SQL: Queries, Constraints, Triggers

Chapter 5
### Example Instances

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

- We will use these instances of the Sailors and Reserves relations in our examples.

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

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
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 relations in relation-list
- **qualification** Comparisons (Attr op const or Attr1 op Attr2, where op is one of <, >, =, ≤, ≥, ≠) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!
Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list. ($\times$)
  - Discard resulting tuples if they fail qualifications. ($\sigma$)
  - Delete attributes that are not in target-list. ($\pi$)
  - If DISTINCT is specified, eliminate duplicate rows.

- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.
**Example of Conceptual Evaluation**

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

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

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

OR

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

It is good style, however, to use range variables always!
Exercise Session (5 minutes)

- Recall the following Semantics of an SQL query:
  - Compute the cross-product of relation-list. \((\times)\)
  - Discard resulting tuples if they fail qualifications. \((\sigma)\)
  - Delete attributes that are not in target-list. \((\pi)\)
  - If DISTINCT is specified, eliminate duplicate rows.

Write the following query using relational algebra:
SELECT  S.sname FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
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?
- 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 and Strings

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.
- `AS` and `=` are two ways to name fields in result.
- `LIKE` is used for string matching. `_' 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).

- If we replace OR by AND in the first version, what do we get?

- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND (B.color='red' OR B.color='green')
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='green'
```
**Find sid’s of sailors who’ve reserved a red and a green boat**

- **INTERSECT:** Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don’t support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

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

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
   AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
   AND B.color='green'
```

Key field!
Nested Queries

Find 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)

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops 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.
- 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; * denotes all attributes. Why do we have to replace * by R.bid?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.
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: \textit{op} ANY, \textit{op} ALL, \textit{op} IN $>,<,=,\ge,\le,\neq$
- Find sailors whose rating is greater than that of some sailor called Horatio:

\begin{verbatim}
SELECT * 
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname='Horatio')
\end{verbatim}
Rewriting INTERSECT Queries Using IN

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

```sql
SELECT  S.sid
FROM    Sailors S, Boats B, Reserves R
WHERE   S.sid=R.sid AND R.bid=B.bid AND B.color='red'
        AND S.sid IN (SELECT  S2.sid
                        FROM    Sailors S2, Boats B2, Reserves R2
                        WHERE   S2.sid=R2.sid AND R2.bid=B2.bid
                                AND   B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find names (not sid’s) of Sailors who’ve reserved both red and green boats, just replace `S.sid` by `S.sname` in SELECT clause. (What about INTERSECT query?)
Division in SQL

Find sailors who’ve reserved all boats.

- Let’s do it the hard way, without EXCEPT:

(1) \[
\text{SELECT } S\text{.sname} \\
\text{FROM Sailors } S \\
\text{WHERE NOT EXISTS} \\
( ( \text{SELECT } B\text{.bid} \\
\text{FROM Boats } B) \\
\text{EXCEPT} \\
( \text{SELECT } R\text{.bid} \\
\text{FROM Reserves } R \\
\text{WHERE R.sid=S.sid}) )
\]

(2) \[
\text{SELECT } S\text{.sname} \\
\text{FROM Sailors } S \\
\text{WHERE NOT EXISTS} \ ( \text{SELECT } B\text{.bid} \\
\text{FROM Boats } B \\
\text{WHERE NOT EXISTS} \ ( \text{SELECT } R\text{.bid} \\
\text{FROM Reserves } R \\
\text{WHERE R.bid=B.bid} \\
\text{AND R.sid=S.sid}) )
\]

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B
Aggregate Operators

- Significant extension of relational algebra.

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

SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
FROM Sailors S2)

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 illegal! (We’ll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
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 (SELECT MAX(S2.age)
       FROM Sailors S2) = S.age
```
Motivation for Grouping

- 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 $i = 1, 2, \ldots, 10$:

\[
\text{SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i}
\]
Queries With GROUP BY and HAVING

The target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).

- The attribute list (i) must be a subset of grouping-list. Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)
Conceptual Evaluation

- The cross-product of relation-list is computed, 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.
- The group-qualification is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors

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

**Sailors instance:**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors and with every sailor under 60.

HAVING COUNT (*) > 1 AND EVERY (S.age <= 60)

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

What is the result of changing EVERY to ANY?
Find age of the youngest sailor with age ≥18, for each rating with at least 2 sailors between 18 and 60.

