Chapter 11
Insert

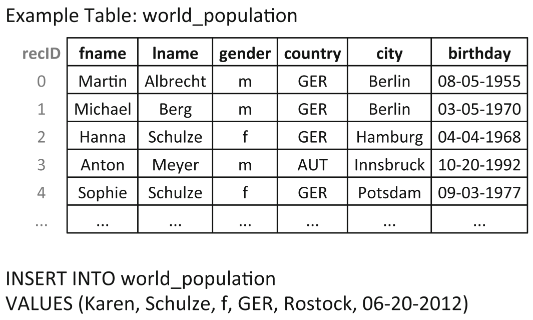
This chapter outlines what happens when inserting a new tuple into a table (execution of an insert statement). Compared to a row-based database, the insert in a column store is a bit more complicated. For a row-oriented database, the new tuple is simply appended to the end of the table, i.e., the tuple is stored as one piece. SanssouciDB uses column-orientation to store the data physically. A detailed description of the differences between row stores and column stores is given in Chap. 8. In a column store, adding a new tuple to the database means to add a new entry to every column that the table consists of. Internally, every column consists of a dictionary and an attribute vector (see Chap. 6). Adding a new entry to a column means to check the dictionary and adding a new value if necessary. Afterwards, the respective value of the dictionary entry is added to the attribute vector of the column. Since the dictionary is sorted, adding a new entry to a column results in three different scenarios:

1. Adding without a new dictionary entry
2. Adding with a new dictionary entry, without resorting the dictionary
3. Adding with a new dictionary entry, with resorting the dictionary

In this chapter, we will give a step by step explanation of the three different scenarios.

11.1 Example

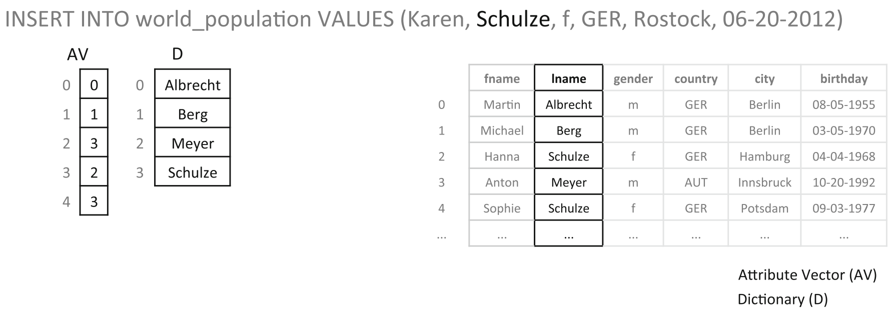
In this example, we insert the data of a new person into the *world_population* table (see Fig. 11.1) that we used before. The example outlines what happens for the column *lname*, representing the last name of a person, and *fname*, representing the first name of a person.


Example Table: world_population

recID	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
...	...	...	...	...	...	...

INSERT INTO world_population
VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)

Fig. 11.1 Example database table named *world_population*


INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)

AV

	AV
0	0
1	1
2	3
3	2
4	3

D

	D
0	Albrecht
1	Berg
2	Meyer
3	Schulze

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
...	...	...	...	...	...	...

Fig. 11.2 Initial status of the *lname* column

11.1.1 Inserting without New Dictionary Entry

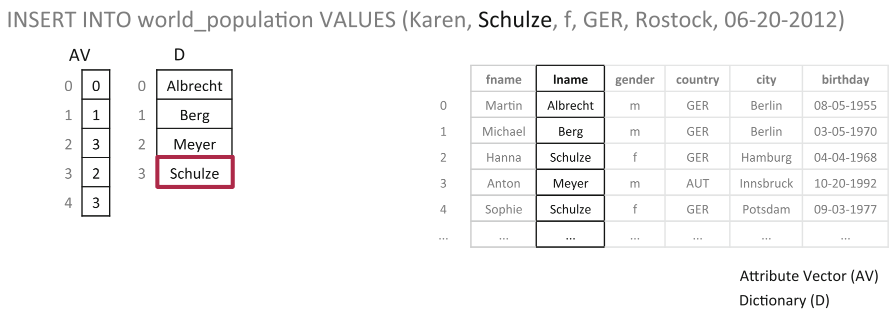
To demonstrate a scenario were we have an insert without a new entry to the dictionary, we will look at the insert of the last name attribute to the *lname* column of our *world_population* table. Attribute vector and dictionary of the *lname* column are initially filled as displayed in Fig. 11.2.

