1 AGGREGATE OPERATIONS

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

- `COUNT ([DISTINCT] A)` : The number of (unique) values in the A column.
- `SUM ([DISTINCT] A)`: The sum of all (unique) values in the A column.
- `AVG ([DISTINCT] A)`: The average of all (unique) values in the A column.
- `MAX (A)`: The maximum value in the A column
- `MIN (A)`: The minimum value in the A column

1: Count the number of different sailor names

```
SELECT COUNT (DISTINCT S.sname)
FROM Sailors S
```

2 The GROUP BY and HAVING Clauses

```
SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

So far we have applied aggregate operations to all rows in a relation. We often want to apply aggregate operations to each of a number of **groups** of a row in a relation.

2: Find the age of the youngest sailor for each rating level

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

- `GROUP By X` means **put all those with the same value for X in the one group**
- `GROUP By X, Y` means **put all those with the same values for both X and Y in the one group**
- Each column that appears in **select-list** must also appear in the **grouping-list**. This is because each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on values of columns in **grouping-list**.

- If `GROUP BY` is ommited, the entire table is regarded as a single group
- `HAVING` applies a selection criteria to the groups. It can be used to select groups which satisfy a given condition. Think of it as a `WHERE` clause for the returned groups
- A column appearing in the **group-qualification** for `HAVING` must also appear in **grouping-list** for `GROUP BY`
- In general `WHERE` clause is evaluating before `HAVING`. The `WHERE` clause acts as a pre filter where as `HAVING` clause as a post filter.
- The difference between `HAVING` and `WHERE` in SQL is that the `WHERE` cannot be used with aggregates, but `HAVING` can. The `WHERE` clause works on row's data, not on aggregated data.

3 EXERCISES

Tables used in this note:
Sailors(sid: integer, sname: string, rating: integer, age: real);
Boats(bid: integer, bname: string, color: string);
Reserves(sid: integer, bid: integer, day: date).

(1) For each red boat, find the number of reservations for this boat.

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

(2) Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT S.rating, AVG(S.age) AS avg_age
FROM Sailors S
GROUP BY S.rating
HAVING COUNT(*) > 1
```

(3) Find age of the youngest sailor with age $\geq$ 18, for each rating with at least 2 sailors between 18 and 60.

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

(4) Find the average age of sailors who are of voting age (i.e. at least 18 yrs old) for each rating level that has at least two sailors.

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

(5) Find the ratings for which the average age of sailors is the minimum over all ratings.

```
SELECT S.rating, AVG(S.age) as avg_age
FROM Sailors S
GROUP BY S.rating
ORDER BY avg_age
LIMIT 1
```

4 INNER Joins

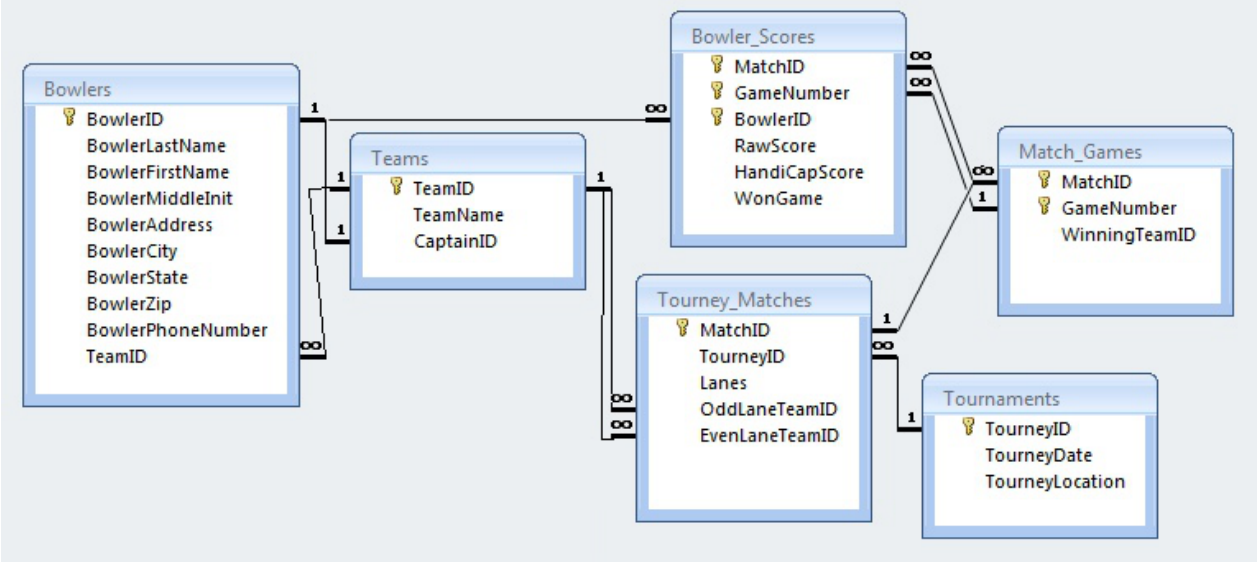
`table1 INNER JOIN table2 ON table1.col1 = table2.col2`

- Creates a virtual table matching rows to the columns specified after `ON`.
- `INNER` can be omitted.
- If a row is cannot be matched in the other table, then it does not appear in the result virtual table.
- Multiple matches will creat a row for *every pair* of matches.

5 EXERCISES

BowlingLeague.sqlite:

More SQL



(1) Print a schedule of all the team matchups over the whole season (Date, Location, OddTeamName, EvenTeamName).

```
SELECT TourneyDate, TourneyLocation, OddTeam.TeamName, EvenTeam.TeamName
FROM Tournaments
JOIN Tourney_Matches ON Tournaments.TourneyID = Tourney_Matches.
   TourneyID
JOIN Teams AS OddTeam ON Tourney_Matches.OddLaneTeamID = OddTeam.TeamID
JOIN Teams AS EvenTeam ON Tourney_Matches.EvenLaneTeamID = EvenTeam.
   TeamID
```

(2) Print game results for Tournament 1, including match ID, game number, team names, bowler names, and raw score.

```
SELECT Bowler_Scores.MatchID, GameNumber, TeamName, Bowlers.
   BowlerFirstName || "" || Bowlers.BowlerLastName AS BowlerName,
   RawScore
FROM Bowler_Scores
JOIN Tourney_Matches ON Bowler_Scores.MatchID = Tourney_Matches.MatchID
JOIN Bowlers ON Bowlers.BowlerID = Bowler_Scores.BowlerID
JOIN Teams ON Teams.TeamID = Bowlers.TeamID
WHERE TourneyID = 1
```

6 The Memory/Storage Hierarchy