Chapter 5
More SQL

In this chapter, we present some additional SQL operators in order to provide a well-rounded, complete overview of the language. We have left some operators for this chapter because they are not central to data analysis (set operations, WHERE clause subqueries) or because they are a bit more advanced (WINDOWS aggregates). At some points, we will revisit some solutions seen in previous chapters in order to show how to write them more efficiently (or, simply, in a different manner).

5.1 More on Joins

We already explained joins in Sect. 3.1.1. In this section, we add a bit more nuance to the behavior of this operator and present some versions of it that are useful in certain situations.

Joins can be combined with all other operations we have seen (including grouping, aggregation, and selection). These operations work on a single table, but that is exactly what the join produces, so one way to think of queries with joins is as follows: all tables mentioned in the `FROM` clause are joined; then, the query proceeds using this result exactly the way it did in the case of a single table.

There are two subtleties to understand about joins: the *existential* effect and the *multiplicative* effect. The existential effect refers to the fact that, when joining two tables, tuples in either table that do not meet the join condition are dropped from the result. As a consequence, a join may not have all the data in a table, and we should not assume otherwise.

Join: Existential Effect

Assume tables `EMP(ssn,name,...)` and `SALES(essn,itemid,amount,date)`, where `essn` is a foreign key to `EMP`. The table `SALES` keeps track of the sales made,

A. Badia, *SQL for Data Science*, Data-Centric Systems and Applications,
https://doi.org/10.1007/978-3-030-57592-2_5

noting the employee who did the sales, the item sold, the amount, and the date. We wish to analyze employee's performance in the last quarter, so we issue this query:

```
SELECT ssn, name, sum(amount) as total
FROM EMP join SALES on (ssn = essn)
WHERE month(date) = 'November' and year(date) = '2015'
GROUP BY ssn, name
ORDER BY total asc
LIMIT 10;
```

The idea is to identify the bottom 10 employees by sales so we can look into their situations and see what led to this not-so-good performance. But we are missing something: suppose that some employees (because they were sick, or absent, or had a really bad month) did not sell anything at all. Will they show in the result with a 0 for `total`? They will not. Instead, we will see the bottom 10 employees *among those employees who had sales during that time period*. So employees are better off selling nothing than selling a small amount.

Why is this happening? Because the employees who sold nothing on that period do not make it past the join, as there are no matching tuples for them in the rows of SALES that pass the filter of the WHERE clause condition. This can be solved by using an *outer join*. On an outer join, we preserve tuples that do not match. We can preserve non-matching tuples in the first (left) table (a 'left outer join'), on the second (right) table (a 'right outer join'), or on both tables (a 'full outer join'). The syntax is

```
#Outer join syntax:
FROM table1 [LEFT|RIGHT] [OUTER] JOIN table2 ON condition
```

Non-matching tuples are missing some attributes, so those attributes are padded with NULLs. Thus, in our previous example, if we use

```
FROM EMP left outer join SALES on (ssn = essn)
```

we will end up with all employees; those that have sales will generate 'regular' tuples; those with no sales will be included in the answer in tuples where all attributes for SALES are padded with NULLs. As a result, the GROUP BY will generate a group for them with a single tuple, and `sum(amount)` will return a 0 (the only value to sum is a NULL).

This approach works especially well with optional attributes (see Sect. 2.1). When we want all the objects of a table together with all attributes (including attributes that may not have values for all objects), an outer join is in order. Remember, however, that non-matching tuples introduce nulls, so those must be cleared before any analysis.

Outer Join Use

Assume a table

```
VAERS(id, received-date, state, age, sex, symptoms, died,
      date-of-death, cause-of-death)
```

that keeps track of adverse effects to vaccinations.[1] Because the last two attributes only make sense when the attribute `died` is TRUE, we could separate this table into two, `VAERS(id, received-date,state,age,sex,symptoms)` and `DEATHS(id,date-of-death,cause-of-death)`, where the table `DEATHS` contains the ids only for those cases where a death occurred. This would avoid having to deal with nulls and 'irregular' data; however, we may want to do an analysis that requires all data combined (for instance, to determine mortality rates). If we simply join the tables

```
SELECT *
FROM VAERS JOIN DEATHS on VAERS.id = DEATHS.id;
```

we will miss all the cases that did not result in death; what we really want is a left outer join:

```
SELECT *
FROM VAERS LEFT OUTER JOIN DEATHS on VAERS.id = DEATHS.id;
```

Note that the attributes `date-of-death, cause-of-death` will have nulls for the cases where no death occurred, and that there is no attribute `dead` to mark such cases explicitly—although adding one is quite easy, as we have seen in past examples.

Another aspect of join that confuses newcomers to SQL is the *multiplicative effect*: when a tuple in table R matches *several tuples* in table S, this tuple gets repeated as many times as there are matches. This is a problem when we use aggregates that are *duplicate sensitive*.[2] In such cases, we may get extraneous results in our analysis.

Join: Multiplicative Effect

Assume the following situation: we manage a bank database, with tables

```
BRANCH(bid, name, address,...),
LOANS(lid,bid,amount,...),
ACCOUNTS(aid,bid,balance,...)
```

[1] Modeled after a real dataset available at https://vaers.hhs.gov/.

