# Database Systems
## *SQL*

**Based on slides by Feifei Li,** University of Utah

# The SQL Query Language

- SQL stands for Structured Query Language

- The most widely used relational query language.  Current standard is SQL:2016
    - (actually there is a new standard with small modifications that has been release in 2019)
  - Many systems like MySQL/PostgreSQL have some “unique” aspects
    - as do most systems.

- Here we concentrate on SQL-92 and SQL:1999

# DDL – Create Table

- CREATE TABLE *table_name* ( { *column_name data_type* [ DEFAULT *default_expr* ] [ *column_constraint* [, ... ] ] | *table_constraint* } [, ... ] )

- Data Types include:

  character(n) – fixed-length character string (CHAR(n))

  character varying(n) – variable-length character string (VARCHAR(n))

  smallint, integer, bigint, numeric, real, double precision

  date, time, timestamp, …

  serial - unique ID for indexing and cross reference

  …

  - you can also define your own type!! (SQL:1999)

# Create Table (w/column constraints)

- CREATE TABLE *table_name* ( { *column_name data_type* [ DEFAULT *default_expr* ] [ *column_constraint* [, ... ] ] | *table_constraint* } [, ... ] )

Column Constraints:

- [ CONSTRAINT *constraint_name* ] { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (*expression*) | REFERENCES *reftable* [ ( *refcolumn* ) ] [ ON DELETE *action* ] [ ON UPDATE *action* ] }

  *action* is one of:

  NO ACTION, CASCADE, SET NULL, SET DEFAULT

  *expression* for column constraint must produce a boolean result and reference the related column's value only.

# Create Table (w/table constraints)

- CREATE TABLE *table_name* ( { *column_name data_type* [ DEFAULT *default_expr* ]  [ *column_constraint* [, ... ] ] | *table_constraint* } [, ... ] )

Table Constraints:

- [ CONSTRAINT *constraint_name* ]
  { UNIQUE ( *column_name* [, ... ] ) |
  PRIMARY KEY ( *column_name* [, ... ] ) |
  CHECK ( *expression* ) |
  FOREIGN KEY ( *column_name* [, ... ] ) REFERENCES *reftable* [ ( *refcolumn* [, ... ] ) ] [ ON DELETE *action* ]  [ ON UPDATE *action* ] }

Here, *expressions, keys, etc can include multiple columns*

# Create Table (Examples)

```
CREATE TABLE films (
   code          CHAR(5) PRIMARY KEY,
   title         VARCHAR(40),
   did           DECIMAL(3),
   date_prod     DATE,
   kind          VARCHAR(10),
   CONSTRAINT production UNIQUE(date_prod)
   FOREIGN KEY did REFERENCES distributors ON DELETE NO ACTION );


CREATE TABLE distributors (
   did    DECIMAL(3) PRIMARY KEY,
   name   VARCHAR(40)
   CONSTRAINT con1 CHECK (did > 100 AND name <> ‘ ’) );
```

# The SQL DML

- Single-table queries are straightforward.

- To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
```

| sid | name | login | age | gpa |
| --- | --- | --- | --- | --- |
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

  - To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```

# Querying Multiple Relations

- Can specify a join over two tables as follows:

  SELECT S.name, E.cid
  FROM Students S, Enrolled E
  WHERE S.sid=E.sid AND E.grade='B'

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

**Note: obviously no referential integrity constraints have been used here.**

**result =**

| S.name | E.cid |
|---|---|
| Jones | History105 |

# Basic SQL Query

| | |
|---|---|
| **SELECT** | **[DISTINCT]** ***target-list*** |
| **FROM** | ***relation-list*** |
| **WHERE** | ***qualification*** |

- *relation-list* : A list of relation names
  - possibly with a *range-variable* after each name
- *target-list* : A list of attributes of tables in *relation-list*
- *qualification* : Comparisons combined using AND, OR and NOT.
  - Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of

    $<, >, =, \leq, \geq, \neq$
- *DISTINCT*: optional keyword indicating that the answer should not contain duplicates.
  - In SQL SELECT, the default is that duplicates are *not* eliminated! (Result is called a "multiset")

# Query Semantics

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do FROM clause: compute *cross-product* of tables (e.g., Students and Enrolled).
  2. do WHERE clause: Check conditions, discard tuples that fail. (called "selection").
  3. do SELECT clause: Delete unwanted fields. (called "projection").
  4. If DISTINCT specified, eliminate duplicate rows.

- Probably the least efficient way to compute a query!
  - An optimizer will find more efficient strategies to get the *same answer*.

# Step 1 – Cross Product

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

X

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'

# Step 2 - Discard tuples that fail predicate

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```

# Step 3 - Discard Unwanted Columns

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```

# Now the Details

We will use these instances of relations in our examples.

Question:

If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

***Reserves***

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

***Sailors***

| sid | sname | rating | age |
|---|---|---|---|
| 22 | Dustin | 7 | 45.0 |
| 31 | Lubber | 8 | 55.5 |
| 95 | Bob | 3 | 63.5 |

***Boats***

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

# Example Schemas

