Databases

Basic SQL

Danilo Montesi

danilo.montesi@unibo.it

SQL

- At first, an acronym for “Structured Query Language”, now a “proper noun”
- Language with many features:
  - Implements both DDL and DML
- There is an standard ISO language, but different DBMSs have their own language grammar
- For the moment we’re going to see the basics of this language

SQL: History

- Its predecessor was **SEQUEL** (1974)
- First implementations were SQL/DS and Oracle (1981)
- SQL has a “de facto” standard since 1983
- Many proposed updates (1986, then 1989, 1992, 1999, 2003, 2006, 2008, …) but still, DBMSs have their own grammar (see some comparisons on http://troels.arvin.dk/db/rdbms/)

SQL Improvements: SQL-base

- **SQL-86:** first proposed standard. It had most of the clauses for expressing queries, but offered a limited support for creating and updating both schemas and data
- **SQL-89:** Referential Integrity is added

SQL Improvements: SQL-2

SQL-92: mostly backward compatible, has new features:

- New functions (e.g., COALESCE, NULLIF, CASE)
- 3 usage levels: entry, intermediate, full

SQL Improvements: SQL-3 (1)

Different subversions:

- **SQL:1999:** proposes the object-relational, triggers and external functions
- **SQL:2003:** extends the object oriented model and allows to perform queries in Java and over semistructured data (XML)

SQL Improvements: SQL-3 (2)

- **SQL:2006:** SQL is extended with other languages (e.g., XQuery) for querying XML data
- **SQL:2008:** some slight edits to the syntax (e.g., trigger with `instead of`)

SQL Improvements

Unofficial Name	Official Name	Features
SQL-Base	**SQL-86**	Basic keywords
	SQL-89	Referential Integrity
SQL-2	**SQL-92**	Modello relazionale New keywords 3 levels: entry, intermediate, full
SQL-3	**SQL:1999**	Relational model with object-oriented Structured in different parts Trigger, external functions, …
	SQL:2003	The support of the Object-Oriented model is extended The no-longer used keywords were removed Extensions: SQL/JRT, SQL/XML, …
	SQL:2006	Extended support for XML data
	SQL:2008	Slight edits (e.g., trigger `instead of`)

Data Definition (1)

CREATE DATABASE:

- Each newly created database contains tables, views, triggers and other things
- For example:

```
CREATE DATABASE db_name
```

Please Note:

- In SQLite **sqlite3 db_name.db sqlite3_open_v2(db_name)**
- In Mimer **CREATE DATABANK db_name**

Data Definition (2)

CREATE SCHEMA:

- A SQL Schema is identified by a name and describes the elements belonging to it (tables, types, constraints, views, domains, …). The schema will belong to the user which has typed the statement
- For example:

```
CREATE SCHEMA schema_name
```

Data Definition (3)

CREATE SCHEMA:

- Such statement could be even followed by the `AUTHORIZATION` keyword, to indicate a specific user owning the schema
- For example:

```
CREATE SCHEMA schema_name
AUTHORIZATION 'user_name'
```

From **MySQL 8.0 Reference Manual … "***CREATE SCHEMA is a synonym for CREATE DATABASE*"

Data Definition (4)

CREATE TABLE:

- Specifies a new relation and creates its empty instance
- It specifies its attributes (with their types) and initial constraints

CREATE TABLE: an Example

```
CREATE TABLE EMPLOYEE (
  Number          CHARACTER(6)
  PRIMARY KEY,
  Name            CHARACTER(20) NOT
  NULL,
  Surname         CHARACTER(20) NOT
  NULL,
  Dept            CHARACTER(15),
  Wage            NUMERIC(9) DEFAULT
  0,
  FOREIGN KEY(Dept) REFERENCES
      DEPARTMENT(Dept),
```

(Attribute) Data Types

- (Attribute) Data Types in SQL correspond to the domains in the relational calculus
  - *Basic* data types (already available)
  - *Custom* data types (called “domains” simple and reusable)

Basic Data Types

- Character-string: data types are either fixed length or varying length
- Numeric, including integer numbers and different floating points
- **DATE**, **TIME**, **INTERVAL**
- Introduced with SQL-3 (SQL:1999):
  - Boolean
  - **BLOB**, **CLOB** (binary/character large object): representing huge data collections (either textual or not)

Custom Data Types

`CREATE DOMAIN`:

- Each custom data type could be used when defining new relations, stating constraints and default values

CREATE DOMAIN: an Example

```
CREATE DOMAIN Grade
  AS SMALLINT DEFAULT NULL
  CHECK ( value >=18 AND value <=
  30 )
```

Table Constraints

- **NOT NULL**
- **UNIQUE** defining keys
- **PRIMARY KEY**: (just one, implies **NOT NULL**; DB2 has a non standard behaviour)
- **CHECK**, let's see it later

UNIQUE and PRIMARY KEY

It could be used when:

- when we define an attribute that defines the key
- as a stand-alone element

CREATE TABLE: an Example

```
CREATE TABLE EMPLOYEE (
  Number        CHARACTER(6)
  PRIMARY KEY,
  Name          CHARACTER(20) NOT
  NULL,
  Surname       CHARACTER(20) NOT
  NULL,
  Dept          CHARACTER(15),
  Wage          NUMERIC(9) DEFAULT
  0,
  FOREIGN KEY(Dept) REFERENCES
      DEPARTMENT(Dept),
```

PRIMARY KEY: Other Options

```
Number CHARACTER(6) PRIMARY KEY
```

```
Number CHARACTER(6),
…
PRIMARY KEY (Number)
```

CREATE TABLE: an Example

```
CREATE TABLE EMPLOYEE (
 Number          CHARACTER(6)
 PRIMARY KEY,
 Name            CHARACTER(20) NOT
 NULL,
 Surname         CHARACTER(20) NOT
 NULL,
 Dept            CHARACTER(15),
 Wage            NUMERIC(9) DEFAULT
 0,
 FOREIGN KEY(Dept) REFERENCES
     DEPARTMENT(Dept),
```

Warning!

```
   Name                    CHARACTER(20) NOT
NULL,
   Surname                 CHARACTER(20) NOT
NULL,
   UNIQUE (Surname,Name)

   Name                   CHARACTER(20) NOT
     NULL UNIQUE,
   Surname                 CHARACTER(20) NOT
NULL UNIQUE,
```

- Are not the same!