To add the string *Schulze* to the column, we need to look up whether it already exists in the dictionary. Since there is another person named *Sophie Schulze* (recordID four of the world_population table) in the database, the dictionary for the *lname* column already contains an entry with the string *Schulze*. As one can see from Fig. 11.3, the dictionary position of *Schulze* is “3”.

Since *Schulze* is on position 3 of the dictionary, we append 3 to the end of the attribute vector (see Fig. 11.4).

11.1.2 Inserting with New Dictionary Entry

When inserting the first name, the first name dictionary is scanned for the string *Karen*. As shown in Fig. 11.5, this name is not present in the dictionary, yet.



```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

AV

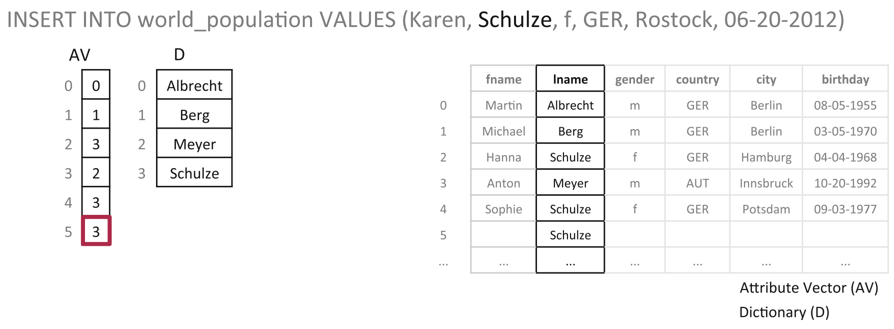
0	0
1	1
2	3
3	2
4	3

D

0	Albrecht
1	Berg
2	Meyer
3	Schulze

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
...	...	...	...	...	...	...

Fig. 11.3 Position of the string *Schulze* in the dictionary of the *lname* column



