SQL

David Zollikofer

This short paper summarizes important SQL concepts. All the content and examples are from *Database Systems: The Complete Book* by Garcia-Molina, Ullman and Widom as well as ETH's lecture slides by Prof. Ce Zhang.

Part I
SQL for Queries

Simple SQL Queries

A basic SQL query has the form `SELECT <attr> FROM <rels> WHERE <cond>` where

- `<attr>` are the attributes of the relation we would like to have
- `<rels>` are the tuples (the relation) we are selecting from
- `<cond>` filters the tuples in the relation only keeping the ones satisfying the condition

A basic query might look as follows:

```
SELECT title, year, producer FROM
movies WHERE id = 10;
```

Conditions are specified with `<attr-name> = <value>` and can be chained together using `AND, OR, NOT, ()`

String Comparison

Strings are lexicographically ordered in SQL and can be compared to each other.
We can also use wildcards:

- `_` matches exactly one character: `WHERE title LIKE 'Ice ___'` would match any movie that has three characters following the space. (such as Ice age or Ice tea).
- `%` matches 0 or more characters: `WHERE title LIKE '%sor%s'` matches or, order, etc.

Dates and Times

Are prefixed with a keyword `DATE, TIMESTAMP, TIME`.

- `DATE '1999-12-01'` is the first of December of 1999
- `TIME '15:00:03.25` is 3 and a quarter minutes past three PM.
- `TIMESTAMP '1966-03-22 13:00:12-1:00'`, notice the `-1:00` meaning one hour behind GTM.

To extract information you can use `SELECT EXTRACT(<selector> FROM <date/time>)` for example `SELECT EXTRACT(MONTH FROM '2000-00-12')`

NULL Values and Unknowns

When a value is not specified in the table it defaults to `NULL`. When we make comparisons and have a `NULL` involved the truth value will become an `UNKNOWN`. On logical operators in behaves similarly to a `FALSE`, though it is not `FALSE`. Arithmetic expressions on `NULL` result in `NULL`.
The database only returns tuples where the `WHERE` clause evaluates to true. Hence if we have an `UNKNOWN` it will not be returned.

Ordering

After a `WHERE, GROUP BY, HAVING` clause we can have an `ORDER BY` clause which looks as follows:
`ORDER BY <attrs> <order>`
where `<attrs>` is the list of attributes we want to order by and `<order>` is either `ASC` or `DESC` with the default being `ASC`.

Union, Intersection and Differences

Let `(Q1), (Q2)` be two queries of the form `SELECT ... FROM ... WHERE ...`. We now define:

- `(Q1) UNION (Q2)` returns records either in `(Q1)` or `(Q2)` or both.
- `(Q1) INTERSECT (Q2)` returns only records which are in `(Q1)` and `(Q2)`
- `(Q1) EXCEPT (Q2)` returns all records being only in `(Q1)` and not in `(Q2)`.

Case Statements

We can modify a row dynamically using the case statement. It is especially useful to deal with `NULL` values.

```
SELECT carId, carName,
CASE
    WHEN carAge = 0 THEN 'new'
    WHEN carAge < 20 THEN 'worthless'
    ELSE 'old-timer'
END AS ageComment
FROM cars;
```

If no `CASE` matches the statement will return `NULL`.

Joins

Cartesian Product / Cross Join

```
SELECT * FROM
Cars c CROSS JOIN Homeowners h
WHERE c.owner = h.id
```

Alternatively there also exist the traditional version with the implicit join:

```
SELECT * FROM
Cars c, Homeowners h
WHERE c.owner = h.id
```

This query build the cross product between all cards and all homeowners and keeps the records where the car owner identifier is the homeowner id.
Note: Sometimes one also sees `Cars as c , Homeowners as h` the keyword `as` is completely optional.
When `h` and `c` both have an attribute `id` we have to reference it as either `c.id` or `h.id` since otherwise there would be ambiguity.

