LASSONDE SCHOOL OF ENGINEERING | YORK UNIVERSITÉ UNIVERSITY

SQL

- *The Basics*
- *Advanced*
- *Manipulation*
- *Constraints*
- *Authorization*

Table of Contents

Parke Godfrey

2016-10-26 initial [v1]
2016-10-31 [v2]
2016-11-07 [v3]

Acknowledgments

Thanks

- to *Jeffrey D. Ullman*
  for initial slidedeck
- to *Jarek Szlichta*
  for the slidedeck with significant refinements on which this is derived

SQL: a *standard* language for accessing databases

Many say SQL stands for *Structured Query Language*. It is effectively *the* standard for relational database systems.

Knowing SQL, you will know how to access and manipulate data in virtually all relational database systems!

E.g., Oracle, Microsoft SQL Server, IBM DB2, SAP Sybase, PostgreSQL, MariaDB, MySQL Teradata, IBM Informix, and Ingres.

Oh, and Microsoft Access, MongoDB, SQLite, Empress,

...

SQL: origins

- SQL is based on the *relational algebra* and the *tuple relational calculus*.

  It is a *declarative* query language. (The database engine finds a “best” way to evaluate the query; this is called *query optimization*.)

- The initial version was developed in the early 1970's and was called SEQUEL, for *Structured English Query Language*.

- SQL became

  - an ANSI (American National Standards Institute) standard in 1986, and
  - and ISO (International Organization for Standardization) standard in 1987.

- SQL includes

  - a *data definition language*,
  - a *data manipulation language*, and
  - a *data control language*

  in addition to being a “data query language”.

The basic block

select … from … where…

select *desired attributes*
from *rel'ns to source from*
where *filter for which tuples to keep*

Running Examples

Our pub schema

- **Beer(name,** manf)
- **Pub(name,** addr, licence)
- **Drinker(name,** addr, phone)
- **Likes(drinker, beer)**
- **Sells(pub, beer,** price)
- **Frequents(drinker, pub)**

Single-relation queries ($\sigma\pi$)

Just lists *one* table in the **from** clause.

---

conceptual evaluation

Think of a *tuple variable* visiting each tuple of the rel'n from the **from** clause.

1. Return the tuple *if* the logical condition in the **where** clause evaluates as **true**
2. *projecting* the attributes defined — possibly an *extended* projection! — by the **select** clause.

Example

What beers are made by *Anheuser Busch*?

```
select name
from Beer
where manf = 'Anheuser Busch';
```

The result of the query

is a table (rel'n), of course!

In this case, it is a single-columned table. E.g.,

name
Bud
Bud Lite
Michelob
…

“*” in the select clause

When there is one relation in the **from** clause, “*” in the **select** clause stands in for *all attr's* of the rel'n.

E.g.,

```
select *
from Beer
where manf = 'Anheuser Busch'
```

Bad practice!

The result of the query

E.g.,

name	manf
Bud	Anheuser Busch
Bud Lite	Anheuser Busch
Michelob	Anheuser Busch
…	…

Renaming attributes

If you want an attribute to have a new name, use “as *new_name*”.

E.g.,

```
select name as beer, manf
from Beer
where manf = 'Anheuser Busch';
```

Expressions in the select clause

Sure! Just as with *extended projection*, most any expression that makes sense can appear as an element of the **select** clause.

E.g.,

```
select pub, beer,
       price * 114 as priceInYen
from Sells
```

Constants as expressions

```
select drinker,
       'likes Bud' as whoLikesBud
from Likes
where beer = 'Bud';
```

The result of the query

E.g.,

drinker	whoLikesBud
Elga	likes Bud
Franck	likes Bud
…	…

Complex conditions in the where clause

- “boolean” operators **and**, **or**, and **not**.
- comparisions “=”, “<>” (SQL's “≠”), “<”, “>”, “<=”, and “>=”.
- And many, many other operators defined in the SQL standards that produce ”boolean”-valued results.

Example: complex condition

```
select price
from Sells
where pub = 'Joe''s Bar'
  and beer = 'Bud';
```

How many tuples does this query return? Why?

String patterns

A condition can compare a string to a pattern by

- *attribute* `like` *pattern*
- *attribute* not `like` *pattern*

A pattern is a quoted string.

- `%` is for *any string*
- _ is for *any character*.

SQL *predates* Java, Perl, Python, etc.! So the *regex* syntax for pattern matching is completely different that the “standard” regex we know and love from, for example, Java. Sigh.