Key and Referential Integrity Constraints

- **CHECK**, let’s see it later
- **REFERENCES** and **FOREIGN KEY** define Referential Integrity Constraints
- They can be defined
  - Over a single attribute
  - Over multiple attributes
- We can define *referential triggered actions* when such constraints are violated

Referential Integrity Constraints (1)

OFFENSES	Code	Date	Officer	State	Number
	34321	95/02/01	3987	IT	AG548UK
	53524	95/03/04	3295	IT	TE395AB
	64521	96/04/05	3295	FR	ZT395AB
	73321	98/02/05	9345	FR	ZT395AB

OFFICER	Id	Surname	Name
	3987	Rossi	Luca
	3295	Neri	Piero
	9345	Neri	Mario
	7543	Mori	Gino

Referential Integrity Constraints (2)

OFFENSES	Code	Date	Officer	State	Number
	34321	95/02/01	3987	IT	AG548UK
	53524	95/03/04	3295	IT	TE395AB
	64521	96/04/05	3295	FR	ZT395AB
	73321	98/02/05	9345	FR	ZT395AB

CAR	State	Number	Surname	Name
	IT	AG548UK	Verdi	Giuseppe
	IT	TE395AB	Verdi	Giuseppe
	FR	ZT395AB	Quinault	Philippe

CREATE TABLE: an Example

```
CREATE TABLE OFFENCES (
  Code CHARACTER(6) PRIMARY KEY,
  Day DATE NOT NULL,
  Officer INTEGER NOT NULL
        REFERENCES OFFICER(Id),
  State CHARACTER(2),
  Number CHARACTER(6),
  FOREIGN KEY(State, Number)
        REFERENCES CAR(State,
      Number)
)
```

Referential Triggered Action

- After each referential constraint, we can specify a triggered action (delete,update) to be invoked if the operation is rejected:

```
ON < DELETE | UPDATE >
   < CASCADE | SET NULL |
     SET DEFAULT | NO ACTION >
```

Referential Triggered Action: Delete

- **CASCADE**: deletes the referencing tuples
- **SET NULL**: the value of the deleted referencing attribute is replaced with NULL
- **SET DEFAULT**: the value of the deleted referencing attributes is replaced with the specified default value
- **NO ACTION**: no removal is allowed

Referential Triggered Action: Update

- **CASCADE**: the value of the referencing foreign key attributes(s) is updated with the new value
- **SET NULL**: the value of the affected referencing attribute is replaced with NULL
- **SET DEFAULT**: the value of the affected referencing attributes is replaced with the specified default value
- **NO ACTION**: no update is allowed

Schema Change Statements

- **ALTER DOMAIN**
- **ALTER TABLE**
- **DROP DOMAIN**
- **DROP TABLE**

ALTER DOMAIN

ALTER DOMAIN:

- Allows to alter previously-defined domains
- Such statement has to be used alongside with those other ones: **SET DEFAULT**, **DROP DEFAULT**, **ADD CONSTRAINT** or **DROP CONSTRAINT**

ALTER DOMAIN: an Example (1)

```
ALTER DOMAIN Grade SET DEFAULT 30
```

- Sets the default Grade to 30
- Such command is applied only when the command is invoked and missing grade value are found

```
ALTER DOMAIN Grade DROP DEFAULT
```

- Removes the default Grade value

ALTER DOMAIN: an Example (2)

```
ALTER DOMAIN Grade
  ADD CONSTRAINT isValid
  CHECK (value >=18 AND value <=30)
```

- Adds the `isValid` constraint to the data type `Grade`

```
ALTER DOMAIN Grade DROP CONSTRAINT isValid
```

- Removes constraint associated to the data type

ALTER TABLE

ALTER TABLE:

- Performs changes to previously defined tables
- Such statement has to be used alongside with these parameters: **ALTER COLUMN**, **ADD COLUMN**, **DROP COLUMN**, **DROP CONSTRAINT** or **ADD CONSTRAINT**

ALTER TABLE: an Example (1)

```
ALTER TABLE EMPLOYEE
  ALTER COLUMN Number SET NOT NULL
```

- `Number` from table `EMPLOYEE` cannot have null values

```
ALTER TABLE EMPLOYEE
  ADD COLUMN Level CHARACTER(10)
```

- An attribute `Level` is added to the table `EMPLOYEE`

ALTER TABLE: an Example (2)

```
ALTER TABLE EMPLOYEE
   DROP COLUMN Level RESTRICT
```

- Removes the attribute `Level` from EMPLOYEE only if it doesn't contain values

```
ALTER TABLE EMPLOYEE
   DROP COLUMN Level CASCADE
```

- Removes the attribute `Level` from `EMPLOYEE` alongside with its values

ALTER TABLE: an Example (3)

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT validNum
  CHECK (char_length(Number) = 10)
```

- Adds the `validNum` constraint to the `Number` attribute from `EMPLOYEE`

```
ALTER TABLE EMPLOYEE
  DROP CONSTRAINT validNum
```

- Removes the previously defined constraint

DROP DOMAIN

`DROP DOMAIN`:

- Removes a user-defined data type

Example:

```
DROP DOMAIN Grade
```

DROP TABLE

DROP TABLE:

- Removes a whole table instance with its schema and its data

Example:

```
DROP TABLE OFFENCES
```

Defining Indices

- They usually enhance the query time, relevant for computation efficiency
- They are defined at the physical level, not logical
- In the old days this was also the only way to define keys
- **CREATE INDEX**

CREATE INDEX: an Example

```
CREATE INDEX idx_Surname
   ON OFFICER (Surname)
```

- Creates the index `idx_Surname` on the attribute `Surname` from the table `OFFICER`

DDL in Practice

- In many systems and projects, different tools are used, instead of SQL statements, in order to define a database schema (e.g., tools with a graphical user interface)

SQL: Data Operations

- Query:
  - **`SELECT`**
- Edit:
  - **`INSERT`**, **`DELETE`**, **`UPDATE`**

How to Interpret the SELECT Clause

3 **SELECT** Number,
1 **FROM** OFFICER
2 **WHERE** Surname = ‘Jones’

1 From the table `OFFICER`

2 Retrieve all the officers having `‘Jones’` as `Surname` attribute

3 Showing for each tuple both `Number` and `Name`

SELECT: Shortcuts (1)

