SQL

Fall, 2018

School of Computer Science
University of Waterloo

Databases CS348

SQL (Structured Query Language)

- Based on the Relational Calculus:

  ⇒ conjunctive queries
  aka. `SELECT` blocks

  ⇒ set operations

  ⇒ update language

  ⇒ non first-order features

- BAG (multiset) Semantics
- NULL values

  ⇒ avoid if at all possible

- A **committee** design

  ⇒ often more pragmatic than logical

  ⇒ evolving *standard*:
  SQL-89, **SQL-92**, SQL-1999,
  SQL:2003/2006/2008/2011/2016

SQL (cont.)

Three major parts of the language:

1. DML (Data Manipulation Language)

   ⇒ query language
   ⇒ update language

   Also: Embedded SQL (SQL/J) and ODBC (JDBC)
   ⇒ necessary for application development

2. DDL (Data Definition Language)

   ⇒ defines *schema* for relations
   ⇒ creates (modifies/destroys) database objects.

3. DCL (Data Control Language)

   ⇒ access control

SQL Data Types

Values of attributes in SQL:

`integer`	integer (32 bit)
`smallint`	integer (16 bit)
`decimal(m,n)`	fixed decimal
`float`	IEEE float (32 bit)
`char(n)`	character string (length `n`)
`varchar(n)`	variable length string (at most `n`)
`date`	year/month/day
`time`	hh:mm:ss.ss

Sample Database Revisited

```
AUTHOR(aid integer, name char(20))

WROTE(author integer, publication char(8))

PUBLICATION(pubid char(8), title char(70))

BOOK(pubid char(8),
     publisher char(50), year integer)

JOURNAL(pubid char(8),
        volume integer, no integer, year integer)

PROCEEDINGS(pubid char(8),
            year integer)

ARTICLE(pubid char(8), crossref char(8),
        startpage integer, endpage integer)
```

… SQL is **NOT** case sensitive.

The Basic "SELECT Block"

Basic syntax:

```
3 SELECT DISTINCT <results>
1 FROM            <tables>
2 WHERE           <condition>
```

- Allows formulation of conjunctive $(\exists, \wedge)$ queries of the form

$$\left\{\texttt{<results>} \mid \exists\texttt{<unused>}.\left(\bigwedge \texttt{<tables>}\right) \wedge \texttt{<condition>}\right\}$$

$\Rightarrow$ a conjunction of `<tables>` with `<condition>`

$\Rightarrow$ `<results>` specifies values in the resulting tuples, and

$\Rightarrow$ `<unused>` are variables *not used in* `<results>`

Example

List all authors in the database:

```
SQL> select distinct *
  2  from author;

AID NAME
--- --------------
  1 Toman, David
  2 Chomicki, Jan
  3 Saake, Gunter
```

The `FROM` clause cannot be used on its own

⇒ the "`SELECT *`" notation

⇒ also reveals all attribute names

Variables vs. Attributes

- Relational Calculus uses *positional* notation, i.e.,

  EMP(x, y, z) is true whenever the x, y, and z components of an answer can be found as a tuple in the instance of EMP

  ⇒ no need for *attribute names*
  ⇒ inconvenient for relations with high arity

- SQL uses *corelations* (tuple variables) and *attribute names* to assign *default variable names* to components of tuples:

  R [AS] p in SQL stands for $R(p.a_1, \ldots p.a_n)$ in RC
  where $a_1, \ldots, a_k$ are the *attribute names* declared for R.

Example

List all publications with at least two authors,

$$\{p \mid \exists a_1, a_2.\texttt{WROTE}(a_1, p) \land \texttt{WROTE}(a_2, p) \land a_1 \neq a_2\}:$$

```
SQL> select distinct r1.publication
  2  from wrote r1, wrote r2
  3  where r1.publication = r2.publication
  4  and r1.author != r2.author;

PUBLICATION
-----------
ChSa98
ChTo98
ChTo98a
```

⇒ cannot *share* a variable (p) in the two `WROTE` relations
need for explicit equality "`r1.publication = r2.publication`"

Example

List titles of all books,

$$\{t \mid \exists p, b, y.\mathtt{PUBLICATION}(p, t) \wedge \mathtt{BOOK}(p, b, y)\} :$$

```
SQL> select distinct title
  2  from publication, book
  3  where publication.pubid = book.pubid;

TITLE
----------------------------------------
Logics for Databases and Information Systems
```

