

Information Systems (Informationssysteme)

Jens Teubner, TU Dortmund
`jens.teubner@cs.tu-dortmund.de`

Summer 2013

Part VI

SQL: Structured Query Language



We already saw the “Hello World!” example of SQL:

```
SELECT  $A_1, \dots, A_n$ 
FROM  $R_1, \dots, R_m$ 
WHERE  $C$ 
```

Semantics:

- All relations R_1, \dots, R_m listed in the **FROM** clause are combined into a **Cartesian product** $R_1 \times \dots \times R_m$.
- The **WHERE** clause **filters** all rows according to the condition C . (Absence of the **WHERE** clause is equivalent to $C \equiv \text{true}$.)
- The **SELECT** clause specifies the **attributes** A_1, \dots, A_n to report in the result ($*$ \equiv all attributes that occur in R_1, \dots, R_m).

Tuple Variables

SQL adopted the notion of **tuple variables**:

```
SELECT i.Name, i.InStock, s.Supplier, s.Price
      FROM Ingredients AS i, SoldBy AS s
      WHERE i.Name = s.Ingredient
            AND s.Price < i.Price
```

Tuple variables **range over tuples**; e.g., *i* represents a **single row** in *Ingredients*.

- If **no tuple variable** is given explicitly, a variable will **automatically be created** with the **name of the table**:

```
FROM Foo ≡ FROM Foo AS Foo
```

(If a variable is given in the query, the implicit variable is **not** declared.)

- The keyword **AS** is optional.

Attribute References

Attributes can be referenced in the form

$$v.A ,$$

where v is a tuple variable and A an attribute name.

If attribute name A is **unambiguous**, the tuple variable may be **omitted**:

```
SELECT Name, InStock, Supplier, s.Price
FROM Ingredients AS i, SoldBy AS s
WHERE Name = Ingredient
AND s.Price < i.Price
```

Personal recommendation:

- Fully qualify all attribute names (except for trivial queries).
- Avoid using $*$.

Consider a query with two tables in the FROM clause:

```
SELECT s.Name, c.Name AS Contact, c.Phone
      FROM Suppliers AS s, ContactPersons AS c
      WHERE s.SuppID = c.SuppID
```

The semantics of this query can be understood as follows:

- Enumerate all pairs of tuples $\langle s, c \rangle$ from the **Cartesian product** $Suppliers \times ContactPersons$ (the number of pairs may be huge).
- Among all pairs $\langle s, c \rangle$, select only those that satisfy the **join condition** $s.SuppID = c.SuppID$.

Most likely, your system will choose a **better evaluation strategy**.

- *E.g.*, using **indexes** or efficient **join algorithms**.
- But the **output** is the **same** as if obtained by full enumeration.



The **join condition** must be specified explicitly in the **WHERE** clause (otherwise, the system will assume you want the Cartesian product).

It is almost always an **error** when two tuple variables are not **linked** by an explicit join predicate (this query most likely returns nonsense):

```
SELECT s.Name, c.Name AS Contact, c.Phone
   FROM Suppliers AS s, ContactPersons AS c
  WHERE s.Name = 'Shop Rite'
        AND c.Phone LIKE '+49 351%'
```

→ In case of **composite keys** (that span multiple attributes), don't forget to link tuple variables via **all** key columns.

 What does the following query return?

```
SELECT c.CocktailID, c.Name
      FROM Cocktails AS c, ConsistsOf AS co,
           Ingredients AS i
      WHERE c.CocktailID = co.CocktailID
            AND co.IngrID = i.IngrID
            AND i.Alcohol > 0
```

To **eliminate duplicates** use the keyword DISTINCT:

```
SELECT DISTINCT c.CocktailID, c.Name
              ...           ...
```


Unnecessary Joins

Do not join **more** tables than needed

→ Query might run slowly if the optimizer overlooks the redundancy.

```
SELECT c.Name, c.Phone
      FROM Suppliers AS s, ContactPersons AS c
      WHERE s.SuppID = c.SuppID
            AND c.Phone LIKE '+49 351%'
```

Unnecessary joins might also lead to **unexpected results**.