Normal (inner) Join

Combines tuples from both relations satisfying join condition.

```
SELECT * FROM
Cars c JOIN Pets p ON c.owner = p.owner
```

or equivalently if both have a column `owner` one would use as a join condition:

```
SELECT * FROM
Cars c JOIN Pets p USING(owner)
```

Natural Joins

Combines tuples from both relations where all columns with the same name match. In the example if they both had a column `owner`, this column would have to match.

```
SELECT * FROM
Cars c NATURAL JOIN Pets p
```

Outer Joins

They can be natural or normal. The key difference is that an outer join also includes non matching records and sets the missing attributes to `NULL`.

```
Cars c FULL JOIN Pets p using(owner)
```

The statement above combines all pet and car combinations as a normal join but also includes all tuples with no match. (all cars not sharing an owner with a cat and vice versa).
Left Outer Join

```
Cars c LEFT OUTER JOIN Pets p using(owner)
```

The statement above includes all pet and car combinations as a normal join but also includes all cars not sharing an owner with a cat. But the cats not sharing an owner with a car are excluded. Analogously there exists a **Right Outer Join**.

Subqueries

There are three places where SQL subqueries can occur:

- Returning a single value to be used in a `WHERE` clause.
- Returning relations to be used in a `WHERE` clause.
- In a `FROM` clause followed by a variable to represent that subquery.

Single Value Returning

```
SELECT * FROM CarOwners co WHERE co.id = (SELECT owner
   FROM Cars WHERE numberPlate = 123 LIMIT 1)
```

Notice that the `LIMIT 1` is important. If multiple or no tuples were returned an error would be raised.

Conditions with Relations

There are four types:

- `s > ALL R`, is true if s is larger than all values in R.
- `s > ANY R`, is true if is larger than at least one value in R.
- `EXISTS R`, is true if R is non-empty.
- `s IN R`, is true if the tuple s is equal to a tuple in R. (analogously there is `NOT IN`)

As an example: (notice that the parentheses around `owner, insurance` are essential to mark it as a tuple). Here we find the car owners who have insurance for their car.

```
SELECT owner FROM CarOwners, Insurance WHERE (owner,
  insurance) IN (SELECT owner, insurance FROM
  OwnerInsurance)
```

Correlated Subquery

We use the tuple we are selecting in a subquery:

```
SELECT * FROM Cars c
WHERE productionYear <= ALL (
    SELECT year FROM Cars
    WHERE model = c.model
)
```

The query above find the oldest cards of every model.

Subqueries in `FROM` Clause

```
SELECT owner FROM
Cars c, (SELECT * FROM OwnerInsurance WHERE payment <
  200) cheapInsOwn
WHERE c.owner = cheapInsOwn.owner;
```

The query above finds all cars with their respective insurance whose owner has a cheap insurance.

Full Relation Operations

Removing Duplicates

To remove duplicates use a the `DISTINCT` keyword as in `SELECT DISTINCT ...`. But note that it is very expensive as all the relations will need to be sorted in order to detect and eliminate duplicates. Note that `NULL` will also be included at most once if it occurs in the result.

Recursive Query

The basic structure is

```
WITH RECURSIVE recRel AS(
non_recursive-term
UNION [ALL]
recursive-term
) SELECT * FROM recRel;
```

The database executes the `recursive-term` as long as the resulting set keeps changing.

```
WITH RECURSIVE anchestor AS (
    SELECT * FROM people
UNION
    SELECT * from people as p
    WHERE p.father = anchestor.id OR p.mother =
      anchestor.id
)

SELECT * from anchestor;
```

Aggregation Operators

SQL has five aggregation operators: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`. All except the `COUNT` are applied to scalar valued columns. The only excpetion being `COUNT(*)` which counts the tuples in the relation constructed from the `FROM` and `WHERE` clause.

```
SELECT AVG(netWorth), COUNT(*)
FROM MovieExec;
```

Grouping

The `GROUP BY` clause follows the `WHERE` clause and lists arguments that, if two tuples match on these arguments, will place them in the same group. The aggregation operators in the `SELECT` are applied on a per group basis. Only attributes mentioned in the `GROUP BY` clause may appear in the `SELECT` clause.

```
SELECT studioName, SUM(length)
FROM Movies
GROUP BY studioName;
```

Grouping, Aggregation and Nulls

- The value NULL is ignored in aggregation. `COUNT(*)` though counts number of tuples, but `COUNT(A)` counts number of attributes `A` and skips NULLS.
- NULL is treated as an ordinary value when forming groups.
- Aggregations over empty bag are NULL, except `COUNT(*)` being 0.