⇒ relations can serve as their own *corelations* when *unambiguous*

`publication` stands for "`publication publication`", i.e.,

`publication(publication.pubid, publication.title)`

The "FROM" Clause (summary)

Syntax:

$$\texttt{FROM}\ R_1\ [\ [\texttt{AS}]\ n_1\ ], \ldots, R_k\ [\ [\texttt{AS}]\ n_k\ ]$$

- R_i are relation (table) names
- n_i are distinct identifiers
- The clause represents a **conjunction** $R_1 \wedge \ldots \wedge R_k$
  $\Rightarrow$ all variables of R_i's are *distinct*
  $\Rightarrow$ we use (co)relation names to resolve ambiguities
- Cannot appear alone
  $\Rightarrow$ only as a part of the *select block*

The "SELECT" Clause

Syntax:

$$\texttt{SELECT DISTINCT } e_1 \text{ [[}\texttt{AS}\text{] } n_1 \text{]}, \ldots, e_k \text{ [[}\texttt{AS}\text{] } n_k \text{]}$$

1. Eliminate superfluous attributes from answers (∃)
2. Form **expressions**:

   ⇒ built-in functions applied to values of attributes
3. Give names to attributes in the answer

Standard Expressions

We can create values in the answer tuples using **built-in** functions:

- On numeric types:

  $+, -, *, /, \ldots$ (usual arithmetic)
- On strings:

  $||$ (concatenation), `substr`, ...
- Constants (of appropriate types)

  "`SELECT 1`" is a valid query in SQL-92
- UDF (user defined functions)

Note: all attribute names **MUST** be “present” in the `FROM` clause.

Example

For every article list the number of pages:

```
SQL> select distinct pubid, endpage-startpage+1
  2  from article;

PUBID     ENDPAGE-STARTPAGE+1
-----     -------------------
ChTo98                     40
ChTo98a                    28
Tom97                      19
```

Naming Attributes in the Results

Results of queries $\iff$ Tables

What are the names of attributes in the result of a `SELECT` clause?

- A single attribute: inherits the name
- An expression: implementation dependent

We can—and should—**explicitly** name the resulting attributes:

$\Rightarrow$ `"<expr> AS <id>"` where `<id>` is the new name

Example

For every article list the number of pages, and name the resulting attributes `id,numberofpages`:

```
SQL> select distinct pubid as id,
  2         endpage-startpage+1 as numberofpages
  3  from article;

ID        NUMBEROFPAGES
-----     -------------
ChTo98               40
ChTo98a              28
Tom97                19
```

The "WHERE" Clause

Syntax:

```
WHERE <condition>
```

Additional conditions on tuples that qualify for the answer.

- Standard atomic conditions:
  1. equality: =, != (on all types)
  2. order: <, <=, >, >=, <> (on numeric and string types)
- Conditions may involve *expressions*

  ⇒ similar conditions as in the SELECT clause

Example(s)

Find all journals printed since 1997:

```
SQL> select * from journal where year >= 1997;

PUBID         VOLUME          NO       YEAR
-------- ---------- ---------- ----------
JLP-3-98          35           3       1998
```

Find all articles with more than 20 pages:

```
SQL> select * from article
  2  where endpage-startpage > 20;

PUBID    CROSSREF  STARTPAGE    ENDPAGE
-------- --------- --------- ----------
ChTo98   ChSa98           31         70
ChTo98a  JLP-3-98        263        290
```

Boolean Connectives

Atomic conditions can be combined using **boolean connectives**:

- `AND` (conjunction)
- `OR` (disjunction)
- `NOT` (negation)

Example

List all publications with at least two authors:

```
SQL> select distinct r1.publication
  2  from wrote r1, wrote r2
  3  where r1.publication = r2.publication
  4  and not r1.author = r2.author;

PUBLICATION
-----------
ChSa98
ChTo98
ChTo98a
```

Summary

- Simple SELECT block accounts for many queries

  $\Rightarrow$ all in $\exists, \wedge$ fragment of relational calculus

- Additional features
  - alternative names for relations
  - expressions and naming in the output
  - built-in atomic predicates and boolean connectives

- Well defined semantics (declarative and operational)

Complex Queries in SQL

- So far we can write only $\exists, \wedge$ queries
  $\Rightarrow$ the SELECT BLOCK queries
  $\Rightarrow$ not sufficient to cover all safe RC queries

- Remaining connectives:
  1. $\vee, \neg$: are expressed using **set operations**
     $\Rightarrow$ easy to enforce *range-restriction requirements*
  2. $\forall$: rewrite using negation and $\exists$
     $\Rightarrow$ the same for $\rightarrow$, $\leftrightarrow$, etc.