What is wrong with these two queries?

- 1 Return all supplier names with an address in 'Dresden':

```
SELECT s.Name
      FROM Suppliers AS s, ContactPersons AS c
     WHERE s.SuppID = c.SuppID
           AND s.Address LIKE '%Dresden%'
```

- 2 Return all cocktails with 'Bacardi' in their name:

```
SELECT c.Name
      FROM Cocktails AS c, ConsistsOf AS co,
           Ingredients AS i
     WHERE c.CocktailID = co.CocktailID
           AND co.IngrID = i.IngrID
           AND c.Name LIKE '%Bacardi%'
```

Non-Monotonic Behavior

SQL queries that use only the constructs introduced above are **monotonic** (↗ slide 104).

→ If further tuples are **inserted** to the database, the query result can only **grow**.

Some real-world queries, however, demand **non-monotonic** behavior.

■ *E.g., “Return all non-alcoholic cocktails (i.e., those without any alcoholic ingredient).”*

→ Insertion of a new *ConsistsOf* tuple could “make” a cocktail alcoholic and thus invalidate a previously correct answer.

Such queries **cannot** be answered with the SQL subset we saw so far.

Indicators for non-monotonic behavior (in natural language):

- “there is no”, “does not exist”, etc.
 - **existential quantification**
- “for all”, “the minimum/maximum”
 - **universal quantification**
 - $\forall r \in R : C(r) \Leftrightarrow \nexists r' \in R : \neg C(r')$

In an equivalent SQL formulation of such queries, this ultimately leads to a test whether a certain **query yields a (non-)empty result**.

Such tests can be expressed with help of the IN (\in) and NOT IN (\notin) keywords in SQL:

```
SELECT c.Name
  FROM Cocktails AS c
 WHERE CocktailID NOT IN (SELECT co.CocktailID
                          FROM ConsistsOf AS co,
                          Ingredients AS i
                          WHERE i.IngrID = co.IngrID
                          AND i.Alcohol <> 0)
```

The IN (NOT IN) keyword tests whether an attribute value appears (does not appear) in a set of values computed by another SQL **subquery**.

→ At least conceptually, the subquery is evaluated before the main query starts.

The existence of a value in a subquery does not depend on multiplicity.

→ The previous query may equivalently be written as:

```
SELECT Name
  FROM Cocktails
 WHERE CocktailID NOT IN (SELECT DISTINCT CocktailID
                          FROM ConsistsOf AS co,
                          Ingredients AS i
                          WHERE i.IngrID = co.IngrID
                          AND i.Alcohol > 0)
```

Whether/how this will affect query performance depends on the particular system and data.

→ The DBMS optimizer likely knows about this equivalence and decide on duplicate elimination/preservation itself.

Consider again the query for all alcoholic cocktails.

 **Do the following queries return the same result?**

```
SELECT Name
  FROM Cocktails
 WHERE CocktailID IN (SELECT DISTINCT CocktailID
                      FROM ConsistsOf AS co,
                      Ingredients AS i
                      WHERE i.IngrID = co.IngrID
                      AND i.Alcohol > 0)
```

```
SELECT DISTINCT c.Name
  FROM Cocktails AS c, ConsistsOf AS co,
       Ingredients AS i
 WHERE c.CocktailID = co.CocktailID
       AND co.IngrID = i.IngrID AND i.Alcohol > 0
```

Remarks:

- In earlier versions of SQL, the subquery must return only a **single output column**.
 - This ensures that the result of the subquery is a set of atomic values and not an arbitrary relation.
- Since SQL-92, comparisons were extended to the **tuple level**. It is thus valid to write, *e.g.*:

```
        ⋮  
WHERE (A, B) NOT IN (SELECT C, D FROM ...)
```


EXISTS / NOT EXISTS

The construct `NOT EXISTS` enables the main (or outer) query to check whether the **result of a subquery is empty**.¹⁰

- In the subquery, tuple variables declared in the `FROM` clause of the outer query may be referenced.

```
SELECT Name
  FROM Cocktails AS c
 WHERE NOT EXISTS (SELECT DISTINCT CocktailID
                   FROM ConsistsOf AS co,
                   Ingredients AS i
                   WHERE i.IngrID = co.IngrID
                        AND co.CocktailID = c.CocktailID
                        AND i.Alcohol > 0)
```

¹⁰Likewise, `EXISTS` tests for non-emptiness.