Having

Assume we would like to filter the groups we have aggregated: We can do this by adding a `HAVING` clause. e.g. we only want the names of sellers which have no sale below or equals 500:

```
SELECT COUNT(*), Name
FROM SellerSale
GROUP BY Name
HAVING MIN(Sale) > 500;
```

Data Modifications

Insertions

Basic syntax:
`INSERT INTO` $R(A_1, \ldots, A_n)$ `VALUES` $(v_1, \ldots v_n)$;
if they are already in standard order we can also write:
`INSERT INTO` R `VALUES` $(v_1, \ldots v_n)$;
or insert programmatically:

```
INSERT INTO Relation1
    SELECT ....
```

We note that values are only inserted after the query has been fully computed preventing interference.

Deletions

We cannot specify the exact tuple to be deleted, instead we must specify a set of tuples (which can contain 1 tuple) to be deleted.

```
DELETE FROM R WHERE <cond>
```

Updates

Basic syntax:
`UPDATE` R `SET <new vals> WHERE <cond>;`
where the new vals are the name of the variable followed by an equals sign and the new assignment.

```
UPDATE Professor
SET name = 'Prof. ' || name
WHERE title = 'Professor';
```

recall that the `||` operator concatenates strings.

Transactions

A transaction is a set of SQL commands that get executed atomically. In PostgreSQL every statement in the shell is automatically wrapped in a transaction:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Wally';
COMMIT;
```

Views

A view is a virtual relation which is built on a query run on a table.

```
CREATE VIEW <viewName> as <viewDefinition>;
```

where `viewDefinition` is a SQL query.
we can delete a view with `DROP VIEW <viewName>;`

Updating Views

We can only update views satisfying some constraints:

- only one base relation involved
- the view involves the key of the base relation
- view does not involve aggregates, group by or duplicate-elimination

Indexes

To add an index:
`CREATE INDEX <indexName> ON Table1(field1);`
to remove an index:
`DROP INDEX <indexName>`

Multicolumn Indexes

`CREATE INDEX <indexName> on Table1(field1,field2);`
Multicolumn indexes first index on field1 and then on field2.

Isolation Level

With the comman `SET TRANSACTION ISOLATION LEVEL <isolation level>` we can set the isolation level for the current transaction.
We can also have a new transaction with a specific isolation level by typing `BEGIN TRANSACTION ISOLATION LEVEL <isolation level>;`
There are the following levels:

	Dirty Reads	Nonrepeatable Reads	Phantoms
`READ UNCOMMITTED`			
`READ COMMITTED`	No		
`REPEATABLE READ`	No	No	
`SERIALIZABLE`	No	No	No

where:

- Dirty Reads: When we read data another transaction has modified and not yet committed.
- Nonrepeatable Reads: When we read a record twice but the value was allowed to change in between.
- Phantoms: When another transactions commits and adds or removes rows to a table we are currently reading.

Part II
SQL for Table Management

Table Creation

The basic syntax is as follows:

```
CREATE TABLE Professor(
    PersNr integer,
    Name varchar(30),
    Level character(2) default "AP",
    PRIMARY KEY (PersNr)
);
```

The default is a value that will be used if no `Level` is specified (otherwise we would have a `NULL`).

Data Types

- `int, bit` integer, 0-1 bit
- `numeric(precision)` a number with fixed precision
- `boolean`
- `timestamp, date, time`
- `varchar(n)` variable length string with limit
- `char(n)` string which will be padded to size n
- `text` unlimited string (in theory)

Relations

Declaring Foreign Keys

We have two options:

```
CREATE TABLE Dog(
    name VARCHAR(20),
    owner INT REFERENCES Person(id)
);
```