Set Operations at Glance

Answers to *Select Blocks* are **relations** (sets of tuples)

$\Rightarrow$ we can apply **set operations** on them

- Set union: Q_1 `UNION` Q_2

  $\Rightarrow$ the set of tuples in Q_1 or in Q_2.
  $\Rightarrow$ used to express "or".
- Set difference: Q_1 `EXCEPT` Q_2

  $\Rightarrow$ the set of tuples in Q_1 but not in Q_2.
  $\Rightarrow$ used to express "and not".
- Set intersection: Q_1 `INTERSECT` Q_2

  $\Rightarrow$ the set of tuples in both Q_1 and Q_2.
  $\Rightarrow$ used to express "and" (redundant, rarely used).

Q_1 and Q_2 must have **union-compatible** signatures:

$\Rightarrow$ same number and types of attributes

Example: Union

List all publication ids for books or journals:

```
SQL> (select distinct pubid from book)
  2  union
  3  (select distinct pubid from journal);

PUBID
-----
ChSa98
JLP-3-98
```

Example: Set Difference

List all publication ids except those for articles:

```
SQL> (select distinct pubid from publication)
  2  except
  3  (select distinct pubid from article);

PUBID
-----
ChSa98
DOOD97
JLP-3-98
```

What About Nesting of Queries?

We can use *SELECT Blocks* (and other *set operations*)
as arguments of *set operations*.

What if we need to use a **set operation** inside of a **SELECT Block**?

- Can use **distributive laws**
  $\Rightarrow (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$
  $\Rightarrow$ often **very** cumbersome
- Nest set operation inside a select block.
  $\Rightarrow$ *common table expressions*

Naming (Sub-)queries

Idea:

Queries denote **relations**. We provide a **naming** mechanism that allows us to assign names to (results of) queries.

⇒ can be used later in place of (base) relations.

- Syntax:

```
WITH foo1 [<opt-schema-1>]
     AS ( <query-1-goes-here> ),
     ...
     foon [<opt-schema-n>]
     AS ( <query-n-goes-here> )
<query-that-uses-foo1-...-foon-as-table-names>
```

Example

List all publication titles for books or journals:

```
SQL> with bookorjournal(pubid) as
       ( (select distinct pubid from book)
         union
         (select distinct pubid from journal)
       )
     select distinct title
     from publication, bookorjournal
     where publication.pubid = bookorjournal.pubid;

TITLE
------------------------------
Logics for Databases and Information Systems
Journal of Logic Programming
```

The `FROM` clause revisited

- Using the `WITH` mechanism is sometimes cumbersome:

  ⇒ we don't want to name every subexpression

- SQL-92 allows us to **inline** queries in the `FROM` clause:

  ```
  FROM ..., ( <query-here> ) <id>,...
  ```

  ⇒ `<id>` stands for the result of `<query-here>`.
  ⇒ unlike for base relations, `<id>` is **mandatory**.

- In "old" SQL (SQL-89) this does NOT work; *views* were the only option.

Example

List all publication titles for journals or books:

```
SQL> select distinct title
  2  from publication,
  3       ( (select distinct pubid from journal)
  4         union
  5         (select distinct pubid from book) ) jb
  6  where publication.pubid = jb.pubid;

TITLE
------------------------------
Logics for Databases and Information Systems
Journal of Logic Programming
```

Can't we just use `OR` instead of `UNION`?

- A **common** mistake:

  ⇒ use of `OR` in the `WHERE` clause instead of the `UNION` operator

- An incorrect solution:

```
select distinct title
from publication, book, journal
where publication.pubid = book.pubid
or publication.pubid = journal.pubid
```

- Often works, but consider where there are no `book`s.