Correlated Subqueries

The reference of an outer tuple makes the subquery **correlated**.

- The subquery is **parameterized** by the outer tuple variable.
- Conceptually, correlated subqueries have to be **re-evaluated** for every new binding of a tuple to the outer tuple variable.
 - Again, the DBMS is free to choose a more efficient evaluation strategy that returns the same result (↷ “query unnesting”)

Correlation can be used with `IN/NOT IN`, too.

- Typically, this yields complicated query formulations (bad style).

Queries with `EXISTS/NOT EXISTS` can be non-correlated.

- The `WHERE` predicate then becomes **independent** of the outer tuple.
- This is rarely desired and almost always an indication of an **error**.

Correlated Subqueries

Subqueries may reference tuple variables from the **outer query**.

The converse (referencing a tuple variable of the subquery in the outer query) is **not** allowed:

```
SELECT c.Name, i.Alcohol ← wrong!
  FROM Cocktails AS c
 WHERE EXISTS (SELECT DISTINCT CocktailID
               FROM ConsistsOf AS co,
               Ingredients AS i
               WHERE i.IngrID = co.IngrID
                  AND co.CocktailID = c.cocktailID
                  AND i.Alcohol > 0)
```

→ Compare this to **variable scoping** in block-structured programming languages (C, Java).

- EXISTS/NOT EXISTS only tests for the **existence** of (at least) one row in the subquery result.
- The **actual tuple value** returned by the query is **immaterial** to the overall query result.
- It is good style to make this explicit in the subquery phrasing:
 - ... EXISTS (SELECT * FROM ...)
 - ... EXISTS (SELECT NULL FROM ...)
 - ... EXISTS (SELECT 42 FROM ...)
- It is legal SQL syntax, though, to specify arbitrarily complex result tuples in the subquery's SELECT clause.


Mathematical logic knows **two quantifiers**:

- $\exists x : \phi$ **existential quantifier**
There is an x that satisfies formula ϕ .
- $\forall x : \phi$ **universal quantifier**
For all x , formula ϕ is satisfied.

We saw an SQL notation to express **existential quantification**.

Universal quantification can be expressed due to the equivalence

$$\forall x : \phi \Leftrightarrow \neg \exists x : \neg \phi .$$

 **State the query “Which is the most expensive cocktail?”**
(*I.e.*, the cocktail that is at least as expensive as all other cocktails.)

For a restricted form of quantification, SQL provides additional notation.

- Comparison of a **single value** with the **values in a set** (that is computed by a subquery).

```
SELECT c1.Name
  FROM Cocktails AS c1
 WHERE c1.Price >= ALL (SELECT c2.Price
                        FROM Cocktails AS c2)
```

- Prices of qualifying outer rows must be greater or equal than **all** prices returned by the subquery.
- Analogously: Comparisons =, <, etc.

ANY can be used instead of ALL if one match should be enough to satisfy the overall predicate.

```
SELECT c1.Name
   FROM Cocktails AS c1
  WHERE NOT c1.Price < ANY (SELECT c2.Price
                             FROM Cocktails AS c2)
```

- SOME can be used as a synonym for ANY.

- ANY/ALL do **not** extend the expressiveness of SQL, since, e.g.,

$$\begin{aligned} A < \text{ANY} (\text{SELECT } B \text{ FROM } \dots \text{ WHERE } \dots) \\ \equiv \\ \text{EXISTS} (\text{SELECT } * \text{ FROM } \dots \text{ WHERE } \dots \text{ AND } A < B) \end{aligned}$$

- $x \text{ IN } S$ is equivalent to $x = \text{ANY } S$.
- The subquery must yield a **single result column**.
- If none of the keywords ALL, ANY, or SOME are present, the subquery must yield **at most one row**.
 - This is a **semantical property** of the query, which the query compiler cannot check for you.
 - This is a common source of trouble (your query might run well when you test, but fail on real data).