```
CREATE TABLE Sailors (sid INTEGER PRIMARY KEY,
    sname CHAR(20),rating INTEGER,age REAL)

CREATE TABLE Boats (bid INTEGER PRIMARY KEY,
    bname CHAR (20), color CHAR(10))

CREATE TABLE Reserves (
    sid INTEGER REFERENCES Sailors,
    bid INTEGER, day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (bid) REFERENCES Boats)
```

# Another Join Query

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

| (sid) | sname | rating | age | (sid) | bid | day |
|---|---|---|---|---|---|---|
| 22 | dustin | 7 | 45.0 | 22 | 101 | 10/10/96 |
| 22 | dustin | 7 | 45.0 | 58 | 103 | 11/12/96 |
| 31 | lubber | 8 | 55.5 | 22 | 101 | 10/10/96 |
| 31 | lubber | 8 | 55.5 | 58 | 103 | 11/12/96 |
| 95 | Bob | 3 | 63.5 | 22 | 101 | 10/10/96 |
| 95 | Bob | 3 | 63.5 | 95 | 103 | 11/12/96 |

# Some Notes on Range Variables

- Can associate "range variables" with the tables in the FROM clause.
  - saves writing, makes queries easier to understand
- Needed when ambiguity could arise.
  - for example, if same table used multiple times in same FROM (called a "self-join")

```
SELECT sname
FROM Sailors,Reserves
WHERE Sailors.sid=Reserves.sid AND bid=103
```

Can be rewritten using range variables as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

# More Notes

- Here’s an example where range variables are required (self-join example):

```
SELECT  x.sname, x.age, y.sname, y.age
FROM Sailors x, Sailors y
WHERE  x.age > y.age
```

- Note that target list can be replaced by “*” if you don’t want to do a projection:

```
SELECT  *
FROM Sailors x
WHERE  x.age > 20
```

# Find sailors who've reserved at least one boat

```
SELECT  S.sid
  FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference (DISTINCT forces the system to remove duplicates from the output)?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause?
  - Would adding DISTINCT to this variant of the query make a difference?

# Expressions

- Can use arithmetic expressions in SELECT clause (plus other operations we'll discuss later)
- Use AS to provide column names (like a renaming operator)

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
  FROM  Sailors S
WHERE  S.sname = ‘Dustin’
```

- Can also have expressions in WHERE clause:

```
SELECT  S1.sname AS name1, S2.sname AS name2
  FROM  Sailors S1, Sailors S2
WHERE  2*S1.rating = S2.rating - 1
```

# String operations

- SQL supports some basic string operations: “LIKE” is used for string matching

```
SELECT  S.age, S.age-5 AS age1, 2*S.age AS age2
  FROM  Sailors S
WHERE  S.sname LIKE ‘J_%m’
```

`_’ stands for any one character and `%’ stands for 0 or more arbitrary characters.

## Find sid's of sailors who've reserved a red or a green boat

- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).

```
SELECT R.sid
  FROM Boats B,Reserves R
WHERE R.bid=B.bid AND
(B.color='red' OR B.color='green')
```

vs.