[2] An aggregate is *duplicate insensitive* when its result over a dataset does not change if duplicate values are removed; examples of this are MIN and MAX. An aggregate is *duplicate sensitive* when it is not duplicate insensitive; SUM, COUNT, and AVG are examples.

The first table lists the branches of the bank, with `bid` as its key. The second one lists each loan the bank currently has, with `lid` its key and `bid` a foreign key to BRANCH, indicating where the loan originated. The third table lists all accounts in the bank, with `aid` its key and `bid` a foreign key to BRANCH indicating where the account was opened. A bank manager is concerned that some branches are underperforming and asks us to find any branches where the total loan amount is larger than the total in deposits. We write the following SQL query:

```
SELECT bid, name
FROM BRANCH as B joins LOANS as L on (B.bid = L.bid)
     joins ACCOUNTS A on (B.bid = A.bid)
GROUP BY bid, name
HAVING sum(amount) > sum(balance);
```

Now imagine a situation where a branch has 2 loans and 3 accounts; for instance,

BRANCH		
Bid	**Name**	**Address**
1	Highlands	1500 Bardstown Road

LOANS		
Lid	**Bid**	**Amount**
10	1	1000
20	1	900

ACCOUNTS		
Aid	**Bid**	**Balance**
15	1	500
25	1	600
35	1	700

The join of all 3 tables is given by

Bid	Name	Address	Lid	Bid	Amount	Aid	Bid	Balance
1	Highlands	1500 Bardstown Road	10	1	1000	15	1	500
1	Highlands	1500 Bardstown Road	10	1	1000	25	1	600
1	Highlands	1500 Bardstown Road	10	1	1000	35	1	700
1	Highlands	1500 Bardstown Road	20	1	900	15	1	500
1	Highlands	1500 Bardstown Road	20	1	900	25	1	600
1	Highlands	1500 Bardstown Road	20	1	900	35	1	700

After the join, that branch appears 6 times; each loan in this branch is repeated 3 times and each account is repeated 2 times. As a consequence, neither sum (on amounts or balances) is correct.

Note that there is no other way to represent this information when all data is put together in a single table. What is happening here is that, while loans and accounts are (indirectly) related through the branch, they are orthogonal to each other. Thus, we cannot associate only some loans with some accounts; this could

create a correlation between loans and accounts which does not truly exist. This is an example of data that cannot be really well analyzed in a single table.

What is the solution in this case? No extra operators are needed here; caution is. What we should do is compute each aggregate separately, one join at a time, in order to get the right results, which can then be compared. As usual, subqueries in the FROM clause are our friends:

```
SELECT bid, name
FROM (SELECT bid, name, sum(amounts) as loans
      FROM BRANCH as B joins LOANS as L on (B.bid = L.bid)
      GROUP BY bid, name) as TEMP1,
     (SELECT bid, name, sum(balance) as accounts
      FROM BRANCH as B joins ACCOUNTS A on (B.bid = A.bid)
      GROUP BY bid, name) as TEMP2
WHERE TEMP1.bid = TEMP2.bid and loans > accounts;
```

Note that one join multiplies the branch, since the relation between branches and loans is one-to-many (see Sect. 2.2), as is the relation between branches and accounts. This is ok; the repetition allows the GROUP BY to do its job by computing the aggregate value over each branch. What we need to avoid is mixing both one-to-many relationships together.

5.2 Complex Subqueries

The SQL standard allows for the expression of complex conditions in the WHERE clause that are specified using whole queries. Like the ones in the FROM clause, these 'embedded' queries are called subqueries. However, unlike the ones in the FROM clause, WHERE clause subqueries come in different flavors and with different predicates. We enumerate the types and give examples as follows:

- *aggregated subqueries*, which we have already seen, are subqueries with only aggregates in the SELECT clause. These are guaranteed to return a single value, which is compared against some attributes.

Example: Aggregated Subqueries

The query

```
SELECT name
FROM chicago-employees
WHERE  salary > (SELECT avg(salary)
           FROM chicago-employees
           WHERE salaried = True);
```

computes the average salary for all salaried employees; this result is then used to pick all employees that make more than this average.

- Subqueries with (NOT) IN: the IN predicate is used in combination with an attribute and a subquery. The subquery is expected to have a single attribute in its SELECT clause; as a result, the subquery returns a table with a single attribute on it—which we can think of as a list of values. The IN predicate checks whether the value of its first argument (attribute) in a given row is among those in the returned list of values.[3]

Example: IN Subquery

Assume tables

`EMPLOYEE(ssn,name,salary,job,dept)`

and

`DEPARTMENT(id,name,manager-ssn)`

and the query "list the names and salaries of all managers." The information about names and salaries is in table `EMPLOYEE`, but the information about who is a manager is in table `DEPARTMENT`, where `manager-ssn` is a foreign key to `EMPLOYEE.ssn`. We can write the following query:

```
SELECT name, salary
FROM EMPLOYEE
WHERE ssn IN (SELECT manager-ssn
              FROM DEPARTMENT);
```

The system runs the IN predicate on each row of `Employee`: it takes the value of `ssn` on the row, and it compares it to the list of ssn returned by the subquery, to see if it is one of them.