Subqueries in the FROM Clause

Since the result of an SQL query is a **table**, it seems most natural to use a subquery result whenever a table might be specified, *i.e.*, in the **FROM** clause.

```
SELECT c.Name AS CocktailName, x.IngrName
      FROM (SELECT co.CocktailID, i.Name AS IngrName
            FROM ConsistsOf AS co, Ingredients AS i
            WHERE co.IngrID = i.IngrID) AS x,
           Cocktails AS c
      WHERE c.CocktailID = x.CocktailID
```

- SQL is **orthogonal** in this sense.
Earlier versions of SQL (up to SQL-86) were not orthogonal in this sense.
- Inside the subquery, tuple variables in the same **FROM** clause **may not be referenced**.

Subqueries in the FROM Clause

Subqueries in the FROM clause may occur implicitly because of **view declarations**, *e.g.*,

```
CREATE VIEW ConsistsOfIngr AS
  SELECT co.CocktailID, i.Name AS IngrName
  FROM ConsistsOf AS co, Ingredients AS i
  WHERE co.IngrID = i.IngrID
```

- This view declaration **permanently registers** the subquery under the name `ConsistsOfIngr`.
- After declaration, the view may be used in queries **just like a table**.

```
SELECT c.Name AS CocktailName, x.IngrName
  FROM ConsistsOfIngr AS x, Cocktails AS c
  WHERE c.CocktailID = x.CocktailID
```

Views are not only for convenience.

- They help to provide **logical data independence**.
 - *E.g.*, replace an actual table by a view declaration that computes the logical table content.
 - See slide 20 for an example.
- They can be used for **access control**.
 - *E.g.*, **deny** a certain user access to the base table(s), but **allow** access to a view over those tables. Access is now restricted to only those data generated by the view.

- **Aggregation functions** are functions from a multiset to a single value, *e.g.*,

$$\min\{42, 57, 5, 13, 27\} = 5 .$$

- SQL defines five main aggregation functions:

COUNT, SUM, AVG, MAX, MIN .

(Some implementations might provide further aggregation functions: STDDEV, VARIANCE, ...)

- Example:

```
SELECT MAX (Price)
FROM Ingredients
WHERE Alcohol = 0
```

Aggregations and Duplicates

Some aggregation functions are sensitive to **duplicates**.

If so, SQL allows to explicitly request to **ignore duplicates**:

```
SELECT COUNT (DISTINCT City)
      FROM Suppliers
      WHERE ZipCode LIKE '0%'
```

If you are only interested in **counting rows**, use COUNT (*):

```
SELECT COUNT (*)
      FROM Ingredients
      WHERE Alcohol > 0
```

There is a subtle difference between COUNT (*) and COUNT (A). The former will count all rows; the latter only those where attribute A does not contain a null value. The latter might be much more expensive to evaluate!

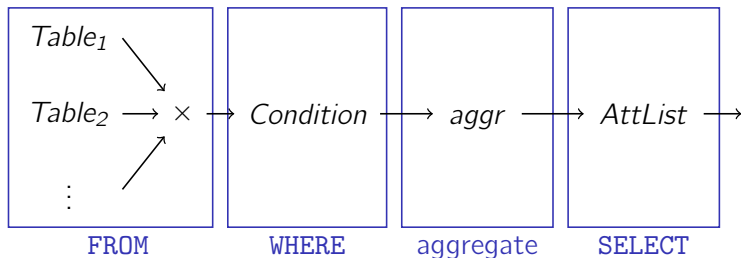
Evaluation of Aggregation Functions

Conceptually, queries with aggregation are evaluated as follows:¹¹

- 1 Evaluate the **FROM** clause
 - Form a **Cartesian product** of all referenced tables/subqueries (see also slide 38).
- 2 Apply **predicates** of the **WHERE** clause.
 - Discard all rows that do not satisfy the **WHERE** predicate.
- 3 Add column values received from 2 to sets/multisets that will be input to the aggregation functions.
 - Remove duplicates if requested by **DISTINCT** keyword within aggregation function(s).
- 4 Compute aggregation result(s) and print a **single row** of aggregated value(s).

