

Prof. Hasso Plattner



A Course in In-Memory Data Management

The Inner Mechanics
of In-Memory Databases

September 4, 2015

This learning material is part of the reading material for Prof. Plattner's online lecture "In-Memory Data Management" taking place at www.openHPI.de. If you have any questions or remarks regarding the online lecture or the reading material, please give us a note at openhpi-imdb@hpi.uni-potsdam.de. We are glad to further improve the material.

Research Group "Enterprise Platform and Integration Concepts",
<http://epic.hpi.uni-potsdam.de>

Aggregate Functions

This chapter discusses aggregate functions. It outlines what aggregate functions are, how they work, and how they can be executed in an in-memory database system.

Aggregate functions are specific functions that take multiple rows as an input to create an output. This means, they work on data sets instead of single values. Grouping the input data based on specified group attributes creates the sets. Basic aggregate functions are, e.g., *COUNT*, *SUM*, *AVERAGE*, *MEDIAN*, *MAXIMUM* and *MINIMUM*. Furthermore, it is possible to create additional functions for special purposes, e.g., OLAP functions that extend basic functions. The basic SQL syntax to use an aggregate function can be seen in Listing 20.1.

Listing 20.1: SQL Aggregate Function Syntax

```
SELECT AggregateFunction(attribute1), attribute2 , attribute3
FROM table_name
WHERE attribute2 = some_value
GROUP BY attribute2 , attribute3
HAVING AggregateFunction(attribute1) > 5;
```

The *GROUP BY* clause specifies the attributes by which the input relation is grouped. All selected attributes that are not part of the *GROUP BY* clause should specify an aggregate function in the select clause, otherwise their value might be undefined. The *WHERE* and the *HAVING* clauses are optional.

20.1 Aggregation Example Using the COUNT Function

Let us consider an example for the use of the *COUNT* aggregate function. Assume an input table containing the complete world population as shown in Figure 20.1.

recID	fname	lname	gender	city	country	birthday
0	John	Smith	m	Chicago	USA	12.03.1964
1	Mary	Brown	f	London	UK	12.05.1964
2	Jane	Doe	f	Palo Alto	USA	23.04.1976
3	John	Doe	m	Palo Alto	USA	17.06.1952
4	Peter	Schmidt	m	Potsdam	GER	11.11.1975
...	...	...	...	...	...	...

Fig. 20.1: Input Relation Containing the World Population

The goal is to count all citizens per country. Using the *COUNT* aggregate function, an SQL query to achieve this is depicted in Listing 20.2:

Listing 20.2: Example SQL Query Using the COUNT Aggregate Function.

```
SELECT country, COUNT(*) AS citizens
FROM world_population
GROUP BY country;
```

Figure 20.2 shows how such a query would be processed. First, the system runs through the attribute vector for the *country* column. For each new encountered country valueID, an entry with initial value “1” is added to the result map. If the encountered valueID has been added before, the respective entry in the result map is increased by one. That way, a result map is created which contains the valueIDs of each country and its number of occurrence. Second, the actual country names are fetched from the country dictionary using the valueIDs and the final result of the *COUNT* function is created. The result contains pairs of country names and the countries’ number of occurrences in the source table, which corresponds to the number of citizens.

20.2 Other Aggregate Functions

Other aggregate functions require a similar calculation function. For *SUM*, the number of occurrences for each valueID is counted in an auxiliary data

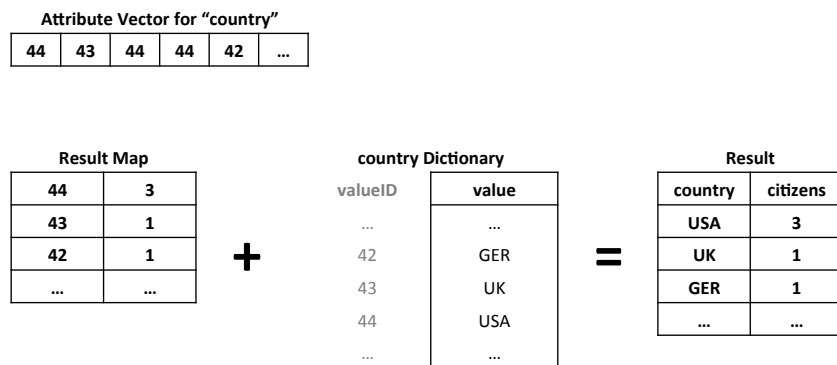


Fig. 20.2: Count Example

structure and the sum is calculated in a final step by summing up the number of occurrences multiplied with the respective value of each valueID. *AVERAGE* can be calculated by dividing *SUM* by *COUNT*. To retrieve the median, the complete relation has to be sorted and the middle value is returned. *MAXIMUM* and *MINIMUM* compare a temporary extreme value with the next value from the relation and replace it if the new value is higher (for *MAXIMUM*) or lower (for *MINIMUM*), respectively.

