Principles of Data Management

Alvin Lin

August 2018 - December 2018

Structured Query Language

Structured Query Language (SQL) was created at IBM in the 80s:

- SQL-86 (first standard)
- SQL-89
- SQL-92 (what all SQL compliant DBs use)
- SQL-1999
- SQL-2003

SQL is a commercial language that is intended to be a data manipulation language, but supports features of data definition languages. It supports the following features:

- Schema
- Domain
- Integrity Constraints
- Indicies
- Security
- Physical Storage

Domain Types

- `char(n)` - n number of characters
- `int` - numbers
- `smallint` - smaller value number
- `numeric(p,d)` - precision decimal
- `real,double,float` - "infinite" precision decimals
- `varchar(n)` - variable length characters

Create Table

```
create table r (A1 D1, A2 D2, ... ,An Dn) (integrity constraints);
```

Example:

```
create table dog(
  ID int,
  Name varchar(20),
  Salary numeric(8,2),
  goodboy smallint
);
```

Example:

```
create table p(
  ID int,
  Name varchar(4) not null,
  DOB varchar(10),
  dog_id int,
  primary key(ID, Name),
  foreign key(dog_id) references dog(ID)
);
```

CRUD

```
insert into r values (v1,v2,vn);
insert into dog values (1,'cat',10.00,0);
delete from dog;
drop table r;
alter table dog add iscat int not null default 0;
alter table dog drop iscat;
```

Select

```
select a1,a2,a3,an from r1,r2,rn where predicate;
select name from dog;
select distinct name from dog;
select all name from dog;
select 'joe' as name;
select name,'lab' as breed from dog;
select * from dog;
```

Explicitly specify what attributes to return, `SELECT *` should be avoided with the exception of manually viewing data.

```
select id,name,salary as pay from dog where salary > 80000;
```

The `where` keyword allows for conditions by which we can filter the returned rows. We can specify modifications to the returned attribute and use boolean conditions to create compound conditionals.

```
select id,name,salary/26 as pay from dog where salary > 80000 and
  breed = 'lab';
```

The `between` modifier allows us to select a range as a predicate.

```
select * from dog where age between 6 and 9;
```

Selecting from multiple tables using `SELECT *` returns a Cartesian product.

```
select * from dog,cat;
```

This is not particularly useful, it is much more useful to use a predicate to match the data.

```
select * from dog,cat where dog.name = cat.name and dog.age = 4;
select * from dog as D, cat as C where D.age = C.age;
```

Examples

table dog	
name	owner
Charlie	Snoopy
Woodstock	Snoopy
Lucy	Ricky
Snoopy	Sam

Finding all the owners with dogs named Woodstock:

```
select d2.owner from dog where name = 'woodstock';
```

Finding all the dogs owned by Snoopy:

```
select name from dog where owner = 'snoopy';
```

Selecting the owner's owner of Charlie:

```
select owner from dog d1, dog d2 where d1.name = 'charlie' and
  d1.owner = d2.name;
```

String Operations

The percent sign (%) is used for partial substring matches while the underscore (_) is used for single character matches.

```
select * from dog where name like '%py';
select * from dog where name like 'Sn\'' escape '\';
```

Concatenation is expressed with double pipes (||).

```
select * from fname||lname like '%foo';
```

Ordering

The `order by` keyword allows for queries to be sorted. Specifying more than one sort option allows for sorting by multiple fields.

```
select name,lname from dog order by name asc;
select name,lname from dog order by name,lname desc;
select name,lname from dog order by name desc,lname;
```

Set Operations

Set operation keywords are valid in SQL.

```
(select name from dog) union (select name from cat);
(select name from dog) intersect (select name from cat);
(select name from dog) except (select name from cat);
```

Null Values

```
select field from table where age is null
```

Boolean evaluation:

1. True or null evaluates to true
2. False or null evaluates to null

3. Null or null evaluates to null
4. True and null evaluates to null
5. False and null evaluates to false
6. Null and null evaluates to null

Aggregate Functions

There are $4\frac{1}{2}$ aggregate functions: `avg, min, max, sum, count`. Count does not entirely count as an aggregate function. We can use this to count the number of non-null rows:

```
select count(*) from penguin;
```

Because no rows can be non-null, this just returns the number of rows in your database. We can also select the number of non-null values for a field:

```
select count(age) from penguin;
```

To count distinct values in a field:

```
select count(distinct name) from dog;
```

The other aggregate functions can be used in a similar way:

```
select avg(salary) from dog where state = 'CA';
```

If there are no dogs, then 0 is returned. If some dogs have a null salary, then null is returned.

```
select avg(salary) as avgs,state from dog group by state;
select avg(salary) as avgs,state,name from dog group by state,name
    ;
```

Results like this can be filtered with the `having` clause:

```
select avg(salary) as avgs,state from dog group by state having
    avgs > 5;
```