¹¹As usual, the system is free to choose a more efficient execution strategy.

Evaluation of Aggregation Functions



Notes:

- Null values are ignored during aggregation. Exception: `COUNT (*)` also counts null values.
- If the aggregation input set is empty, aggregation functions return `NULL`. Exception: `COUNT` returns 0.

Restrictions:

- Aggregations **must not be nested** (makes no sense).
- Aggregations **must not be used in the WHERE clause**.
 - Aggregation is performed only **after** the WHERE clause has been evaluated.
- The result of an aggregation query is a **single output tuple**.
- If aggregation is used, **no attributes** may appear in the SELECT clause.
 - Would make no sense, because aggregation yields a single output row.
 - But see GROUP BY clause below.

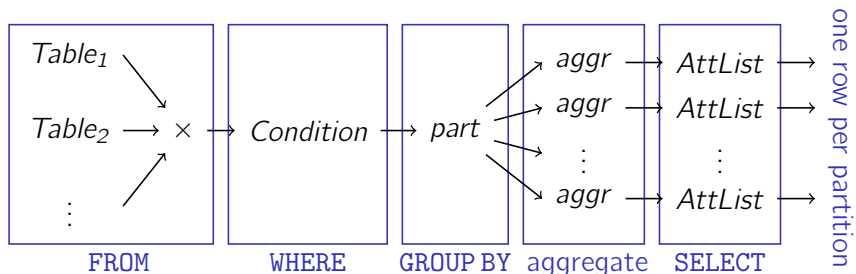
- The GROUP BY clause **partitions** the tuples of a table into **disjoint groups**.
- Aggregation functions are then applied **for each tuple group** separately.

```
SELECT GlassID, COUNT(*) AS cnt
FROM Cocktails
GROUP BY GlassID
```

GlassID	cnt
7	12
3	19
4	8

→ The tuple group with *GlassID* = 7 counts 12 rows, etc.

Evaluation with GROUP BY



- Query returns as many result rows as there are distinct values in the **GROUP BY** attribute(s).
- Any attribute that appears in the **GROUP BY** clause may also be used in the **SELECT** clause.

GROUP BY Examples

The GROUP BY clause may contain more than one column:

```
SELECT Year, Month,  
       SUM(Amount) AS Amt  
FROM Sales  
WHERE Month LIKE 'J%'  
GROUP BY Year, Month
```

Year	Month	Amt
2008	Jan	115154.86
2008	Jul	116348.82
2008	Jun	114418.37
2009	Jan	113908.68
2009	Jul	108407.65
2009	Jun	113489.23

 **What is the result of this query?**

```
SELECT Month  
FROM Sales  
WHERE Month LIKE 'J%'  
GROUP BY Month
```

 **This one?**

```
SELECT Month, SUM(Amount)  
FROM Sales  
WHERE Month LIKE 'J%'  
GROUP BY Year, Month
```

GROUP BY: Columns in SELECT

Only columns (and aggregation functions) listed in the GROUP BY clause may appear in the SELECT part.


```
SELECT c.CocktailID, c.Name, COUNT (*)  
  FROM Cocktails AS c, ConsistsOf co  
 WHERE c.CocktailID = co.CocktailID  
 GROUP BY c.CocktailID
```

wrong!

Solution: Group by CocktailID **and** Name.

→ Since CocktailID is a key, this will not actually affect grouping.

```
SELECT c.CocktailID, c.Name, COUNT (*)  
  FROM Cocktails AS c, ConsistsOf co  
 WHERE c.CocktailID = co.CocktailID  
 GROUP BY c.CocktailID, c.Name
```



Conditions over Aggregates

Remember that aggregations must not be used in the **WHERE** clause.

- With **GROUP BY**, it makes sense to **filter out entire groups**, based on some aggregate group property.
- *E.g.*, Report average sales amount per month only for those months where there were at least 5 transactions.



Can we express that with the SQL constructs we learned so far?