Summary on First-Order SQL

- SQL introduced so far captures all of RC (relational calculus)

  $\Rightarrow$ optionally with duplicate semantics
  $\Rightarrow$ powerful (many queries can be expressed)
  $\Rightarrow$ efficient (PTIME, LOGSPACE)

- Shortcomings:

  $\Rightarrow$ some queries are hard to write (syntactic sugar)
  $\Rightarrow$ no *"counting"* (aggregation)
  $\Rightarrow$ no *"path in graph"* (recursion)

WHERE Subqueries

- Additional (complex) search conditions

  $\Rightarrow$ query-based search predicates

- Advantages
  - simplifies writing queries with negation
- Drawbacks
  - complicated semantics
    (especially when duplicates are involved)
  - **very** easy to make mistakes
- **VERY COMMONLY** used to formulate queries

Overview of WHERE Subqueries

- Presence/absence of a *single* value in a query

```
<attr> IN ( <query> )
<attr> NOT IN ( <query> )
```

- Relationship of a value to some/all values in a query

```
<attr> op SOME ( <query> )
<attr> op ALL ( <query> )
```

- Emptiness/non-emptiness of a query

```
EXISTS ( <query> )
NOT EXISTS ( <query> )
```

In the first two cases `<query>` must be unary.

Example: "<attr> in (<query>)"

```
SQL> select distinct title
  2  from publication
  3  where pubid in (select pubid from article);

TITLE
-------------------------------
Temporal Logic in Information Systems
Datalog with Integer Periodicity Constraints
Point-Based Temporal Extension of Temporal SQL
```

“Pure” SQL Equivalence

Nesting in the WHERE clause is mere syntactic sugar:

```
select r.b
from r
where  r.a in (
  select b
  from s
)
```

```
select r.b
from r, (
  select distinct b
  from s
) as s
where  r.a = s.b
```

All of the remaining constructs can be rewritten in similar fashion.

Example: "<attr> not in (<query>)"

All author-publication ids for all publications except books and journals:

```
SQL> select *
  2  from wrote
  3  where publication not in (
  4     ( select pubid from book )
  5     union
  6     ( select pubid from journal ) );

AUTHOR  PUBLICAT
------- -----
      1 ChTo98
      1 ChTo98a
      1 Tom97
      2 ChTo98
      2 ChTo98a
```

…search conditions may contain complex queries.

“<attr> not in (<query>)” (cont.)

. . . another formulation:

```
SQL> select *
  2  from wrote
  3  where publication not in (
  4          select pubid from book
  5  ) and publication not in (
  6          select pubid from journal
  7  )
```

. . . and may be combined using boolean connectives.

Example: “<attr> op SOME/ALL (<query>)”

Find the longest articles (a way expressing max):

```
SQL> select distinct pubid
     from article
     where endpage-startpage >= all (
         select endpage-startpage
         from   article
     );

PUBID
-----
ChTo98
```

“<attr> = some (<query>)” ≡ “<attr> in (<query>)”

“<attr> <> all (<query>)” ≡ “<attr> not in (<query>)”

Parametric Subqueries

- So far, *subqueries* were **independent** of the *main* query

  $\Rightarrow$ not correlated
  $\Rightarrow$ not much fun (good only for simple queries)

- SQL allows **parametric** (correlated) subqueries.

Parametric subqueries have the form "`<query>`" mentioning

$$\texttt{<attr>}_1, \texttt{<attr>}_2, \ldots$$

where $\texttt{<attr>}_i$ is an attribute in the main query.

The truth of a predicate defined by a subquery is determined for each substitution (tuple) in the main query:

1. instantiate all the parameters and
2. check for the truth value as before …

Example: "EXISTS (<query>)"

Parametric subqueries are most common for "existential" subqueries:

```
SQL> select *
  2  from wrote r
  3  where exists ( select *
  4       from wrote s
  5       where r.publication = s.publication
  6       and r.author <> s.author );

    AUTHOR PUBLICAT
---------- --------
         1 ChTo98
         1 ChTo98a
         2 ChTo98
         2 ChTo98a
         2 ChSa98
         3 ChSa98
```

Example: "NOT EXISTS (<query>)"

It is easy to now complement conditions:

```
SQL> select *
  2  from wrote r
  3  where not exists (
  4        select *
  5        from wrote s
  6        where r.publication = s.publication
  7        and r.author <> s.author
  8  );

    AUTHOR PUBLICAT
---------- --------
         1 Tom97
```