```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

AV

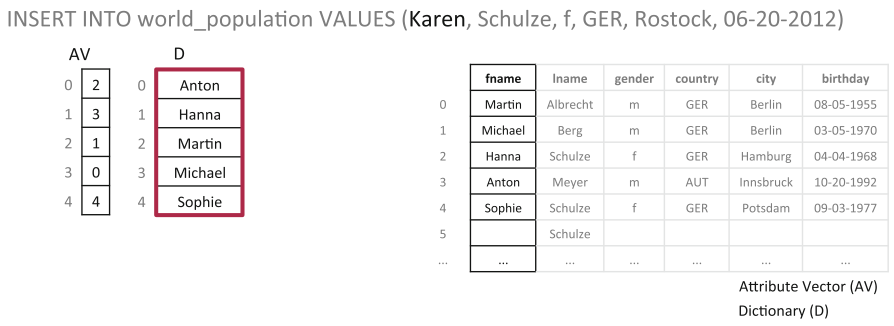
0	0
1	1
2	3
3	2
4	3
5	3

D

0	Albrecht
1	Berg
2	Meyer
3	Schulze

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Fig. 11.4 Appending valueID of *Schulze* to the end of the attribute vector



```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

AV

0	2
1	3
2	1
3	0
4	4

D

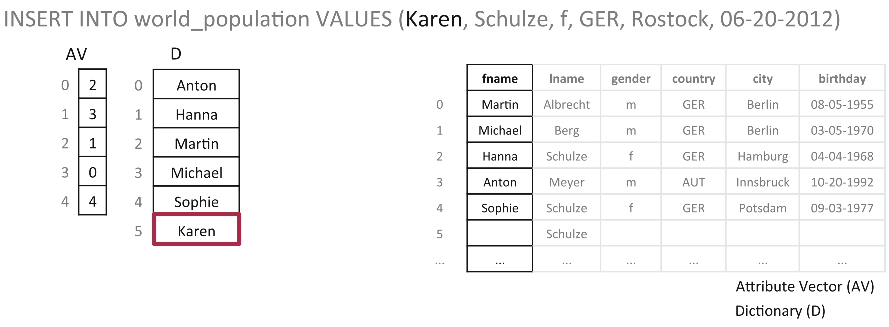
0	Anton
1	Hanna
2	Martin
3	Michael
4	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Fig. 11.5 Dictionary for first name column

Therefore, the name is appended to the end of the first name dictionary (see Fig. 11.6).

As outlined in Chap. 6, the dictionary needs to be kept sorted. After appending *Karen* to the end of the dictionary, the dictionary needs to be resorted. Therefore, as shown in Fig. 11.7, a new dictionary is created with a sorted order. In the new



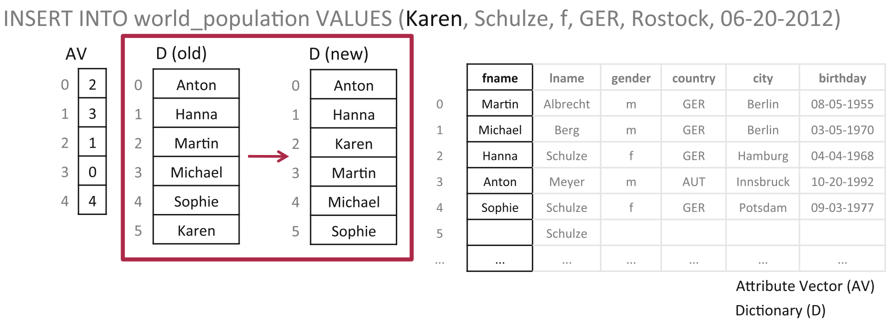
```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

	AV
0	2
1	3
2	1
3	0
4	4

	D
0	Anton
1	Hanna
2	Martin
3	Michael
4	Sophie
5	Karen

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Fig. 11.6 Addition of *Karen* to *fname* dictionary



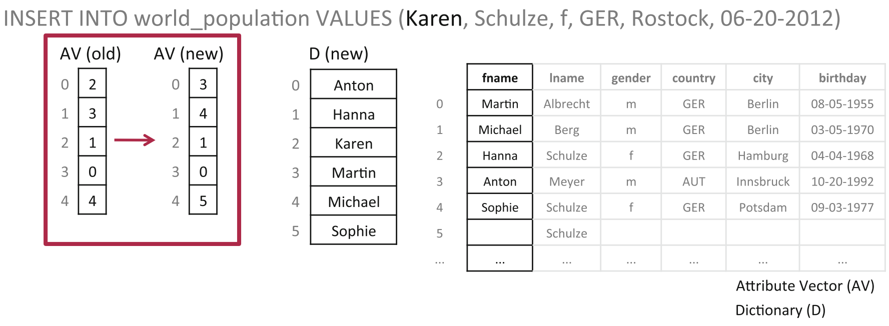
```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

	AV
0	2
1	3
2	1
3	0
4	4

	D (old)		D (new)
0	Anton	0	Anton
1	Hanna	1	Hanna
2	Martin	2	Karen
3	Michael	3	Martin
4	Sophie	4	Michael
5	Karen	5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Fig. 11.7 Resorting the *fname* dictionary