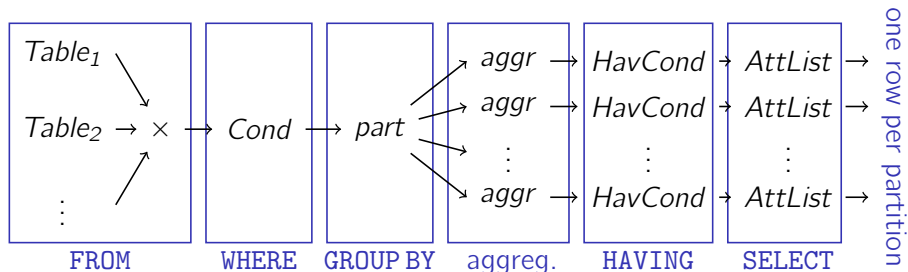
The SQL **HAVING** clause is a convenient means to describe exactly such types of queries.

```
SELECT Year, Month, AVG (Amount) AS Average
FROM Sales
GROUP BY Year, Month
HAVING COUNT (*) >= 5
```

- In the **HAVING** clause, the same types of expressions may be used as in the **SELECT** clause, *i.e.*,
 - aggregation functions,
 - columns listed in the **GROUP BY** clause.

Evaluation with GROUP BY and HAVING

The **HAVING** clause is applied **after** grouping and aggregation (**WHERE** is applied before).



→ Conditions that **only** refer to **GROUP BY** columns may be put into **WHERE** or **HAVING**.

UNION

The SQL keyword `UNION` allows to collect results from multiple queries into a single output relation (\rightsquigarrow algebra operator \cup).

```
SELECT Name, Price
      FROM Ingredients
      WHERE Alcohol > 0
UNION
SELECT Name, Price
      FROM Cocktails
```

`UNION` is **strictly needed** (no other way in SQL to express such queries).

Typical use case:

- Specializations of a general concept are stored in separate tables. They can be re-combined using `UNION`.

- Combined relations must be **schema-compatible**.
 - But SQL is less strict than relational algebra.
 - Both operands must have the same number of columns; columns of compatible types must be listed in same order. Column names, however, do not matter (need not be identical).
- The **other set operators** are available in SQL, too:
 - UNION implements \cup
 - EXCEPT implements $-$ (MINUS is synonym)
 - INTERSECT implements \cap
- All three operators **remove duplicates**.
- To **keep duplicates**: combine with ALL

SELECT ... FROM ... WHERE ...

UNION ALL (or: EXCEPT ALL, INTERSECT ALL)

SELECT ... FROM ... WHERE ...

All SQL queries return result rows in **arbitrary order**.

- You might observe that the system produces the same order when you run the same query multiple times. But there is **no guarantee**: the next run might already lead to a different order.
- This is **intentional**. The system might find a **better execution strategy** if it is allowed to produce results in any order.

Sometimes it is desirable to present the **overall result** of a query **in a particular order** to the user.

→ SQL keyword `ORDER BY`.

```
SELECT LastName, FirstName, Phone
FROM ContactPersons
ORDER BY LastName, FirstName
```

Conceptually, the ORDER BY specification is applied as the **last operation**, only to **present results to the user**.

- May reference columns and aggregation functions just like the SELECT part.
- ORDER BY does **not** make sense in **subqueries** and is thus forbidden there.

The ORDER BY clause is a **list** of ordering criteria.

- **lexicographic ordering** according to this list
- Append DESC to a sort key to sort in descending order.
(↪ ... ORDER BY Year DESC, SUM(Amount) DESC)

SQL Keyword JOIN

Joins can be expressed in SQL by listing the relations in the **FROM** clause and constraining the Cartesian product in the **WHERE** clause.

```
SELECT s.Name, c.Name AS Contact, c.Phone
       FROM Suppliers AS s, ContactPersons AS c
       WHERE s.SuppID = c.SuppID
```

→ Don't be afraid. The system will recognize the pattern and **not** build up the Cartesian product.

Alternatively, joins can be made explicit as follows:

```
SELECT s.Name, c.Name AS Contact, c.Phone
       FROM Suppliers AS s JOIN ContactPersons AS c
       ON s.SuppID = c.SuppID
```

That is, you can write

$$Table_1 \text{ JOIN } Table_2 \text{ ON } JoinCondition$$

in the FROM part of your query.