Example: "<attr> IN (<query>)"

```
SQL> select *
     from wrote r
     where publication in (
         select publication
         from wrote s
         where r.author <> s.author
     );

  AUTHOR PUBLICAT
-------- -----
       1 ChTo98
       1 ChTo98a
       2 ChTo98
       2 ChTo98a
       2 ChSa98
       3 ChSa98
```

More levels of Nesting

- WHERE subqueries are **just queries**
  - $\Rightarrow$ we can nest again and again and ...
  - $\Rightarrow$ every nested subquery can use attributes from the enclosing queries as parameters.
  - $\Rightarrow$ correct naming is imperative
- Used to formulate very complex **search conditions**
  - $\Rightarrow$ attributes present **in the subquery only** CANNOT be used to construct the result(s).

Example

List all authors who always publish with someone else:

```
SQL> select distinct a1.name
     from author a1, author a2
     where not exists (
        select *
        from   publication p, wrote w1
        where  p.pubid = w1.publication
          and  a1.aid = w1.author
          and  a2.aid not in (
                   select author
                   from   wrote
                   where  publication = p.pubid
                     and  author <> a1.aid
          )
     );
```

Summary

- WHERE subqueries enable easy formulation of queries of the form

  "All x in R such that (a part of) x doesn't appear in S".

  - Subqueries only stand for **`WHERE` conditions**

    $\Rightarrow$ CANNOT be used to produce results.

  - You can use input parameters, but these must be *bound* in the main query

- All of these are just a syntactic sugar and can be expressed using queries nested in the FROM clause

  - but it might be quite hard . . .
  - and it is easy to make mistakes (be **very** careful)

How do we Modify a Database?

- Naive approach:

  $DBSTART;$
  $r_1 := Q_1(DB);$
  $\ldots$
  $r_k := Q_k(DB);$
  $DBCOMMIT;$

- Not an acceptable solution in practice

Incremental Updates

Idea

Tables are large but **updates are small** ⇒ Incremental updates

1. Insertion of a tuples (`INSERT`)
   ⇒ constant tuple
   ⇒ results of queries
2. Deletion of tuples (`DELETE`)
   ⇒ based on match of a condition
3. Modification of tuples (`UPDATE`)
   ⇒ allows updating "in place"
   ⇒ based on match of a condition

SQL Insert

- One constant tuple (or a fixed number):

```
INSERT INTO r[(a1,...,ak)]
        VALUES (v1,...,vk)
```

$\Rightarrow$ adds tuples $(v_1, \ldots, v_k)$ to r.
$\Rightarrow$ the type of $(v_1, \ldots, v_k)$ must match the schema definition of r.

- Multiple tuples (generated by a query):

```
INSERT INTO r ( Q )
```

$\Rightarrow$ adds result of Q to r

Example: inserton of a tuple

Add a new author:

```
SQL> insert into author
  2        values (4, 'Niwinski, Damian',
  3               'zls.mimuw.edu.pl/~niwinski');

1 row created.

SQL> select distinct aid,name,url from author;

AID NAME              URL
--- --------------    ---------------------
  1 Toman, David      db.uwaterloo.ca/~david
  2 Chomicki, Jan     cs.monmouth.edu/~chomicki
  3 Saake, Gunter
  4 Niwinski, Damian zls.mimuw.edu.pl/~niwinski
```

Example: use of a query

Add a new author (without looking up author id):

```
SQL> insert into author (
 2     select max(aid)+1, 'Snodgrass, Richard T.',
 3            'www.cs.arizona.edu/people/rts'
 4     from author );
1 row created.

SQL> select distinct aid, name from author;

AID NAME
-- ---------------
  1 Toman, David
  2 Chomicki, Jan
  3 Saake, Gunter
  4 Damian Niwinski
  5 Snodgrass, Richard T.
```

SQL Delete

- Deletion using a condition:

```
DELETE FROM r
WHERE cond
```

$\Rightarrow$ deletes **all** tuples that match `cond`.

