primary key in Company.pid is the primary key in Person.cname is the foreign key in Person referencing the primary key cname in Company. If cascading delete is **disallowed**, deletion of ('Apple', 'Cupertino') will delete the tuple in relation Company. **False**.

Quiz 2: Relational Model, SQL & TRC

(1) Consider the sentence $\exists x \text{girl}(x)$ where the variable x comes from the domain of persons. Which of the following is the most appropriate interpretation of this sentence. **There exists a person who is a girl.**

(2) The following TRC expression represents a constraint: $\exists S(\text{Skill}(S) \wedge S.\text{skill} = \text{'AI'})$.

(3) Which of the following is NOT correct for Tuple Relational Calculus (TRC)?

- TRC applies on relational model
- TRC is a declarative language
- **TRC is a procedural language**
- Variables in TRC range over tuples

Quiz 3: TRC, RA

(1) Which of the following statement is True:

- **Rewrite rules can transform a RA expression into another equivalent RA expression that is more efficient to evaluate.**
- RA is a declarative language.
- RA and SQL express the same queries with aggregation functions.

(2) $\{()|\exists w \in \textbf{worksFor} \wedge w.\textbf{cname} = \textbf{'Google'}\}$. Which of the statements correspond to the above TRC. **Some person works for Google.**

(3) Which of the following can be used in SQL to represent an existential quantifier in TRC? 1. **EXISTS**; 2. **SOME**; 3. ANY; 4. ALL.

(4) Consider the RA expression: $\pi_{\text{Person.pname, worksFor.salary}}(\sigma_{\text{Person.city}='Seattle'}(\text{Person} \bowtie \text{worksFor}))$. Which of the following statements best describes the operation? **It retrieves the names of all people living in Seattle along with their salaries, but only if they work for a company.**

(5) Which of the following is FALSE about query plans

- The leaf nodes of a query plan are typically relational tables.
- **Projection pushdown (moving projection operators closer to leaf nodes) always yields an equivalent query plan.**
- For query plans with only joins (one the same key), changing the join orders always yields an equivalent query plan.
- Query plans depict the logical execution sequence of a query

(6) Match the following logically equivalent pairs.

- $\neg(F \to G) \iff F \wedge \neg G.$
- $\exists t(F(t) \wedge G(t)) \iff \neg\forall t(F(t) \to \neg G(t))$
- $\forall t(F(t) \to G(t)) \iff \neg\exists t(F(t) \wedge \neg G(t))$
- $F \to (G \to H) \iff (F \wedge G) \to H$

Quiz 4: Views, Joins

(1) What is the output of the following recursive query? **Syntax Error** since the base case fails.

```
WITH RECURSIVE t(no1) AS (
    SELECT 1 FROM t
    UNION ALL
    SELECT no1+1 FROM t WHERE n < 5
)
SELECT sum(no1) FROM t;
```

(2) Select all the types which are not Regular Join: CROSS JOIN; JOIN; NATURAL JOIN; **[NOT] EXISTS**; INNER JOIN; **[NOT] IN**.

(3) Consider RA expressions E1 and E2 with schemas E1(A1, A2, ..., An), and E2(B1, B2, ..., Bn). Does the following equivalence hold (**False**). $E_1 \cap E_2 = E_1 \bowtie E_2$

(4) The view StudentSkill_AI is defined on the studentSkill relation as follows:

```
CREATE VIEW StudentSkill AS
SELECT Ss.sid FROM StudentSkill Ss WHERE Ss.Skill = 'AI';
```

Assuming sid=1011 exists in studentSkill, does the following DELETE statement deletes the tuple (1011, 'AI') from the studentSkill relation? **False**

```
DELETE FROM StudentSkill_AI WHERE sid = 1011;
```

(5) In SQL, how can semijoins be simulated?

- Using the NATURAL JOIN operator.
- Using the IN predicate.
- Using the UNION operator.
- **Both A and B.**

(6) What is the output of the following recursive query? **120**

```
WITH RECURSIVE t(no1, no2) AS (
    VALUES (1, 1) UNION ALL
    SELECT no1+1, (no1+1) + no2 FROM t WHERE n < 5
)
SELECT max(no2) FROM t;
```

(7) Query evaluation precedes view expansion. **False**

(8) Which of the following is FALSE about materialized views

- CREATE MATERIALIZED VIEW is similar to CREATE TABLE
- Materialized view need to be updated after every base table insert
- REFRESH MATERIALIZED VIEW is used to update a materialized view
- **Materialized view query need to be expanded at every evaluation**

(9) Mark the statement that is False about Natural Joins:

- Natural Join is associative.
- SQL supports a Natural Join operator.
- **R1 Natural Join R2 is a subset of R1.**
- A Natural Join with no common attributes resembles a Cartesian Product.

