Chapter 16
Materialization Strategies

SQL is the most common language to interact with databases. Users are accustomed to the table-oriented output format of SQL. To provide the same data interfaces as known from row stores in column stores, the returned results have to be transformed into tuples in row format. The process of transforming encoded columnar data into row-oriented tuples is called materialization.

Especially for column-oriented databases with lightweight compression, an appropriate materialization strategy is essential. Abadi et al. [AMDM07] analyzed different materialization strategies for column-oriented databases. Depending on the storage technique (e.g. compressed vs. uncompressed data, dictionary encoding vs. no dictionary encoding), different materialization strategies can be superior. Grund et al. [GKK+11] analyzed database operators and the impact of materialization strategies for intermediate results, in particular for dictionary-encoded columnar data structures.

16.1 Aspects of Materialization

Abadi et al. [AMDM07] divide the topic of materialization into two aspects, the execution of materialization and the time of materialization. The execution can be divided into parallel and pipelined materialization. The advantages and disadvantages of both approaches are discussed in detail in [GKK+11] and are not part of this learning material. All the following examples use a non-pipelined execution, where each operator is independent from the others.

There are two different strategies concerning the time aspect of materialization: early and late materialization. Early materialization describes the strategy, where data is decoded early (using dictionary lookups) during the query execution. For example, consider a dictionary-encoded string column. It contains the attribute vector of integer values and the sorted dictionary of strings. Here, the actual string replaces the positional integer value representing the corresponding dictionary position early. Hence, a row-oriented tuple representation is created early on.

H. Plattner, *A Course in In-Memory Data Management*,
DOI: 10.1007/978-3-642-36524-9_16, © Springer-Verlag Berlin Heidelberg 2013

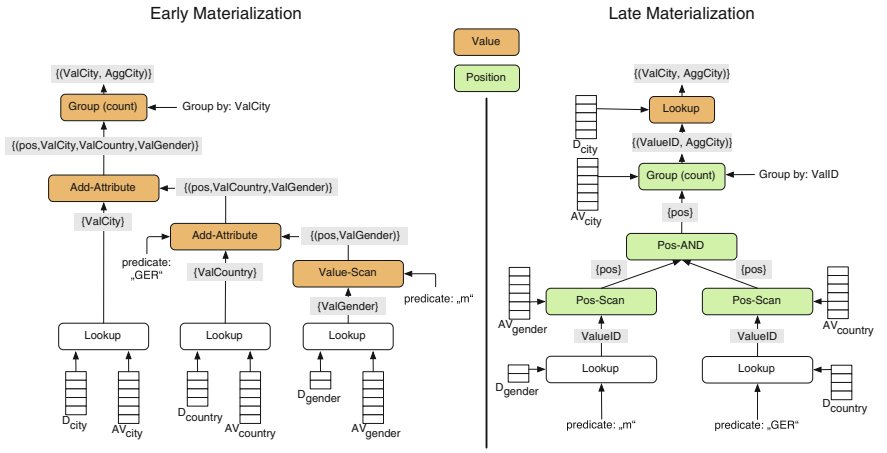

Fig. 16.1 Example comparison between early and late materialization

With the late materialization strategy, column-orientation and the positional information instead of the actual value are used as long as possible during query execution. Ideally, the row-oriented tuple will be materialized in the very last step before returning the result to the user.

Figure 16.1 shows in an example where actual values and positions are used in early and late materialization.

In many cases, late materialization can improve the performance for column stores, especially when light-weight compression techniques are used [AMDM07]. The following sections will discuss both strategies based on an example query.

16.2 Example

To discuss the difference between early and late materialization, we will examine the query "List the number of male inhabitants per city in Germany", see SQL query in Listing 16.1.

```
SELECT city, COUNT(*)
FROM world_population
WHERE gender = "m"
    AND country = "GER"
GROUP BY city
```

Listing 16.1: Example query

In both following examples, one strategy will be used throughout the whole query execution for exemplary purposes, even though a combination is often advantageous in real world situations. Example data of the *World Population Table* which is used in the query is shown in Fig. 16.2.