- Deletion using cursors (later)

$\Rightarrow$ available in embedded SQL
$\Rightarrow$ only way to delete one out of two duplicate tuples

Example

Delete all publications that are not articles or the collections an article appears in:

```
SQL> delete from publication
  2          where pubid not in (
  3                 select pubid
  4                 from article
  5          ) and pubid not in (
  6                 select crossref
  7                 from article
  8          );

0 rows deleted.
```

SQL Update

- Two components:
  1. an update statement (`SET`)

     ⇒ an assignment of values to attributes.
  2. a search condition (`WHERE`)

- Syntax:

```
UPDATE r
SET    <update statement>
WHERE  <condition>
```

Example

```
SQL> update author
  2  set    url = 'brics.dk/~david'
  3  where  aid in (
  4             select aid
  5             from author
  6             where name like 'Toman%'
  7           );

1 row updated.

SQL> select * from author;

AID NAME                URL
--- --------------- ---------------------
  1 Toman, David     //brics.dk/~david
  ...
```

Support for Transactions

The DBMS guarantees noninterference (serializability) of all data access requests to tables in a database instance

- Transaction starts with first **access** of the database

  ⇒ until it sees:
- `COMMIT`: make changes permanent

  ```
  SQL> commit;
  Commit complete.
  ```
- or `ROLLBACK`: discard changes

  ```
  SQL> rollback;
  Rollback complete.
  ```

Aggregation

Standard and very useful extension of First-Order Queries.

- Aggregate (column) functions are introduced to
  $\Rightarrow$ find number of tuples in a relation
  $\Rightarrow$ add values of an attribute (over the whole relation)
  $\Rightarrow$ find minimal/maximal values of an attribute
- Can apply to *groups* of tuples that have equal values for (selected) attributes
- Can **NOT** be expressed in Relational Calculus

Aggregation in SQL

The same in SQL syntax:

```
SELECT   x1,...,xk, agg1,...,aggl
FROM     Q
GROUP BY x1,...,xk
```

Restrictions:

- All attributes in the `SELECT` clause that are **NOT** in the scope of an aggregate function **MUST** appear in the `GROUP BY` clause.
- `aggi` are of the form `count(*)`, `count(<expr>)`, `sum(<expr>)`, `min(<expr>)`, `max(<expr>)`, or `avg(<expr>)` where `<expr>` is usually an attribute of `Q` (and usually not in the `GROUP BY` clause).

Operational Reading

1. Partition the input relation to groups with equal values of **grouping** attributes
2. On each of these partitions apply the **aggregate function**
3. Collect the results and form the answer

Example (count)

For each publication, count the number of authors:

```
SQL> select publication, count(author)
  2  from wrote
  3  group by publication;

PUBLICAT COUNT(AUTHOR)
-------- -------------
ChSa98               2
ChTo98               2
ChTo98a              2
Tom97                1
```

Example (sum)

For each author, count the number of article pages:

```
SQL> select author,sum(endpage-startpage+1) as pgs
  2  from wrote, article
  3  where publication=pubid
  4  group by author;

    AUTHOR       PGS
---------- --------
         1       87
         2       68
```

. . . not quite correct: it doesn't list 0 pages for author 3.

The HAVING clause

- The WHERE clause can't impose conditions on values of aggregates
  $\Rightarrow$ WHERE clause has to be used **before** GROUP BY

- SQL allows a HAVING clause instead
  $\Rightarrow$ like WHERE, but for aggregate values...

- The aggregate functions used in the HAVING clause may be different from those in the SELECT clause; the grouping, however, is common.

The HAVING clause is mere *SYNTACTIC SUGAR*

... and can be replaced by a nested query and a WHERE clause.

Example

List publications with exactly one author:

```
SQL> select publication, count(author)
  2  from wrote
  3  group by publication
  4  having count(author) = 1;

PUBLICAT COUNT(AUTHOR)
----- ---------
Tom97                1
```

. . . This query *can* be written without aggregation.

Example (revisited.)

For every author, count the number of books and articles:

```
SQL> select distinct aid, name, count(publication)
     from author, (
      ( select distinct author, publication
        from wrote, book
        where publication = pubid )
     union all
      ( select distinct author, publication
        from wrote, article
        where publication = pubid ) ) ba
     where aid = author
     group by name, aid;
```

Summary

- SQL covered so far:
  1. Simple SELECT BLOCK
  2. Set operations
  3. Formulation of complex queries, nesting of queries, and views
  4. Updating Data
  5. Aggregation
- This covers pretty much all of the useful SQL DML
  $\Rightarrow$ the Bad and Ugly coming next . . .