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18 AND S.age <= 60
GROUP BY S.rating
HAVING COUNT(*) > 1
```

**Sailors instance:**

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
For each red boat, find the number of reservations for this boat

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

- Grouping over a join of three relations.
- What do we get if we remove `B.color='red'` from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?
Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT(*)
           FROM Sailors S2
           WHERE S.rating = S2.rating)

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) > 1
Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested! **WRONG:**

```
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN (AVG (S2.age)) FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG (S.age) AS avgage
     FROM Sailors S
     GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN (Temp.avgage)
                     FROM Temp)
```
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 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.
Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC’s are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)

  - *Types of IC’s*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
    - *Domain constraints*: Field values must be of right type. Always enforced.
General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE Sailors
    ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK ( rating >= 1 AND rating <= 10 )
)
```

```
CREATE TABLE Reserves
    ( sname CHAR(10),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid,day),
    CONSTRAINT noInterlakeRes
    CHECK (`Interlake` <>
        ( SELECT B.bname
        FROM Boats B
        WHERE B.bid=bid)))
```
Declaring a primary key

Add to the relation schema a line of the form:

```
PRIMARY KEY (<list of attributes>)
```

If the primary key has just one attribute, we may instead write PRIMARY KEY immediately after the definition of the data type of the attribute, e.g.:

```
id INT PRIMARY KEY,
```

NULL values are not allowed in attributes of a primary key.
When a key constraint is violated, an error message is produced.

The state of the database (i.e., the data it contains) is restored to what it was before the action that caused the violation.

Updates in SQL are grouped in units called transactions (more about transactions later in the course).

Constraint-violating transactions are undone (or rolled back).
--- Primary Key Example, slide 1 of 2 ---

```
SQL> CREATE TABLE Students (  
2       cpr VARCHAR(10) PRIMARY KEY,  
3       name VARCHAR(20),  
4       address VARCHAR(20)  
5     );
Table created.
```

```
SQL> INSERT INTO Students VALUES ('0602751127','Ethan Longwinder','My Way 2');
1 row created.
```

```
SQL> INSERT INTO Students VALUES ('0602751127','Ethan Longwinder II','My Way 2');
INSERT INTO Students VALUES ('0602751127','Ethan Longwinder II','My Way 2')
ERROR at line 1:
ORA-00001: unique constraint (ESBEN.SYS_C005079) violated
```
SQL> INSERT INTO Students VALUES (NULL,'Mysterio Student','No Way 8');
INSERT INTO Students VALUES (NULL,'Mysterio Student','No Way 8')
ERROR at line 1:
ORA-01400: cannot insert NULL into ("ESBEN"."STUDENTS"."CPR")

SQL> SELECT * FROM Students;

<table>
<thead>
<tr>
<th>CPR</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0602751127</td>
<td>Ethan Longwinder</td>
<td>My Way 2</td>
</tr>
</tbody>
</table>
--- Declaring other candidate keys ---

If we want the DBMS to check other uniqueness constraints, we may add to the SQL relation schema any number of lines of the form:

```
UNIQUE (<list of attributes in key>)
```

Uniqueness is *not* guaranteed for tuples having NULL values in the key attributes. However, NULL values can be prevented by adding a *NOT NULL* constraint after the declaration of each key attribute.
--- Unique Key Example, slide 1 of 3 ---

SQL> CREATE TABLE Students (  
2    cpr VARCHAR(10) PRIMARY KEY,  
3    name VARCHAR(30) NOT NULL,  
4    address VARCHAR(20)  
5    CONSTRAINT my_constraint UNIQUE (name,address) );

Table created.

SQL> INSERT INTO Students VALUES ('0602751127','Ethan Longwinder','My Way 2');
1 row created.

SQL> INSERT INTO Students VALUES ('0602751129','Ethan Longwinder','My Way 2')
ERROR at line 1:
ORA-00001: unique constraint (ESBEN.MY_CONSTRAINT) violated
SQL> INSERT INTO Students VALUES ('2103780002', 'H. Omeless', NULL);
1 row created.