Table "world_population"

fname	lname	gender	country	city	birthday
Martin	Albrecht	m	GER	Berlin	08-05-1955
Michael	Berg	m	GER	Berlin	03-05-1970
Hanna	Schulze	f	GER	Bonn	04-04-1968
Ulrich	Schulze	m	GER	Bonn	10-20-1992
...	...	...	...	...	...

Dictionary encoded attribute vectors

53946	10435	0	68	357	15556
54368	25063	0	68	357	20882
30145	99645	1	68	443	20182
99312	99645	0	68	443	29147
fname	lname	gender	country	city	birthday

Fig. 16.2 Example data of table "*world_population*"

16.3 Early Materialization

When early materialization is used as the materialization strategy throughout the complete query, all required columns are materialized first. In our case, required columns are all columns that are used as predicates in the query (i.e., *country* and *gender*), as well as all columns that are part of the result (i.e., *city*). Dictionary lookups are performed for each of these columns using the valueIDs in the corresponding attribute vectors. For the gender column, the result of these lookups is the vector `{ValGender}` with the actual values (see Fig. 16.3a).

The next step is to scan the intermediate vector `{ValGender}` for the gender predicate 'm'. To all qualifying lines the corresponding position is added and copied to the intermediate vector `{(pos, ValGender)}` (see Fig. 16.3b).

In the next step, the columns are combined as shown in Fig. 16.4. Hereby, the `{ValCountry}` vector is added to the intermediate result `{(pos, ValGender)}` while scanning for the predicate value 'GER'.

The final step is to aggregate and return the requested data of the SQL query. For that the intermediate result `{(pos, ValGender' ValCountry' ValCity)}` is grouped by ValCity and aggregated. The result is `{(ValCity' AggCity)}`, as shown in Fig. 16.5.

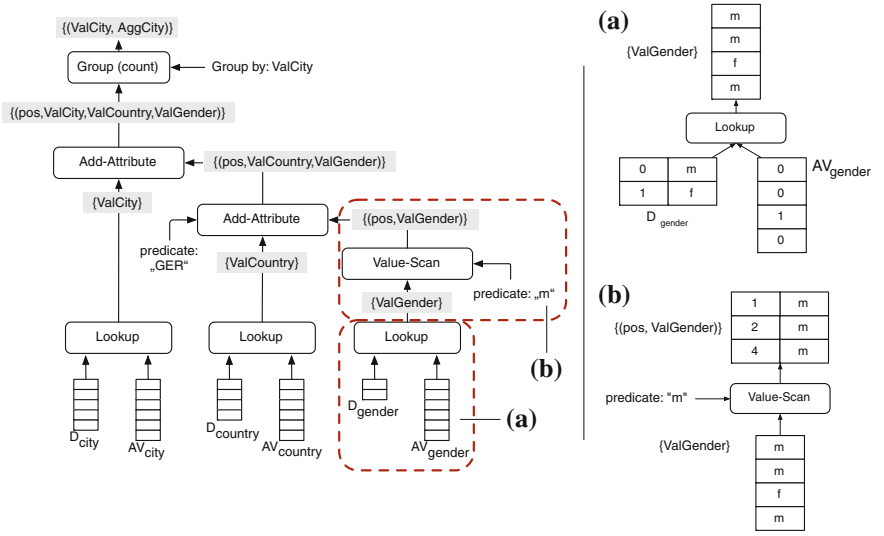

Fig. 16.3 Early materialization: materializing column via dictionary lookups and scanning for predicate