Quiz 5: Joins and Semijoins

(1) Which of the following is always true:

- $E_1 \ltimes E_2 \subseteq E_1 \cap E_2$
- $E_1 \ltimes E_2 \subseteq E_1$

(2) Consider the following equality $E_1 \ltimes E_2 = E_1 \bowtie \pi_c(E_2)$. Which of the following represents the schemas of E1 and E2 for the above equality to hold

- **E1(a, b, c), E2(a, b, c)**
- **E1(a, b, c), E2(d, c)**

(3) Consider $E_1 \ltimes E_2$. Which of the following represents the schemas of E1 and E2 for the semijoin to run in linear time.

- **E1(a, b, c), E2(a, b, c)**
- **E1(a, b, c), E2(d, c)**

(4) Consider the schemas Student(sid, sname, age) and Enroll(sid, cno, grade). Finding the names of students along with the courses they are taking can be done using a semijoin. **False**

(5) Consider the following equality $E_1 \ltimes E_2 = E_1 - E_2$. Which of the following represents the schemas of E1 and E2 for the above equality to hold

- **E1(a, b, c), E2(a, b, c)**
- E1(a, b, c), E2(d, c)

(6) Select the options that are FALSE regarding the SEMI JOIN.

- **SEMI JOIN is associative.**
- **SEMI JOIN always returns the same result as a NATURAL JOIN.**
- SEMI JOIN can be performed by the "IN" predicate in an SQL query.
- SEMI JOIN only returns columns from the left relation.

(7) Mark the statement that is False about Natural Joins:

- SQL supports a Natural Join operator.
- A Natural Join with no common attributes resembles a Cartesian Product.
- **R1 Natural Join R2 is a subset of R1.**
- Natural Join is associative.

(8) What of the SQL Keywords support Regular Join: **[NOT] IN**; **[NOT] EXISTS**; **INNER JOIN**; **JOIN**; **CROSS JOIN**; **NATURAL JOIN**.

Quiz 6: SQL Expressions and Aggregate Functions

(1) Which of the following is not a valid usage of SQL expression?

- SELECT q.one from (select 1 as one) as q
- **SELECT SELECT 1**
- SELECT q.one from (select 1 as one) as q where (select (select 1)) = (select 2)
- SELECT (SELECT 1)

(2) SQL expression can be a single constant, variable, column or a scalar function but must always evaluate to a single value. **True**, a tuple is a value as well.

(3) Provided the same input values and database state, AVG(), RANK(), GETDATE() are: **Deterministic, deterministic, non-deterministic**.

(4) Provided that the following function is created successfully,

```
CREATE FUNCTION return_same(input_text text)
RETURNS SETOF RECORD AS
$$
SELECT input_text AS one, input_text AS two
$$ LANGUAGE SQL STABLE;
```

running the second query returns **Error**.

```
Select from return_same('example')
```

(5) Which of the following Boolean query checks for foreign key constraint?

```
SELECT NOT EXISTS (
    SELECT E.Sid FROM Enroll E WHERE E.Sid NOT IN (
        SELECT S.Sid FROM Student S))
```

(6) Select all that are collections: **Graphs, Vectors, Maps, Relations.**

(7) Functions SUM(), AVG() and COUNT() returns NULL when queried on an empty table. **False**, SUM() returns NULL, COUNT() returns 0.

(8) The following query returns the number of courses taken by each student for all the students in the Student relation. **False**

```
SELECT S.Sid, Count(E.Cno)
FROM Student S, Enroll E
WHERE S.Sid = E.Sid
GROUP BY (S.Sid)
```

(9) Provided the following Numbers table:

	num_1	num_2
1	1	0
2	2	3
3	1	3
4	4	0

Running the below will result in? **Error**, see **Week 6, Example of GROUP BY**.

```
SELECT n.num_1 FROM Numbers n GROUP BY (n.num_1 + n.num_2)
```

(10) Partitioning by subject, ordering by score: using RANK() and Dense_Rank() provides the same result? **Not necessarily.**

- Use RANK() if gaps in ranking are acceptable or if you need to know how many rows are tied at each rank. RANK() will return (1, 1, 3, 4).
- Use DENSE_RANK() if you want consecutive ranking numbers with no gaps, even when there are ties. DENSE_RANK() will return (1, 1, 2, 3).