```
CREATE TABLE Dog(
    name VARCHAR(20),
    owner INT,
    FOREIGN KEY (owner) REFERENCES Person(id)
);
```

With the second version we can also define composite foreign keys by writing
`FOREIGN KEY (<attributes>) REFERENCES <table>(<attributes>)`

Referential Integrity

What should happen to a record (called child record in the following discussion) if a parent record (the one referred to in one of the childs foreign key fields) gets updated or deleted?
There are multiple options: (usually the key a child refers to is a parents primary key)

- `CASCADE`
  - `UPDATE` if the parents primary key is updated the child's foreign key will also be updated.
  - `DELETE` if parent gets deleted, child will be deleted too.
- `RESTRICT`
  - `UPDATE` attempt to update parent's primary key will fail with an error
  - `DELETE` attempt to update parent's primary key will fail with an error
- `NO ACTION`
  - `UPDATE` attempt to update parent's primary key will fail with an error
  - `DELETE` attempt to update parent's primary key will fail with an error
- `SET DEFAULT`
  - `UPDATE` when parent's primary key gets updated child's foreign key will be set to DEFAULT or NULL
  - `DELETE` when parent gets deleted child's foreign key will be set to DEFAULT or NULL

Difference between `NO ACTION` and `RESTRICT`? With `RESTRICT` the constraint will be checked and immediately fails whereas with `NO ACTION` the database tries to update/delete but then fails on committing (failure of deferred check). However some databases (e.g. MySQL) do not differentiate between the two options.
Usually `NO ACTION` is the **default** if no specific constraint has been chosen.
The constraints can be specified during table creation as follows:

```
create table Lecture (
    ...,
    PersNr integer references Professor
    on delete {cascade | restrict | no action |
        set null |set default}
    on update {cascade | restrict | no action |
        set null |set default}
);
```

Enforcing 1:1 Relations

This is best done using a third table:

```
CREATE TABLE Employee(
    emp_id VARCHAR(20) NOT NULL UNIQUE
);
CREATE TABLE Office(
    room VARCHAR(20) NOT NULL UNIQUE
);

CREATE TABLE HasOffice(
    emp_id VARCHAR(20) NOT NULL UNIQUE REFERENCES
      Employee (emp_id),
    room VARCHAR(20) NOT NULL UNIQUE REFERENCES Office
       (room)
);
```

Constraints

NOT NULL

Attribute cannot be `NULL`

```
..
sName text NOT NULL,
...
```

PRIMARY KEY

Only one primary key constraint can be specified per relation. Two options, either a single attribute as primary key like

```
...
sID int PRIMARY KEY,
...
```

or a combination of attributes:

```
CREATE TABLE Student
(
    sID int,
    sName text,
    GPA real
    PRIMARY KEY (sID, sName)
);
```

where the combination must be unique and all attributes need to be non null. **UNIQUE**

Usage like primary key: A value must be unique, but different from `PRIMARY KEY` the value can be `NULL` and we can have multiple `UNIQUE` constraints per relation.

CHECK

Can happen at tuple or attribute level which will get checked at update of tuple or single attribute respectively.

```
CREATE TABLE Employee
(
salary real,
bonus real,
CHECK(salary + bonus > 10000)
);
```

```
CREATE TABLE Employee
(
salary real CHECK(salary  > 5000)
);
```

Assertions & Triggers

Assertion

A logical expression that must be true at all times:

```
CREATE  ASSERTION <assertName> CHECK (<condition>)
```

Trigger

A trigger is an SQL statement that gets executed when a certain predefined action is performed. It is best to just look at an example or read the docs if a trigger is needed:

```
CREATE TRIGGER CeoCantGetPayCut
AFTER UPDATE OF (salary) ON Ceo
REFERENCING
    OLD ROW AS oldRow,
    NEW ROW AS newRow,
FOR EACH ROW
WHEN(oldRow.salary > newRow.salary)
    UPDATE Ceo
    SET salary = oldRow.salary
    WHERE id = oldRow.id;
```