```
SELECT *
FROM PEOPLE
WHERE Age < 30

SELECT Name, Age, Income
FROM PEOPLE
WHERE Age < 30
```

Basic SELECT Statement

```
SELECT <AttributeList>
FROM <TableList>
[ WHERE <Condition> ]
```

- Target list
- **FROM** statement
- **WHERE** statement

Database Example

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

MOTHERHOOD	
Mother	Child
Abby	Alice
Abby	Louis
Jesse	Olga
Jesse	Phil
Alice	Jim
Alice	James

FATHERHOOD	
Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

Selection and Projection

Return name and income of people under 30 yo

$$\pi_{\text{Name, Income}}\ (\sigma_{\text{Age<30}}\ (\text{PEOPLE}))$$

```sql
SELECT Name,
 Income
FROM PEOPLE
WHERE Age < 30
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
Phil	26	30
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]

Selection and Projection

Return name and income of people under 30 yo

$$\pi_{\text{Name, Income}} (\sigma_{\text{Age<30}} (\text{PEOPLE}))$$

```
SELECT Name,
 Income
FROM PEOPLE
WHERE Age < 30
```

Name	Income
Jim	21
James	15
Phil	30

SELECT: (Attribute) Renaming

```
SELECT P.Name AS GivenName,
       P.Income AS Revenue
FROM PEOPLE AS P
WHERE P.Age < 30
```

Name	Income
Jim	21
James	15
Phil	30

GivenName	Revenue
Jim	21
James	15
Phil	30

Pure Selection

Provide the Name, Age and Income of people under 30 yo

$$\sigma_{Age<30}\ (PEOPLE)$$

```sql
SELECT *
FROM PEOPLE
WHERE Age < 30
```

PEOPLE		
Name	**Age**	**Income**
Jim	27	21
James	25	15
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
Phil	26	30
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]

Selection without Projection

Provide the Name, Age and Income of people under 30 yo

Name	Age	Income
Jim	27	21
James	25	15
Phil	26	30

$\sigma_{Age<30}$ (PEOPLE)

```
SELECT *
FROM PEOPLE
WHERE Age < 30
```

Projection without Selection

Return the peoples' name and income

$$\pi_{\text{Name,Income}}\ (\text{PEOPLE})$$

```
SELECT Name,
 Income
FROM PEOPLE
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

Projection without Selection

Return the peoples' name and income

$\pi_{\text{Name,Income}}$ (PEOPLE)

```
SELECT Name,
 Income
FROM PEOPLE
```

Name	**Income**
Jim	21
James	15
Alice	42
Jesse	35
Phil	30
Louis	40
Frank	20
Olga	41
Steve	35
Abby	87

SELECT: Shortcuts (2)

Given a relation R(A,B)

```
SELECT *
FROM R
```

It corresponds to:

```
SELECT X.A AS A, X.B AS B
FROM R AS X
WHERE true
```

Composed Conditions

```
SELECT *
FROM PEOPLE
WHERE Income>25 AND (Age<30 OR
         Age>60)
```

PEOPLE		
Name	Age	Income
Phil	26	30
Frank	60	20
Olga	30	41
Steve	85	35

Composed Conditions

```
SELECT *
FROM PEOPLE
WHERE Income>25 AND (Age<30 OR
         Age>60)
```

Name	Age	Income
Phil	26	30
Steve	85	35

LIKE Predicate (1)

- It returns the people having a name starting with 'J' and have a 'm' as a third letter:

```
SELECT *
FROM PEOPLE
WHERE Name LIKE 'J_m%'
```

LIKE Predicate (2)

```
SELECT *
FROM PEOPLE
WHERE Name LIKE 'J_m%'
```

PEOPLE

Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20

Handling NULL Values

EMPLOYEE			
Number	Surname	Agency	Age
5998	Neri	Milan	45
9553	Bruni	Milan	NULL

- Return the employees being either more than 40 yo or NULL value

$$\sigma_{(Age > 40)\ OR\ (Age\ IS\ NULL)}\ (EMPLOYEE)$$

Example

- Return the employees being either more than 40 yo or NULL value

$\sigma_{\text{(Age > 40) OR (Age IS NULL)}}$ (EMPLOYEE)

```
SELECT *
FROM EMPLOYEE
WHERE Age>40 OR Age IS NULL
```

Projection (Relational Algebra)

EMPLOYEE			
Number	Surname	Agency	Age
7309	Neri	Naples	55
5998	Neri	Milan	64
9553	Rossi	Rome	44
5698	Rossi	Rome	64

- Return the surname and the agency for all the employees

$$\pi_{\text{Surname,Agency}} (\text{EMPLOYEE})$$

Projection (SQL and DISTINCT)

```
SELECT
   Surname, Agency
FROM EMPLOYEE
```

Surname	**Agency**
Neri	Naples
Neri	Milan
Rossi	Rome
Rossi	Rome

```
SELECT DISTINCT
   Surname, Agency
FROM EMPLOYEE
```

Surname	**Agency**
Neri	Naples
Neri	Milan
Rossi	Rome

Select, Project, Join

- By using only one relation in the `FROM` clause, one single SQL query can express: *select*, *project* and *rename*
- Using more relations in the `FROM` clause we have *joins* (and cartesian products)

SQL vs Relational Algebra (1)

- R1(A1,A2) R2(A3,A4)

```
SELECT          DISTINCT R1.A1, R2.A4
FROM            R1, R2
WHERE           R1.A2 = R2.A3
```

- Cartesian products (**FROM**)
- Selection (**WHERE**)
- Projection (**SELECT**)

SQL vs Relational Algebra (2)

- R1(A1,A2) R2(A3,A4)

