SQL
Spring 2017

David Toman
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 and Aggregation
  - Update language
- BAG (multiset) Semantics
- NULL values
  - avoid if at all possible
- A committee design
  - often more pragmatic than logical
  - evolving standard:
    SQL/89, SQL/92, SQL3('96),

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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>SQL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer (32 bit)</td>
<td></td>
</tr>
<tr>
<td>smallint</td>
<td>integer (16 bit)</td>
<td></td>
</tr>
<tr>
<td>decimal(m,n)</td>
<td>fixed decimal</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>IEEE float (32 bit)</td>
<td></td>
</tr>
<tr>
<td>char(n)</td>
<td>character string (length n)</td>
<td></td>
</tr>
<tr>
<td>varchar(n)</td>
<td>variable length string (at most n)</td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>year/month/day</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>hh:mm:ss.ss</td>
<td></td>
</tr>
</tbody>
</table>
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 (∃, ∧) queries of the form

\[ \{\langle \text{results} \rangle \mid \exists \langle \text{unused} \rangle. (\land \langle \text{tables} \rangle) \land \langle \text{condition} \rangle \} \]

⇒ a conjunction of <tables> with <condition>
⇒ <results> specifies values in the resulting tuples, and
⇒ <unused> are variables not used in <results>

Example

List all authors in the database:

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

<table>
<thead>
<tr>
<th>AID</th>
<th>NAME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toman, David</td>
<td><a href="http://db.uwaterloo.ca/~david">http://db.uwaterloo.ca/~david</a></td>
</tr>
<tr>
<td>2</td>
<td>Chomiccki, Jan</td>
<td><a href="http://cs.buffalo.edu/~chomiccki">http://cs.buffalo.edu/~chomiccki</a></td>
</tr>
<tr>
<td>3</td>
<td>Saake, Gunter</td>
<td></td>
</tr>
</tbody>
</table>

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.,
  \( \text{EMP}(x, y, z) \) is true whenever the \( x, y, \) and \( z \) components
  of an answer can be found as a tuple in the instance of \( \text{EMP} \)
⇒ 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:
  \( R[\text{AS}] p \) 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 \( R \).
List all publications with at least two authors,
\[ \{ p \mid \exists a_1, a_2. \text{wrote}(a_1, p) \land \text{wrote}(a_2, p) \land a_1 \neq a_2 \} : \]

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

<table>
<thead>
<tr>
<th>PUBLICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChSa98</td>
</tr>
<tr>
<td>ChTo98</td>
</tr>
<tr>
<td>ChTo98a</td>
</tr>
</tbody>
</table>

⇒ cannot share a variable \((p)\) in the two \text{wrote} relations
need for explicit equality \(r1.\text{publication}=r2.\text{publication}\)

Example

List titles of all books,
\[ \{ t \mid \exists p, b, y. \text{publication}(p, t) \land \text{book}(p, b, y) \} : \]

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

<table>
<thead>
<tr>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logics for Databases and Information Systems</td>
</tr>
</tbody>
</table>

⇒ relations can serve as their own \text{corelations} when \text{unambiguous}
publication stands for “publication publication”, i.e.,
publication(publication.pubid, publication.title)

The "FROM" Clause (summary)

Syntax:
```
FROM R_1[n_1], \ldots, R_k[n_k]
```

- \(R_i\) are relation (table) names
- \(n_i\) are distinct identifiers
- the clause represents a conjunction \(R_1 \land \ldots \land R_k\)
  ⇒ all variables of \(R_i\)'s are distinct
  ⇒ we use (co)relation names to resolve ambiguities
- can NOT appear alone
  ⇒ only as a part of the \text{select} block

The "SELECT" Clause

Syntax:
```
SELECT DISTINCT e_1[AS i_1], \ldots, e_k[AS i_k]
```

1. eliminate superfluous attributes from answers (\(\exists\))
2. form \text{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.