Naturally, NOT IN is simply the negation of IN: it returns true if the attribute value is not one of those in the returned list of values.

- Subqueries with (NOT) EXISTS: The EXISTS predicate takes a subquery and checks whether it returns an empty answer (no queries or not). EXISTS is satisfied if the subquery answer is *not* empty (i.e. if something exists in the answer).

Example: EXISTS Subquery

Assume the same database as the previous example, and suppose we want the name and salary of all employees in the Research (id "RE") department but only if there is a Marketing department (id "MK"); we are not sure whether this department exists. We can write the query

```
SELECT name, salary
FROM EMPLOYEE
```

[3]The SQL standard actually allows a more complex IN predicate, but most systems do not implement it.

```
WHERE dept = 'RE' and EXISTS (SELECT *
                              FROM DEPARTMENT
                              WHERE id = 'MK');
```

Note that the subquery uses '*', because the attributes returned are irrelevant; the EXISTS simply checks whether any rows are returned.

As before, NOT EXISTS is the negation of EXISTS; it is satisfied if the subquery returns an empty answer.

- Subqueries with ANY, ALL: the ANY and ALL predicates take an attribute and a subquery. The value of the attribute is compared to all values returned by the subquery; ANY requires that at least one comparison returns TRUE, while ALL requires that all comparisons return TRUE.

Example: Subqueries with ANY/ALL

Assume the EMPLOYEE-DEPARTMENT database. We want to find out which employees make more money than everyone in the Marketing department, so we run this query:

```
SELECT ssn, name
FROM EMPLOYEE
WHERE salary > ALL (SELECT salary
                    FROM EMPLOYEE
                    WHERE dept = "MK");
```

It is easy to see that several types of subqueries are redundant; for instance, the condition `attribute IN Subquery` is equivalent to `attribute = ANY Subquery`, and the condition `attribute NOT IN Subquery` is equivalent to `attribute <> ALL Subquery`. There are more equivalences, although some intuitive ones are disrupted by the presence of nulls.

Example: Subquery Equivalence and Nulls

The query used above to retrieve employees that make more than everyone in the Marketing department would seem to be equivalent to the query

```
SELECT ssn, name
FROM EMPLOYEE
WHERE salary > (SELECT max(salary)
                FROM EMPLOYEE
                WHERE dept = "MK");
```

but these queries may return different results when attribute `salary` in table `EMPLOYEE` contains nulls. The reason is that the comparison with ALL requires all comparisons between salaries return TRUE, but the result is unknown for nulls; this

causes the ALL predicate to fail. However, the aggregate `max` will happily compute a result while ignoring any nulls. As far as there are also some non-null values in `salary`, this result will be non-null and will be used by the comparison. Thus, the second query may or may not return something, depending on the data in table `EMPLOYEE`.

Subqueries in the WHERE clause can be *correlated*. Such subqueries mention an attribute that comes from the outer query that uses them. For instance, assume that we want to find all employees who make above their department's average salary; one way to write this is

```
SELECT name
FROM Employee E1
WHERE salary > (SELECT avg(salary)
                FROM Employee E2
                WHERE E2.dept = E1.dept);
```

This query is understood as follows: results from E1 (the copy of Employee used in the outer query). On each row, we return employee x's name if x's salary is greater than the average salary, calculated over the rows of E2 (another copy of Employee) that match x's department (i.e. the average salary in x's department). All types of WHERE clause subqueries can be correlated. In fact, it is very common for some of them (like EXISTS) to be used primarily in correlated contexts.

Thanks in part to the subquery predicates, there are several different ways to write most queries in SQL. The analyst has a choice as to how and when to use subqueries, as we can see with some simple examples.

Example: Moving Subqueries from FROM to WHERE

Example 3.1.2 of Sect. 3.1.2 is repeated here with a subquery in the WHERE clause, instead of in the FROM clause.

```
SELECT name
FROM chicago-employees
WHERE  salary > (SELECT avg(salary) as avgsal
                 FROM chicago-employees
                 WHERE salaried = True);
```

The system will first evaluate the subquery, obtaining a value for the average (mean) of across all values of table `Chicago-employees`. It will then use this value to evaluate the condition. Suppose, for instance, that the average found was 45.6; the condition in `WHERE` will be treated as `salary > 45.6`. Note that the same table is mentioned in both `FROM` clauses; this is not unusual at all. One can think of this as having two copies of the same table, being used independently to evaluate the subquery and the main query.

Most subqueries in SQL can be avoided. Aggregated, non-correlated subqueries can be used in the WHERE clause but they can also be put in the FROM clause and its result is used. For instance, the query above was written originally with a subquery in the FROM clause.

Aggregated and correlated subqueries can also be moved to the FROM clause, but we need to add a grouping to simulate the effects of the correlation. For instance, the query above can be rewritten as

```
SELECT name
FROM Employee, (SELECT dept, avg(salary) as avgsal
                FROM Employee
                GROUP BY dept)
WHERE salary > avgsal;
```

Queries with IN and EXIST can be turned into joins, whether they are correlated or not. For instance, the example with IN above can be also written as

```
SELECT name, salary
FROM EMPLOYEE, DEPARTMENT
WHERE ssn = manager-ssn;
```

Also, queries with NOT IN and NOT EXIST can be rewritten using EXCEPT, as will be explained when set predicates are introduced in Sect. 5.4.