```
INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)
```

	AV (old)		AV (new)
0	2	0	3
1	3	1	4
2	1	2	1
3	0	3	0
4	4	4	5

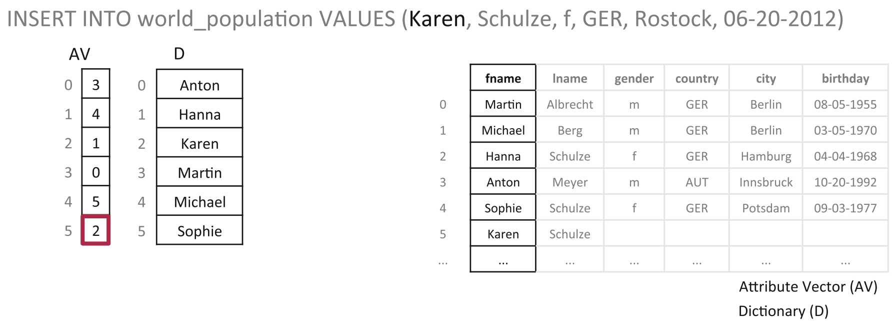
	D (new)
0	Anton
1	Hanna
2	Karen
3	Martin
4	Michael
5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5		Schulze				
...	...	...	...	...	...	...

Fig. 11.8 Rebuilding the *fname* attribute vector

dictionary most of the values have been moved to a new position. For instance, the valueID for *Michael* changed from 3 to 4.

Based on the changed valueIDs of the new first name dictionary, all valueIDs of the first name attribute vector need to be updated as well. Figure 11.8 shows the changes to the attribute vector. For instance at position 1, the valueID for *Michael* is changed from 3 to 4.



INSERT INTO world_population VALUES (Karen, Schulze, f, GER, Rostock, 06-20-2012)

	AV
0	3
1	4
2	1
3	0
4	5
5	2

	D
0	Anton
1	Hanna
2	Karen
3	Martin
4	Michael
5	Sophie

	fname	lname	gender	country	city	birthday
0	Martin	Albrecht	m	GER	Berlin	08-05-1955
1	Michael	Berg	m	GER	Berlin	03-05-1970
2	Hanna	Schulze	f	GER	Hamburg	04-04-1968
3	Anton	Meyer	m	AUT	Innsbruck	10-20-1992
4	Sophie	Schulze	f	GER	Potsdam	09-03-1977
5	Karen	Schulze				
...	...	...	...	...	...	...

Fig. 11.9 Appending the valueID representing *Karen* to the attribute vector

In case the newly added dictionary value is inserted at the end based on the sorting order of the dictionary, those two steps are omitted. The dictionary does not need to be resorted and therefore the attribute vector does not need to be rebuilt.

Finally the valueID 2, representing the dictionary position of the string *Karen*, is appended to the attribute vector (see Fig. 11.9).

11.2 Performance Considerations

When thinking of the *world_population* example, there are about 8 billion people and 5 million unique first names. Every new entry to the dictionary may cause an overhead regarding resorting of the dictionary and reorganization of the respective attribute vector. Triggering resorting and reorganization at every single insert would lead to a performance penalty, which compromises the overall performance of the system. Therefore, an additional insert layer needs to be added, the *differential buffer*. Chapter 25 explains in detail how write performance is kept at a high level using periodic merges of the differential buffer and the main store.

The vulnerability of a column to reorganization heavily depends on the column cardinality (the number of distinct values in a dictionary). When the dictionary only has a few entries, it is most likely that a column needs to be reorganized with a new insert. However, especially with attributes of low column cardinality, e.g., gender or country, the likelihood of reorganization decreases over time, since most of the possible values for the respective column have been inserted into the dictionary already. In real world applications, the dictionary only changes occasionally after it has reached a certain size. The additional steps necessary for new unique dictionary entries will occur less frequent and therefore expensive reorganization becomes less frequent.

11.3 Self Test Questions

1. **Access Order of Structures during Insert**
   When doing an insert, what entity is accessed first?
   (a) The attribute vector
   (b) The dictionary
   (c) No access of either entity is needed for an insert
   (d) Both are accessed in parallel in order to speed up the process

2. **New Value in Dictionary**
   Given the following entities:
   Old dictionary: ape, dog, elephant, giraffe
   Old attribute vector: 0, 3, 0, 1, 2, 3, 3
   Value to be inserted: lamb
   What value is the lamb mapped to in the new attribute vector?
   (a) 1
   (b) 2
   (c) 3
   (d) 4

3. **Insert Performance Variation over Time**
   Why might real world productive column stores experience faster insert performance over time?
   (a) Because the dictionary reaches a state of saturation and, thus, rewrites of the attribute vector become less likely.
   (b) Because the hardware will run faster after some run-in time.
   (c) Because the column is already loaded into main-memory and does not have to be loaded from disk.
   (d) An increase in insert performance should not be expected.

4. **Resorting Dictionaries of Columns**
   Consider a dictionary encoded column store (without a differential buffer) and the following SQL statements on an initially empty table: INSERT INTO students VALUES(‘Daniel’, ‘Bones’, ‘USA’);
   INSERT INTO students VALUES(‘Brad’, ‘Davis’, ‘USA’);
   INSERT INTO students VALUES(‘Hans’, ‘Pohlmann’, ‘GER’);
   INSERT INTO students VALUES(‘Martin’, ‘Moore’, ‘USA’);
   How often do attribute vectors have to be completely rewritten?
   (a) 2
   (b) 3
   (c) 4
   (d) 5

5. **Insert Performance**
   Which of the following use cases will have the worst insert performance when all values will be dictionary encoded?

   (a) A city resident database, that store all the names of all the people from that city
   (b) A database for vehicle maintenance data which stores failures, error codes and conducted repairs
   (c) A password database that stores the password hashes
   (d) An inventory database of a company storing the furniture for each room