SQL> INSERT INTO Students VALUES ('2103780004', NULL, NULL);
INSERT INTO Students VALUES ('2103780004', NULL, NULL)
ERROR at line 1:
ORA-01400: cannot insert NULL into ("ESBEN"."STUDENTS"."NAME")

SQL> SELECT * FROM Students;

<table>
<thead>
<tr>
<th>CPR</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0602751127</td>
<td>Ethan Longwinder</td>
<td>My Way 2</td>
</tr>
<tr>
<td>2103780002</td>
<td>H. Omeless</td>
<td></td>
</tr>
</tbody>
</table>
**Unique Key Example, slide 3 of 3**

```
SQL> INSERT INTO Students VALUES ('0602751129', 'Bullie Bank', 'Goa Way 10');
1 row created.

SQL> INSERT INTO Students VALUES ('0602751131', 'Ethan Longwinder', 'My Way 4');
1 row created.

SQL> UPDATE Students SET address = 'Urban Collective';
UPDATE Students SET address = 'Urban Collective'
ERROR at line 1:
ORA-00001: unique constraint (ESBEN.MY_CONSTRAINT) violated
```

```
SQL> SELECT * FROM Students;

<table>
<thead>
<tr>
<th>CPR</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0602751127</td>
<td>Ethan Longwinder</td>
<td>My Way 2</td>
</tr>
<tr>
<td>2103780002</td>
<td>H. Omeless</td>
<td></td>
</tr>
<tr>
<td>0602751129</td>
<td>Bullie Bank</td>
<td>Goa Way 10</td>
</tr>
<tr>
<td>0602751131</td>
<td>Ethan Longwinder</td>
<td>My Way 4</td>
</tr>
</tbody>
</table>
```
Should every relation have a primary key?

Short answer: Not necessarily.

Consider for example the relation corresponding to a simple multivalued attribute in an E-R diagram:

- Typically, the only candidate key would consist of both attributes.
- Thus, a primary key constraint would only serve the purpose of eliminating duplicate tuples.
Next: Assertion-based constraints.
The constraint NOT NULL may be specified for any attribute in a relation schema, indicating that NULL is not a legal value.

In general, any attribute that does not correspond to an optional attribute in the E-R diagram should be declared NOT NULL.
CHECK constraints

Many business rules can be expressed as so-called CHECK constraints, which are assertions (i.e., conditions that must be true) about attributes or tuples of a relation.

- A CHECK constraint on an attribute is checked every time
  - a value of this attribute is modified.
  - a new tuple is inserted.

- A CHECK constraint on tuples is checked every time
  - an attribute value changes.
  - a new tuple is inserted.

- If a constraint is violated, the current transaction is rolled back, and an error message is produced.
Writing attribute-based CHECK constraints

A constraint $C$ on an attribute is declared by writing

```
CHECK $C$
```

immediately after the datatype definition.

The condition $C$ may refer to other attributes of the relation, and even to other relations, using a subquery.

(However, Oracle does not allow SQL queries in $C$.)

Examples:

- percentage INT CHECK (percentage>=0 AND percentage<=100)
- cpr CHAR(10) CHECK (cpr IN (SELECT cpr FROM students))
Writing tuple-based CHECK constraints

A constraint $C$ on tuples is declared by adding the line

```
CHECK $C$
```

to the relation schema definition.

The only difference to attribute-based CHECK constraints is *when* the constraint is checked.

Examples:

- CHECK (upper-bound => lower-bound)
- CHECK (cpr IN (SELECT cpr FROM students))
Next: Foreign keys.
Foreign key constraints

A foreign key constraint on an attribute is a constraint saying that its attribute values can always be found in exactly one place in another relation.

Foreign key constraints are typically used to express referential integrity, i.e., that values supposed to refer to tuples in other tables indeed do so.

If we want the DBMS to check foreign key constraints, we may add to the SQL relation schema any number of declarations of the form:

```
FOREIGN KEY (<attribute name>)
    REFERENCES <table name>(<attribute name>)
```
Composite foreign keys

Foreign keys may be **composite**, i.e., consist of several attributes.

