Structured Query Language (SQL)

CSE462 Database Concepts

Demian Lessa/Jan Chomicki

Department of Computer Science and Engineering
State University of New York, Buffalo

Fall 2013

Introduction

What is Structured Query Language (SQL)?

- SQL is the most common DML/DDL for relational databases.
- Virtually all RDBMS vendors implement SQL based on standards.
  - SQL-86, SQL-92 (SQL2), SQL-99 (SQL3), SQL:2003, SQL:2008, SQL:2011.
  - They also provide proprietary extensions (eg, PL/SQL, T-SQL).
- SQL's DML is conceptually very similar to RA.
- SQL standardizes commands beyond DML/DDL (cf, chapters 7 and 9).

Introduction

How do SQL and RA compare?

- RA is a conceptual query language, SQL is implemented by RDBMSs.
- RA's semantics is defined on sets, SQL's on bags.
- RA does not support `NULL`s, SQL does.
- Neither RA nor SQL can specify order within sets of tuples.
  - However, top-level SQL results can be ordered (not subqueries).
- SQL is relationally complete.
  - All RA queries are expressible in SQL.
  - Not all SQL queries are expressible in RA.
  - RA cannot express aggregation or recursion.
  - RA cannot handle duplicates or `NULL`s.

Select-Project (SP) Queries

```
SELECT selectList
FROM fromList
[WHERE condition]
[ORDER BY orderList];
```

Syntax:

- **SELECT**: lists attributes/expressions to project.
  - Aliasing: "`studioName AS SName`" renames attribute `studioName` to `SName`.
  - Wildcard: "`*`" returns all attributes from all relations, "`R.*`" all attributes in `R`.
- **FROM**: lists relation expressions to which the query refers.
  - Tuple variables: "`relexpr AS T`" assigns tuple variable `T` to `relexpr`.
- **WHERE**: defines a boolean expression that tuples must satisfy.
  - SQL and RA: operands, comparison operators, boolean connectives.
  - SQL only: `||`, `LIKE`, `COALESCE`, `NULLIF`, `CASE`, etc.
- **ORDER BY**: lists attributes/expressions on which tuples are sorted.
  - **ASC** (**DESC**): attribute values sorted lowest (highest) first. Default is **ASC**.
  - Ties on the 1st attribute are broken using the 2nd attribute, etc.

Select-Project (SP) Queries

```
SELECT selectList
FROM fromList
[WHERE condition]
[ORDER BY orderList];
```

Semantics:

1. Compute the Cartesian product of the relations in `fromList`.
2. Discard all tuples obtained in (1) that do not satisfy `condition`.
3. For each tuple obtained in (2), return one tuple for the `selectList`.
   - Evaluate all projected expressions, eg, "`Price * Qty AS Total`".
4. Sort tuples obtained in (3) based on `orderList`.
   - Sorting is a blocking operation!

Note: practical query evaluation is quite efficient!

Example: SP Queries

Find all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Sample Output

title	year	length	genre	studioName	producerC#
Pretty Woman	1990	119	comedy	Disney	999

Example: SP Queries

Find the title and length of all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title,
  length
FROM
  Movies
WHERE
  studioName = 'Disney' AND
  year = 1990;
```

Sample Output

title	length
Pretty Woman	119

Example: SP Queries

Find the name and duration of all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title AS name,
  length AS duration
FROM
  Movies
WHERE
  studioName = 'Disney' AND
  year = 1990;
```

Sample Output

name	duration
Pretty Woman	119

Example: SP Queries

Find the title and length, in minutes and hours, of all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title,
  length AS minutes,
  length/60.0 AS hours
FROM
  Movies
WHERE
  studioName = 'Disney' AND
  year = 1990;
```

Sample Output

name	minutes	hours
Pretty Woman	119	1.98334

Example: SP Queries

Find the title of all MGM movies produced after 1970 or that run for less than 90 minutes.

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title
FROM
  Movies
WHERE
  studioName = 'MGM' AND
  (year > 1970 OR length < 90);
```

Sample Output

title
Shaft
Shaft's Big Score
The Champ

Expressions Involving Strings

- Comparisons observe lexicographic order.
- Concatenation of two or more strings is provided by the `||` operator.
  - Syntax: `str1 || str2 [|| str3 [...]]`
  - Semantics: returns the concatenated string if all strings are non-`NULL`, or `NULL` otherwise.
- The SQL standard defines other string functions.
  - Computing/searching/replacing substrings, trimming, padding, etc.
- Pattern matching is supported via the `LIKE` operator.
  - Syntax: `operand [NOT] LIKE pattern`, where
    - `pattern` is a string possibly containing wildcards _ and `%`.
  - Semantics:
    - _ matches any single character.
    - `%` matches zero or more characters.
    - Any other character matches itself exactly once.

Example: SP Queries

We remember a movie "Star *something*" in which *something* has four letters. What movies could it be?

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title
FROM
  Movies
WHERE
  title LIKE 'Star ____';
```

Sample Output

title
Star Wars
Star Trek

Example: SP Queries

What are the titles of the movies with a possessive (’s) in their titles?

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title
FROM
  Movies
WHERE
  title LIKE '%''s%';
```

Sample Output

title
Logan’s Run
Alice’s Restaurant

Expressions Involving `NULL`

What is `NULL`?