```
SELECT  R.sid
  FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
UNION
SELECT R.sid
  FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'
```

# Find sid’s of sailors who’ve reserved a red <u>and</u> a green boat

- If we simply replace OR by AND in the previous query, we get the wrong answer. (Why?)
- Instead, could use a self-join:

```
SELECT R1.sid
  FROM Boats B1, Reserves R1,
            Boats B2, Reserves R2
WHERE R1.sid=R2.sid
     AND R1.bid=B1.bid
     AND R2.bid=B2.bid
     AND (B1.color=‘red’ AND B2.color=‘green’)
```

## Find sid’s of sailors who’ve reserved a red <u>and</u> a green boat

- Or you can use AS to “rename” the output of a SQL block:

```
SELECT R1.sid
  FROM Boats B1, Reserves R1,
         (SELECT R2.sid
          FROM Boats B2, Reserves R2
          WHERE B2.color ='green'
                 AND B2.bid=R2.bid) AS GR
WHERE R1.sid=GR.sid
    AND R1.bid=B1.bid
    AND B1.color='red'
```

```
SELECT RR.sid
  FROM (SELECT R1.sid
         FROM Boats B1, Reserves R1,
         WHERE B1.color='red'
                 AND B1.bid=R1.bid) AS RR,
         (SELECT R2.sid
          FROM Boats B2, Reserves R2
          WHERE B2.color ='green'
                 AND B2.bid=R2.bid) AS GR
WHERE RR.sid=GR.sid
```

# AND Continued...

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- EXCEPT (sometimes called MINUS)
- many systems don't support them.

```
SELECT S.sid        Key field!
FROM Sailors S, Boats B,
         Reserves R
WHERE S.sid=R.sid
         AND R.bid=B.bid
         AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B,
         Reserves R
WHERE S.sid=R.sid
         AND R.bid=B.bid
    AND B.color='green'
```

## Find sid's of sailors who've reserved a red <u>but did not reserve</u> a green boat

```
SELECT S.sid
FROM Sailors S, Boats B,
          Reserves R
WHERE S.sid=R.sid
           AND R.bid=B.bid
           AND B.color='red'
EXCEPT
SELECT S.sid
FROM Sailors S, Boats B,
          Reserves R
WHERE S.sid=R.sid
          AND R.bid=B.bid
    AND B.color='green'
```

# Nested Queries

- Powerful feature of SQL: WHERE clause can itself contain an SQL query!
  - Actually, so can FROM and HAVING clauses.

  ***Names of sailors who've reserved boat #103:***

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN (SELECT R.sid
            FROM  Reserves R WHERE  R.bid=103)
```

- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries:
  - think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

# Nested Queries with Correlation

***Find names of sailors who've reserved boat #103:***

```
SELECT  S.sname
FROM  Sailors S
WHERE EXISTS (SELECT  *
          FROM  Reserves R
          WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- Can also specify NOT EXISTS
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103.
  - UNIQUE checks for duplicate tuples in a subquery;
  - UNIQUE returns true for empty subquery (assumes that two NULL values are different)
- Subquery must be recomputed for each Sailors tuple.
  - Think of subquery as a function call that runs a query!

# More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL
- Find sailors whose rating is greater than that of some sailor called Horatio:

```
SELECT  *
FROM  Sailors S
WHERE  S.rating > ANY (SELECT  S2.rating FROM  Sailors S2
                                    WHERE S2.sname='Horatio')
```

# Semantics of nested operators

- v is a value, A is a *multi-set*
- v IN A evaluates to true iff $v \in A$. v NOT IN A is the opposite.
- EXISTS A evaluates to true iff $A \neq \emptyset$. NOT EXISTS A is the opposite.
- UNIQUE A evaluates to true iff A is a *set*. NOT UNIQUE A is the opposite.
- v OP ANY A evaluates to true iff $\exists x \in A$, such that v OP x evaluates to true.
- v OP ALL A evaluates to true iff $\forall x \in A$, v OP x *always* evaluates to true.

# Rewriting INTERSECT Queries Using IN

*Find sid's of sailors who've reserved both a red and a green boat:*

```
SELECT  R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid
    AND B.color='red'
    AND R.sid IN (SELECT R2.sid
                FROM  Boats B2, Reserves R2
                WHERE  R2.bid=B2.bid
                AND  B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- How would you change this to find *names* (not *sid*'s) of Sailors who've reserved both red and green boats?

