Fundamentals of Database Systems

[SQL – II]

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2019

1. Data Manipulation
   - Principle Structure
   - Relational Operations
   - Logical Operations
   - Set Operations
   - Other Features

2. Problems

Principle structure of manipulating a table

A typical SQL query for data manipulation appears as follows:

$$\begin{array}{l}\ \texttt{select}\ \ A_1, A_2, \ldots, A_m \\ \texttt{from}\ \ R_1, R_2, \ldots, R_n \\ \texttt{where}\ \ P\,; \end{array}$$

Here, each A_i represents an attribute, each R_i denotes a relation and P is a predicate.

Principle structure of manipulating a table

A typical SQL query for data manipulation appears as follows:

```
 select  $A_1, A_2, \ldots, A_m$
from  $R_1, R_2, \ldots, R_n$
where  $P$;
```

Here, each A_i represents an attribute, each R_i denotes a relation and P is a predicate.

- The `select` clause corresponds to the projection operation of the relational algebra.
- The `from` clause corresponds to the Cartesian-product operation of the relational algebra.
- The `where` clause corresponds to the selection predicate of the relational algebra.

An example

Given the IPL table, the SQL query “`select * from IPL where PoS = 'Shane Watson';`” will yield the following.

YEAR	VENUE	WINNER	PoS
2008	India	Rajasthan Royals	Shane Watson
2013	India	Mumbai Indians	Shane Watson

Relational operations

The following relational operators are available in SQL.

Operator	Description
=	Equal
<> or !=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
`LIKE ...`	Search for a pattern
`BETWEEN ... AND ...`	Between an inclusive range
`IN (..., ..., ...)`	Verify multiple values for an attribute

Note: These operators are used in the `where` clause.

Relational operations – {=, <>/ !=, >, <, >=, <=}

These are the standard ones!!!

Relational operations – Like

Like helps to perform the pattern matching operation on strings. The '%' and '_' are used to match any substring and any character, respectively.

Relational operations – Like

Like helps to perform the pattern matching operation on strings. The ‘%’ and ‘_’ are used to match any substring and any character, respectively.

“`select * from IPL where PoS like ‘%ell’;`” will yield

YEAR	VENUE	WINNER	PoS
2014	India, UAE	Kolkata Knight Riders	Glenn Maxwell
2015	India	Mumbai Indians	Andre Russell

Relational operations – Like

Like helps to perform the pattern matching operation on strings. The ‘%’ and ‘_’ are used to match any substring and any character, respectively.

“`select * from IPL where PoS like ‘%ell’;`” will yield

YEAR	VENUE	WINNER	PoS
2014	India, UAE	Kolkata Knight Riders	Glenn Maxwell
2015	India	Mumbai Indians	Andre Russell

“`select * from IPL where PoS like ‘S%a____’;`” will yield

YEAR	VENUE	WINNER	PoS
2008	India	Rajasthan Royals	Shane Watson
2012	India	Kolkata Knight Riders	Sunil Narine
2013	India	Mumbai Indians	Shane Watson
2018	India	Chennai Super Kings	Sunil Narine

Relational operations – Between-And

“`select * from IPL where YEAR between 2013 and 2017;`” will yield

YEAR	VENUE	WINNER	PoS
2013	India	Mumbai Indians	Shane Watson
2014	India, UAE	Kolkata Knight Riders	Glenn Maxwell
2015	India	Mumbai Indians	Andre Russell
2016	India	Sunrisers Hyderabad	Virat Kohli
2017	India	Mumbai Indians	Ben Stokes

Logical operations – Not

“`select * from IPL where not YEAR between 2013 and 2017;`” will yield

YEAR	VENUE	WINNER	PoS
2008	India	Rajasthan Royals	Shane Watson
2009	South Africa	Deccan Chargers	Adam Gilchrist
2010	India	Chennai Super Kings	Sachin Tendulkar
2011	India	Chennai Super Kings	Chris Gayle
2012	India	Kolkata Knight Riders	Sunil Narine
2018	India	Chennai Super Kings	Sunil Narine
2019	India	Mumbai Indians	Andre Russell