```
SELECT          DISTINCT R1.A1,
  R2.A4
FROM            R1, R2
WHERE           R1.A2 = R2.A3
```

$$\pi_{A1,A4}\,(\sigma_{A2=A3}\,(R1 \bowtie R2))$$

SQL: Alias and Renaming

- Renaming could be required
  - in the cartesian product
  - in the target list

```
SELECT      X.A1 AS B1, ...
FROM        R1 AS X, R2 AS Y,
 R1 AS Z
WHERE       X.A2 = Y.A3
 AND ...
```

SQL vs Relational Algebra (3)

```
SELECT  DISTINCT X.A1 AS B1, Y.A4 AS B2
FROM            R1 AS X, R2 AS Y, R1 AS Z
WHERE X.A2 = Y.A3 AND Y.A4 = Z.A1
```

$$\rho_{B1,B2 \leftarrow A1,A4} (\pi_{A1,A4} (\sigma_{A2 = A3 \text{ AND } A4 = C1} (R1 \bowtie R2 \bowtie \rho_{C1,C2 \leftarrow A1,A2} (R1))))$$

SQL: Evaluating the Queries

- SQL is a declarative language. We are providing the semantics by examples
- DBMS have *query execution plans* for efficient evaluations:
  - Selections are run as soon as possible
  - When possible, join are ran instead of cartesian products

SQL: Formulating the Queries

- We don’t necessarily have to write efficient queries since DBMS embed query optimizers
- Hereby it is more important that the provided queries are easy to understand (avoiding errors when formulating the query)

Database Example

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

MOTHERHOOD	
Mother	Child
Abby	Alice
Abby	Louis
Jesse	Olga
Jesse	Phil
Alice	Jim
Alice	James

FATHERHOOD	
Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

Example 1

- People's fathers earning more than 20

$$\pi_{Father}\ (FATHERHOOD \bowtie_{Child\ =Name}\ \sigma_{Income>20}\ (PEOPLE))$$

- Same query using SQL:

```sql
SELECT DISTINCT Father
FROM PEOPLE, FATHERHOOD
WHERE Name=Child AND Income > 20
```

Example 2

- Return the people's name, income and their father's income, where such people earn more than their fathers

$$\pi_{\text{Name, Income, IF}} (\sigma_{\text{Income>IF}} (\rho_{\text{NF,AF,IF} \leftarrow \text{Name,Age,Income}} (\text{PEOPLE}) \bowtie_{\text{NF=Father}} (\text{FATHERHOOD} \bowtie_{\text{Son=Name}} \text{PEOPLE})))$$

```
SELECT C.Name, C.Income, F.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Name = Father AND Child =
C.Name          AND C.Income > F.Income
```

SELECT with Renaming

```
SELECT C.Name AS Name,
       C.Income AS Income,
       F.Income AS fatherIncome
FROM PEOPLE F, FATHERHOOD,
 PEOPLE C
WHERE F.Name = Father AND
      Child = C.Name AND
      C.Income > F.Income
```

Using Expressions in the Target List

```
SELECT Income/2 AS
halvedIncome
FROM PEOPLE
WHERE Name = 'Louis'
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Louis	50	40

halvedIncome
20

JOIN Statement

- Return each person's mother and father
  - Implicit JOIN:

```
SELECT F.Child, Father, Mother
FROM MOTHERHOOD M, FATHERHOOD F
WHERE M.Child = F.Child
```

  - Explicit JOIN:

```
  SELECT Mother, FATHERHOOD.child,
Father
  FROM MOTHERHOOD JOIN FATHERHOOD ON
       FATHERHOOD.Child =
MOTHERHOOD.Child
```

SELECT with JOIN: Syntax

```
SELECT    …
FROM LeftTable { … JOIN
 RightTable
     ON joincondition }, …
[ WHERE otherPredicate ]
```

Example

- Return name, income and father's income of those people having a greater income than their father's

```sql
SELECT C.Name, C.Income, F.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Name = Father AND Child = C.Name
  AND
      C.Income > F.Income
```

```sql
SELECT C.Name, C.Income, F.Income
FROM (PEOPLE F JOIN FATHERHOOD ON
      F.Name = Father) JOIN PEOPLE C ON
      Child = C.Name
WHERE C.Income > F.Income
```

Natural Join

$$\pi_{\text{Child,Father,Mother}}\ (\text{FATHERHOOD} \bowtie_{\text{Child=Name}} \rho_{\text{Name}\leftarrow\text{Child}}\ (\text{MOTHERHOOD}))$$

$$\text{FATHERHOOD} \bowtie \text{MOTHERHOOD}$$

```
SELECT Mother, FATHERHOOD.Child,
Father
FROM MOTHERHOOD JOIN FATHERHOOD ON
     FATHERHOOD.Child =
MOTHERHOOD.Child
```

```
SELECT Mother, Child, Father
FROM MOTHERHOOD NATURAL JOIN
 FATHERHOOD
```

Outer Join

- With the previous joins, also called inner joins, some of the tuples could be discarded from the final result: this happens if they don't have a correspondent tuple in the other table
- In order to avoid this information loss, we can use:

  LEFT/RIGHT/FULL **OUTER JOIN**

- When such join is either left or right, the **OUTER** keyword could be omitted because left and right are "outer" by definition

Left (Outer) Join

- Return the father and the mother, if known

```
   SELECT FATHERHOOD.Child, Father, Mother
   FROM FATHERHOOD LEFT [OUTER] JOIN
        MOTHERHOOD ON FATHERHOOD.Child =
```

FATHERHOOD.Child	Father	Mother
Frank	Steve	NULL
Olga	Louis	Jesse
Phil	Louis	Jesse
Jim	Frank	Alice
James	Frank	Alice

Outer Join

```
SELECT FATHERHOOD.Child, Father, Mother
FROM MOTHERHOOD JOIN FATHERHOOD ON
     MOTHERHOOD.Child = FATHERHOOD.Child

SELECT FATHERHOOD.Child, Father, Mother
FROM MOTHERHOOD LEFT OUTER JOIN FATHERHOOD
ON
     MOTHERHOOD.Child = FATHERHOOD.Child

SELECT FATHERHOOD.Child, Father, Mother
FROM MOTHERHOOD FULL OUTER JOIN FATHERHOOD
ON
     MOTHERHOOD.Child = FATHERHOOD.Child
```