# Division in SQL (For All query)

**Find sailors who've reserved all boats.**

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
                                        FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                            AND R.sid=S.sid))
```

*Sailors S such that ...*

*there is no boat B without ...*

*a Reserves tuple showing S reserved B*

# Division in SQL (For All query) Another way..

**Find sailors who've reserved all boats.**

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  ( (SELECT  B.bid
                       FROM  Boats B)
                      EXCEPT
                      (SELECT  R.bid
                       FROM   Reserves R
                       WHERE   R.sid=S.sid))
```

*Sailors S such that ...*

*there is no boat B without ...*

*a Reserves tuple showing S reserved B*

# Basic SQL Queries - Summary

- An advantage of the relational model is its well-defined query semantics.
- SQL provides functionality close to that of the basic relational model.
  - some differences in duplicate handling, null values, set operators, etc.
- Typically, many ways to write a query
  - the system is responsible for figuring a fast way to actually execute a query regardless of how it is written.
- Lots more functionality beyond these basic features

# Aggregate Operators

- Significant extension from set based queries.

COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)

*single column*

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```

```
SELECT AVG ( DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

# Find name and age of the oldest sailor(s)

- The first query is incorrect!
- Third query equivalent to second query.

```
SELECT  S.sname, MAX (S.age)
FROM  Sailors S
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX (S2.age)
           FROM  Sailors S2)
```

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age >= ALL (SELECT  S2.age
                     FROM  Sailors S2)
```

# GROUP BY and HAVING

- So far, we've applied aggregate operators to all (qualifying) tuples.
  - Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

**For *i* = 1, 2, ... , 10:**

```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.rating = i
```

# Queries With GROUP BY

- To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
[WHERE      qualification]
GROUP BY  grouping-list
```

The *target-list* contains

(i) list of column names &

(ii) terms with aggregate operations (e.g., MIN (*S.age*)).

- column name list (i) **can contain only attributes from the *grouping-list***, since the output for each group must represent a consistent value from that group.

# Group By Examples

For each rating, find the average age of the sailors

```
SELECT  S.rating,  AVG (S.age)
FROM  Sailors S
GROUP BY  S.rating
```

For each rating find the age of the youngest sailor with age $\geq$ 18

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
```

# Conceptual Evaluation

- The cross-product of *relation-list* is computed first, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.

- One answer tuple is generated per qualifying group.

# An illustration

SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.0 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

1. Form cross product

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.0 |
| 10 | 35.0 |

2. Delete unneeded rows, columns; form groups

Answer Table

3. Perform Aggregation

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 35.0 |
| 8 | 55.0 |
| 10 | 35.0 |

## Find the number of reservations for each red boat.

```
SELECT B.bid, COUNT(*) AS numres
FROM Boats B, Reserves R
WHERE  R.bid=B.bid
      AND B.color=‘red’
GROUP BY  B.bid
```

- Grouping over a join of two relations.

# An illustration

SELECT B.bid, COUNT (*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid

| b.bid | b.color | r.bid |
|---|---|---|
| 101 | blue | 101 |
| 102 | red | 101 |
| 103 | green | 101 |
| 104 | red | 101 |
| 101 | blue | 102 |
| 102 | red | 102 |
| 103 | green | 102 |
| 104 | red | 102 |

1

| b.bid | b.color | r.bid |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| 102 | red | 102 |
| | | |
| | | |

2

| b.bid | scount |
|---|---|
| 102 | 1 |

answer

# Queries With GROUP BY and HAVING

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set

# Conceptual Evaluation

- Form groups as before.

- The *group-qualification* is then applied to eliminate some groups.
  - Expressions in *group-qualification* must have a *single value per group*!
  - That is, attributes in *group-qualification* must be arguments of an aggregate op or must also appear in the *grouping-list*.

- One answer tuple is generated per qualifying group.

## Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

| sid | sname | rating | age |
|---|---|---|---|
| 22 | Dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 10 | 35.0 |

**2**

| rating | m-age | count |
|---|---|---|
| 1 | 33.0 | 1 |
| 7 | 35.0 | 2 |
| 8 | 55.0 | 1 |
| 10 | 35.0 | 1 |

**3**

| rating | |
|---|---|
| 7 | 35.0 |

*Answer relation*

# Find sailors who've reserved all boats.

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
                                        FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                          AND R.sid=S.sid))
```

*Sailors S such that ...*

*there is no boat B*

*without ...*

*a Reserves tuple showing S reserved B*

## Find sailors who’ve reserved all boats.

- Can you do this using Group By and Having?