- SQL allows attributes to be assigned `NULL`.
- `NULL` may be interpreted to represent values that:
  - are unknown or missing;
  - are inapplicable;
  - should not be displayed.
- Semantics:
  - `NULL` cannot be used explicitly as an arithmetic/boolean operand.
  - `value op NULL` evaluates to `NULL`, `op` $\in \{+, -, *, /, ||\}$.
  - `value cop NULL` evaluates to `UNKNOWN`,
    `cop` $\in \{<, <=, =, >=, >, <>, \texttt{LIKE}\}$.
  - `value IS NULL` evaluates to `TRUE` when `value` is assigned `NULL`.
  - `value IS NOT NULL` evaluates to `TRUE` when `value` is not assigned `NULL`.
- What is the result of "`value = NULL`"? and "`value <> NULL`"?

Expressions Involving `NULL`

Three-Valued Logic: Truth Values

X	Y	X AND Y	X OR Y	NOT X
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

NULL Handling Functions

The COALESCE function returns the first non-NULL value from a list.

- Syntax: `COALESCE(val1, ..., valn)`
- If all values in the list are NULL, returns NULL.

The NULLIF function returns NULL if two values are equal.

- Syntax: `NULLIF(val1, val2)`

CASE Expressions

```
-- Simple
CASE s_expr
  WHEN t_expr1 THEN s_expr1;
  ...
  WHEN t_exprM THEN s_exprM;
  [ELSE s_exprN;]
END
```

Semantics:

- `s_expr*` and `t_expr*` are scalar expressions.
- Tests `s_expr` for equality against each `t_expr1, ..., t_exprM`, in order.
- Returns `s_exprk` for the first *k* such that `s_expr = t_exprk`.
- If no such *k* exists, returns `s_exprN` if avaliable, or `NULL` otherwise.

CASE Expressions

```
-- Searched
CASE
  WHEN b_expr1 THEN s_expr1;
  ...
  WHEN b_exprM THEN s_exprM;
  [ELSE s_exprN;]
END
```

Semantics:

- `s_expr*` are scalar expressions, `b_expr*` are boolean expressions.
- Tests each `b_expr1, ..., b_exprM`, in order.
- Returns `s_exprk` for the first *k* such that `b_exprk` is `TRUE`.
- If no such *k* exists, returns `s_exprN` if avaliable, or `NULL` otherwise.

Example: SP Queries

What does the query below return?

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title
FROM
  Movies
WHERE
  length <= 120 OR
  length > 120;
```

Answer

All tuples with non-`NULL` lengths, therefore, the query does not output the entire relation for all database instances.

Example: SP Queries

What does the query below return?

`Movies(title, year, length, genre, studioName, producerC#)`

SQL Query

```
SELECT
  title
FROM
  Movies
WHERE
  length <= 120 OR
  length > 120 OR
  length IS NULL;
```

Answer

All `Movies` tuples, for all database instances.

Select-Project-Join (SPJ) Queries

```
SELECT selectList
FROM fromItem [, fromItem [, ...]]
[WHERE condition]
[ORDER BY orderList];
```

Syntax:

- `fromItem` is a table (`tabExpr`) or join expression (`joinExpr`).
  - `tabExpr` is a table name with an optional tuple variable.
  - `joinExpr` has the form:
    `fromItem joinType JOIN fromItem [ON joinCond]`.
  - `joinType` identifies the type of the join operation:
    - Cartesian product: **CROSS**.
    - Inner join (default): [**NATURAL**] [**INNER**].
    - Outer join: [**NATURAL**] (**LEFT** | **RIGHT** | **FULL**) [**OUTER**].
  - `joinCond` is a boolean expression specifying the match criteria.
    - Must be omitted ('**ON**' keyword as well) for **CROSS** and **NATURAL** joins.

Select-Project-Join (SPJ) Queries

Outer Joins.

- Retain dangling tuples that fail to match the join condition.
  - Dangling tuples are padded with `NULL`s for their missing components.
- Variants and their dangling tuple retention behavior:
  - **`LEFT JOIN`**: dangling tuples from the left operand of the join.
  - **`RIGHT JOIN`**: dangling tuples from the right operand of the join.
  - **`FULL JOIN`**: dangling tuples from both left and right operands of the join.
- All joins above are theta joins and require a join condition.
  - Their natural join variants do not require the join condition.

Example: SPJ Queries

Find the name of the producer of ‘Star Wars’.

Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)

SQL Query

```
SELECT
  name
FROM
  Movies, MovieExec
WHERE
  producerC# = cert# AND
  title = ‘Star Wars’;
```

Example: SPJ Queries

Find the name of the producer of ‘Star Wars’.

`Movies(title, year, length, genre, studioName, producerC#)`
`MovieExec(name, address, cert#, netWorth)`

SQL Query (join syntax)

```
SELECT
  name
FROM
  Movies
  JOIN MovieExec ON (producerC# = cert#)
WHERE
  title = ‘Star Wars’;
```

Example: SPJ Queries

List the unique pairs of movie stars sharing an address.

`MovieStar(name, address, gender, birthDate)`

SQL Query (join syntax)

```
SELECT
  S1.name, S2.name
FROM
  MovieStar AS S1
  INNER JOIN MovieStar AS S2 ON (S1.address = S2.address
                                 AND S1.name < S2.name);
```

Observation

The second part of the join condition guarantees the uniqueness of pairs. In particular, a star is not paired with itself, and every pair of matching stars appears only once in the result, namely, in alphabetical order.

Example: SPJ Queries

For each movie, display its title and the name of its producer. Include all movies, even those in which the producer is missing.

`Movies(title, year, length, genre, studioName, producerC#)`
`MovieExec(name, address, cert#, netWorth)`

SQL Query

