Assume relations $EMP(ssn, \text{name, sex, salary})$ and $PROJ(essn, pnumber, hours)$.

1. **Grouping**

1. “Give the payroll (sum of salaries) per project”

   ```sql
   SELECT pnumber, sum(salary) as payroll
   FROM EMP, PROJ
   WHERE ssn = essn
   GROUP BY pnumber
   ```

2. “Compute the average salary per project”

   ```sql
   SELECT pnumber, avg(salary)
   FROM EMP, PROJ
   WHERE ssn = essn
   GROUP BY pnumber
   ```

3. “List the payroll for each project but using only salaries greater than 20,000”

   ```sql
   SELECT pnumber, sum(salary) as payroll
   FROM EMP, PROJ
   WHERE ssn = essn AND salary > 20,000
   GROUP BY pnumber
   ```

4. “List the payroll for each project but list only those project whose payroll is greater than 100,000”

   ```sql
   SELECT pnumber, sum(salary) as payroll
   FROM EMP, PROJ
   WHERE ssn = essn
   GROUP BY pnumber
   HAVING payroll > 100,000
   ```

5. The following is an example where $HAVING$ references an aggregation which is not (but could have been) part of the grouping. “List the projects whose payrolls exceed their maximum allowed cost. This cost is computed as the number of hours put by all employees in the project times 300”

   ```sql
   SELECT pnumber, sum(salary) as payroll
   FROM EMP, PROJ
   WHERE ssn = essn
   GROUP BY pnumber
   HAVING payroll > sum(hours) * 100
   ```
6. “Compute the average payroll”

This is done in two steps. In the first one, create a table or view, say PAYROLLS, with the first query. In the second one, compute the average from this table or view:

```
SELECT avg(payroll) FROM PAYROLLS
```

7. “Select the projects with a payroll higher than the average payroll”.

This is also done in two steps. The first step is as above. The second one picks up projects with payroll higher than average:

```
SELECT pnumber
FROM PAYROLLS
WHERE payroll > (SELECT avg(payroll) FROM PAYROLLS)
```

**NOTE:** You need to know the payroll before you can compute the average. Thus, the following will fail:

```
SELECT pnumber, sum(salary) as payroll
FROM EMP, PROJ
WHERE ssn = essn
GROUP BY pnumber
HAVING payroll > avg(payroll)
```

This condition does not make much sense since since the HAVING applies to each tuple resulting from the GROUP BY and there is only one payroll per tuple; hence the average will be the payroll itself. What we ask for is the average of all payrolls.

## 2 Subqueries

1. “Retrieve the names of employees that work on some (i.e. one or more) projects that Smith works on”.

   (a) As a non correlated query:
   
   ```
   SELECT name
   FROM EMP, PROJ
   WHERE EMP.ssn = PROJ.essn AND PROJ.pnumber IN
   (SELECT pnumber
    FROM PROJ, EMP
    WHERE PROJ.essn = EMP.ssn AND EMP.name = “Smith”)
   ```

   Observe that the nested query is a regular query which makes no reference to the outer query. The nested subquery is evaluated once and the obtained result plugged in for the query; then the outer query is evaluated.

   (b) Same question with correlated subquery:
   
   ```
   SELECT name
   FROM EMP, PROJ P
   WHERE EMP.ssn = PROJ.essn AND EXISTS
   (SELECT pnumber
    FROM PROJ, EMP
    WHERE PROJ.essn = EMP.ssn AND EMP.name = “Smith”
    AND P.pnumber = PROJ.pnumber)
   ```

   The nested query has a reference to the outer query: P is a tuple variable for PROJ on the FROM clause of the outer query, so P.pnumber refers to the outside query. This is like a parameter: in
order to evaluate the nested subquery, we need to plug a value for \( P.pnumber \). The evaluation goes like this: evaluate the outer query except the EXISTS predicate (i.e. go through the join of EMP and PROJ); for every possible value of \( P.pnumber \), evaluate the subquery and check the EXISTS predicate. If the predicate returns true, the tuple that provided the value for \( P.pnumber \) is part of the answer; otherwise it is not.

(c) Note that most queries using \( \text{IN} \) (or = \( \text{SOME} \)) can be expressed without subqueries:

```sql
SELECT EMP.name
FROM EMP E, PROJ P
WHERE EMP.ssn = PROJ.essn AND E.essn = P.essn
AND E.name = "Smith" AND PROJ.pnumber = P.pnumber
```

Note the similarity with Relational Algebra, where self-joins are introduced to have all elements necessary for comparison in a single tuple.

2. “Retrieve the names of employees that work on all project that Smith works on”.

SQL, like the Algebra, has no direct counterpart for situations where we need to express a condition that affects all (every, each) element of a set. Therefore we must use embedded subqueries to express the idea:

```sql
SELECT name
FROM EMP E1, PROJ
WHERE EMP.ssn = PROJ.essn AND NOT EXISTS
(SELECT pnumber
FROM PROJ, EMP
WHERE PROJ.essn = EMP.ssn
AND EMP.name = "Smith"
AND PROJ.pnumber NOT IN
(SELECT pnumber
FROM EMP E, PROJ
WHERE E.essn = PROJ.essn
and E.essn = E1.essn))
```

The intuition is the following: all \( x \) have property \( P \) is the same as there is no \( x \) that does not have property \( P \) (think about it for a moment!). Therefore, the above query says: for a given employee, there are no project numbers for projects that Smith works on that are not project numbers of the given employee (repeat aloud several times).

Note that the latter embedded subquery is correlated (with \( E1.essn \) being the parameter that introduces correlation); we could have simply written that subquery as

```sql
(SELECT pnumber
FROM PROJ
WHERE E1.essn = PROJ.essn)
```

### 2.1 Equivalences

One of the characteristics of SQL is that there are several ways to write all but the most trivial queries. Some versions are more efficient than others. However, SQL semantics makes true equivalence a very tricky thing, because of duplicate tuples and nulls. Example: joining tables \( R \) and \( S \). The above are almost equivalent, but not quite, because of duplicates (can you see why?).

```sql
SELECT R.B FROM R, S WHERE R.A = S.D
SELECT R.B FROM R WHERE R.A = ANY (SELECT S.D FROM S)
SELECT R.B FROM R WHERE R.A IN (SELECT S.D FROM S)
SELECT R.B FROM R WHERE EXISTS (SELECT S.D FROM S WHERE S.D = R.A)
```