```
SELECT  S.sname
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.sname, S.sid
HAVING
        COUNT(DISTINCT R.bid) =
              ( Select COUNT (*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause.  Why?

## An Illustration

```
SELECT  S.name
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
            ( Select COUNT (*) FROM
                              Boats)
```

| sname | sid | bid |
|---|---|---|
| Frodo | 1 | 102 |
| Bilbo | 2 | 101 |
| Bilbo | 2 | 102 |
| Frodo | 1 | 102 |
| Bilbo | 2 | 103 |

| sname | sid | bid |
|---|---|---|
| Frodo | 1 | 102,102 |
| Bilbo | 2 | 101, 102, 103 |

| sname | sid | count |
|---|---|---|
| Frodo | 1 | 1 |
| Bilbo | 2 | 3 |

| count |
|---|
| 3 |

**Sailors**

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Frodo | 7 | 22 |
| 2 | Bilbo | 2 | 39 |
| 3 | Sam | 8 | 27 |

**Boats**

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

**Reserves**

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12 |
| 2 | 102 | 9/12 |
| 2 | 101 | 9/14 |
| 1 | 102 | 9/10 |
| 2 | 103 | 9/13 |

# Find the names of the sailors who've reserved most number of boats for each rating group

```
SELECT  S.sname
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.sname, S.sid
HAVING
        COUNT(R.bid) =
            ( Select MAX(C) FROM
                (SELECT  S1.sid, COUNT(*) AS C FROM
                        Sailors S1, reserves R1
                        WHERE  S1.sid = R1.sid AND S1.rating = S.rating
                        GROUP BY S1.sid) )
```

## Find the names of the sailors who’ve reserved most number of boats for each rating group

```
SELECT  S.sname
FROM  Sailors S, reserves R
WHERE  S.sid = R.sid
GROUP BY S.sname, S.sid
HAVING
        COUNT(R.bid) >= ALL
                (SELECT  COUNT(*) FROM
                        Sailors S1, reserves R1
                        WHERE  S1.sid = R1.sid AND S1.rating = S.rating
                        GROUP BY S1.sid)
```

# INSERT

INSERT [INTO] *table_name* [(*column_list*)]
VALUES ( value_list)

INSERT [INTO] *table_name* [(*column_list*)]
*<select statement>*

INSERT INTO Boats VALUES ( 105, ‘Clipper’, ‘purple’)

INSERT INTO Boats (bid, color) VALUES (99, ‘yellow’)

You can also do a “bulk insert” of values from one table into another:

```
INSERT INTO TEMP(bid)
SELECT r.bid FROM Reserves R WHERE  r.sid = 22;
```

(must be type compatible)

# DELETE & UPDATE

**DELETE [FROM]** ***table_name***
**[WHERE** ***qualification]***

DELETE FROM Boats WHERE color = ‘red’

```
DELETE FROM Boats b
WHERE b. bid =
    (SELECT r.bid FROM Reserves R WHERE  r.sid = 22)
```

Can also modify tuples using UPDATE statement.

```
UPDATE Boats
SET Color = “green”
WHERE bid = 103;
```

# Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null*. IS NULL/IS NOT NULL
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

# NULLs

e.g. :

branch2=

| bname | bcity | assets |
|---|---|---|
| Downtown | Boston | 9M |
| Perry | Horse | 1.7M |
| Mianus | Horse | .4M |
| Kenmore | Boston | NULL |

**What does this mean?**

- **We don't know Kenmore's assets?**
- **Kenmore has no assets?**
- **………………..**

**Effect on Queries:**

**SELECT * FROM branch2
WHERE assets = NULL**

| bname | bcity | assets |
|---|---|---|

**SELECT * FROM branch2
WHERE assets IS NULL**

| bname | bcity | assets |
|---|---|---|
| Kenmore | Boston | NULL |

# NULLs

- Arithmetic with nulls:
  - n op null = null

    op : + , - , *, /, mod, ...

```
SELECT   ...........
FROM     .............
WHERE boolexpr IS UNKNOWN
```