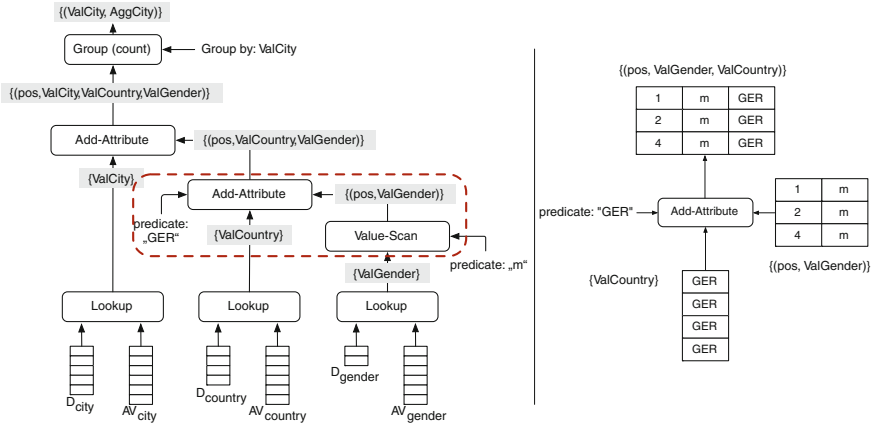

Fig. 16.4 Early materialization: scan for constraint and addition to intermediate result

16.4 Late Materialization

Instead of materializing the values of the dictionary lookup early (as done in the early materialization strategy), the dictionary-encoded value (`valueID`) contained in the attribute vector is being used. Ideally, the lookup into the dictionary for materialization is performed in the very last step before returning the result.

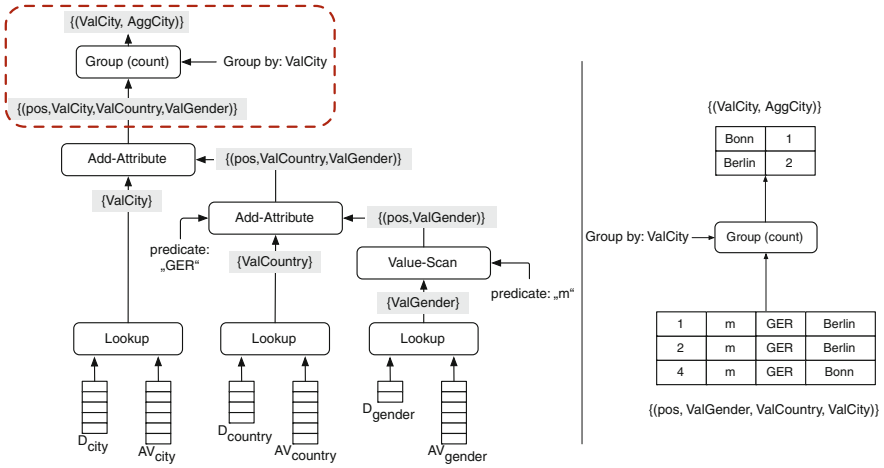

Fig. 16.5 Early materialization: group by *ValCity* and aggregation

Figure 16.6 shows the first step. Here, the predicates *gender* = '*m*' and *country* = '*GER*' are used for the lookup using the corresponding dictionaries. The outcome is a vector of dictionary positions (valueIDs) per column that qualify for the given predicates. Notice that the dictionary for the column *city* is not accessed, since it is not required for the actual processing of the query right now. Only the `valueID` of the columns *gender* and *country* are looked up, as they are required for the succeeding scan operation.