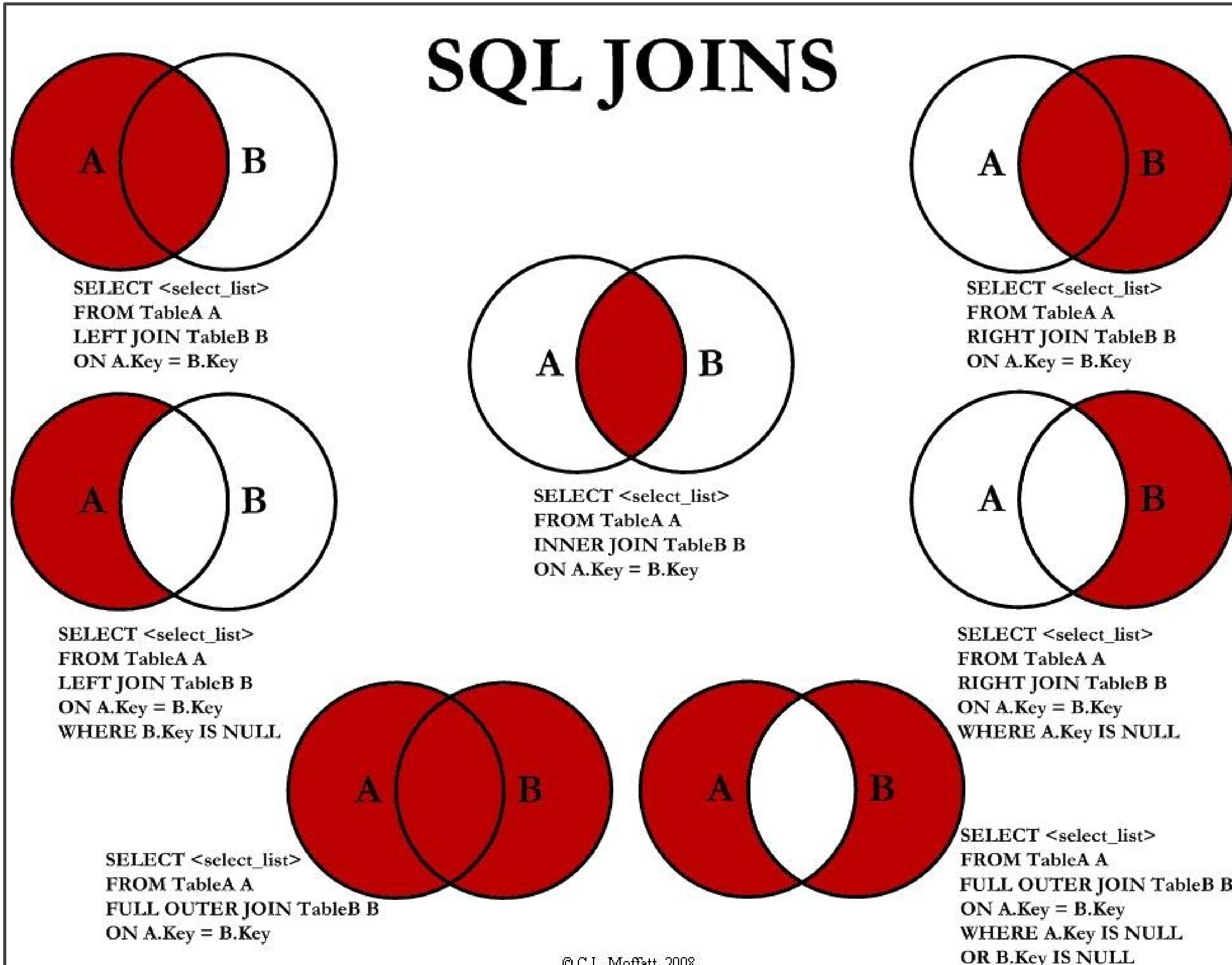
- What does the last query return?

Full Outer Join: an Example

FATHERHOOD.Child	Father	Mother
NULL	NULL	Abby
NULL	NULL	Abby
Olga	Louis	Jesse
Phil	Louis	Jesse
Jim	Frank	Alice
James	Frank	Alice
Frank	Steve	NULL

The full outer join returns all the tuples that were excluded on both left and right operand

Recap


SQL JOINS
A
B
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
© C.L. Moffatt, 2008

Sorting the Answer

- Provide the name and the income of people being less than 30 yo sorted by **alphabetic order**

```
SELECT Name, Income
FROM PEOPLE
WHERE Age < 30
ORDER BY Name ASC
```

- **ASC** ascending order (default)
- **DESC** descending order

Sorting the Answer: an Example

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

Sorting the Answer (1)

```
SELECT Name,
  Income
FROM PEOPLE
WHERE Age <= 30
```

Name	Income
Jim	21
James	15
Phil	30

```
SELECT Name,
  Income
FROM PEOPLE
WHERE Age <= 30
ORDER BY Name
```

Name	Income
James	15
Jim	21
Phil	30

- **ORDER BY** ‘s default sorting order is ascending

Sorting the Answer (2)

```
SELECT Name,
  Income
FROM PEOPLE
WHERE Age <= 30
O
```

Name	Income
James	15
Jim	21
Phil	30

```
SELECT Name,
  Income
FROM PEOPLE
WHERE Age <= 30
O
```

Name	Income
Phil	30
Jim	21
James	15

Union, Intersection, Difference

- The **SELECT** requires a specific statement for performing unions:

```
SELECT    …
UNION          [ALL]
SELECT    …
```

- In the result the rows are unique (except when **ALL** is used. In this case we have a *multiset union*)

Set Union

MOTHERHOOD	
Mother	Child
Abby	Alice
Abby	Louis
Jesse	Olga

FATHERHOOD	
Father	Child
Steve	Frank
Louis	Olga
Louis	Phil

```
SELECT Child
FROM
MOTHERHOOD
UNION
SELECT Child
FROM
FATHERHOOD
```