- **Booleans with nulls: One can write:**

**3-valued logic (true, false, unknown)**

**What expressions evaluate to UNKNOWN?**

1. **Comparisons with NULL (e.g. assets = NULL)**
2. **FALSE OR UNKNOWN (but: TRUE OR UNKNOWN = TRUE)**
3. **TRUE AND UNKNOWN**
4. **UNKNOWN AND/OR UNKNOWN**

# NULLs

branch2=

| bname | bcity | assets |
|---|---|---|
| Downtown | Boston | 9M |
| Perry | Horse | 1.7M |
| Mianus | Horse | .4M |
| Kenmore | Boston | NULL |

Aggregate operations:

SELECT SUM(assets)
FROM branch2

returns

SUM
---------
11.1M

NULL is ignored
Same for AVG, MIN, MAX

But.... COUNT(assets) retunrs 4!

Let branch3 an empty relation
Then: SELECT SUM(assets)
FROM branch3 returns NULL

but COUNT(<empty rel>) = 0

# Joins

```
SELECT (column_list)
FROM  table_name
  [INNER | {LEFT |RIGHT | FULL } OUTER] JOIN
table_name
    ON qualification_list
WHERE …
```

Explicit join semantics needed unless it is an INNER join (INNER is default)

# Inner Join

Only the rows that match the search conditions are returned.

```
SELECT s.sid, S.sname, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats

```
SELECT s.sid, S.sname, r.bid
FROM Sailors s NATURAL JOIN Reserves r
```

“NATURAL” means equi-join for each pair of attributes with the same name

# An illustration

**SELECT s.sid, S.sname, r.bid**
**FROM Sailors s INNER JOIN Reserves r**
**ON s.sid = r.sid**

| sid | sname | rating | age |
|---|---|---|---|
| 22 | Dustin | 7 | 45.0 |
| 31 | Lubber | 8 | 55.5 |
| 95 | Bob | 3 | 63.5 |

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

| s.sid | s.name | r.bid |
|---|---|---|
| 22 | Dustin | 101 |
| 95 | Bob | 103 |

# Left Outer Join

Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause (use nulls in fields of non-matching tuples)

```
SELECT s.sid, S.sname, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid
```

Returns all sailors & information on whether they have reserved boats

# An illustration

**SELECT s.sid, S.sname, r.bid**
**FROM Sailors s LEFT OUTER JOIN Reserves r**
**ON s.sid = r.sid**

| sid | sname | rating | age |
|---|---|---|---|
| 22 | Dustin | 7 | 45.0 |
| 31 | Lubber | 8 | 55.5 |
| 95 | Bob | 3 | 63.5 |

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

| s.sid | s.name | r.bid |
|---|---|---|
| 22 | Dustin | 101 |
| 95 | Bob | 103 |
| 31 | Lubber | |

# Right Outer Join

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

```
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & information on which ones are reserved.

# An illustration

**SELECT r.sid, b.bid, b.name**
**FROM Reserves r RIGHT OUTER JOIN Boats b**
**ON r.bid = b.bid**

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

| r.sid | b.bid | b.name |
|---|---|---|
| 22 | 101 | Interlake |
| | 102 | Interlake |
| 95 | 103 | Clipper |
| | 104 | Marine |

# Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & all information on reservations

# An illustration

**SELECT r.sid, b.bid, b.name**
**FROM Reserves r FULL OUTER JOIN Boats b**
**ON r.bid = b.bid**

| sid | bid | day |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 95 | 103 | 11/12/96 |

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

| r.sid | b.bid | b.name |
|---|---|---|
| 22 | 101 | Interlake |
|  | 102 | Interlake |
| 95 | 103 | Clipper |
|  | 104 | Marine |

**Note: in this case it is the same as the ROJ because bid is a foreign key in reserves, so all reservations must have a corresponding tuple in boats.**