The syntax for declaring composite foreign keys is the obvious extension of what we saw before:

```
FOREIGN KEY (<list of attribute names>)
    REFERENCES <table name>(<list of attribute names>)
```
— Semantics of a foreign key constraint —

Suppose the schema for relation $R$ contains the declaration

\[
\text{FOREIGN KEY } (A_1, \ldots, A_n) \text{ REFERENCES } S(B_1, \ldots, B_n).
\]

Then the relation $S$ must have $B_1, \ldots, B_n$ as primary keys or contain a declaration like

\[
\text{UNIQUE } (B_1, \ldots, B_n).
\]

This means that the DBMS checks that any values of $A_1, \ldots, A_n$ in a tuple of $R$ can also be found as values of $B_1, \ldots, B_n$ in a tuple of $S$. 
SQL> CREATE TABLE ITUpeople (  
2      cpr VARCHAR(10) PRIMARY KEY,  
3      name VARCHAR(30) NOT NULL,  
4      address VARCHAR(20)  
5  );
Table created.

SQL> CREATE TABLE Students (  
2      cpr VARCHAR(10) PRIMARY KEY  
3          CONSTRAINT ValidCPR REFERENCES ITUpeople(cpr),  
4      enrolled VARCHAR(10),  
5      graduated VARCHAR(10),  
6      gpa REAL CHECK (gpa>=6 AND gpa<=13),  
7          CONSTRAINT PositiveStudyTime CHECK (enrolled < graduated)  
8  );
Table created.

SQL> INSERT INTO ITUpeople VALUES ('0602751129','Bullie Bank','Goa Way 10');
1 row created.
SQL> INSERT INTO Students VALUES ('0602751129', '2003-08-01', NULL, NULL);

1 row created.

SQL> UPDATE Students SET graduated = '2001-02-28' WHERE cpr='0602751129';

UPDATE Students SET graduated = '2001-02-28' WHERE cpr='0602751129'

ERROR at line 1:
ORA-02290: check constraint (ESBEN.POSITIVESTUDYTIME) violated

SQL> DELETE FROM ITUpeople WHERE cpr='0602751129';

DELETE FROM ITUpeople WHERE cpr='0602751129'

ERROR at line 1:
ORA-02292: integrity constraint (ESBEN.VALIDCPR) violated - child record found

SQL> SELECT * FROM ITUpeople;

<table>
<thead>
<tr>
<th>CPR</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0602751129</td>
<td>Bullie Bank</td>
<td>Goa Way 10</td>
</tr>
</tbody>
</table>
SQL> ALTER TABLE Students DROP CONSTRAINT ValidCPR;
Table altered.

SQL> ALTER TABLE Students ADD CONSTRAINT ValidCPR 2 FOREIGN KEY (cpr) REFERENCES ITUpeople(cpr) ON DELETE CASCADE;
Table altered.

SQL> DELETE FROM ITUpeople WHERE cpr='0602751129';
1 row deleted.

SQL> SELECT * FROM ITUpeople;
no rows selected

SQL> SELECT * FROM Students;
no rows selected
— Problem session (5 minutes) —

What is the difference (if any) between the CHECK constraint

```sql
    cpr CHAR(10) CHECK (cpr IN (SELECT cpr FROM students))
```

and the referential integrity constraint

```sql
    cpr CHAR(10) REFERENCES students(cpr)
```
Referential integrity from E-R diagrams

If a relationship in our E-R diagram has an “exactly one” cardinality constraint, it can be expressed as a foreign key constraint.

This means that the DBMS maintains the referential integrity of the relationship.

There seems to be no general way to express an “at least one” cardinality constraint.

Note that in supertype-subtype relationships there is an implicit “exactly one” cardinality constraint.
Maintaining referential integrity

The default (i.e., standard) policy when a transaction violates a foreign key constraint is to roll the transaction back.

However, for each referential constraint we may choose from two other policies for handling changes to the referenced relation:

- **The cascade policy**:
  - If the foreign key attribute values of a tuple were changed, change all references to this tuple to the new value.
  - If a tuple is deleted, delete all tuples referencing it.

- **The set-null policy**:
  - If some reference became invalid, set all its attribute values to NULL.
Next: Triggers.
**Triggers** is a general mechanism for:

- Enforcing constraints/business rules, and more generally
- Making the DBMS perform actions on certain events.