```
SELECT
  title, name AS producer
FROM
  Movies, MovieExec
WHERE
  producerC# = cert# OR
  producerC# IS NULL;
```

Observation

What is the user's intention in this query? Is the test for `NULL` in the `producerC#` component an independent selection condition or part of the join criteria?

Example: SPJ Queries

For each movie, display its title and the name of its producer. Include all movies, even those in which the producer is missing.

`Movies(title, year, length, genre, studioName, producerC#)`

`MovieExec(name, address, cert#, netWorth)`

SQL Query (join syntax)

```
SELECT
  title, name AS producer
FROM
  Movies
  LEFT JOIN MovieExec ON (producerC# = cert#);
```

Example: SPJ Queries

Compute the three natural outer joins of the given relation instances.

MovieStar(name, address, gender, birthDate)

name	address	gender	birthdate
Mary Tyler Moore	Maple St.	F	9/9/99
Tom Hanks	Cherry Ln.	M	8/8/88

MovieExec(name, address, cert#, netWorth)

name	address	cert#	networth
Mary Tyler Moore	Maple St.	12345	$100M
George Lucas	Oak Rd.	23456	$200M

Example: SPJ Queries

Natural Left Outer Join

name	address	gender	birthdate	cert#	networth
Mary Tyler Moore	Maple St.	F	9/9/99	12345	$100M
Tom Hanks	Cherry Ln.	M	8/8/88	NULL	NULL

`MovieStar(name, address, gender, birthDate)`

name	address	gender	birthdate
Mary Tyler Moore	Maple St.	F	9/9/99
Tom Hanks	Cherry Ln.	M	8/8/88

`MovieExec(name, address, cert#, netWorth)`

name	address	cert#	networth
Mary Tyler Moore	Maple St.	12345	$100M
George Lucas	Oak Rd.	23456	$200M

Example: SPJ Queries

Natural Right Outer Join

name	address	gender	birthdate	cert#	networth
Mary Tyler Moore	Maple St.	F	9/9/99	12345	$100M
George Lucas	Oak Rd.	NULL	NULL	23456	$200M

`MovieStar(name, address, gender, birthDate)`

name	address	gender	birthdate
Mary Tyler Moore	Maple St.	F	9/9/99
Tom Hanks	Cherry Ln.	M	8/8/88

`MovieExec(name, address, cert#, netWorth)`

name	address	cert#	networth
Mary Tyler Moore	Maple St.	12345	$100M
George Lucas	Oak Rd.	23456	$200M

Example: SPJ Queries

Natural Full Outer Join

name	address	gender	birthdate	cert#	networth
Mary Tyler Moore	Maple St.	F	9/9/99	12345	$100M
Tom Hanks	Cherry Ln.	M	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	$200M

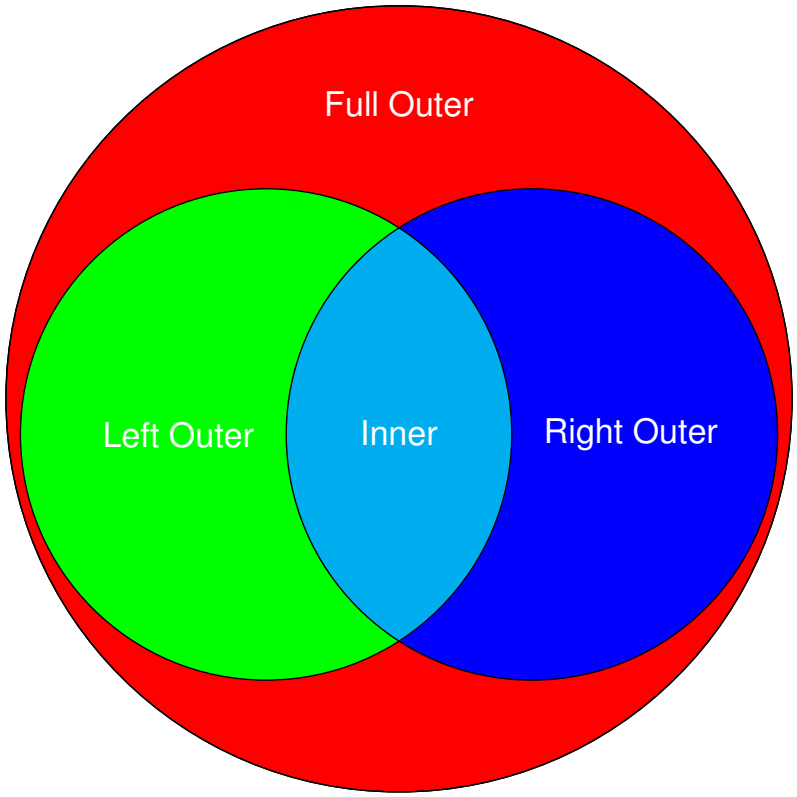
MovieStar(name, address, gender, birthDate)

name	address	gender	birthdate
Mary Tyler Moore	Maple St.	F	9/9/99
Tom Hanks	Cherry Ln.	M	8/8/88

MovieExec(name, address, cert#, netWorth)

name	address	cert#	networth
Mary Tyler Moore	Maple St.	12345	$100M
George Lucas	Oak Rd.	23456	$200M

SPJ Queries



Select-Project-Join (SPJ) Queries

Advanced Joins.

- Semijoin.
  - Join between relations that returns tuples only from one of them.
  - Left semijoin returns tuple from the tuple on the left of the join.
  - Right semijoin returns tuple from the tuple on the left of the join.
  - Typically implemented using correlated subqueries (more later).
- Anti-joins (including anti-semijoins).
  - Join between relations that returns tuples if the join condition is not satisfied.
  - For instance, a left anti-semijoin of R and S on condition φ returns tuples of R (left/semi) that do not join with any tuples of S on φ (anti).
  - Typically implemented as a combination of an outer join with one or more `NULL` checks in the body.

Set and Bag Operations

