3.7 Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five standard built-in aggregate functions:[9]

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

3.7.1 Basic Aggregation

Consider the query "Find the average salary of instructors in the Computer Science department." We write this query as follows:

```
select avg (salary)
from instructor
where dept_name = 'Comp. Sci.';
```

The result of this query is a relation with a single attribute containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an awkward name to the result relation attribute that is generated by aggregation, consisting of the text of the expression; however, we can give a meaningful name to the attribute by using the **as** clause as follows:

```
select avg (salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.';
```

In the *instructor* relation of Figure 2.1, the salaries in the Computer Science department are \$75,000, \$65,000, and \$92,000. The average salary is \$232,000/3 = \$77,333.33.

Retaining duplicates is important in computing an average. Suppose the Computer Science department adds a fourth instructor whose salary happens to be \$75,000. If du-

[9] Most implementations of SQL offer a number of additional aggregate functions.

plicates were eliminated, we would obtain the wrong answer ($232,000/4 = $58,000) rather than the correct answer of $76,750.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2018 semester." In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation *teaches*, and we write this query as follows:

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

Because of the keyword **distinct** preceding *ID*, even if an instructor teaches more than one course, she is counted only once in the result.

We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count** (*). Thus, to find the number of tuples in the *course* relation, we write

```
select count (*)
from course;
```

SQL does not allow the use of **distinct** with **count** (*). It is legal to use **distinct** with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct** to specify duplicate retention, but since **all** is the default, there is no need to do so.

3.7.2 Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;
```

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 3.13 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

Figure 3.13 shows the tuples in the *instructor* relation grouped by the *dept_name* attribute, which is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 3.14.

In contrast, consider the query "Find the average salary of all instructors." We write this query as follows:

select avg (*salary*)
from *instructor*;

In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.14 The result relation for the query "Find the average salary in each department".

As another example of aggregation on groups of tuples, consider the query "Find the number of instructors in each department who teach a course in the Spring 2018 semester." Information about which instructors teach which course sections in which semester is available in the *teaches* relation. However, this information has to be joined with information from the *instructor* relation to get the department name of each instructor. Thus, we write this query as follows:

```
select dept_name, count (distinct ID) as instr_count
from instructor, teaches
where instructor.ID= teaches.ID and
      semester = 'Spring' and year = 2018
group by dept_name;
```

The result is shown in Figure 3.15.

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause. In other words, any attribute that is not present in the **group by** clause may appear in the **select** clause only as an argument to an aggregate function, otherwise the query is treated as erroneous. For example, the following query is erroneous since *ID* does not appear in the **group by** clause, and yet it appears in the **select** clause without being aggregated:

```
/* erroneous query */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

In the preceding query, each instructor in a particular group (defined by *dept_name*) can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output. As a result, such cases are disallowed by SQL.

The preceding query also illustrates a comment written in SQL by enclosing text in "/* */"; the same comment could have also been written as "-- erroneous query".

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.15 The result relation for the query "Find the number of instructors in each department who teach a course in the Spring 2018 semester."

dept_name	*avg_salary*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.16 The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

3.7.3 The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used in the **having** clause. We express this query in SQL as follows:

select *dept_name*, **avg** (*salary*) **as** *avg_salary*
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

The result is shown in Figure 3.16.

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "For each course section offered in 2017, find the average total credits (*tot_cred*) of all students enrolled in the section, if the section has at least 2 students."

select *course_id*, *semester*, *year*, *sec_id*, **avg** (*tot_cred*)
from *student*, *takes*
where *student.ID*= *takes.ID* **and** *year* = 2017
group by *course_id*, *semester*, *year*, *sec_id*
having count (*ID*) >= 2;

Note that all the required information for the preceding query is available from the relations *takes* and *student*, and that although the query pertains to sections, a join with *section* is not needed.

3.7.4 Aggregation with Null and Boolean Values

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the *instructor* relation have a null value for *salary*. Consider the following query to total all salary amounts:

select sum (*salary*)
from *instructor*;

The values to be summed in the preceding query include null values, since we assumed that some tuples have a null value for *salary*. Rather than say that the overall sum is itself *null*, the SQL standard says that the **sum** operator should ignore *null* values in its input.

In general, aggregate functions treat nulls according to the following rule: All aggregate functions except **count (*)** ignore null values in their input collection. As a result of null values being ignored, the collection of values may be empty. The **count** of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection. The effect of null values on some of the more complicated SQL constructs can be subtle.

A **Boolean** data type that can take values **true**, **false**, and **unknown** was introduced in SQL:1999. The aggregate functions **some** and **every** can be applied on a collection of Boolean values, and compute the disjunction (**or**) and conjunction (**and**), respectively, of the values.

Note 3.2 SQL AND MULTISET RELATIONAL ALGEBRA - PART 2

As we saw earlier in Note 3.1 on page 80, the SQL **select**, **from**, and **where** clauses can be represented in the multiset relational algebra, using the multiset versions of the select, project, and Cartesian product operations.

The relational algebra union, intersection, and set difference ($\cup, \cap$, and $-$) operations can also be extended to the multiset relational algebra in a similar way, following the corresponding definitions of **union all**, **intersect all**, and **except all** in SQL, which we saw in Section 3.5; the SQL **union**, **intersect**, and **except** correspond to the set version of $\cup, \cap$, and $-$.

The extended relational algebra aggregate operation γ permits the use of aggregate functions on relation attributes. (The symbol $\mathcal{G}$ is also used to represent the aggregate operation and was used in earlier editions of the book.) The operation ${}_{dept\_name}\gamma_{\textbf{average}(salary)}(instructor)$ groups the *instructor* relation on the *dept_name* attribute and computes the average salary for each group, as we saw earlier in Section 3.7.2. The subscript on the left side may be omitted, resulting in the entire input relation being in a single group. Thus, $\gamma_{\textbf{average}(salary)}(instructor)$ computes the average salary of all instructors. The aggregated values do not have an attribute name; they can be given a name either by using the rename operator ρ or for convenience using the following syntax:

$$ {}_{dept\_name}\gamma_{\textbf{average}(salary)\ \textbf{as}\ \text{avg\_salary}}(instructor) $$

More complex SQL queries can also be rewritten in relational algebra. For example, the query:

select A_1, A_2, **sum**(A_3)
from r_1, r_2, ..., r_m
where P
group by A_1, A_2 **having count**(A_4) > 2

is equivalent to:

$$ t1 \leftarrow \sigma_P(r_1 \times r_2 \times \cdots \times r_m) $$
$$ \Pi_{A_1, A_2, SumA3}(\sigma_{countA4 > 2}({}_{A_1, A_2}\gamma_{\text{sum}(A_3)\ \textbf{as}\ SumA3,\ \textbf{count}(A_4)\ \textbf{as}\ countA4}(t1))) $$

Join expressions in the **from** clause can be written using equivalent join expressions in relational algebra; we leave the details as an exercise for the reader. However, subqueries in the **where** or **select** clause cannot be rewritten into relational algebra in such a straightforward manner, since there is no relational algebra operation equivalent to the subquery construct. Extensions of relational algebra have been proposed for this task, but they are beyond the scope of this book.