Of note, queries with EXISTS and NOT EXISTS can be turned into aggregated subqueries, whether they are correlated or not. For instance, the example with EXISTS above can be written as

```
SELECT name, salary
FROM EMPLOYEE
WHERE dept = 'RE' and 0 < (SELECT count(*)
                           FROM DEPARTMENT
                           WHERE id = 'MK');
```

We change the SELECT clause in the subquery from '*' to `count(*)`, which returns the number of rows in the answer. If this number is greater than 0, then there is at least one row in the answer, so EXISTS is satisfied (using equality instead of $<$ will express NOT EXISTS).

In the end, there is no real need to use subqueries in the WHERE clause in SQL. They were introduced early in the standard and the desire to keep the standard backward compatible has resulted in subqueries still being there even though they are not strictly necessary.

5.3 Windows and Window Aggregates

Windows have been added to the SQL standard to give more flexibility than GROUP BY allows. A *window* is a set of rows from a table, specified by the user, on which certain calculations are performed. In a sense, they are similar to groups, since a

table can be partitioned into a set of windows. However, unlike groups, windows are not collapsed to a row; it is possible to operate *within* them, as the individual rows that make up a window can be accessed and manipulated as convenient.

Window functions can only appear in SELECT and ORDER BY clauses. To specify a window, we use a WINDOW clause, which has the syntax

```
WINDOW name AS (PARTITION-BY ...ORDER-BY ...FRAME ...)
```

These three components, which are all optional, are understood as follows:

- PARTITION BY takes a list of attributes as argument and is equivalent to GROUP BY: it creates the windows by putting together all rows of the input table that have the same values for the specified attributes.
- ORDER BY takes a list of attributes as arguments and is equivalent to an ORDER BY clause in that it sorts the tuples within a window. Each window is sorted separately, so there is a first row, second row, etc. on each window. The user can also specify where to put nulls (first or last).
- FRAME is an expression that specifies a subset of rows within each window to which an aggregate applies. That is, in each window, the aggregate takes as input only the rows denoted by the FRAME, not all of them. A common way to denote certain rows is to use the previously defined order. It is assumed that, as the system applies an aggregate over a window, it will scan it row-by-row, so that it will advance from the first row to the last ('first' and 'last' defined according to the order). As it does that, there is a 'current' row that is used as a reference. The frame specifies which rows around the current row are involved in computing the aggregate—that is, it defines a neighborhood of the current row. This is defined by ROWS (number of rows before/after current one) or RANGE (values in ordering attribute are in a range relative to current one); such rows can be PRECEDING or FOLLOWING the current row.

  A few examples will show the basic idea:[4] assume a table
  `Sales(storeid, productid, day, month, year, amount)`
  The query

```
SELECT storeid, month, amount, avg(amount) over w
FROM Sales
WHERE month BETWEEN 2015/09 and 2015/12
WINDOW w AS (PARTITION BY storeid, ORDER BY month,
             ROWS 2 PRECEDING);
```

computes the *moving average* over 3 months. Here the FRAME is

```
ROWS 2 PRECEDING
```

which means 'the current row and the two rows before it (in the order given, i.e. by month).' If the FRAME used were

```
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
```

[4]These examples come from the description of the SQL standard on SIGMOD Record.

this would also give a moving average, but this time using the past, present ('current'), and next month. Finally, the window

```
RANGE BETWEEN INTERVAL '1' month PRECEDING
              AND INTERVAL '1' month FOLLOWING)
```

we would get the same as in the previous case: the frame consists of the current row, the previous one (`'1' month PRECEDING`), and the next one (`'1' month FOLLOWING`). There is a subtle difference, though:

- ROWS is physical aggregation; if there are gaps or repetitions on the data, it will still pick the previous, current, and following rows, giving dubious results.
- RANGE is logical aggregation: it will skip gaps or repetitions and always find the values that precede and follow the current month. However, RANGE can only be used with one numerical grouping aggregate.

The keyword UNBOUNDED can be used in ROWS instead of a number; it means to use all rows up to the window boundary. For instance, the FRAME

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

would give a *cumulative* aggregate, as each aggregate would include all rows from the beginning (relative to ORDER) of the group until the current one, which progresses row-by-row from beginning to end (again, relative to ORDER).

Multiple windows can be defined in the same query, in the same WINDOW clause:

```
SELECT storeid, productid,
       sum(amount) OVER everything,
       sum(amount) OVER bystore,
       sum(amount) over byproduct
FROM sales
WINDOW everything AS (),
       bystore AS (PARTITION BY storeid),
       byproduct AS (PARTITION BY productid);
```

This query computes three sums: one over the whole table, one per store (partitioned by `storeid`), and one per product (partitioned by `productid`). Note that this can be written with just regular grouping but would require three SELECT statements.

Exercise 5.1 Write the equivalent to the query above using regular grouping. Hint: use three subqueries in WITH or FROM.