```
queryExpr
setOp [ALL] queryExpr
[setOp [ALL] queryExpr [setOp [ALL] ...]]
[ORDER BY orderList];
```

Syntax:

- `queryExpr` is a simple SQL query.
- `setOp` is one of the following operators:
  - **`UNION`**, for set union.
  - **`INTERSECT`**, for set intersection.
  - **`EXCEPT`**, for set difference.
  - Use **`ALL`** for the respective bag operation.
- The operations do not distinguish `NULL`s from different tuples.

Example: Set and Bag Operations

What does the query below compute?

MovieStar(name, address, gender, birthDate)

MovieExec(name, address, cert#, netWorth)

SQL Query

```
 SELECT name, address
 FROM MovieStar
 WHERE gender = 'F'
INTERSECT
 SELECT name, address
 FROM MovieExec
 WHERE netWorth > 10,000,000;
```

Answer

The set of female movie stars that are also movie executives and whose net worth is above 10M.

Example: Set and Bag Operations

What does the query below compute?

MovieStar(name, address, gender, birthDate)

MovieExec(name, address, cert#, netWorth)

SQL Query

```
 SELECT name, address
 FROM MovieStar
EXCEPT
 SELECT name, address
 FROM MovieExec;
```

Answer

The set of movie stars that are not movie executives.

Example: Set and Bag Operations

What does the query below compute?

MovieStar(name, address, gender, birthDate)

MovieExec(name, address, cert#, netWorth)

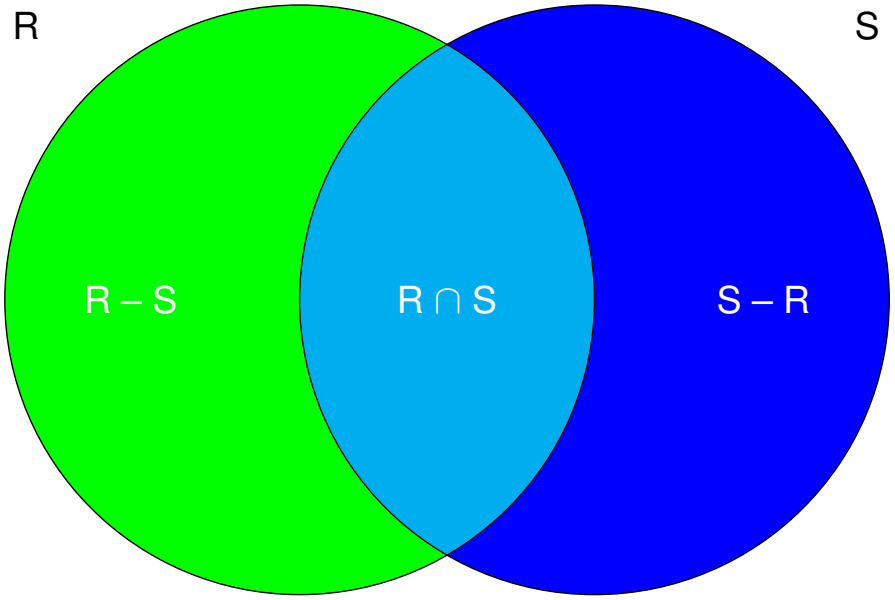
SQL Query

```
 SELECT name, address
 FROM MovieStar
 WHERE gender = 'F'
UNION
 SELECT name, address
 FROM MovieExec
 WHERE netWorth > 10,000,000;
```

Answer

The set of movie industry persons that are either female stars or executives whose net worth is above 10M or both.

Set and Bag Operations



Subqueries

A subquery is a query that appears within another query.

- As an operand of a set or bag operation.
- As a derived relation, nested within the **FROM** clause:
  - Must be within parenthesis and be assigned a tuple variable.
  - Cannot access attributes of other relations in the **FROM** clause.
- As a scalar subquery, wherever a single scalar is required:
  - Must return exactly one tuple.
- As a relation operand, wherever a relation is required:
  - Testing for set (non-)emptyness: [**NOT**] **EXISTS**.
  - Testing for set (non-)membership: [**NOT**] **IN**.
  - Comparisons using **ANY**
- Correlated subqueries.
  - Nested in clauses other than **FROM**.
  - May access attributes values from outer queries.
- If used as operands, schemas must be compatible with operations.

Subqueries

Set operations involving subqueries.

- **`EXISTS`** `R`
  - `TRUE` if the subquery `R` is not empty.
- `value` **`IN`** `R`
  - `TRUE` if `value` belongs to `R`.
  - `value` is a tuple with the same number of components as `R`.
  - Typically, `value` is a scalar and `R` a unary relation.
- `value cop` **`ALL`** `R`, `cop` $\in \{<, \leq, =, >, \geq, <>, \texttt{LIKE}\}$.
  - `TRUE` if for every tuple `t` $\in$ `R`, `value cop t` holds.
- `value cop` **`ANY`** `R`, `cop` $\in \{<, \leq, =, >, \geq, <>, \texttt{LIKE}\}$.
  - `TRUE` if for some tuple `t` $\in$ `R`, `value cop t` holds.
- Negated forms.
  - **`NOT EXISTS`** `R`
  - **`NOT`** `value cop (`**`ALL`**` | | `**`ANY`**`) R`
  - `value` **`NOT IN`** `R`

Example: Subqueries

Consider relations R(A) and S(A) and explain the queries below.

SQL Query

