LASSONDE SCHOOL OF ENGINEERING | YORK UNIVERSITÉ UNIVERSITY

SQL

- *The Basics*
- *Advanced*
- *Manipulation*

Version

initial	*2020-02-25*
last modified	*2020-08-17*

Acknowledgments

Thanks to

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

Printable version of talk

- Follow this link:
  SQL [to **pdf**].
- Then *print* to get a **PDF**.

(*This only works correclty in **Chrome** or **Chromium**.*)

SQL

a standard language for accessing databases

Many say SQL stands for *Structured Query Language*. (That is not quite right, but close enough!) 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, 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
  - an 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".

create tables

add, delete, update tables

The basic block

select … from … where…

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

π
ρ
σ, $\bowtie$

does projection ← Select → Π (relational algebra)

Running Examples

our YRB schema

- **yrb_customer(cid,** name, city)
- **yrb_club(club,** desp)
- **yrb_member(club, cid)**
- **yrb_category(cat)**
- **yrb_book(title, year,** language, cat, weight)
- **yrb_offer(club, title, year,** price)
- **yrb_purchase(cid, club, title, year, whenp,** qnty)
- **yrb_shipping(weight,** cost)

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

which customers are from New York?

```
select name
from yrb_customer
where city = 'New York';
```

The result of the query

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

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

name
Lux Luthor
Clark Kent
Phil Regis

“*” 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 yrb_customer
where city = 'New York';
```

Bad practice!

The result of the query

E.g.,

cid	name	city
23	Lux Luthor	New York
24	Clark Kent	New York
28	Phil Regis	New York

Renaming attributes

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

E.g.,

```
select name as person, city
from yrb_customer
where city = 'New York';
```

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 title, year, club, price,
  price*1.3 AS priceInCAD
from yrb_offer;
```

Constants as expressions

```
select name,
       'from New York' as peopleFromNY
from yrb_customer
where city = 'New York';
```

The result of the query

E.g.,

name	peoplefromny
Lux Luthor	from New York
Clark Kent	from New York
Phil Regis	from New York

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 yrb_offer
 where club  = 'AAA'
   and title = 'Vegetables are Good!';
```

How many tuples does this query return? *Why?*

```
select price
  from yrb_offer
 where club  = 'AAA'
   and title = 'Vegetables are Good!'
   and year  = '1987';
```

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.

Used in a a WHERE clause to search for a specified pattern in a column.

Example: like

```
select club
from yrb_club
where club like '%YRB%';
```

% → represents zero, one, or multiple characters

_ → (underscore) represents a single character

upto here on 29 Oct

tuple → row

NULL “values”

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

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.

Null → missing value
Null → inapplicable

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

club	title	year	price
YRB_3421	Richmond Underground	1997	NULL

```
select title, year
from yrb_offer
where club = 'YRB_3421'
  and price <  2.00
   or price >= 2.00;
```

The query returns the *empty* table!

(because price is NULL)

bag → multi-set, allows duplicates

Distinct vs All

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

```
select distinct …
```

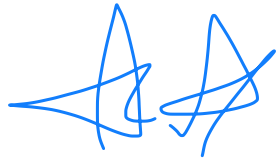
will return a *set* of tuples (that is, with any duplicates removed).

```
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`
- For `union`, the default is `distinct`!

Multi-relation queries ($\sigma\pi\bowtie$)



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 club, cat
from   yrb_book, yrb_offer
where  yrb_book.title = yrb_offer.title
and    yrb_book.year = yrb_offer.year;
```

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

In English?

How many tuples will return?

Variables / “aliases”

```
select club, cat
from   yrb_book as b, yrb_offer as o
where  b.title = o.title
and    b.year = o.year;
```

This is a nice shorthand, and can improve readability.

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

Variables / “aliases”

self-join → regular join, but the table is joined with itself

What if we need a self-join?

```
select ...
from   yrb_purchase as p1,
           yrb_purchase as p2
where  ...
```

→ aliasing

After rename, the alia:

- is the unique identifier for the table in the current query
- only valid in the current query
- The alias becomes the new name of the table reference so far as the current query is concerned — it is not allowed to refer to the table by the original name elsewhere in the query.

Intersection, Union, and Except ("—")

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

Distinct → used to return distinct / unique values

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

- Takes an optional keyword of `distinct` or `all` after (with `distinct` 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 clubs are there?

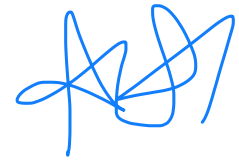
```
select count(*) as numOfClubs
from yrb_club;
```

Example

How many members does club AAA have?