Even though the visualization of the late materialization strategy implies a parallel execution of the lookups, the execution can also be done sequentially. Actually, with

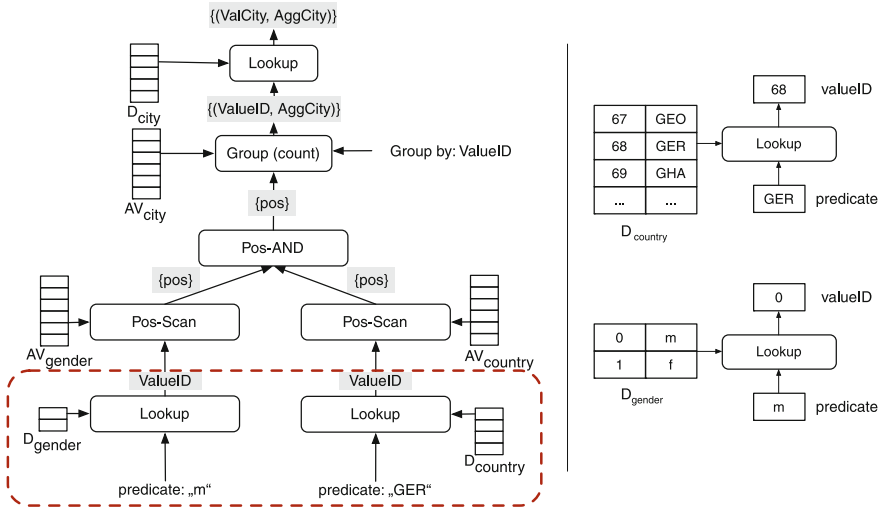

Fig. 16.6 Late materialization: lookup predicate values in dictionary

a predicate as *country* = '*GER*', for which less than 2 % of the world population qualify, a sequential execution is advantageous (see Chap. 15 for more details).

Figure 16.7a shows the scan phase. With the valueIDs from the first step, now the attribute vectors are scanned. The position of each matching valueID in the attribute vector is added to the output vector of this step ($\{pos\}$). The merge of these positional lists is shown in Fig. 16.7b. Here, each value that is existent in both vectors is appended to the result vector of this step.

Figure 16.8a shows the group by operation. Hereby, the intermediate vectors are taken to group the positions in $\{pos\}$ by the valueIDs in the city attribute vector and add the count of each city to the output vector. In the last step the actual lookup of the city valueIDs is performed, as shown in Fig. 16.8b.

Compared to the early materialization strategy, the late materialization strategy might have to perform an additional lookup, e.g. when the gender would also be part of the result. This penalty can diminish the advantages, for example when many columns have to be materialized (consequently many dictionary lookups, what typically occurs when using `'SELECT*'`) or when the result set is very large (i.e., many output rows).

In general, the question to which extend—and even if—late materialization is in favor of early materialization depends on many variables like the used query operations and selectivity, among others [GKK+11].