```
-- Q1
SELECT A
FROM R
WHERE A <> ALL(SELECT A FROM S);

-- Q2
SELECT A
FROM R
WHERE A >= ALL(SELECT A FROM R);

-- Q3
SELECT A
FROM R
WHERE A >= ALL(SELECT A FROM S);
```

Answer

- Q1: left anti-semijoin
- Q2: maximal value
- Q3: dominating subset

Example: Subqueries

Consider relations R(A) and S(A) and explain the queries below.

SQL Query

```
-- Q1
SELECT A
FROM R
WHERE A = ANY(SELECT A FROM S);

-- Q2
SELECT A
FROM R
WHERE A <> ANY(SELECT A FROM R);

-- Q3
SELECT A
FROM R
WHERE A >= ANY(SELECT A FROM S);
```

Answer

- Q1: left semijoin
- Q2: non-singleton
- Q3: non-minimal value

Example: Subqueries

Consider relations R(A) and S(A) and explain the queries below.

SQL Query

```
-- Q1
SELECT A
FROM R
WHERE A IN
  (SELECT A FROM S);

-- Q2
SELECT A
FROM R
WHERE A NOT IN
  (SELECT A FROM S);
```

Answer

- Q1: left semijoin
- Q2: left anti-semijoin

Example: Subqueries

Consider relations R(A) and S(A) and explain the queries below.

SQL Query

```
-- Q1
SELECT A
FROM R
WHERE
  EXISTS(SELECT A FROM S
         WHERE S.A=R.A);

-- Q2
SELECT A
FROM R
WHERE
  NOT EXISTS(SELECT A FROM S
             WHERE S.A=R.A);
```

Answer

- Q1: left semijoin
- Q2: left anti-semijoin

Example: Subqueries

Find all producers of Harrison Ford's movies.

Movies(title, year, length, genre, studioName, producerC#)

MovieExec(name, address, cert#, netWorth)

StarsIn(movieTitle, movieYear, starName)

Answer

```
SELECT name
FROM MovieExec
WHERE cert# IN
       (SELECT producerC#
        FROM Movies
        WHERE (title, year) IN
               (SELECT movieTitle, movieYear
                FROM StarsIn
                WHERE starName = 'Harrison Ford'));
```

Subqueries

- Observations
  - Analize queries starting from the innermost subqueries.
  - Most queries can be rewritten without subqueries. (How?)
  - Simple subqueries are evaluated just once.
  - Correlated subqueries are evaluated once per assignment to some term in the subquery coming from a tuple variable outside the subquery.
  - You may define a constant subquery using the following syntax:
    (**VALUES** (V_{1_1}, ..., V_{1_k}), ..., (V_{n_1}, ..., V_{n_k})) **AS** S

Example: Subqueries

Find the title and year of any movie that has a remake. If a movie has one remake, list only the original. If it has $n > 1$ remakes, list all but the latest.

`Movies(title, year, length, genre, studioName, producerC#)`

Answer

```
SELECT title, year
FROM Movies AS Old
WHERE year < ANY
      (SELECT year FROM Movies
       WHERE title = Old.title);
```

Required

- Read sections 6.1 to 6.3 from chapter #6.
- Go over exercises 6.3.1, 6.3.3, 6.3.7, 6.3.8 from chapter #6.

Duplicate Elimination

```
SELECT [DISTINCT] selectList
FROM fromItem [, fromItem [, ...]]
[WHERE condition]
[ORDER BY orderList];
```

Semantics:

1. Compute the Cartesian product of the relations in `fromList`.
2. Discard all tuples obtained in (1) that do not satisfy `condition`.
3. For each tuple obtained in (2), return one tuple for the `selectList`.
4. If **`DISTINCT`** is specified, eliminate duplicate tuples obtained in (3).
5. Sort tuples obtained in (4) based on `orderList`.

Grouping and Aggregation

Grouping and Aggregation.

- Sometimes it is useful to partition tuples of a relation based on the values of one or more of their attributes.
- Any number of computations may then be carried out over the collection of tuples in each partition.
- Each computation is an aggregate of the values of the individual tuples in the partition, such as a sum, maximum, minimum, average, etc.

Grouping and Aggregation

```
SELECT [DISTINCT] selectList
FROM fromItem [, fromItem [, ...]]
[WHERE condition]
[GROUP BY groupList]
[HAVING aggCondition]
[ORDER BY orderList];
```

Syntax:

- `selectList` must only contain:
  - grouped attributes/expressions, and
  - aggregate functions/expressions.
- **`GROUP BY`**: lists attributes/expressions on which tuples are partitioned.
  - All non-aggregate expressions in `selectList` must be in `groupList`.
- **`HAVING`**: boolean expression which aggregate tuples must satisfy.
  - Consists of literals, grouped attributes, and aggregate expressions.

Grouping and Aggregation