Child
Alice
Louis
Olga
Frank
Phil

Multiset Union

MOTHERHOOD	
Mother	**Child**
Abby	Alice
Abby	Louis
Jesse	Olga

FATHERHOOD	
Father	**Child**
Steve	Frank
Louis	Olga
Louis	Phil

```
SELECT Child
FROM MOTHERHOOD
UNION ALL
SELECT Child
FROM FATHERHOOD
```

Child
Alice
Louis
Olga
Frank
Olga
Phil

Olga appears twice

Positional Notation (1)

```
SELECT Father, Child
FROM FATHERHOOD
UNION
SELECT Mother, Child
FROM MOTHERHOOD
```

- When two tables have different schema, how could we resolve the conflict by renaming?
    - Either fictitious or none
    - We always assume the names of the first operand
    - Merge the conflicting attributes

Positional Notation: First Operand

Father	Child
Steve	Frank
Louis	Olge
Louis	Phil
Frank	Jim
Frank	James
Abby	Alice
Abby	Louis
Jesse	Olga
Jesse	Phil
Alice	Jim
Alice	James

Positional Notation (2)

```
SELECT Father,
  Child
FROM FATHERHOOD
   UNION
SELECT Child,
  Mother
FROM MOTHERHOOD
```

```
SELECT Father,
  Child
FROM FATHERHOOD
   UNION
SELECT Mother,
  Child
FROM MOTHERHOOD
```

- The resulting tables’ resulting scheme in both cases is `(Father, Child)`

Difference

```
SELECT Name
FROM EMPLOYEE
  EXCEPT
SELECT Surname AS Name
FROM EMPLOYEE
```

- We could later on express such operator through *nested* select *queries*

Intersection

```
SELECT Name
  FROM EMPLOYEE
      INTERSECT
  SELECT Surname AS Name
  FROM EMPLOYEE
```

- It is the same as

```
SELECT E.Name
  FROM EMPLOYEE E, EMPLOYEE F
  WHERE E.Name = F.Surname
```

Database Example

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

MOTHERHOOD	
Mother	Child
Abby	Alice
Abby	Louis
Jesse	Olga
Jesse	Phil
Alice	Jim
Alice	James

FATHERHOOD	
Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

Nested Queries

Predicates allow to:

- compare one (or more, as we will see later) attributes with the result of a nested (“sub”) query
- use the existential quantifier (*exists*, ∃)

Nested Queries: an Example (1)

- Provide the name and the income of Frank’s father

```
SELECT Name, Income
FROM PEOPLE, FATHERHOOD
WHERE Name=Father AND Child='Frank'
```

Cartesian product and **WHERE** (equi-join)

```
SELECT Name, Income
  FROM PEOPLE
  WHERE Name=(SELECT Father
              FROM FATHERHOOD
              WHERE Child='Frank')
```

WHERE clause is true when subquery result is equal to Name. Moreover, only one tuple is produced by the subquery

Nested Queries: Discussion

- Nested queries are “less declarative”, but sometimes more readable since they requires less variables
- Nested and non-nested queried could be combined
- The “subqueries” within nested ones cannot express set operations (“the union can be performed within the *outer query*”); this limitation is not significative
- Comparison operators require single values as operands. A solution is needed to compare a value with the result of a query (i.e., a relation)

Nested Queries: ANY & ALL

- Nested queries can be formulated through a predicate using either **`ANY`** or **`ALL`** alongside with a comparison operator (>, <, =, >=, ..), solving the homogeneity problem

```
Attribute op ANY( Expr )
```

  - An outer query tuple is matched if it satisfies the predicate with respect to any tuples within *Expr*

```
Attribute op ALL( Expr )
```

  - A outer query tuple is matched if it satisfies the predicate with respect to all tuples within *Expr*

Nested Queries: IN

`Attribute IN( Expr )`

- An outer query tuple is matched if its values in `Attribute` is contained within the elements returned by *Expr*
- `ANY`, `ALL` and `IN` can be negated through using the word `NOT` before
- Some interesting equivalences:
  - *A* **IN**(*Expr*) ≡ *A* = **ANY**(*Expr*)
  - *A* **NOT IN**(*Expr*) ≡ *A* ≠ **ALL**(*Expr*)

Nested Queries: an Example (2a)

- Provide name and income of the fathers' having child earning more than 20

```
SELECT DISTINCT F.Name, F.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Name = FATHERHOOD.Father AND
      FATHERHOOD.Child = C.Name AND C.Income > 20
```

- We can rewrite it without DISTINCT, because we will not join tables so the fathers' names will not be repeated for each child:

```
SELECT Name, Income
FROM PEOPLE
WHERE Name IN (SELECT Father
               FROM FATHERHOOD
               WHERE Child = ANY (SELECT Name
                                  FROM PEOPLE
                                  WHERE Income >
20))
```

Nested Queries: an Example (2b)

- Provide name and income of the fathers' having child earning more than 20

```
SELECT DISTINCT F.Name, F.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Nome = Father AND Child = C.Name
  AND
      C.Income > 20
```

- We can rewrite it without DISTINCT:

```
SELECT Name, Income
FROM PEOPLE
WHERE Name IN (SELECT Father
                  FROM FATHERHOOD, PEOPLE
               WHERE Child=Name AND
Income>20)
```

Nested Queries: an Example (3)

- Provide name and income of the fathers' having child earning more than 20, and provide the child's income too

```
SELECT DISTINCT F.Name, F.Income, C.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Name = Father AND Child = C.Name AND
      C.Income > 20
```

- Does the following one provide the same answer?

```
SELECT Name, Income
FROM PEOPLE
WHERE Name IN (SELECT Father
               FROM FATHERHOOD
               WHERE Child = ANY (SELECT Name
                                  FROM PEOPLE
                                  WHERE
Income>20))
```

ANY meaning: clause is true if Child value is equal to any of the values returned by nested query

Nested Queries Visibility

```
SELECT Name, Income
FROM PEOPLE
WHERE Name IN     (SELECT Father
                   FROM FATHERHOOD
                   WHERE Child=ANY (SELECT
  Name
                                    FROM
  PEOPLE
```