Logical operations – Or

“`select * from IPL where YEAR < 2013 or YEAR > 2017;`” will yield

YEAR	VENUE	WINNER	PoS
2008	India	Rajasthan Royals	Shane Watson
2009	South Africa	Deccan Chargers	Adam Gilchrist
2010	India	Chennai Super Kings	Sachin Tendulkar
2011	India	Chennai Super Kings	Chris Gayle
2012	India	Kolkata Knight Riders	Sunil Narine
2018	India	Chennai Super Kings	Sunil Narine
2019	India	Mumbai Indians	Andre Russell

Logical operations – And

“`select * from IPL where WINNER = ‘Chennai Super Kings’ and PoS = ‘Sachin Tendulkar’;`” will yield

YEAR	VENUE	WINNER	PoS
2010	India	Chennai Super Kings	Sachin Tendulkar

Set operations – Difference

```
 (select distinct VENUE from IPL) except
(select VENUE from IPL where VENUE = ‘South Africa’);
```

VENUE
India
India, UAE

Set operations – Difference

```
 (select distinct VENUE from IPL) except
(select VENUE from IPL where VENUE = ‘South Africa’);
```

VENUE
India
India, UAE

```
 (select VENUE from IPL) except all
(select VENUE from IPL where VENUE = ‘India’);
```

VENUE
South Africa
India, UAE
South Africa

Note: The `except` operation automatically eliminates duplicates. To retain all duplicates, `except all` is to be used.

Set operations – Difference (in MySQL)

As there is no except operator in MySQL, we can write the following equivalent query:

```
 select VENUE from IPL where VENUE not in (select
VENUE from IPL where VENUE = ‘South Africa’);
```

VENUE
India
India, UAE

Set operations – Union

```
 (select YEAR, WINNER
from IPL
where VENUE = ‘India, UAE’)
union
(select YEAR, WINNER
from IPL
where VENUE = ‘South Africa’);
```

YEAR	WINNER
2014	Kolkata Knight Riders
2009	Deccan Chargers
2019	Mumbai Indians

Note: The `union` operation automatically eliminates duplicates. To retain all duplicates, `union all` is to be used.

Set operations – Intersection

```
 (select *
from IPL
where VENUE = ‘India’)
intersect
(select *
from IPL
where WINNER = ‘Chennai Super Kings’);
```

YEAR	VENUE	WINNER	PoS
2010	India	Chennai Super Kings	Sachin Tendulkar
2011	India	Chennai Super Kings	Chris Gayle
2018	India	Chennai Super Kings	Sunil Narine

Note: The `intersect` operation automatically eliminates duplicates. To retain all duplicates, `intersect all` is to be used.

Set operations – Intersection (in MySQL)

As there is no `intersect` operator in MySQL, we can write the following equivalent query:

```
 select * from IPL where VENUE = ‘India’ and WINNER =
‘Chennai Super Kings’);
```

VENUE
India
India, UAE

Ordering tuples

“select * from IPL where YEAR <= 2013 order by WINNER;” will yield the following.

YEAR	VENUE	WINNER	PoS
2010	India	Chennai Super Kings	Sachin Tendulkar
2011	India	Chennai Super Kings	Chris Gayle
2009	South Africa	Deccan Chargers	Adam Gilchrist
2012	India	Kolkata Knight Riders	Sunil Narine
2013	India	Mumbai Indians	Shane Watson
2008	India	Rajasthan Royals	Shane Watson

Grouping by

To group the tuples based on same values on the attribute VENUE, we write the following.

```
select VENUE from IPL group by VENUE;
```

VENUE
India
South Africa
India, UAE

Grouping by – More features

We can group the tuples and count based on same values of an attribute as follows.

```
select VENUE, count(WINNER) from IPL group by VENUE;
```

VENUE	count(WINNER)
India	10
South Africa	1
India, UAE	1

Join operations

Consider another relation as follows.