There are a number of restrictions on what can be used as a *JoinCondition*:

- The condition must **only** refer to columns of the two referenced tables.
- The condition **must not** contain any subqueries.

The JOIN clause can be nested:

$$(Table_1 \text{ JOIN } Table_2 \text{ ON } JoinCond_1) \text{ JOIN } Table_3 \text{ ON } JoinCond_2$$

The JOIN syntax also allows to specify **outer joins**:

```
SELECT s.Name, c.Name AS Contact, c.Phone
FROM Suppliers AS s
      LEFT OUTER JOIN ContactPersons AS c
      ON s.SuppID = c.SuppID
```

- Likewise: RIGHT OUTER JOIN, FULL OUTER JOIN.
- JOIN is synonym for INNER JOIN.

Further syntactic sugar:

- $Table_1$ NATURAL JOIN $Table_2$
- $Table_1$ JOIN $Table_2$ USING (*ColumnList*)

SQL also uses **null values** and **three-valued logic** (↗ slide 80).

- NULL is the literal for the null value.
(INSERT INTO Suppliers VALUES (42, 'Foo Inc.', NULL))
- **Test** for null values with IS NULL (or IS NOT NULL)

```
SELECT Name, www
FROM Suppliers AS s
WHERE www IS NOT NULL
```



Do **not** use = NULL in tests.

Comparisons =, <=, etc. with NULL **always** yield NULL
(i.e., “unknown”; also NULL = NULL → NULL).

So far we only looked at the **data retrieval language part** of SQL.

SQL also offers syntax to

- create or delete tables, to modify their schema, etc.,
→ **data definition language**
- add, delete, or modify rows in the database,
→ **data manipulation language**
- define access rights on data.
→ **data control language**

Systems also implement further commands, not strictly part of SQL:

- physical schema management (index creation), backup, etc.

Table Creation

To create a new table, use the `CREATE TABLE` statement:

```
CREATE TABLE Ingredients ( IngrID    INTEGER NOT NULL,  
                            Name      CHAR(30),  
                            Alcohol   DECIMAL(3,1),  
                            Flavor    CHAR(20) )
```

Data types (somewhat system-dependent):

- `INTEGER`, `SMALLINT`, `BIGINT`
- `DECIMAL (m,n)`: m digits total, n of which are decimals
- `CHAR (n)`: fixed-length strings
- `VARCHAR (n)`: variable-length strings (up to length n)
- `DATE`, `TIME`, `DATETIME`, etc.

Table Creation

- Allow (NULL; default) or disallow (NOT NULL) **null values**.
- Specify **key constraints**:

```
CREATE TABLE Suppliers (SupplID  INTEGER NOT NULL,  
                          Name     CHAR(30) NOT NULL,  
                          www      VARCHAR(200),  
                          PRIMARY KEY (SupplID) )
```

```
CREATE TABLE Contacts (ContactID  INTEGER NOT NULL,  
                        SupplID    INTEGER NOT NULL,  
                        Name        CHAR(40),  
                        Phone       CHAR(20),  
                        PRIMARY KEY (ContactID),  
                        FOREIGN KEY (SupplID)  
                        REFERENCES Suppliers (SupplID) )
```

Dropping or Altering Tables

Deleting an entire table (including its schema definition):

```
DROP TABLE Suppliers
```

- All data in the table is **irrecoverably lost**.
- Many systems implicitly **commit** transactions upon DDL statements (see later).

Change the schema of existing tables using the `ALTER TABLE` statement, *e.g.*,

```
ALTER TABLE Contacts ADD COLUMN Email VARCHAR(30)
```

`CREATE VIEW` is also a data definition statement (since it changes the database schema; ↗ slide 175):

```
CREATE VIEW ConsistsOfIngr AS
  SELECT co.CocktailID, i.Name AS IngrName
     FROM ConsistsOf AS co, Ingredients AS i
     WHERE co.IngrID = i.IngrID
```

To remove a view declaration from the schema, use the `DROP VIEW` statement:

```
DROP VIEW ConsistsOfIngr
```

Insert new rows into a table using the `INSERT` statement:

```
INSERT INTO Suppliers (SupplID, Name, www)
VALUES (42, 'Seven Eleven', NULL)
```

- List tuple values in same order as list of column names.
- The list of column names (`SupplID, ...`) can be omitted (must then give values for all columns).
- You may choose to not specify all columns, but only if the missing columns allow null values or are declared with a default value.