Name	Income
Frank	20
Louis	40
Louis	40

Father
Frank
Louis
Louis

Name
Jim
Alice
Jesse
Phil
Louis
Olga
Steve
Abby

- **Answer:** no, because for each father we do not view the child’s income

Nested Queries: Considerations

- **Visibility Rules**
  - It is not possible to refer to variables declared within inner blocks
  - If a variable's name is omitted, we assume to take the "nearest" declared one
- We can refer to a variable defined:
  - within the scope of the query in which it is defined (*i.e.*, outer blocks)
  - or within the scope of a nested query, at any level, (*i.e.*, inner block) within it

Semantics of Nested Queries with Variables

- The inner query is performed one time for each tuple within the outer query
- The only way to avoid this is by creating a view, which, however, modifies the database schema

Existential Quantification

`EXISTS` `(` *`Expr`* `)`

- The predicate is true if *`Expr`* returns at least one tuple (i.e., *`Expr`* returns a non-empty set)
- Typically *`Expr`* is a nested query
- Useful with a linking variable between the outer query and the nested query

Existential Quantification: an Example (1)

- People having at least one child

```
SELECT *
FROM PEOPLE
WHERE EXISTS (SELECT *
              FROM FATHERHOOD
              WHERE Father = Name)
      OR
      EXISTS (SELECT *
              FROM MOTHERHOOD
              WHERE Mother = Name)
```

Name column is taken from the relation declared in the outer block

Existential Quantification: an Example (2)

- Fathers having all their children earning more than 20

```
SELECT DISTINCT Father
FROM FATHERHOOD Z
WHERE NOT EXISTS (SELECT *
                  FROM FATHERHOOD W,
                       PEOPLE
                  WHERE
W.Father=Z.Father
                                   AND
W.Child=Name
                                   AND
Income<=20)
```

`Z.Father` is taken from the relation obtained in the outer block

Existential Quantification: Error

- People of the same age and income

```
SELECT DISTINCT *
FROM PEOPLE
WHERE EXISTS (SELECT *
              FROM PEOPLE
              WHERE Age=Age AND
                    Income=Income)
```

Scope rule: Age and Income, without table reference, implicitly refers to to closest **FROM** clause

Existential Quantification: Correct

- People of the same age and income

```
SELECT DISTINCT *
FROM PEOPLE P
WHERE EXISTS (SELECT *
              FROM PEOPLE
              WHERE P.Name≠Name AND
                    P.Age=Age AND
                    P.Income=Income)
```

Visibility: Wrong!

```
SELECT *
FROM EMPLOYEE
WHERE Dept IN (SELECT Name
               FROM DEPARTMENT D1
               WHERE
Name='Production')
      OR
      Dept IN (SELECT Name
               FROM DEPARTMENT D2
               WHERE D2.City =
D1.City)
```

Wrong because in the last selection the city of D1 is not visible

Set Difference and Nested Queries

```
SELECT Name
FROM EMPLOYEE
  EXCEPT
SELECT Surname AS Name
FROM EMPLOYEE

SELECT E.Name
FROM EMPLOYEE E
WHERE NOT EXISTS (SELECT *
                  FROM EMPLOYEE
                  WHERE Surname =
  E.Name)
```

Nested Queries Positions

- **WHERE** clause: standard use, we have seen several examples so far
- **FROM** clause: a new data source (i.e., a relation) is required, the alternative is to create a view that, nevertheless, modifies the database schema
- **SELECT** clause: uncommon use, it is equivalent to a join. It necessarily requires a tuple as a result

Nested Queries in FROM clause

- Provide the name and the income of Jim’s children

```
SELECT Name, Income
   FROM PEOPLE P, (SELECT Child
                        FROM FATHERHOOD
                        WHERE
Father='Jim') AS
                        JIMCHILD
WHERE Name = JIMCHILD.Child
```

Nested Queries in SELECT clause

- Provide the total shipping charges for each customer in the customer table

```
SELECT CUSTOMER.Num,
       (SELECT SUM(ShipCharge)
        FROM ORDERS
        WHERE
     CUSTOMER.Num=ORDERS.Num)
       AS TotalShipCharge
  FROM CUSTOMER
```

The query can be rewritten by simply using the join between the two tables!

Aggregate Functions

In the target list, we can put expressions that compute values from a set of tuples through aggregate functions:

- *Aggr:* COUNT | MIN | MAX | AVG | SUM
- Basic syntax:

  Aggr([**DISTINCT**] *)

  Aggr([**DISTINCT**] *Attribute*)

Aggregate Functions: COUNT (1)

- The number of Frank's children

```
  SELECT COUNT(*) AS
NumFrankChildren
  FROM FATHERHOOD
  WHERE Father = 'Frank'
```

- The aggregate function (**COUNT**) is applied to the tuples of the following result:

```
  SELECT *
  FROM FATHERHOOD
  WHERE Father = 'Frank'
```

Aggregate Functions: COUNT (2)

FATHERHOOD	
Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

NumFrankChildren
2

COUNT DISTINCT

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	24
Alice	55	36
Jesse	50	36

```
SELECT
COUNT(*)
FROM PEOPLE
```

COUNT(*)
4

```
SELECT COUNT(DISTINCT
Income)
FROM PEOPLE
```

COUNT(DISTINCT Income)
3

Some Other Aggregate Functions

- SUM, AVG, MAX, MIN
- Average of the income of Frank's children

```
SELECT AVG(Income)
FROM PEOPLE JOIN FATHERHOOD ON
     Name=Child
WHERE Father='Frank'
```

COUNT with NULL Values (1)

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	NULL
Alice	55	36
Jesse	50	36

```
SELECT
COUNT(*)
FROM PEOPLE
```

COUNT(*)
4

```
SELECT COUNT(Income)
FROM PEOPLE
```

COUNT(Income)
3

COUNT with NULL Values (2)

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	NULL
Alice	55	21
Jesse	50	35

```sql
SELECT COUNT(DISTINCT Income)
FROM PEOPLE
```

COUNT(DISTINCT Income)
2

Aggregate Functions and NULLs

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	NULL
Alice	55	21
Jesse	50	35

```
SELECT AVG(Income) AS AvgInc
FROM PEOPLE
```

AvgInc
25.6

Aggregate Functions and Target List

- A wrong query:

```
SELECT Name, MAX(Income)
FROM PEOPLE
```

- Whose the name? We cannot extract the name having the max income. The *Target List* must have all the same types of attributes

```
SELECT MIN(Age), MAX(Income)
FROM PEOPLE
```