One window can also be defined in terms of another window:

```
SELECT storeid, month, sum(amount) over W2a,
                       avg(amount) over W2b
FROM sales
WHERE month BETWEEN 2001/09 and 2001/12
WINDOW w AS (PARTITION BY storeid, ORDER BY month),
```

```
w2a AS (w ROWS BETWEEN UNBOUNDED PRECEDING
                AND CURRENT ROW),
w2b AS (w ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING);
```

This query gives the cumulative sales per month and a centered average per month. Windows `w2a` and `w2b` are based on window `w`, so they 'inherit' their PARTITION and ORDER from `w`.

Exercise 5.2 Write a query over `sales` that computes the moving average sale amount over 5 months per customer.

The use of windows on analytics relies on the fact that it provides us with this fine grained control over how to compute aggregates (so we can get *cumulative* and *moving* aggregates) but also on the fact that they can be used as part of a query where arbitrary attributes (whether mentioned in the window or not) can be retrieved. This makes it more flexible then GROUP BY. For instance, our first example of this section retrieved both the individual sale amount and a running average computed for each store. This means that the result will display, for each store, as many rows as sales there were in that store, with the individual amount of each, and the computed aggregate added to each row. To do this with GROUP BY we need to use a subquery, due to the restrictions on GROUP BY syntax.

We now show how window aggregates can be used to accomplish some of the tasks already explained in the book, but expressed in a more concise (and sometimes more efficient) manner. As a simple example, in Sect. 3.2 we calculated cumulative totals as follows:

```
SELECT V2.Value, Sum(V.Value) as Cumulative
FROM Values as V2, Values as V
WHERE V.Order <= V2.Order
GROUP BY V2.Value;
```

With windows, this can be accomplished without a Cartesian product, which is more efficient:

```
SELECT V.Value,
       sum(Value) OVER(PARTITION BY Order
                       ORDER BY Order
                       ROWS BETWEEN UNBOUNDED PRECEDING
                            AND CURRENT ROW) AS Cumulative
FROM Values as V
GROUP BY V.Value;
```

Rankings come in handy for computing ranking-based correlation measures like Spearman's or Kendall's (see Sect. 3.2.2). The window function `RANK()` creates an explicit attribute with the ranking of a row based on the ordering specified in the ORDER BY:

```
SELECT *,
       rank() OVER (PARTITION BY store-id
                    ORDER BY amount DESC)  AS position;
```

will return a table with an attribute called `position` with values 1, 2, ... on each group created by `store-id`, with the value in `position` based on the `amount` attribute: the largest amount is 1, the second largest is 2, ... (note that we are sorting in descending order). The ranking is restarted at 1 for each group (each store).

Using this, we can simplify the computation of `top k` results, which we did earlier using ORDER BY and LIMIT:

```
SELECT *,
       rank() OVER (PARTITION BY store-id
                    ORDER BY amount DESC)  AS position
WHERE position < 11;
```

will retrieve, for each store, the top 10 sales by amount (i.e. the 10 largest sales). However, it must be noted that this query may return more than the intended 10 rows in case of ties. To deal with this, SQL distinguishes between RANK() and DENSE_RANK(), which does not skip ranks. As a simple example, if in some stores the top 4 sales are 1000, 900, 900, and 800, RANK() will produce 1, 2, 2, 4 and DENSE_RANK() will produce 1, 2, 2, 3.

Explicit ranking could be used for calculating percentiles. However, many systems (including Postgres) have 'ordered aggregates' that do this directly:

`percentile_disc(fraction) WITHIN GROUP (ORDER BY column(s))`

This is the *discrete percentile* function; it returns the first input value whose position in the ordering equals or exceeds the specified fraction (`fraction` must be a value between 0 and 1.0). There is also

`percentile_cont(fraction) WITHIN GROUP (ORDER BY column(s))`

This is the *continuous percentile* function; it returns the value corresponding to the specified fraction in the ordering. Note that `percentile_disc` returns a value from the dataset, while `percentile_cont` may create a value by interpolation if needed. Hence, we can use

```
SELECT percentile_cont(0.5) within group (order by A)
FROM Dataset;
```

to get the median value of column `A`. We can also use this as an aggregate function, in a query with GROUP BY,[5] to find medians within groups:

```
SELECT storeid,
       percentile_cont(0.5) WITHIN GROUP (ORDER BY amount)
                            AS StoreMedian
FROM sales
GROUP BY storeid;
```

Exercise 5.3 Write a query over `sales` that computes both the mean and the median sale amount per customer.

[5]The percentile aggregates are not windows aggregates, so they cannot be used with OVER.

A small aside: even though computing the mode is not hard, Postgres also has a window aggregate for that:

`mode() WITHIN GROUP (ORDER BY attrib)`

will compute the mode of attribute `attrib`.

Checking outliers with MAD is doable in Postgres by calculating the median with `percentile_cont`. Assume that in the people dataset we have a `height` column that contains some suspicious values. We can use

```
SELECT percentile_cont(0.5) WITHIN GROUP
                           (ORDER BY height - median)
FROM Dataset,
     (SELECT percentile_cont(0.5) WITHIN GROUP
                                 (ORDER BY height)
             as median
      FROM Dataset);
```

to get the MAD value and decide if those values are outliers or errors.