Subqueries

Subquery expressions can be used for set membership, comparisons, and cardinality expressions.

```
select name,age from dog where age in (select age from cat where
  state = 'sleeping');
select name,age from dog where (age,legs) in (select age,legs from
    cat where
  state = 'sleeping');
select name from dog where age > some(select age from cat where
   state = 'CA');
select name from dog where age > all(select age from cat where
   state = 'CA');
select name from dog where age > (select min(age) from cat where
  state = 'CA');
```

Because nested subqueries can get very messy, we use the `with` clause as a way of defining a temporary relation that only exists in the clause in which the `with` clause occurs.

```
with dept_total(dept_name, value) as
  (select dept_name, sum(salary) from instructor group by
     dept_name),
  dept_total_avg(value) as (select avg(value) from dept_total)
select dept_name from dept_total,dept_total_avg
  where dept_total.value > dept_total_avg.value;
```

Scalar subqueries must be used where only a single value is expected.

```
select dept_name,
  (select count(*) from instructor
    where department.dept_name = instructor.dept_name) as
       num_instructors
  from department;
```

Database Modification

Deleting records (also accepts a valid where clause):

```
delete from instructor;
delete from instructor where dept_name = 'Finance';
delete from instructor where dept_name in
  (select dept_name from department where building = 'Watson');
delete from instructor where salary < (select avg(salary) from
   instructor);
```

Records can be inserted into databases:

```
insert into course(course_id, title, dept_name, credits)
  values('CS-437', 'Database Systems', 'Comp Sci', 4);
insert into course values('CS-437', 'Database Systems', 'Comp Sci
   ', 4);
```

```
insert into student select ID, name, dept_name, 0 from instructor;
```

Records can be updated in databases:

```
update instructor
  set salary = salary * 1.03
  where salary > 100000;
update instructor
  set salary = salary * 1.05
  where salary <= 100000;
```

As an aside, since we have no guarantee to the order of execution of the previous statements, some instructors might get more than one raise. We can use the `case` statement to solve this.

```
update instructor set salary = case
  when salary <= 100000 then salary * 1.05
  else salary * 1.03
end
```

Updates can also take scalar subqueries:

```
update student S
  set total_credits = (select sum(credits) from takes,course
    where takes.course_id = course.course_id and
      S.ID = takes.ID and
      takes.grade <> 'F' and
      takes.grade is not null);
```

Database Joins

Join types:

- `inner`
- `left outer`
- `right outer`
- `full outer`

Conditions:

- `natural`
- `on <predicate>, using (attributes)`

Natural joins match on all attributes with the same name in both tables. Inner joins only select rows on the matching attribute (essentially enforcing that no row in the output has null values) while full outer joins preserve all information in both tables.

Dog	
Name	Age
A	5
B	10
C	7

Cat		
ID	Name	State
1	C	VT
2	D	IA

```
select * from dog natural left outer join cat;
```

Name	Age	ID	State
A	5	NULL	NULL
B	10	NULL	NULL
C	7	1	VT

```
select * dog natural right outer join cat;
```

Name	Age	ID	State
C	7	1	VT
D	NULL	2	IA

```
select * from dog natural full outer join cat;
```

Name	Age	ID	State
A	5	NULL	NULL
B	10	NULL	NULL
C	7	1	VT
D	NULL	2	IA

```
select * from dog inner join cat on cat.name = dog.name;
```

Name	Age	ID	State
C	7	1	VT

Suppose we have two cats with the same name:

Dog	
Name	Age
A	5
B	10
C	7

Cat		
ID	Name	State
1	C	VT
2	D	IA
3	C	MI

```
select * from dog natural full outer join cat;
```

Name	Age	ID	State
A	5	NULL	NULL
B	10	NULL	NULL
C	7	1	VT
D	NULL	2	IA
C	7	3	MI

Views

Views are used to restrict access to attributes or rows. It represents a virtual table.

```
create view old_dogs as (select name,age from dog where age > 10);
```

Executing any query on old_dogs is equivalent to using the view as a subquery. This is generally used to restrict access to the table, prevent users from querying all dogs or querying certain attributes of dogs. We can also insert into views.

```
insert into old_dogs values('Jeff', 13) values('Cheese', 8);
```

While this might work, you will never see Cheese by querying the view because the view restricts access to dogs whose age is over 10.

```
create view dogs2 as (select name from dog,cat where
  dog.lives_in = cat.state);
```

Inserting into this view would lead to ambiguity, so in general, there are certain restrictions for inserting into views.

1. Only a single table is present in the from clause.
2. No aggregations, distinct, etc.
3. No group by or having clauses.
4. Any attribute not in the selection can be set to null.

There exist the concept of materialized views, which are physical tables that contain the results of a view. It is a cache of data used for long running queries that needs to be updated.