Example: like

```
select name
from Drinker
where phone like '%555-____';
```

NULL “values”

A tuple in SQL reln's can have NULL as a “value” for one or more of its attr's.

The meaning is contextual. Two common cases.

- *missing value*. E.g., we know that Joe's Bar has an address, but we do not know what it is.
- *inapplicable* (There is no value). E.g., the value of attr. *spouse* for someone who is unmarried.

SQL is a three-valued logic

not a two-valued (*boolean*) logic, because of *nulls*!

The values are **true**, **false**, and **unknown**.

- "*anything* = NULL" is **unknown**
- "*anything* <> NULL" is **unknown**

This includes "NULL = NULL" and "NULL <> NULL"!

In evaluating a query, we only accept tuples that evaluate to **true** wrt the **where** clause; anything that evaluates to **false** or to **unknown**, we reject.

(Surprising) example

pub	beer	price
Joe's Bar	Bud	NULL

```
select pub
from Sells
where price <  2.00
   or price >= 2.00;
```

The query returns the *empty* table!

Distinct vs All

SQL allows us to choose *set* or *bag* semantics per query (or sub-query).

```
select distinct …
```

will return a *set* of tuples (that is, with any duplicates removed).

```
select all …
```

will return a *bag* of tuples (that is, without duplicates being removed).

The keyword `distinct` or `all` after `select` is optional; the default is `all`.

Multi-relation queries (σπ⋈)

We may have more than one table listed (*sourced*) in the **from** clause. Distinguish attr's of the same name by *reln*.*attr*.

---

conceptual evaluation

1. Apply the cross-product across the reln's in the **from** clause.
2. For each tuple in the result, if the tuple evaluates to **true** wrt the **where clause**, then
3. return the *projection* of the tuple wrt the **select** clause.

Note that any *join* criteria that we have in mind *must* be explicitly stated in the **where** clause.

Example: joining two tables

```
select beer
from Likes, Frequents
where pub = 'Joe''s Bar'
  and Frequents.drinker = Likes.drinker;
```

(Any attr. name only in one of the tables does not need to be disambiguated — have a table prefix — in SQL.)

In English?

Variables / “aliases”

```
select beer
from Likes as L, Frequents as F
where pub = 'Joe''s Bar'
  and F.drinker = L.drinker;
```

This is a nice shorthand, and can improve readability.

But additionally, we must have table aliases supported by SQL! Why?

Intersection, Union, and Except ("—")

Simply place the keyword between two **select … from … where …** blocks. E.g.,

```
select … from … where …
union distinct
select … from … where …
```

- Takes an optional keyword of `distinct` or `all` after (with `all` as the default).
- The two blocks must be *schema compatible*.
- The names of the attr's in the answer-table schema are inherited from the first block.

And that's all!

for the basics, that is

Oh…and *anywhere* that a table can appear in an SQL query, *another* SQL query can appear *instead*!

We call this a *sub-query*.

This is because SQL is extremely *composable*, just as is the *relational algebra*.

Aggregation

We add *aggregate* operators that can be used in the **select** clause as attr. / column definitions.

E.g., `sum`, `count`, `avg`, `min`, and `max`.

Rule. May not mix non-aggregate and aggregate operators with a **select** clause.

A `select … from … where …` query with aggregrate operators returns *exactly one* tuple in the answer table.

Example

How many drinkers are there?

```
select count(*) as #drinkers
from Drinker;
```

Example

How many drinkers live on Spadina Ave?

```
select count(*) as #drinkers
from Drinker
where addr like '%Spadina Ave%';
```

Extending aggregation

Aggregation is quite powerful. But it looks limited since

- one may not mix non-aggr. and aggr. columns in the **select** clause, and
- in that an aggr. query returns just one tuple (the "aggregate").

For example, say we want to know, for each type of beer, how many (`count`) drinkers *like* that beer. We want a two-column answer table — beer and #drinkers — that would report that with multiple rows (one for each type of beer).

By composition

By composing with sub-queries, we can actually *already* write this!

```
select beer,
       (select count(*)
        from Likes L
        where L.beer = B.beer)
           as #drinkers
from (select distinct beer
      from Likes) as B
```

The result of the query

E.g.,

beer	#drinkers
Bud	35
Bud Lite	2
Michelob	47
Molsons	59
…	…

Wait, what?!