To find quartiles with this approach, we could use

```
SELECT storeid,
      percentile_cont(0.25) WITHIN GROUP (ORDER BY amount)
  AS quart1,
      percentile_cont(0.5) WITHIN GROUP (ORDER BY amount)
  AS quart2,
      percentile_cont(0.75) WITHIN GROUP (ORDER BY amount)
  AS quart3,
      percentile_cont(1.0) WITHIN GROUP (ORDER BY amount)
  AS quart4
FROM sales
GROUP BY storeid;
```

It is possible to also use the `ntile(`n`)` function, which sorts the values and breaks them into n buckets. To get quartiles with this approach, it is enough to use

```
SELECT DISTINCT storeid,
       ntile(4) OVER (PARTITION BY storeid ORDER BY amount)
                        as quartile
FROM sales;
```

The function creates an attribute called `quartile` with values 1, 2, 3, 4. A value i is assigned to each tuple, depending on whether the amount is on the i-th quartile, within each store group. Once this is done, the result can be used to get an aggregate within each quartile; for instance, the largest/last value would be close to the result from using `percentile_disc` as above, with values at each .25 of the total.

Exercise 5.4 Calculate deciles using `percentile_disc` or `percentile_cont`.

Exercise 5.5 Calculate deciles using `ntile`.

Another example of the usefulness of this additional aggregates is the computation of the *k (k%) trimmed mean*. Recall that this is the mean calculated excluding

the top k (or $k\%$ and the bottom k (or $k\%$) of all values, in order to defuse the influence of outliers. Support, for instance, that we want to exclude the top and bottom 5%; this can be achieved with

```
SELECT avg(attr)
FROM Data,
  (SELECT percentile_disc(.05) WITHIN GROUP (ORDER BY attr)
          as mincut,
          percentile_disc(.95) WITHIN GROUP (ORDER BY attrb)
          as maxcut
   FROM Data) as cuts
WHERE attr > mincut and attr < maxcut;
```

If we want to exclude the top (and bottom) k values, we can use `rank()` to sort them and then use the new attribute created by `rank()` to exclude the appropriate number of values. Note that excluding the top k is easy with descending order, but the bottom k requires some additional computation to determine which values to exclude (of course, the situation using ascending order is symmetrical).

Exercise 5.6 Write the query to calculate the top 10 trimmed mean on the generic `Data` table.

Windows aggregates can be used to compute distances more efficiently too. We saw in Sect. 4.3.1 that computing distances between any two points in a dataset required an expensive Cartesian product, but we can avoid that in some cases. Assume in table `Data` there is a numerical attribute `A` and we want to compute the distances between any two points using their differences in value of `A`; instead of writing

```
SELECT *, abs(R.A - S.A)
FROM Data as R, Data as S;
```

we can use window aggregate `lag`, which computes the difference between a row and its preceding one (where 'preceding' is determined, as usual in windows aggregates, by some ordering). This is accomplished with

```
SELECT *, lag(A) OVER (PARTITION BY id, ORDER BY id, A)
          as previous,
          A - lag(A) OVER (PARTITION BY id, ORDER BY id, A)
          as diff
FROM Data;
```

The attribute `diff` can be used as a distance based on attribute `A`. This can be extended to distances based on several attributes by computing (and combining) separate lags for each attribute. Of course, distances more sophisticated than `abs(R.A - S.A)` may require additional computation.

Exercise 5.7 Repeat the clustering algorithm based on random pivots of Sect. 4.3.1 using `lag` to calculate distances.

Exercise 5.8 Repeat the clustering algorithms based on thresholds of Sect. 4.3.1 using `lag` to calculate distances.

Finally, we introduce a couple of extensions of regular GROUP BY, `ROLLUP`, and `CUBE`. Even though not really window functions, these may come in handy in some cases. A query using

`GROUP BY ROLLUP(`$attribute_1$`,...,`$attribute_n$`)`

is equivalent to a query using all these groupings:

- no grouping at all;
- a grouping with $attribute_1$;
- a grouping with $attribute_1$, $attribute_2$;
- ...
- a grouping with $attribute_1$, $attribute_2$, ..., $attribute_{n-1}$; and
- a grouping with $attribute_1$, ..., $attribute_n$.

That is, one grouping per *prefix* of $attribute_1$, ..., $attribute_n$.

Conversely, a query using

`GROUP BY CUBE(`$attribute_1$`,...,`$attribute_n$`)`

is equivalent to a query using all groupings that can be done with any subset of $attribute_1$, ..., $attribute_n$. A simple example will illustrate their usage.

Example: Rollup and Cube

Recall table `SALES(storeid, productid, time, day, month year, amount)`.

- the query

```
SELECT year, month, day, sum(amount) as total
FROM Sales
GROUP BY ROLLUP(year, month, day);
```

  will calculate aggregates over the following groups : `(year, month, day)` (sums per day), `(year, month)` (sums per month), `(year)` (sums per year), the sum of the whole table. The result table will look as follows:

Year	Month	Day	Total
Some year	Some month	Some day	Total for year, month, day
...			
Some year	Some month	Null	Total for year and month
...			
Some year	Null	Null	Total for year
...			
Null	Null	Null	Total for whole table

  Since each one of those is a different grouping, this query is equivalent to 4 queries with regular GROUP BY (in general, a query with roll-up on n attributes is equivalent to $n + 1$ queries with regular GROUP BY).