# Views

**CREATE VIEW** *view_name*
**AS** *select_statement*

**Makes development simpler**
**Often used for security**
**Not instantiated - makes updates tricky**

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

# An illustration

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

| b.bid | scount |
|---|---|
| 102 | 1 |

**Reds**

# Views Instead of Relations in Queries

```
CREATE VIEW Reds
AS SELECT  B.bid,  COUNT (*) AS scount
    FROM Boats B, Reserves R
    WHERE  R.bid=B.bid AND   B.color='red'
     GROUP BY  B.bid
```

| bid | scount |
|---|---|
| 102 | 1 |

Reds

```
SELECT  bname, scount
    FROM Reds R, Boats B
    WHERE  R.bid=B.bid
         AND scount < 10
```

# Views

## Create View vs INTO

(1)
```
SELECT bname, bcity
FROM   branch
INTO   branch2
```

vs

(2)
```
CREATE VIEW branch2 AS
SELECT bname, bcity
FROM   branch
```

(1) creates new table that gets stored on disk

(2) creates "virtual table" (materialized when needed)

Therefore: changes in branch are seen in the view version of branch2 (2) but not for the (1) case.

# Sorting the Results of a Query

- ORDER BY *column* [ ASC | DESC] [, ...]

```
SELECT  S.rating, S.sname, S.age
    FROM  Sailors S, Boats B, Reserves R
    WHERE  S.sid=R.sid
        AND R.bid=B.bid AND B.color='red'
    ORDER BY  S.rating, S.sname;
```

- Can order by any column in SELECT list, including expressions or aggs, and select top-k:

```
SELECT  S.sid, COUNT (*) AS redrescnt
    FROM  Sailors S, Boats B, Reserves R
    WHERE  S.sid=R.sid
        AND R.bid=B.bid AND B.color='red'
    GROUP BY S.sid
    ORDER BY  redrescnt DESC
    LIMIT 10;
```

# Discretionary Access Control

**GRANT** ***privileges*** **ON** ***object*** **TO** ***users*** **[WITH GRANT OPTION]**

- Object can be a Table or a View
- Privileges can be:
  - Select
  - Insert
  - Delete
  - References (cols) – allow to create a foreign key that references the specified column(s)
  - All
- Can later be REVOKED
- Users can be single users or groups
- See Chapter 17 for more details.

# Two more important topics

- Constraints (such as triggers)
- SQL embedded in other languages (not discussed here)
- We will not review them in further details in this class

# IC's

What are they?

- predicates on the database
- must always be true (checked whenever db gets updated)

There are the following 4 types of IC's:

Key constraints (1 table)

e.g., 2 accts can't share the same acct_no

Attribute constraints (1 table)

e.g., 2 accts must have nonnegative balance

Referential Integrity constraints ( 2 tables)

E.g. bnames associated w/ loans must be names of real branches

Global Constraints (n tables)

E.g., a loan must be carried by at least 1 customer with a svngs acct

# Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)

E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

**SQL examples:**

**1) single relation: All BOSTON branches must have assets > 5M**

```
CREATE TABLE branch (
        ..........
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'BOS') OR assets > 5M))
```

**Affects:**

**insertions into branch**

**updates of bcity or assets in branch**

# Global Constraints

SQL example:
2) Multiple relations: every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
        SELECT   *
        FROM     loan AS L
        WHERE  NOT EXISTS(
                 SELECT   *
                 FROM borrower B, depositor D, account A
                 WHERE B.cname = D.cname  AND
                       D.acct_no = A.acct_no  AND
                       L.lno  = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

```
CREATE ASSERTION loan-constraint
    CHECK(  ..... )
```

Checked with EVERY DB update!
very expensive.....

# Global Constraints

Issues:

1) How does one decide what global constraint to impose?
2) How does one minimize the cost of checking the global constraints?

Ans: Semantics of application and Functional dependencies.

# Summary: Integrity Constraints

| Constraint Type | Where declared | Affects... | Expense |
|---|---|---|---|
| Key Constraints | CREATE TABLE<br>(PRIMARY KEY, UNIQUE) | Insertions, Updates | Moderate |
| Attribute Constraints | CREATE TABLE<br>CREATE DOMAIN<br>(Not NULL, CHECK) | Insertions, Updates | Cheap |
| Referential Integrity | Table Tag<br>(FOREIGN KEY ....<br>REFERENCES ....) | 1.Insertions into referencing rel'n<br>2. Updates of referencing rel'n of relevant attrs<br>3. Deletions from referenced rel'n<br>4. Update of referenced rel'n | 1,2: like key constraints. Another reason to index/sort on the primary keys<br>3,4: depends on<br>a. update/delete policy chosen<br>b. existence of indexes on foreign key |
| Global Constraints | Table Tag (CHECK)<br>or<br>outside table<br>(CREATE ASSERTION) | 1. For single rel'n constraint, with insertion, deletion of relevant attrs<br>2. For assesrtions w/ every db modification | 1. cheap<br>2. very expensive |

# Triggers (Active database)

- Trigger: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)

- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.