Inserting Rows

The inserted row(s) may also be the result of an SQL query:

```
INSERT INTO SalesStat (Year, Month, Amount)
  SELECT Year, Month, SUM(Amount)
    FROM Sales
   GROUP BY Year, Month
```

DML statements are executed with **snapshot semantics**.

- Conceptually, new values are computed based on a **snapshot** of the database. **Then** the updates are applied.
- The statement does not “see” its own effects.

```
INSERT INTO Budget (Project, Year, Amount)
  SELECT Project, 2012, AVG(Amount) * 1.10
    FROM Budget
   GROUP BY Project
```

Changing Values in Existing Rows

Values in existing rows can be changed with **UPDATE**:

```
UPDATE Employee
  SET Salary = Salary * 1.05,
      Bonus = Bonus + 500
  WHERE EmpType = 'Manager'
```

- In the table listed in the **UPDATE** part, all rows that satisfy the **WHERE** clause are assigned new values as stated by the **SET** clause.
- Without a **WHERE** clause, all rows are updated.
- Again: snapshot semantics

New column values can be computed via `SELECT` statements:

```
UPDATE Sales AS s1
  SET CumulativeAmount = ( SELECT SUM (Amount)
                           FROM Sales s2
                           WHERE s2.Year <= s1.Year )
```



The subquery must return at most one row!

Tuples can be deleted with help of the `DELETE` statement:

```
DELETE FROM Customers
WHERE CustomerID = 42
```

- `DELETE` without a `WHERE` clause deletes **all** rows of the table. But the table itself remains existent.
 - Use `DROP TABLE` to remove the table.

SQL is **not** a complete programming language.

→ It is not even meant to provide such expressiveness (↗ slide 147)

Application programs typically use SQL to interact with the database.

- They generate SQL statements (*e.g.*, based on user input), ship them to the DBMS, and present results to the user (*e.g.*, via a GUI).

Challenge: Impedance mismatch

- Different **type systems**
- SQL ↔ Object-oriented concepts
- declarative, set-oriented ↔ imperative, record-oriented
- concurrency models, exception handling

Various forms of **SQL ↔ programming language integration** exist.

- Embedded SQL (*e.g.*, for C): SQL used in PL with special markup
- SQL as a language subset (*e.g.*, 4GL programming languages)
- PL constructs that are compiled into SQL code (*e.g.*, Linq, ActiveRecord)
- Libraries for SQL interaction (*e.g.*, JDBC, ODBC)

Example: Embedded SQL (for DB2 and C; next slide)

```

EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
short IngrID;
char  Name[31];
EXEC SQL END DECLARE SECTION;

void main (void) {

    EXEC SQL CONNECT TO DEMO;

    EXEC SQL DECLARE IngrCursor CURSOR FOR
        SELECT IngrID, Name
           FROM Ingredients;

    EXEC SQL OPEN IngrCursor;

    while (1) {
        EXEC SQL FETCH IngrCursor into :IngrID, :Name;

        if (sqlca.sqlcode == 100)
            break;

        printf (" %8i    | %30s\n", IngrID, Name);
    }

    EXEC SQL CLOSE IngrCursor;
}

```

- Instructions for **DB2 preprocessor** marked with `EXEC SQL`.
 - Preprocessor turns these into “real” C code, which is then compiled by a regular C compiler.
- **Variable declarations** marked, so preprocessor knows where to convert SQL types ↔ C types.
 - Reference C variables in SQL code using `:varname`.
 - Extended SQL syntax to interact with C variables, e.g.,
`SELECT ... INTO VarList FROM ...`.
- To **iterate** over result sets, use **cursors**.
 - `OPEN, FETCH, ..., FETCH, CLOSE`
 - Make sure you properly close cursors; the database may release **locks** then.