```
SELECT [DISTINCT] selectList
FROM fromItem [, fromItem [, ...]]
[WHERE condition]
[GROUP BY groupList]
[HAVING aggCondition]
[ORDER BY orderList];
```

Semantics:

1. Compute the Cartesian product of the relations in `fromList`.
2. Discard all tuples obtained in (1) that do not satisfy `condition`.
3. Partition tuples in (2) on attributes in `groupList`.
4. For each partition obtained in (3),
   - compute aggregates in `selectList` and `aggCondition`,
   - generate one aggregate tuple for each partition, and
   - discard aggregate tuples that do not satisfy `aggCondition`.
5. For each tuple obtained in (4), return one tuple for the `selectList`.
6. If **`DISTINCT`** is specified, eliminate duplicate tuples obtained in (5).
7. Sort tuples obtained in (6) based on `orderList`.

Grouping and Aggregation

Grouping.

- Independent from aggregate computation.
- However, aggregate computation requires grouping.
- If **`GROUP BY`** is omitted, the relation is treated as a single partition.

Aggregate expressions.

- May appear in `selectList`, `aggCondition`, and `orderList`.
- Syntax.
  - **`AGG`**`([`**`DISTINCT`**`] expr)`.
  - **`AGG`** is one of **`SUM`**, **`COUNT`**, **`MIN`**, **`MAX`**, **`AVG`**, **`STDEV`**, etc.
  - Many RDBMSs support user-defined aggregate functions.
- Semantics.
  - Aggregate expressions in the query are computed for each partition.
  - **`AGG`** accumulates `expr` over all (distinct) tuples in each partition.

Grouping and Aggregation

Dealing with NULLs:

- NULL is treated as an ordinary value when grouping.
  - A group may have one or more attributes assigned NULL.
- All aggregate computations ignore NULLs.
  - NULL components do not contribute towards **SUM**, **AVG**, **STDEV**, or **COUNT**.
  - However, all aggregates except **COUNT** return NULL for an empty bag.
- Counting tuples.
  - **COUNT** returns zero for an empty bag.
  - **COUNT**(*) counts all tuples, even if all components are NULL.
  - **COUNT**(A) counts all non-NULL values in column A.
  - **COUNT**(**DISTINCT** A) counts all distinct, non-NULL values in column A.
- Aggregates cannot be composed directly.
  - **AGG2**(**AGG1**([**DISTINCT**] expr)) is not allowed!
  - To achieve this behavior, you must compose queries that compute the respective aggregates.

Example: Grouping and Aggregation

Explain what each query below computes.

`StarsIn(movieTitle, movieYear, starName)`

SQL Queries: Counting

```
-- Q1
SELECT COUNT(*)
FROM StarsIn;

-- Q2
SELECT COUNT(starName)
FROM StarsIn;

-- Q3
SELECT
  COUNT(DISTINCT starName)
FROM StarsIn;
```

SQL Queries: Duplicates

```
-- Q4
SELECT starName
FROM StarsIn;

-- Q5
SELECT DISTINCT starName
FROM StarsIn;

-- Q6
SELECT starName
FROM StarsIn
GROUP BY starName;
```

Example: Grouping and Aggregation

Explain what the query below computes.

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

SQL Queries: Counting

```
SELECT name, SUM(length)
FROM MovieExec
JOIN Movies ON (producerC# = cert#)
GROUP BY name;
```

Answer

For each executive, compute the total number of movie minutes produced.

Example: Grouping and Aggregation

Explain what the query below computes.

Movies(<u>title, year</u>, length, genre, studioName, producerC#)
MovieExec(name, address, <u>cert#</u>, netWorth)

SQL Queries: Counting

```
SELECT name, SUM(length)
FROM MovieExec
JOIN Movies ON (producerC# = cert#)
WHERE netWorth > 10,000,000
GROUP BY name
HAVING MIN(year) >= 1950
ORDER BY name;
```

Answer

For each executive whose net worth is above 10M and whose earliest movie was not produced before 1950, compute the total number of movie minutes produced.

Example: Grouping and Aggregation

Given the relation instance below, what do the given queries compute?

SQL Queries: NULLs

```
-- Q1
SELECT COUNT(*) FROM R;

-- Q2
SELECT A, COUNT(*)
FROM R GROUP BY A;

-- Q3
SELECT A, COUNT(B)
FROM R GROUP BY A;

-- Q4
SELECT A, SUM(B)
FROM R GROUP BY A;
```

Relation $R(A, B)$

A	B
NULL	NULL

Answer

- Q1:
- Q2:
- Q3:
- Q4:

Required

- Read section 6.4 from chapter #6.
- Go over exercises 6.4.1, 6.4.4, 6.4.5, 6.4.8 from chapter #6.

Views

A view is a virtual relation defined as a named query expression.

- Definitions are stored but results are not.
- Definitions may contain references to other views.
- Views may be read-only or updatable.
  - Views names may appear anywhere stored relation names may appear.
  - But only updatable views may appear as targets of update operations.
  - A view definition determines whether a view is updatable.
  - From the user's perspective, updatable views behave exactly like tables.

Views