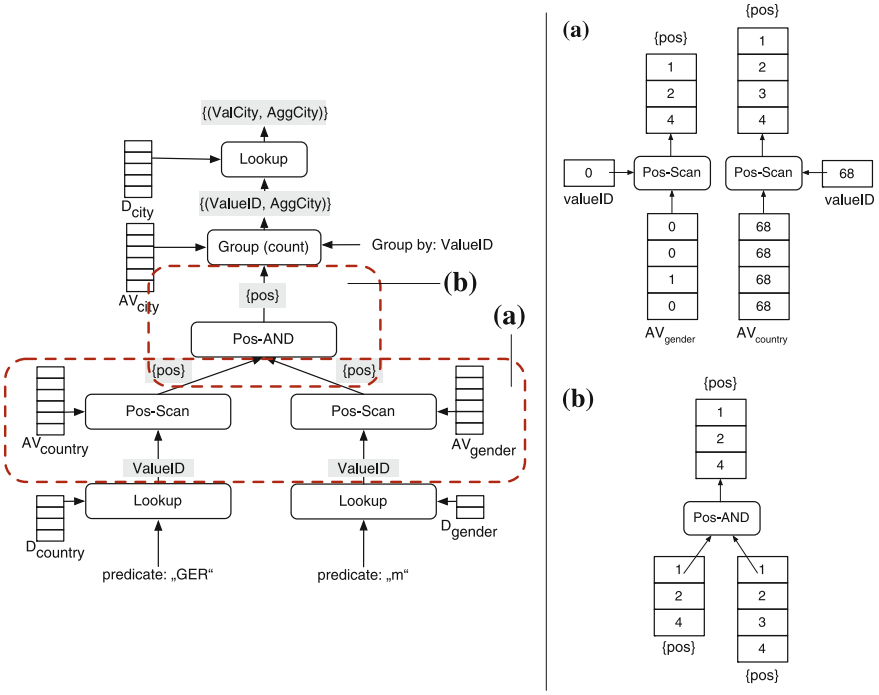

Fig. 16.7 Late materialization: scan and logical *AND*

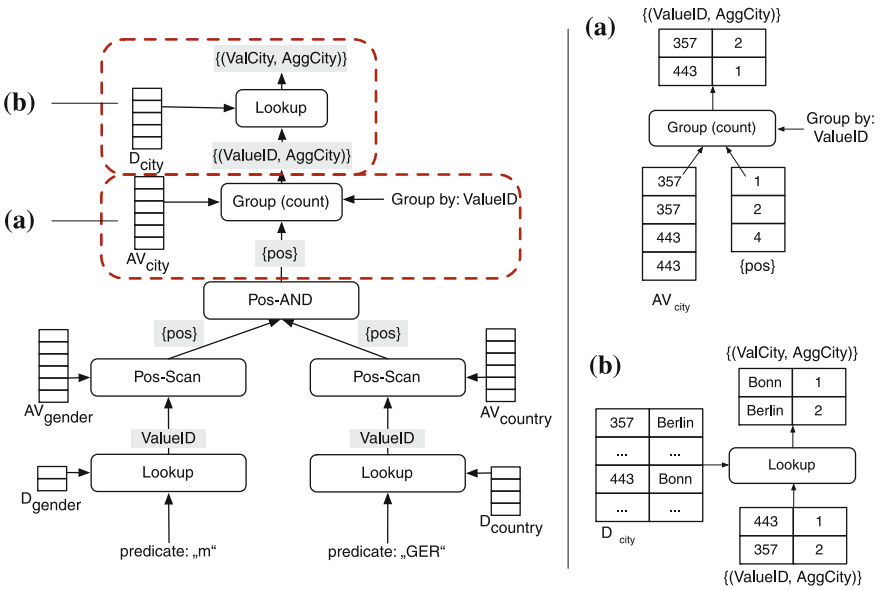

Fig. 16.8 Late materialization: filtering of attribute vector and dictionary lookup

16.5 Self Test Questions

1. **Which Strategy is Faster?**
   Which materialization strategy—late or early materialization—provides the better performance?

   (a) Early materialization
   (b) Late materialization
   (c) Depends on the characteristics of the executed query
   (d) Late and early materialization always provide the same performance.

2. **Disadvantages of Early Materialization**
   Which of the following statements is true?

   (a) The execution of an early materialized query plan can not be parallelized
   (b) Whether late or early materialization is used is determined by the system clock
   (c) Early materialization requires lookups into the dictionary, which can be very expensive and are not required when using late materialization
   (d) Depending on the persisted value types of a column, using positional information instead of actual values can be advantageous (e.g. in terms of cache usage or SIMD execution).

References

[AMDM07] D.J. Abadi, D.S. Myers, D.J. DeWitt, S. Madden, Materialization strategies in a column-oriented dbms, in *ICDE*, ed. by R. Chirkova, A. Dogac, M.T. Ã-zsu, T.K. Sellis (IEEE, New York, 2007), pp. 466–475 Url: http://dblp.uni-trier.de/db/conf/icde/icde2007.html#AbadiMDM07

[GKK+11] M. Grund, J. Krueger, M. Kleine, A. Zeier, H. Plattner, Optimal Query Operator Materialization Strategy for Hybrid Databases, in *DBKDA* (IARIA, Cancun, 2011), pp. 169–174