Okay, that query is a *bit* hard to grôk…

Likely because it uses a *correlated* sub-query, as well as a sub-query within the **select** clause!

But it is a beautiful illustration of just how powerful composition is.

We will be coming back to that query — and to correlated sub-queries — shortly. It will start to make sense after that.

The group-by clause

The **group-by** clause provides us a meaningful way to mix non-aggregate attr's and aggregate attr's in the same **select** clause.

- In the **group by**, we list the (non-aggr.) attr's that we want to use in the **select**.
- There will be *one* answer tuple per value combination over the **group-by** attr's that results in the underlying query.
- The values of the aggr. attr's for that tuple will be wrt aggregation over the tuples having that **group-by** value.

Previous example

Number of drinkers who like each beer

```
select beer, count(*) as #drinkers
from Likes
group by beer
```

Conceptual evaluation: group by

1. Evaluate the “underlying” query — the query without the aggregate operators or the **group by**.
2. Partition the resulting tuples by the **group-by** attributes' values.
3. For each resulting *group* from the partition, compose the answer tuple
   a. with the values of the **group-by** attributes' as that of the group, and
   b. computing the aggregate values over the tuples of the underlying answer set that belong to the group.

The having clause

The **having** clause is a counterpart to *group-by*.

- It optionally comes after the *group-by* clause.
- It lists conditions that the aggregated tuples — resulting from the "select … from … where …" and then "group by" evaluation — must *meet* (i.e., evaluate to **true** wrt).

  The conditions may only refer to aggregate columns. Why?

Note that **having** is extra in that we do not technically need it to write the same queries.

Example: Having

Query. *For each beer (that at least ten drinkers like), how many drinkers like that beer?*

```
select beer, count(*) as #drinkers
from Likes
group by beer
having count(*) >= 10.
```

Example: Having (2)

```
select beer, count(*) as #drinkers
from Likes
group by beer
having #drinkers >= 10.
```

By the *standards*, this is illegal syntax! the name `#drinkers` is not within the *scope* of the **having** clause.

Some database systems *do* allow it, though. (Sadly, not DB2!)

Example: Having (3)

without the having!

```
select beer, #drinkers
from (  select beer,
                count(*) as #drinkers
        from Likes
        group by beer
     ) as WhoLikesWhat
where #drinkers >= 10;
```

So, **having** is “syntactic sugar”, but is so useful, it is worth having in SQL.

But this *sub-query* “trick” can be useful when we want to use the names that we have given our aggregate columns.

Sub-queries

Where are we allowed to put sub-queries?

Anywhere that a table is expected! *And more…*

1. In the **from** clause, replacing a table name with a query instead.
2. In the **where** clause, using special predicates.
3. In the **select** clause!

Sub-queries in the *from* clause

- This provides us a simple way to *nest* queries.
- By the *standards*, one needs to provide the nested query an *alias*, regardless of whether it is used.
- Such a sub-query *cannot* be *correlated*.

Example

```
select Z.name as pub, Z.addr
from (  select distinct P.name,
                        P.addr
        from Pub P, Sells S
        where P.name = S.pub
     ) as Z
```

Sub-queries in the *where* clause

- SQL provides *predicates* to compare column values with sub-query results.
  - coln_1 `>` `all (select` coln_2 `from …)`
    and for "`= all`", "`< all`", "`>= all`", etc.
  - coln_1 `>` `any (select` coln_2 `from …)`
    and for "`= any`", "`< any`", "`>= any`", etc.
  - $(\text{coln}_1$`,` …`,` $\text{coln}_n)$ `in`
    `(select` coln_{n+1}`,` …`,` coln_{2n} `from …)`
  - `exists (select * from …)`

And, of course, all these can be used with "`not`".

Example

```
select D.name as drinker, D.addr as home
       P.name as pub, P.addr as address
from Drinker D, Pub P
where (D.name, P.name) in (
              select F.drinker, F.pub
              from Frequents F
         )
```

Sub-queries in the *where* clause (2)

correlation

- Sub-queries in the **where** clause *can* be *correlated*.
  - All variables / table aliases in the containing query (and *above*, if that query is nested) are visible to it.

Example w/ correlation

```
select D.name, D.addr
from Drinker D
where not exists (
        select S.beer
        from Sells S
        except
        select L.beer
        from Likes L
        where D.name = L.drinker
      )
```

Sub-queries in the *select* clause

- Such a sub-query should have a *single-column* schema and return *at most one* value per “invocation”. (It is a runtime error if it returns more.)
- If the sub-query returns *no* answer tuple for an invocation, the return “value” is taken as NULL.
- *Variables* (table aliases) in the containing query (and *above*) are within its *scope*, so can be used for *correlation*.