Table: WC

YEAR	VENUE	WINNER	PoS
2003	South Africa, Zimbabwe, Kenya	Australia	Sachin Tendulkar
2007	West Indies	Australia	Glenn McGrath
2011	India, Sri Lanka, Bangladesh	India	Yuvraj Singh
2015	Australia, New Zealand	Australia	Mitchell Starc
2019	England, Wales	England	Kane Williamson

Join operations

Inner join: `select * from IPL inner join WC;`

Natural inner join: `select * from IPL natural inner join WC;`

Left outer join: `select * from IPL left outer join WC;`
Natural left outer join: `select * from IPL natural left outer join WC;`

Right outer join: `select * from IPL right outer inner join WC;`

Natural right outer join: `select * from IPL natural right outer inner join WC;`

Full outer join: `select * from IPL full outer join WC;`

Cartesian product

The Cartesian product of the two relations IPL and WC can be obtained as follows.

```
select * from IPL, WC;
```

Aggregate functions

The following functions take a collection of values (generally through attribute names) as input and return a single value.

Function	**Description**
`count()`	Number of items
`sum()`	Summation
`avg()`	Average value
`max()`	Maximum value
`min()`	Minimum value

Note: The aggregate functions work at the `select` line and takes attributes (not relations) as the arguments.

Aggregate functions – sum() and avg()

We can compute average of the values selected over a particular attribute as follows.

```
select avg(YEAR) from IPL where YEAR < 2011;
```

avg(YEAR)
2010

Note: `sum()` and `avg()` work only on numeric data.

Aggregate functions – count(), max() and min()

We can compute the minimum of the values selected over a particular attribute as follows.

```
 select WINNER from IPL where YEAR = (select
max(YEAR) from IPL);
```

WINNER	PoS
Mumbai Indians	Andre Russell

Note: `count()`, `max()` and `min()` can work on both numeric and nonnumeric data.

Nested structure

```
 select VENUE, PoS
from IPL
where WINNER not in
(select WINNER
from IPL
where YEAR >= 2010);
```

VENUE	PoS
India	Shane Watson
South Africa	Adam Gilchrist

Running SQL on cloud services

Try this out!!!

Oracle Live SQL – Learn and share SQL

`https://livesql.oracle.com`

Problems

1. Consider the following schema representing a train reservation database:
   - Passenger = $\langle \underline{pid} : integer, pname : string, age : integer \rangle$
   - Reservation = $\langle pid : integer, class : string, tid : integer, tamount : number \rangle$

   Note that, a single transaction (through *tid*) can include multiple reservations of passengers travelling in a group. Write the following queries in SQL.

   (i) Find the *pname*s (names of passengers) that comprise firstname and surname both.

   (ii) Find the *pid*s of passengers who are not adults and have a reservation in the 'Sleeper' class.

   (iii) Calculate the total amount paid by all the senior citizens (age more than 60) together through the system.

Problems

2. Consider the following schema representing the population of some cities in United States along with the names states to which they belong to:
   - Census = $\langle \underline{id} : integer, city : string, state : string, population : number \rangle$

   Write queries in SQL that will return the names of least and most populous cities included in *Census*. If there are more then return all.

3. Write an SQL query that performs a division operation on a pair of relations without using the division operator (i.e., $\div$). **Hint:** Use the Cartesain product and other operations.

Problems

4. Consider the following schema representing the costs charged by the instructors for the courses on a MOOC platform:
   - Courses $= \langle \underline{cid} : integer, cname : string, ctype : string \rangle$
   - Instructors $= \langle \underline{iid} : integer, iname : string, affiliation : string \rangle$
   - Catalog $= \langle \underline{cid} : integer, \underline{iid} : integer, cost : real \rangle$

   The *Catalog* relation lists the costs charged for courses by the Instructors. Write the following queries in SQL.

   (i) Find the *cid*s of free courses offered from Indian Statistical Institute, Kolkata.

   (ii) Find the *iid*s of instructors who offer only part-time courses (there can be other course types than full-time too).

   (iii) Find the *cid*s of courses offered by multiple instructors.