```
CREATE VIEW viewName[(attrList)] AS
  queryExpr;
```

Syntax:

- `viewName`: view name, unique across tables and views.
- `(attrList)`: optional list of attribute names for the view's schema.
  - If omitted, names are computed from `queryExpr`.
- `(queryExpr)`: valid query without an **`ORDER BY`** clause.

Views

Advantages.

- Views provide an effective abstraction/decoupling mechanism.
  - When table schemas change (data/constraints), applications may break.
  - Views may keep the same schema and update their definition as necessary.
  - Low-level changes to the database become transparent to applications.
- Views may provide precomputed, high-level views of the data.
  - Instead of having users perform joins among several tables. . .
  - Provide views that perform common joins.
- Views can provide computed values, including aggregates.
- Views take little space to store.
  - The database maintains only the definition of a view, not a copy of it result.
- Views may provide extra security.
- Views may limit the degree of exposure of sensitive data.

Example: Views

Create a view named `DisneyMovies` that returns all `Movies` produced by Disney. Do not include the `studioName` attribute in the output. Using the view, write a query that returns all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

Answer

```
CREATE VIEW DisneyMovies AS
  SELECT title, year, length, genre, producerC#
  FROM Movies
  WHERE studioName = 'Disney';

SELECT *
FROM DisneyMovies
WHERE year = 1990;
```

Common Table Expressions (CTE)

A common table expression (CTE) is a temporary named relation obtained from a query expression and defined within a DML statement, usually a complex query.

- Available only during the execution of the statement.
- May contain references to CTEs defined earlier within the statement.
  - Analogous to linear notation of RA.
- Evaluated once per execution of the statement.
  - Even if referenced multiple times by the statement or sibling CTEs.
- Addresses several important use cases:
  - Breaking complex queries into smaller parts.
  - Factoring out common and/or expensive expressions.
- Conceptually, can be thought of as a temporary view.
  - But no need to request the DBA to create a view for you.
- Would be nothing more than syntactic convenience. However,
  - Provides syntax for defining recursive queries!

Common Table Expressions

```
WITH [RECURSIVE]
  cteName1[(attrList1)] AS (queryExpr1)[,
  ...
  cteNameN[(attrListN)] AS (queryExprN)]
queryExpr;
```

Syntax:

- `cteNameK`: unique name across all tables, views, and earlier CTEs.
- `(attrListK)`: optional list of attribute names for the CTE's schema.
  - If omitted, names are computed from `queryExprK`.
- `(queryExprK)`: query referencing tables, views, and earlier CTEs.
- Using **`RECURSIVE`**, CTEs may reference their own name as follows:
  - Query expression: `nrQuery` **`UNION`** `[`**`ALL`**`] rQuery`.
  - `nrQuery`: arbitrary query, but must not reference the CTE.
  - `rQuery`: SPJ query, may reference the CTE once in the **`FROM`** clause.
  - In practice, multiple non-recursive CTEs are allowed.
  - Mutual recursion is (usually) not supported.

Example: Common Table Expressions

Create a CTE `DisneyMovies` that returns all `Movies` produced by Disney. Do not include the `studioName` attribute in the output. Using the CTE, write a query that returns all Disney movies produced in 1990.

`Movies(title, year, length, genre, studioName, producerC#)`

Answer

```
WITH DisneyMovies AS
  (SELECT title, year, length, genre, producerC#
   FROM Movies
   WHERE studioName = 'Disney')

SELECT *
FROM DisneyMovies
WHERE year = 1990;
```

Example: Common Table Expressions

Create a recursive CTE `BossChain` that returns all pairs of employees such that the first one is the boss of the second, either directly or indirectly. Assume that the boss of your boss is also your boss and that the `Boss` relation contains only each employee's immediate boss. Using the CTE, write a query that returns all pairs working in the same department.

`Employee(empId, firstName, lastName, SSN, deptId)`

`Boss(bossId, empId)`

Answer

```
WITH RECURSIVE BossChain AS
  (SELECT bossId, empId FROM Boss  -- base case
   UNION
   SELECT BC.bossId, B.empId  -- B's boss' boss is also B's boss
   FROM BossChain BC JOIN Boss AS B ON (B.bossId = BC.empId))

SELECT BC.*
FROM BossChain BC
JOIN Employee B ON (B.empId = BC.bossId)
JOIN Employee E ON (E.empId = BC.empId AND E.deptId = B.deptId);
```

Required

- Read section 8.1 from chapter #8 and 10.2 in chapter #10.
- Go over exercises 8.1.1 and 8.1.2 from chapter #8.

Data Modification

SQL provides three basic data modification commands:

- `INSERT`: inserts new tuples.
- `UPDATE`: modifies existing tuples.
- `DELETE`: excludes existing tuples.

Insertion

INSERT INTO R[(A$_1$,...,A$_k$)]
VALUES (val$_1^1$,...,val$_k^1$) [, (val$_1^2$,...,val$_k^2$) [, ...]];

Semantics:

1. Each list of values in the **VALUES** clause must provide values for:
   - all attributes in $(A_1, \ldots, A_k)$, in that order, or
   - all attributes of R (in standard order), if the list of attributes for R is omitted.
   - Each list of values must be compatible with the corresponding attribute list of R.
2. If some tuple in the **VALUES** clause violates a constraint in R, the insert fails.
3. Otherwise, inserts each $(v_1^i, \ldots, v_k^i)$ into R with value v_j^i for attribute A_j.
4. Default values are used for missing attributes of R in $(A_1, \ldots, A_k)$.

Insertion