### 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:

\[
\Rightarrow "<\text{expr}> \text{ AS } <\text{id}>" \text{ where } <\text{id}> \text{ is the new name}
\]
The "WHERE" Clause

Additional conditions on tuples that qualify for the answer.

```
WHERE C
```

- 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 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
  ⇒ all in $\exists, \land$ 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 $\exists, \land$ queries
  ⇒ the SELECT BLOCK queries
  ⇒ not sufficient to cover all RC queries
- remaining connectives:
  1. $\lor, \neg$: are expressed using set operations
     ⇒ easy to enforce range-restriction requirements
  2. $\forall$: rewrite using negation and $\exists$
     ⇒ the same for $\to, \leftrightarrow$, etc.

Set Operations at Glance

Answers to Select Blocks are relations (sets of tuples)

- we can apply set operations on them:
  - set union: $Q_1 \text{ UNION } Q_2$
    ⇒ the set of tuples in $Q_1$ or in $Q_2$.
    ⇒ used to express “or”.
  - set difference: $Q_1 \text{ EXCEPT } Q_2$
    ⇒ the set of tuples in $Q_1$ but not in $Q_2$.
    ⇒ used to express “and not”.
  - set intersection: $Q_1 \text{ INTERSECT } Q_2$
    ⇒ the set of tuples in both $Q_1$ and $Q_2$.
    ⇒ used to express “and” (redundant, rarely used).

$Q_1$ and $Q_2$ must have union-compatible signatures:
  ⇒ same number and types of attributes

Example: Union

List all publication ids for books or journals:

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

PUBID
-----
ChSa98
JLP-3-98
Example: Set Difference

List all publication ids except those for articles:

```
SQL> (select pubid from publication) 
  2   except 
  3 (select 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 is we need to use a `Set Operation` inside of a `SELECT Block`?

- we can use distributive laws
  \[(A \lor B) \land C \equiv (A \land C) \lor (B \land C)\]
  \[\Rightarrow \text{often very cumbersome}\]
- nest set operation inside a select block.
  \[\Rightarrow \text{common table expressions}\]

Naming (Sub-)queries

Idea:

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

\[\Rightarrow \text{can be used later in place of (base) relations.}\]

Syntax:

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

Example

List all publication titles for books or journals:

```
SQL> with bookorjournal as 
  2   ( (select pubid from book) 
  3   union 
  4   (select pubid from journal) 
  5   ) 
  6 select title 
  7 from publication, bookorjournal 
  8 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 books or journals:

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

---

**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:
  ```sql
  SELECT title
  FROM publication, book, journal
  WHERE publication.pubid=book.pubid
  OR publication.pubid=journal.pubid
  ```

- often works; but imagine there are no **books**...

---

**Summary on First-Order SQL**

- **SQL introduced so far** captures all of **Relational Calculus**
  ⇒ optionally with duplicate semantics
  ⇒ powerful (many queries can be expressed)
  ⇒ efficient (PTIME, LOGSPACE)

- **Shortcomings:**
  ⇒ some queries are hard to write (syntactic sugar)
  ⇒ no “**counting**” (aggregation)
  ⇒ no “**path in graph**” (recursion)
WHERE Subqueries

- Additional (complex) search conditions
  ⇒ 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

Advantages
- simplifies writing queries with negation

Drawbacks
- complicated semantics
  (especially when duplicates are involved)
- very easy to make mistakes

Example: "attr IN ( Q )"

```
SQL> select title
2  from publication
3  where pubid in ( 4   select pubid from article
5 )  

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

Overview of WHERE Subqueries

- presence/absence of a single value in a query
  Attr IN ( Q )
  Attr NOT IN ( Q )

- relationship of a value to some/all values in a query
  Attr op SOME ( Q )
  Attr op ALL ( Q )

- emptiness/non-emptiness of a query
  EXISTS ( Q )
  NOT EXISTS ( Q )

In the first two cases Q has to be unary.

“Pure” SQL Equivalence

Nesting in the WHERE clause is mere syntactic sugar:

```
SELECT r.b FROM r, (  SELECT distinct b
WHERE r.a IN (  FROM s
    SELECT b  ) s
    FROM s
  ) s
WHERE r.a=s.b

All of the remaining constructs can be rewritten in similar fashion…```
Example: "attr NOT IN ( Q )"
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 ChTo98    2 ChTo98a
```

... search conditions may contain complex queries.

Example: "attr op SOME/ALL ( Q )"
Find article with most pages:

```
SQL> select pubid
2   from article
3   where endpage-startpage>=all ( 
4       select endpage-startpage
5       from article
6     )

  PUBID
  ------
  ChTo98
```

...another way of saying max

```
attr = SOME (Q) is the same as attr IN (Q)
attr <> ALL (Q) is the same as attr NOT IN (Q)
```

"attr NOT IN ( Q )" (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.

Parametric Subqueries

- so far subqueries were independent on the main query
  ⇒ not correlated
  ⇒ not much fun (good only for simple queries)

- SQL allows parametric (correlated) subqueries:
  ⇒ of the form Q(p1,...pk) where
    pis are attributes 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**

Parametric subqueries are most common for “existential” subqueries:

```sql
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**

…and now it is easy to complement conditions:

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

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

Example: **IN**

```sql
SQL> select *
2   from wrote r
3   where publication in ( select publication
4       from wrote s
5       where r.author<>s.author
6     )

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

More levels of Nesting

- **WHERE subqueries are just queries**
  - we can nest again and again and . . .
  - every nested subquery can use attributes from the enclosing queries as parameters.
  - correct naming is imperative

- **used to formulate very complex search conditions**
  - 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
SQL> select a1.name, a2.name
2     from author a1, author a2
3     where not exists (
4           select *
5           from    publication p, wrote w1
6           where p.pubid=w1.publication
7           and a1.aid=w1.author
8           and a2.aid not in ( 
9               select author
10                  from    wrote
11                  where publication=p.pubid
12                  and author<>a1.aid
13         )
14       )
```

Summary

- WHERE subqueries—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... 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...
- easy to make mistakes though (be very careful)

How do we Modify a Database?

- Naive approach:
  ```
  DBSTART;
  r_1 := Q_1(DB);
  ...
  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)
  
  ⇒ adds tuples (v1,...,vk) to r.
  ⇒ the type of (v1,...,vk) must match the schema definition of r.
  ```

- multiple tuples (generated by a query):

  ```
  INSERT INTO r ( Q )
  
  ⇒ 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 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    Damian Niwinski zls.mimuw.edu.pl/~niwinski
```

Example: use of a query

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

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

```
-- SQL> select 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
  
  ⇒ deletes all tuples that match cond.
  ```

- Deletion using cursors (later)

  ⇒ available in embedded SQL
  ⇒ only way to delete one out of two duplicate tuples
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.
```

Example

```
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)
  2. a search condition (WHERE)

- **Syntax:**
  ```
  UPDATE r
  SET <update statement>
  WHERE <condition>
  ```

Example

```
Example

Update author

```
```
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

- **COMMIT:** make changes permanent
  ```
  SQL> commit;
  Commit complete.
  ```

- **ROLLBACK:** discard changes
  ```
  SQL> rollback;
  Rollback complete.
  ```
**Aggregation**

Standard and most useful extension of First-Order Queries.

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

**Aggregation in SQL**

The same in SQL syntax:

```
SELECT x1,...,xk, agg1,...,agg1
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(y), sum(y), min(y), max(y), or avg(y)` where `y` is an attribute of `Q` (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
```

<table>
<thead>
<tr>
<th>AUTHOR</th>
<th>PGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td>68</td>
</tr>
</tbody>
</table>

... 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
  ⇒ **WHERE** clause has to be used **before** **GROUP BY**

- SQL allows a **HAVING** clause instead
  ⇒ 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 nested query and a **WHERE** clause

Example (revisited.)

For every author count the number of books and articles:

```
SQL> select name, count(aid)
2 from author, ( 
3 ( select author 
4 from wrote, book 
5 where publication=pubid )
6 union all 
7 ( select author 
8 from wrote, article 
9 where publication=pubid ) ) ba
10 where aid=author
11 group by name,aid;
```

<table>
<thead>
<tr>
<th>PUBLICAT</th>
<th>COUNT(AUTHOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom97</td>
<td>1</td>
</tr>
</tbody>
</table>

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

⇒ the Bad and Ugly coming next...