- the query

```
SELECT year, month, day, sum(amount)
FROM Sales
GROUP BY CUBE(year, month, day);
```

will form the following groups: `(year, month, day)`, `(year, month)`, `(year, day)`, `(year)`, `(month, day)`, `(month)`, `(day)`, `()`. Again, each one of those requires a separate group by, so the query with CUBE is equivalent to 8 queries with regular grouping(in general, a query with CUBE on n attributes is equivalent to 2^n queries with regular GROUP BY). The result table will look as follows:

Year	Month	Day	Total
Some year	Some month	Some day	Total for year, month, day
		...	
Some year	Some month	Null	Total for year and month
		...	
Some year	Null	Null	Total for year
		...	
Null	Null	Null	Total for whole table
Some year	Null	Some day	Total for year and day
		...	
Null	Some month	Some day	Total for month and day
		...	
Null	Some month	null	Total per month
		...	
Null	Null	Day	Total per day

While queries with ROLLUP and CUBE can be written without them, they clearly make calculations simpler to write (and, in most cases, much more efficient to compute).

Exercise 5.9 Simulate the rollup query in the example without using ROLLUP, with simple GROUP BY. Hint: use the union of several queries (see next section), one for each specific grouping.

Exercise 5.10 Simulate the cube query in the example without using CUBE, with simple GROUP BY. Hint: use the union of several queries (see next section), one for each specific grouping.

Exercise 5.11 Assume the spreadsheet

State/Year	2000	2001	...	
AL	500	350	...	AL-total
AK	500	350	...	AL-total
...	...	...	...	...
	2000-total	2001-total	...	Total

We know that this would be represented as a table with schema `(state,year, total)` and that all values with 'total' in the name are not raw data—so they would not be part of the table. Calculating those *margin* totals is not complicated, but it used to require 3 separate queries (one for the row totals, one for the column totals, one for the whole total). Write a single query that will compute all totals at once with one of our new found friends.

5.4 Set Operations

SQL allows *set operations*; in particular, taking the union, intersection, and difference of two tables is allowed. This is expressed in SQL by using two queries and combining them with the keywords UNION, INTERSECT, EXCEPT. For instance, to take the union of two queries we write

```
SELECT ... FROM ... WHERE ...
UNION
SELECT ... FROM ... WHERE ...
```

The system will run the first (topmost) query, as well as the second (bottom) one; it will then combine the rows from both answers into a single table. If we use INTERSECT instead of UNION, the system computes the intersection of both answers (i.e. the rows that are present in both answers). If we use EXCEPT instead, we get all the rows of the first (topmost) answer that are *not* present in the second (bottom) answer.

However, the system requires that the tables to be combined are *schema compatible*. Two tables are schema compatible iff they have the same number and type of attributes. For instance, if the answer to a table includes three attributes, one of them being an integer, the second a string, and the third a date, a schema compatible table will also have three attributes, and the first one will be an integer, the second a string, and the third one a date. This ensures that the tables can be meaningfully combined into one. Note that this refers to the tables that are input to the UNION (INTERSECT, EXCEPT) operators, that is, to what is used in the SELECT clause of the queries, not to the tables in the FROM clause of the queries.

Example: UNION Operation

Assume we have several tables named

`PsychologyRank(school-name, state, type, ranking-position)`

for schools ranked according to their Psychology programs; another table

`EconomyRank(school-name, state, type, ranking-position)`

where the schools are ranked according to their Economy programs; another table

`HistoryRank(school-name, state, type, ranking-position)`

and so on.

```
SELECT 'Psychology' as program, *
FROM PsychologyRank
UNION
SELECT 'Economy' as program, *
FROM EconomyRank
UNION
SELECT 'History' as program, *
FROM HistoryRank;
```

Note that we have applied two UNIONs to three tables; as their set counterparts, the operations are binary (take two arguments) but yield an object of the kind (table or set) and so can be used several times. Moreover, the union is associative and symmetric, so the order of tables does not matter (intersection is also associative and symmetric, but the difference is neither). Note also that, if the schemas of the tables being analyzed are not exactly the same, or we do not want all the data, we can use the SELECT clause to pick the common (or desired) attributes of similar tables.

Exercise 5.12 Assume we have data about New York City real estate sales, and we have 5 different datasets, one for each one of the 5 boroughs that are part of the city: Manhattan, Brooklyn, Queens, The Bronx, and Staten Island. Each dataset is in a table with the name of the borough and a similar schema: `(address, type, date-sold, amount-sold)`. Put all data together in a single table with schema `(borough,address,date-sold,amount-sold)`.

Example: INTERSECT and EXCEPT Operations

Assume the same tables as in the previous example and suppose that this time we just want to know if there are schools (names) that are tops on both Psychology and Economy. The following query will take care of this:

```
SELECT name
FROM PsychologyRank
INTERSECT
SELECT name
FROM EconomyRank;
```

If we want schools that are top ranked in Psychology but not in Economy, we use this instead:

```
SELECT name
FROM PsychologyRank
EXCEPT
SELECT name
FROM EconomyRank;
```

Note how this time we focused on attribute `name`; this made the results schema compatible while giving us all the information required.