```
select count(*) as numOfMembers
from yrb_member
where club = 'AAA';
```

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 club, how many (`count`) memebrs do they *'own'*. We want a two-column answer table — club and numOfMembers — that would report that with multiple rows (one for each club).

By composition

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

```
select club,
   (select count(*)
   from yrb_member as a
   where b.club=a.club) as numOfMember
from (select club
         from yrb_club) as b;
```

The result of the query

E.g.,

club	numofmember
AARP	7
AAA	18
CNU Club	7
W&M Club	10
...	...

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 members for each club

```
select      club, count(*) as numOfMembers
from        yrb_member
group by    club;
```

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*.

Example: Having

Query. *For each club (that at least has 15 members), how many members in that club?*

```
select club, count(*) as numOfMembers
from   yrb_member
group  by club
having count(*) > 15;
```

Example: Having (2)

```
select    club, count(*) as numOfMembers
from      yrb_member
group by club
having    numOfMembers> 15;
```

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

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

Example: Having (3)

without the having!

```
select club, numOfMembers
from (  select club,
                count(*) as numOfMembers
        from   yrb_member
        group by club
     ) as countOfMember
where numOfMembers > 15;
```

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 club, numOfMembers
from (  select club,
               count(*) as numOfMembers
        from   yrb_member
        group by club
      ) as countOfMember
where numOfMembers > 15;
```

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.
  - `(`coln_1`, …,` 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 club
from   yrb_offer
where  price = (select MAX(price)
                from yrb_offer);
```

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 title, year
from yrb_book as b
where not exists (
   select club
   from yrb_club
   except
   select club
   from yrb_offer as o
   where o.title = b.title
   and o.year = b.year
   );
```

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*.

Example

```
select club,
   (select count(*)
   from yrb_member as a
   where b.club=a.club) as NumOfMember
from (select club from yrb_club) as b;
```

Extension

What if a sub-query is referred more than once in one query?

How to divide and conquer complicated queries? Can we do that?

The **with** queries! (Common Table Expressions)

- **with** provides a way to write auxiliary statements.

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 of *insert*

Add the tuple to **yrb_offer** that club *Basic* offers book *Richmond Underground, 1997* for *15.95*:

```
insert into yrb_offer (title, year, club, price) values
    ('Richmond Underground',1997,'Basic',15.95);
```

Transactions

What is the difference between

```
insert into yrb_offer (title, year, club, price) values
    ('Richmond Underground',1997,'YRB_Bronze',15.45),
    ('Richmond Underground',1997,'W&M Club',12.95);
```

and

```
insert into yrb_offer (title, year, club, price) values
    ('Richmond Underground',1997,'YRB_Bronze',15.45);

insert into yrb_offer (title, year, club, price) values
    ('Richmond Underground',1997,'W&M Club',12.95);
```

?

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 yrb_customers (
    cid    smallint primary key,
    name   varchar(20),
    city   varchar(15)
           default 'Toronto'
);
```

Example of *insert* w/ default

```
insert into yrb_customers (cid) values
    ('1');
```

resulting in

cid	name	city
1	NULL	Toronto

Anywhere a table…

…you can put a (sub)query!

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 **CommonPurchase** (*cidA,nameA,cidB,nameB*) those customer pairs who buy at least one book in common.

```
insert into CommonPurchase (
    select distinct
        C.cid  as cidA,
        C.name as nameA,
        D.cid  as cidB,
        D.name as nameB
    from yrb_customer C, yrb-purchase P,
                         yrb_customer D, yrb_purchase Q
    where C.cid   = P.cid
      and D.cid   = Q.cid
      and P.title = Q.title
      and P.year  = Q.year
      and C.cid   > D.cid
);
```

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 of *update*

Change customer *Jack Daniels*'s address(city) to ‘Blacksburg’.

```
update yrb_customer
set city = 'Blacksburg'
where name = 'Jack Daniels';
```

Does this only update *one* tuple?

Example

update of multiple tuples

Make $70 the maximum price for a book.

```
update yrb_offer
set price = 70.00
where price > 70.00;
```

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 of *delete*

Delete from **yrb_offer** the fact that club *AAA* offers *Richmond Underground, 1997*.

```
delete from yrb_offer
where club = 'AAA'
  and title = 'Richmond Underground'
  and year = 1997
  ;
```

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**.