The definition of a trigger consist of an event, a condition, and an action.

- Triggers are awakened (or triggered) when the **event**, a certain change to the database, occurs.
- If the **condition** associated with the trigger is true, then the **action** is performed.
Triggers in SQL

Key features of triggers in SQL:

- Triggering events are insertions, deletions, and updates of tuples.
- The action can be any SQL statement.  
  (But most RDBMSs have restrictions on the SQL allowed in the action.)
- The action can refer to values from both before and after the event.
- The action can be performed either
  - After each event that activates the trigger, or
  - At the end of each transaction where one or more events activated the trigger.
### Trigger Example, slide 1 of 4

```sql
SQL> select * from MovieExec;

<table>
<thead>
<tr>
<th>NAME</th>
<th>ADDRESS</th>
<th>CERT</th>
<th>NETWORTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>George Lucas</td>
<td>Oak Rd.</td>
<td>555</td>
<td>2000000000</td>
</tr>
<tr>
<td>Ted Turner</td>
<td>Turner Av.</td>
<td>333</td>
<td>1250000000</td>
</tr>
<tr>
<td>Stephen Spielberg</td>
<td>123 ET road</td>
<td>222</td>
<td>1000000000</td>
</tr>
<tr>
<td>Merv Griffin</td>
<td>Riot Rd.</td>
<td>199</td>
<td>1120000000</td>
</tr>
<tr>
<td>Calvin Coolidge</td>
<td>Fast Lane</td>
<td>123</td>
<td>200000000</td>
</tr>
</tbody>
</table>
```

```sql
SQL> CREATE TABLE NetworthHistory (  
    2 name VARCHAR(25),
    3 oldnetworth INT,
    4 newnetworth INT
    5 );

Table created.
```
SQL> CREATE TRIGGER NetWorthTrigger
  2     AFTER UPDATE OF netWorth ON MovieExec
  3     REFERENCING
  4     OLD AS Oldtuple
  5     NEW AS Newtuple
  6     FOR EACH ROW
  7     WHEN (Oldtuple.networth <> NewTuple.networth)
  8     BEGIN
  9     INSERT INTO NetworthHistory
 10     VALUES (:Oldtuple.name,:Oldtuple.networth,:Newtuple.networth);
 11     END;
 12     /

Trigger created.
SQL> UPDATE MovieExec
2       SET netWorth = 29000000
3       WHERE name='George Lucas';

1 row updated.

SQL> SELECT * FROM NetworthHistory;

<table>
<thead>
<tr>
<th>NAME</th>
<th>OLDNETWORTH</th>
<th>NEWNETWORTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>George Lucas</td>
<td>2000000000</td>
<td>290000000</td>
</tr>
</tbody>
</table>
SQL> UPDATE MovieExec
    2     SET netWorth = 25000000
    3     WHERE name='George Lucas';

1 row updated.

SQL> SELECT * FROM NetworthHistory;

<table>
<thead>
<tr>
<th>NAME</th>
<th>OLDMETWORTH</th>
<th>NEWMETWORTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>George Lucas</td>
<td>200000000</td>
<td>290000000</td>
</tr>
<tr>
<td>George Lucas</td>
<td>290000000</td>
<td>250000000</td>
</tr>
</tbody>
</table>
Trigger definition syntax, simplified

Syntax in Oracle (differs slightly from SQL definition):

```sql
CREATE TRIGGER <name of trigger> AFTER
  INSERT | DELETE | UPDATE
    [OF <attribute name>] ON <name of relation or view>
    [REFERENCING OLD AS <name>, NEW AS <name>]
    [FOR EACH ROW]
    [WHEN <condition>]]
BEGIN
  <PL/SQL commands>
END;
```

Vertical lines | between alternatives. Brackets [] around optional parts.
Variables in <PL/SQL commands> must be prefixed by semicolon (:old.a).
Most important points in this lecture

As a minimum, you should after this week:

- Know how to declare key constraints and referential integrity (i.e., foreign key) constraints in SQL.
- Understand the basic mechanisms for maintaining referential integrity.
- Know how to declare tuple-based CHECK constraints, and know how these are checked.
- Understand how to define triggers, and the mechanism for executing triggers in SQL.