```
INSERT INTO R[(A1,...,Ak)]
  queryExpr;
```

Semantics:

1. The schema of `queryExpr` must be compatible with:
   - the list of attributes $(A_1, \ldots, A_k)$, if provided, or
   - R, if the list of attributes for R is omitted.
2. Computes the result of the `queryExpr`.
3. If some tuple obtained in (2) violates a constraint in R, the insert fails.
4. Otherwise all tuples obtained in (2) are inserted into R.
5. Default values are used for missing attributes of R in $(A_1, \ldots, A_k)$.

Example: Insertion

Explain what the statement below accomplishes.

`StarsIn(movieTitle, movieYear, starName)`

SQL: Insertion

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES ('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

Example: Insertion

Explain what the statement below accomplishes.

`Studio(name)`

SQL: Insertion

```
INSERT INTO Studio(name)
  SELECT DISTINCT studioName
  FROM Movies
  WHERE studioName NOT IN
        (SELECT name FROM Studio);
```

Deletion

```
DELETE FROM R
[WHERE delete_cond];
```

Semantics:

1. If the removal of a tuple obtained in (1) violates a constraint in the database, the deletion fails.
2. Otherwise, deletes all tuples satisfying the (optional) condition.

Example: Deletion

Explain what the statement below accomplishes.

`StarsIn(movieTitle, movieYear, starName)`

SQL: Deletion

```
DELETE FROM StarsIn
WHERE
  movieTitle = 'The Maltese Falcon' AND
  movieYear = 1942 AND starName = 'Sydney Greenstreet';
```

Example: Deletion

Explain what the statement below accomplishes.

`MovieExec(name, address, cert#, netWorth)`

SQL: Deletion

```
DELETE FROM MovieExec
WHERE netWorth < 10,000,000;
```

Update

```
UPDATE R SET
  A_i1 = expr_i1 [,
  A_i2 = expr_i2 [,
  ...]]
[WHERE updateCond];
```

Semantics:

1. Computes the set of tuples from R satisfing `updateCond`.
2. If the update of a tuple obtained in (1) violates a constraint in the database, the update fails.
3. Otherwise, assigns every attribute $A_{i_1}, \ldots, A_{i_n}$ the specified values.

Example: Update

Explain what the statement below accomplishes.

MovieExec(name, address, cert#, netWorth)

Studio(name, presC#)

SQL: Update

```
UPDATE MovieExec SET
  name = 'Pres. ' || name
WHERE cert# IN (SELECT presC# FROM Studio);
```

Required

- Read section 6.5 from chapter #6.
- Go over exercise 6.5.1 from chapter #6.

Data Definition Language

```
CREATE TABLE tableName (
  attName1 attType1 [DEFAULT def1] [constraint1a [...]],
  ...
  attNameN attTypeN [DEFAULT defN] [constraintNa [...]] [,
  tableConstraint1 [, tableConstraint2 [...]]
  ]
);
```

Syntax:

- `tableName`: unique table name.
- `attName`: unique name of attribute.
- `attType`: data type of the attribute.
- `def`: default value of the attribute.
  - When inserting a new tuple, if no value is specified, the default is used.
- `constraint`: column-level constraint, applies to that attribute only.
- `tableConstraint`: table-level constraint, applies to the entire table.

Data Definition Language

```
CREATE TABLE tableName (
  attName1 attType1 [DEFAULT def1] [constraint1a [...]],
  ...
  attNameN attTypeN [DEFAULT defN] [constraintNa [...]] [,
  tableConstraint1 [, tableConstraint2 [...]]
  ]
);
```

Constraints:

1. Column constraints.
   - **NULL**
   - **NOT NULL**
   - **UNIQUE**
   - **PRIMARY KEY**
   - **CHECK** (expression)
   - **REFERENCES** refTable (attList)

Data Definition Language

```
CREATE TABLE tableName (
  attName1 attType1 [DEFAULT def1] [constraint1a [...]],
  ...
  attNameN attTypeN [DEFAULT defN] [constraintNa [...]] [,
  tableConstraint1 [, tableConstraint2 [...]]
  ]
);
```

Constraints:

- Table constraint: `[CONSTRAINT constraintName] constraint`
  - **PRIMARY KEY** (attList)
  - **UNIQUE** (attList)
  - **CHECK** (expression)
  - **FOREIGN KEY** (attList) **REFERENCES** refTable (attList)

Data Definition Language

Primary Key

- At most one per table.
- Fields in the key cannot be NULL.
- Duplicate combinations of fields not allowed.

Data Definition Language

Unique Key

- Any number per table.
- Fields in may be NULL.
- NULL values considered equal for comparison purposes.

Data Definition Language

Foreign Key

- Fields in may be NULL.
- Number and type of constrained fields must match the number and type of referenced field.
- Foreign key field values in each tuple must match the values of the referenced fields in some tuple of the referenced table.

Data Definition Language

Check

- Fields in may be NULL.
- The check expression is a boolean expression involving one or more fields.
- The check is satisfied if the expression evaluates to TRUE or UNKNOWN.

Data Definition Language

```
DROP TABLE tableName;
```

Syntax:

- `tableName`: existing table name.