Detour: Examples

See SQL Examples over Colour Schema (PDF).

Data *Manipulation* Language

How to update data in the database

- **insert**
  - add new tuples to a table
- **update**
  - change tuples' columns' values in a table
- **delete**
  - delete certain tuples from a table

insert

```
insert into <table> (<attr's>) values
    (<tuple #1>),
    (<tuple #2>),
        …
    (<tuple #n>);
```

where the *tuples* are lists of values.

The attribute list after *table* is optional. If left out, we have to match the schema in the *tuples* as declared in the table's *create*.

Example insert

Add the tuple to **Like** that *Sally* likes *Bud*:

```
insert into Like (drinker, beer) values
    ('Sally', 'Bud');
```

Transactions

What is the difference between

```
insert into Like (drinker, beer) values
    ('Sally', 'Bud'),
    ('Franck', 'Grolsch');
```

and

```
insert into Like (drinker, beer) values
    ('Sally', 'Bud');
insert into Like (drinker, beer) values
    ('Franck', 'Grolsch');
```

?

insert is a *transaction*

As such, it is under the *all-or-nothing* principle: all the transaction is completed (*commit*) or none of it is (*rollback*).

Why might an insert transaction fail?

Specifying the attributes

We may add to the relation name a list of attributes.

Two reasons to do so.

1. We forgot the standard order of attributes for the relation.
2. We do not have values for all attributes, so we want the system to fill in missing components with NULL or a default value.

Note that it is good practice always to specify the attributes, for the same reasons it is good practice to not use "$*$" in **select**.

Adding default values

In a **create table** statement, we can follow an attribute by **default** and a value. When an inserted tuple has no value for that attribute, the default will be used.

E.g.,

```
create table Drinker (
    name char(30) primary key,
    addr char(50)
         default '123 Front St',
    phone char(16)
);
```

Example insert w/ default

```
insert into Drinker (name) values
    ('Sally');
```

resulting in

name	address	phone
Sally	123 Front St	NULL
...	...	...

Anywhere a table…

Okay, you've said that anywhere I put a table name in SQL, I can put a subquery instead. *Ha!* What about in an **insert**?! Can I replace the table name with a sub-query?

Of course! *With caveats*.

The rules about what is a legal sub-query in an **insert** are involved. But essentially, it must be *unambiguous* what the update to what base table is to be made.

And a subquery instead of *values*?

Yes.

E.g., enter into a table **PotentialFriends** (*name*) those drinkers who frequent at least one pub that *Sally* also frequents.

```
insert into PotentialFriends (
    select F.drinker
    from Frequents S, Frequents F
    where S.drinker = 'Sally'
      and S.pub = F.pub
      and S.drinker <> F.drinker
);
```

values

Hey, **values** is cool! Can I use it as a sub-query in other places in place of a table name?

Yes.

update

To change certain attr's in certain tuples of a rel'n.

```
update <table>
set <list of attr assigments>
where <condition on tuples>
```

Example: *update*

Change drinker *Fred*'s phone number to '555-1212'.

```
update Drinker
set phone = '555-1212'
where name = 'Fred';
```

Does this only update *one* tuple?

Example: *update* of multiple tuples

Make $4 the maximum price for a beer.

```
update Sells
set price = 4.00
where price > 4.00;
```

delete and sub-queries

- We can use sub-queries (with correlation too) in **set**.
- We can use sub-queries (with correlation too) in the **where** clause, of course.
- And we can use *a* sub-query in place of the *table name* after **delete** (with caveats).

Again, just as with **insert**, the rules about what is a legal sub-query to replace the *table name* are involved. But essentially, it must be *unambiguous* to what base table the deletions are to be made.

Something sad: we are not allowed use of the **with** clause here!

delete

To *delete* tuples from some table satisfying some condition:

```
delete from <table>
where <condition>;
```

Example: *delete*

Delete from **Like** the fact that *Sally* likes *Bud*.

```
delete from Likes
where drinker = 'Sally'
  and beer = 'Bud';
```

delete & sub-queries

- Can use sub-queries (with correlation) in the **where** clause, of course.
- Can replace the *table name* in the **delete**'s **from** clause, with the same caveats for **insert** and **update**.