Transactions

In most flavors of SQL, transactions begin with most statements and usually end with a commit or rollback. You can specify for a group of statements to be atomic however.

```
begin atomic;
  select * from dog;
  insert into cat ...
  update turtle ...
commit;
```

Integrity Constraints

Integrity contraints for a single table include the following:

- `not null`
- `unique`
- `primary key`: not null, unique, indexed
- `check <predicate>`

```
create table dog (
  id int primary key,
  name varchar(30) not null,
  salary int,
  check (salary > 5),
  check (age < 10),
  check (breed in ('Lab', 'Dalmatian', 'GSD'))
)
```

Integrity constraints can apply across multiple tables. Referential integrity uses foreign keys to ensure some condition.

```
create table dog (
  id int primary key,
  name varchar(20) not null,
  foreign key(owner) references person(id)
  on update cascade
)
```

If the referenced person is deleted or updated in database, we specify how the database handles the constraint. `on update cascade` makes any changes from the

person table cascade into the dog table to update its value. Alternatively, we can specify `on delete set null` to set the value to null if the referenced person is deleted. Another option is to specify `set default 0` to specify a default value if the referenced person is deleted. Complex check statements can also be used like this, but it is not usually recommended since there are better options.

```
create table dog (
  check owner_id in (select id from person)
)
```

Additional Data Types

- Date - yyyy-mm-dd
- Time - HH:MM:SS
- Timestamp - Date and time
- Interval - the resulting of subtracting a date/time related data type
- Blob - binary large object (cannot be indexed)
- Clob - character large object (cannot be indexed)

Blobs and clobs are referenced by a pointer stored in the database. Because they are not indexed, relevant metadata needs to be stored as well.

Indices

Suppose have a dog table with the attributes id, name, age, and state:

```
create index dog_index on dog(state, age);
```

Creating an index affects the structure of how the data is stored on disk to allow efficient access. Queries that use an index must use all the attributes that the index applies to. An index creates a table of all the distinct values for an attribute currently in the table. Each entry in the index holds a list of pointers to all matching entries in the table. The index does not have to be specified in order for it to be used to optimize a query. After it is created, the database will use automatically for any matching predicates.

Functions/Procedures

We can define functions and procedures according to the SQL-1999 standard. The syntax for this will often be vendor-specific.

```
create function dog_count(state varchar(2)) returns integer begin
  declare dc integer;
  select count(*) into dc from dog where dog.state = state;
  return dc;
end
```

After defining this function, we can use it in future queries:

```
select name,age from cat where dog_count('OR') > 5;
```

Subqueries can be also generally be used as arguments in functions. All functions must have a `begin` and `end` clause, a return type specified, and the actual value returned. Tables can also be returned as of SQL-2003:

```
create function dogs(st varchar(2)) returns table(
    id int, name varchar(10), breed varchar(5)) begin
  return table(
    select id,name,breed from dog where dog.state = state
  )
end
```

Procedures can also be used as function like constructs:

```
create procedure dog_count(state varchar(2), out dcount integer)
   begin
  select count(*) into dcount from dog where dog.state = state;
end
```

This could be used in subsequent SQL statements:

```
declare dc int;
call dog_count('OR', dc);
```

Iteration

SQL-1999 also supports iteration using `while`:

```
while boolean_exp do
  ...
end while
```

and also `repeat`:

```
repeat
  ...
until expression
end repeat
```

Another option available is `for` loops:

```
declare r integer default 0;
for r as select age from dog; do
  set n = n + r.age
end for
```

Triggers

Triggers are statements that happen automatically upon an event. A condition to trigger upon and a subsequent action must be specified. Like functions and procedures, the syntax here will likely be vendor-specific. Triggers can be set on insert, update, and delete events.

```
create trigger trigger1 setnull before update of dog
  referencing new row as nrow
  for each row when (nrow.age > 15)
  begin atomic
    set nrow.age = null;
  end;
create trigger myT after update of dog
  referencing new row as nrow
  for each row when (nrow.age > 30)
  begin atomic
    set nrow.age = null
  end;
create trigger doginfo_trigger after update of dog on (age)
  referencing new row as nrow
  referencing old row as orow
  for each row when (nrow.breed <> 'GSD' and nrow.breed is not
     null) and
    (orow.breed = 'GSD' or orow.breed is null)
  begin atomic
    update cat set cat.age = cat.age +
      (select age from dog where cat.id = nrow.id)
      where cat.breed = orow.breed
  end;
```

Triggers are generally used to accumulate summary data, perform data replication, and set default values (although these are all really bad reasons to use them). They

have issues with cascading runs and their best use is probably with audit data to record when changes are made in a database.

You can find all my notes at `http://omgimanerd.tech/notes`. If you have any questions, comments, or concerns, please contact me at alvin@omgimanerd.tech