As stated earlier in Sect. 5.2, several complex predicates express negation, including NOT IN and NOT EXISTS. These conditions can also be written using EXCEPT instead. The idea is to write a query `Q1 EXCEPT Q2`, where `Q2` is the subquery of the NOT IN (or NOT EXISTS) condition, and `Q1` is the main query. Thus,

```
SELECT attrA
FROM Table1
WHERE attrB NOT IN (SELECT attrC
                    FROM Table2);
```

becomes

```
SELECT attrA
FROM (SELECT attrB
      FROM Table1
      EXCEPT
      SELECT attrC
      FROM Table2) as Temp, Table1
WHERE Temp.attrB = Table1.attrB;
```

When `attrA = attrB`, the transformation is even simpler, as the next example shows.

Example: NOT IN Transformed into EXCEPT

Assume we are running several trial experiments over a population, and table `Participation(subject-id, trial-id, date)` which keeps track of which subjects participate in which trials. We want to get all subject that participated in trial 'ACK' but not in trial 'PYL.' We can write this using NOT IN as

```
SELECT subject-id
FROM Participation
WHERE trial-id = 'ACK' and
      subject-id NOT IN (SELECT subject-id
                         FROM Participation
                         WHERE trial-id = 'PYL');
```

but we can also write it with EXCEPT as

```
SELECT subject-id
FROM Participation
WHERE trial-id = 'ACK'
EXCEPT
SELECT subject-id
FROM Participation
WHERE trial-id = 'PYL';
```

5.5 Expressing Domain Knowledge

Domain knowledge (sometimes called *subject matter knowledge*) refers to the facts and constraints that we have about whatever real-world domain the data refers to. Such knowledge usually allows us to predict in advance what kinds of attributes to expect and, for each attribute, what values are normal or common. For instance, in a medical database of patients, we can expect attributes expressing measurements of medically significant factors like blood pressure, cholesterol levels, etc. A *subject matter expert* (in this case, an MD) could tell us what are typical and atypical values for many such attributes. This is very useful for EDA, since it makes it much easier to determine if there are errors in our data, or if we have outliers or missing values.

Another way in which this domain knowledge can be used is by telling the system to check attribute values for us. For instance, for closed domain values, we can tell that only certain values are acceptable. In this case, other values should be rejected. For this situation we have the CHECK statement. This statement can be used when creating a table or can be added later to the table definition using ALTER TABLE. It has the syntax

`CHECK condition`

where `condition` is a condition similar to those of a WHERE clause. In most systems, referencing other tables is not allowed; the condition is restricted to the table that contains the CHECK. Even with this limitation, CHECKS can be extremely useful to find 'bad' data.

Example: CHECK Statement

Assume, for the New York flights database, that we know (since this is a dataset of flights *from* New York) that the only valid values for attribute `origin` are 'EWR,' 'LGA,' and 'JFK.' We also know that the arrival time should be later than the departure time, attribute `month` should be a number between 1 and 12, and `day` should be a number between 1 and 31.[6] In general, constraints that involve only

[6]Note that this is still not enough to catch bad dates; transforming these values into dates (see Sect. 3.3.1.3) is the right thing to do.

one attribute value, or several attribute values in the same row, are expressible with CHECK statements.

```
CREATE TABLE NY-FLIGHTS(
flightid int,
year int,
month int CHECK (month BETWEEN 1 and 12),
day int CHECK (day BETWEEN 1 and 31),
dep_time int,
sched_dep_time int,
dep_delay int,
arr_time int,
sched_arr_time int,
arr_delay int,
carrier char(2),
flight char(4),
tailnum char(6),
origin char(3) CHECK (origin IN ('EWR', 'LGA', 'JFK')),
....
CHECK (sched_dep_time < sched_arr_time));
```

Note that the check on departure and arrival time is added at the end of the attribute list, while other checks that involve a single attribute are added after the attribute definition. Note also that we have used < to express 'earlier than' because both the departure and arrival time are expressed as integers.

Exercise 5.13 In the `NY-Flights` data, there are several constraints that should hold of the data in each flight (each record):

- `arr_time` should also always be later than `dep_time` and `sched_dep_time`.
- `dep_delay` should be the difference between `dep_time` and `sched_dep_time`.
- `arr_delay` should be the difference between `arr_time` and `sched_arr_time`.

Write a more complete `CREATE TABLE` statement that enforces all these constraints using CHECKs.

In most systems, there exists the option of *disabling* checking when loading data. The main advantage of this is speed: if there are CHECK statements in a table, the system tests them every time there is an insertion into the table; when loading data from a file, this means one check per row (see Sect. 2.4.1). This slows down the loading; that is why many systems provide a way to circumvent the checking. The price to pay for this is that we may upload bad data into our database and we will have to search for it manually: if we enable the CHECKS back once the data is loaded, the system will not test the data already in the table. Thus, unless we are loading a very large amount of data and have limited time, it is a good idea to let the system check the data for us as it loads the table. We will still have to deal with the errors manually, but at least we know exactly which data causes problems.