# An example of Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 100,000)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, ‘Violation of Minimum Professor Salary’);
END;
```

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.

# Triggers – Event,Condition,Action

- Events could be :

  `BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>`

  e.g.: `BEFORE INSERT ON Professor`

- Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)

- Action can be many different choices :
  - SQL statements , and even DDL and transaction-oriented statements like “commit”.

# Example Trigger

Assume our DB has a relation schema :

Professor (pNum, pName, salary)

We want to write a trigger that :

Ensures that any new professor inserted has salary >= 70000

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

    for what context  ?

BEGIN

    check for violation here ?


END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
     FOR EACH ROW
BEGIN
       Violation of Minimum Professor Salary?
END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     FOR EACH ROW

BEGIN

   IF (:new.salary < 70000)
        THEN RAISE_APPLICATION_ERROR (-20004,
        'Violation of Minimum Professor Salary');
   END IF;

END;
```

# Details of Trigger Example

- BEFORE INSERT ON Professor
  - This trigger is checked before the tuple is inserted
- FOR EACH ROW
  - specifies that trigger is performed for each row inserted
- :new
  - refers to the new tuple inserted
- If (:new.salary < 70000)
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
- Use error code: -20004;
  - this is in the valid range

# Example Trigger Using Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 70000)
BEGIN
   RAISE_APPLICATION_ERROR (-20004,            'Violation of Minimum
   Professor Salary');
END;
```

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.

# Triggers: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 70000)

BEGIN
   RAISE_APPLICATION_ERROR (-20004,
   'Violation of Minimum Professor Salary');
END;
```

# Example Trigger

```
CREATE TRIGGER updSalary
     BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20004, ‘Salary
  Decreasing !!’);
END;
```

- Ensure that salary does not decrease

# Another Trigger Example (SQL:99)

```
CREATE TRIGGER  youngSailorUpdate
    AFTER  INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```

# Row vs Statement Level Trigger

- Row level:  activated once per modified tuple
- Statement level: activate once per SQL statement

- Row level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- Statement level triggers will be more efficient if we do not need to make row-specific decisions

# Row vs Statement Level Trigger

- Example: Consider a relation schema

  Account (num, amount)

  where we will allow creation of new accounts
  only during normal business hours.

# Example: Statement level trigger

```
CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT                    --- is default
BEGIN
   IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun'))
   OR
   (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00' AND '17:00')
    THEN
      RAISE_APPLICATION_ERROR(-20500,'Cannot   create new account
  now !!');
   END IF;
END;
```

# When to use BEFORE/AFTER

- Based on efficiency considerations or semantics.
- Suppose we perform statement-level after insert, then all the rows are inserted first, then if the condition fails, and all the inserted rows must be "rolled back"
- Not very efficient !!

# Combining multiple events into one trigger

```
CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 70000) THEN
  RAISE_APPLICATION_ERROR (-20004, 'below min salary'); END IF;
IF (UPDATING AND :new.salary < :old.salary) THEN
  RAISE_APPLICATION_ERROR (-20004, ‘Salary Decreasing !!'); END
  IF;
END;
```

# Summary : Trigger Syntax

```
CREATE TRIGGER <triggerName>
BEFORE|AFTER   INSERT|DELETE|UPDATE
   [OF <columnList>] ON <tableName>|<viewName>
   [REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
```

# Constraints versus Triggers

- Constraints are useful for database consistency
  - Use IC  when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- Triggers  are flexible and powerful
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence …

- But can be hard to understand ……
  - Several triggers      (Arbitrary order → unpredictable !?)
  - Chain triggers         (When to stop ?)
  - Recursive triggers  (Termination?)