Maximum and Nested Queries

- Return the people having the (same) maximum income

```
SELECT *
FROM PEOPLE
WHERE Income = (SELECT
MAX(Income)
                FROM PEOPLE)
```

Aggregate Functions and Grouping (1)

- Aggregate function can operate over relations' groups via the **GROUP BY** statement:

GROUP BY *AttrList*

Aggregate Functions and Grouping (2)

- The number of the fathers' children

```
SELECT Father, COUNT(*) AS NumberOfChildren
FROM FATHERHOOD
GROUP BY Father
```

FATHERHOOD

Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

Father	NumberOfChildren
Steve	1
Louis	2
Frank	2

GROUP BY: Semantics

1. Perform the query without aggregate functions and without aggregate operators

```
SELECT *
FROM FATHERHOOD
```

2. Then perform the grouping and apply the aggregate function over each group

Grouping and Target Lists

Wrong:

```
SELECT Father, AVG(C.Income), F.Income
FROM PEOPLE C JOIN FATHERHOOD ON
  C.Name=Child
     JOIN PEOPLE F ON Father=F.Name
GROUP BY Father
```

We also need to group by **F.Income**

Correct:

```
SELECT Father, AVG(C.Income), F.Income
FROM PEOPLE C JOIN FATHERHOOD ON
  C.Name=Child
     JOIN PEOPLE F ON Father=F.Name
GROUP BY Father, F.Income
```

Conditions on Groups

- Provide those fathers whose children have an average income greater than 25; return the father and their children's average income

```
SELECT Father, AVG(C.Income)
FROM PEOPLE C JOIN FATHERHOOD
ON
     C.Name=Child
GROUP BY Father
HAVING AVG(C.Income) > 25
```

WHERE vs. HAVING

- Provide the fathers whose children under 30 yo have an average income greater than 20

```
  SELECT Father, AVG(C.Income)
  FROM PEOPLE C JOIN FATHERHOOD
ON
        C.Name=Child
  WHERE C.Age < 30
  GROUP BY Father
  HAVING AVG(C.Income) > 20
```

Grouping and NULLs

R	A	B
	1	11
	2	11
	3	NULL
	4	NULL

```
SELECT B,
  COUNT(*)
FROM R GROUP BY
  B
```

B	COUNT(*)
11	2
NULL	2

```
SELECT A,
  COUNT(*)
FROM R GROUP BY
  A
```

A	COUNT(*)
1	1
2	1
3	1
4	1

```
SELECT A,
  COUNT(B)
FROM R GROUP BY A
```

A	COUNT(B)
1	1
2	1
3	0
4	0

SELECT Syntax: Summary

```
SELECT AttList1 + Exprs
FROM TableList + Joins
[ WHERE Condition ]
[ GROUP BY AttList2]
[ HAVING AggrCondition]
[ ORDER BY OrderingAttr1]
```

Updating Operations

- Such operations are
  - **INSERT**
  - **DELETE**
  - **UPDATE**
- … of one or more tuples within a table
- … on the basis of a predicate that may involve other relations

INSERT

```
INSERT INTO Table [(AttList)]
  VALUES( Vals )
```

or

```
INSERT INTO Table [(AttList)]
  SELECT ...
```

INSERT: Examples

```
INSERT INTO PEOPLE(Name, Age, Income)
  VALUES('Jack', 25, 52)

INSERT INTO PEOPLE VALUES('John', 25,
 52)
INSERT INTO PEOPLE(Name, Income)
  VALUES('Robert', 55)

INSERT INTO PEOPLE(Name)
  SELECT Father
  FROM FATHERHOOD
  WHERE Father NOT IN (SELECT Name
                       FROM PEOPLE)
```

INSERT: Discussion

- The attributes’ and the values’ **ordering is relevant**
- Both lists should have the **same number of arguments**
- If the attribute list is omitted, we assume that all the attributes are considered and each value corresponds to a **specific attribute as declared in the relation’s schema**
- If the attribute list does not contain all the relation’s attributes, either a **NULL value or a default value are emplaced**

Deleting Tuples

```
DELETE FROM Table
   [ WHERE Condition ]
```

Deleting Tuples: some Examples

```
DELETE FROM PEOPLE
   WHERE Age < 35

DELETE FROM FATHERHOOD
   WHERE Child NOT IN (SELECT Name
                       FROM PEOPLE)

DELETE FROM FATHERHOOD
```

Deleting Tuples: Discussion

- Removes the tuples satisfying a given condition
  - It could cause the removal of other tuples (if the constraints are defined using `CASCADE`)
  - If no condition is provided, such has to be intended as `WHERE TRUE`

Updating Tuples

```
UPDATE TableName
SET Attribute = < Expr |
                  SELECT … |
                  NULL |
                  DEFAULT >
  [ WHERE Condition ]
```

Updating Tuples (1)

BEFORE

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

```
UPDATE PEOPLE
SET Income = 45
WHERE Name = Bob
```

AFTER

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	45

Updating Tuples (2)

BEFORE

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

```
UPDATE PEOPLE
SET Income =
Income*1.1
WHERE Age < 30
```

AFTER

PEOPLE		
Name	Age	Income
Jim	27	33
James	25	16.5
Bob	55	36

Updating Tuples (3)

```
UPDATE PEOPLE
SET Income =
    (SELECT Income FROM
     PEOPLE WHERE
      Name=Jim)
WHERE Name = Bob
```

BEFORE

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

AFTER

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	30

Updating Tuples (4)

BEFORE

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

```
UPDATE PEOPLE
SET Income = NULL
WHERE Age < 30
```

AFTER

PEOPLE		
Name	Age	Income
Jim	27	NULL
James	25	NULL
Bob	55	36

Updating Tuples (5)

BEFORE

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

```
UPDATE PEOPLE
SET Income = DEFAULT
WHERE Age < 30
```

AFTER

PEOPLE		
Name	Age	Income
Jim	27	0
James	25	0
Bob	55	36

Assuming that in CREATE TABLE we specified 0 as the DEFAULT value for Income