

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>

Indices

In row-oriented database systems indices are used to speed up access on columns that are often used for data selection (e.g. primary key attributes). Without indices a full table scan has to be performed when the data is stored row-wise. In column-oriented databases systems a data select operation on any attribute only requires a column scan. This is performed much faster than a complete table scan. Consequently, in many use cases additional indices are not required. The following chapter describes the usage of additional index structures in column-oriented tables, called inverted indices.

18.1 Indices: A Query Optimization Approach

Usually, applications work only with a subset of records at a time. Therefore, before processing the portion, it must be located within the database. Hence, records should be stored in a manner that makes it possible to locate them efficiently whenever they are needed. The process of locating a specific set of records is determined by the predicates that are used to characterize these records.

SanssouciDB organizes its records in columns (see Chapter 8). To determine the records, it is necessary to perform a scan on all the columns, which are used as filter criteria. In main memory column-oriented databases, which store column values continuously, i.e. in adjacent memory blocks, searching for a value with a full scan (by iterating through all items placed in memory sequentially) can be done orders of magnitude faster than in row-oriented databases. Therefore, the creation of additional index structures in such databases is not essential, such as in row-oriented systems. Nevertheless, because the complexity of a full column scan is linear ($O(n)$), with the number of tuples, it is just a matter of data volume that will make the speed advantage of indices relevant to main memory column-oriented databases.

In this chapter, we discuss the topic of inverted indices in the context of main memory databases in more detail.

18.2 Technical Considerations

In the *world_population* table shown in Figure 18.1 we want to locate the records of all people from Berlin. The dictionary and the attribute vector of

Example Table: *world_population*

recID	fname	lname	gender	country	city	birthday
0	Daniel	Specht	m	GER	Iserlohn	08-05-1955
1	Michael	Schardt	m	GER	Dresden	03-05-1970
2	Miriam	Faust	f	GER	Hamburg	04-04-1968
3	Toni	Meier	m	GER	Dresden	10-20-1992
4	Marlit	Mattes	f	GER	Berlin	09-03-1977
5	Tanja	Berger	f	GER	Aachen	07-10-1983
6	Thomas	Peters	m	GER	Berlin	04-07-1989
7	Ralf	Wagner	m	GER	Potsdam	01-11-1981

Fig. 18.1: Example Database Table Named *world_population*

the column are shown in Figure 18.2.

Column[city]					
Attribute Vector (AV)			Dictionary (D)		
0	4	Iserlohn	0	Aachen	
1	2	Dresden	1	Berlin	
2	3	Hamburg	2	Dresden	
3	2	Dresden	3	Hamburg	
4	1	Berlin	4	Iserlohn	
5	0	Aachen	5	Potsdam	
6	1	Berlin			
7	5	Potsdam			

Fig. 18.2: City Column of the *world_population* Table

To determine the set of *Berlin* records, we need to set the filter criterion on the *city* attribute of the table. The respective SQL query could look like the one depicted in Listing 18.1.

Listing 18.1: Query to select all people from Berlin

```
SELECT * FROM world_population
WHERE city = 'Berlin'
```

Now, let us do a back of the envelope calculation. Assuming the table contains 8 billion records, our CPU is able to process 4 GB per second per core, and the city column is encoded using 20 bit. The memory footprint of the *city* column's attribute vector can be calculated using

$$8 \text{ billion} \cdot 20 \text{ bit} = 160 \text{ billion bit} = 20 \text{ billion Byte} \approx 18.6 \text{ GB}$$

The time it takes a single core to scan that amount of data can be calculated using

$$18.6 \text{ GB} \div 4 \text{ GB/sec} \approx 4.65 \text{ sec}$$

If the scan operation is parallelized between 40 cores, it takes $\approx 116 \text{ ms}$. While this speed is unthinkable for a row-oriented database, the speed might not be sufficient for all applications.

18.3 Inverted Index

Now, let us investigate a more complicated, but more efficient algorithm presented in detail in [FSKP12]. We consider an inverted index for the attribute *city*. An inverted index maps each distinct value to a position list, which contains all positions where the distinct value can be found in the column. The index for the dictionary-encoded column consists of the following two parts:

- Index offsets (IO) : this vector stores for each dictionary entry (or in other words, for each unique value of the attribute vector) the offset for the position list in the positions vector. This means that the offset vector stores references to the first occurrence of the particular dictionary value in the positions vector.
- Index positions (IP) : the index position vector contains a position list of all distinct values of the attribute vector sorted by the integer valueID. In contrast, the attribute vector stores valueIDs by position.

Let us see how much data the CPU has to read using an index. We continue with the query shown in Listing 18.1. The following steps need to be executed to determine the position of *Berlin* in the attribute vector.

1. We need to perform a binary search on the dictionary to determine the dictionary position related to *Berlin*. As depicted in Figure 18.4, *Berlin* is at position 1.
2. The dictionary position of *Berlin* corresponds directly to the position of the index offset vector shown in Figure 18.5. In this example, the dictionary

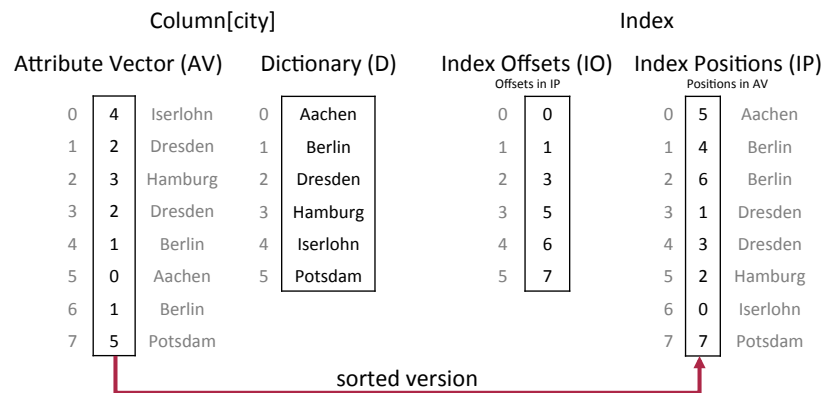


Fig. 18.3: Index Offset and Index Positions

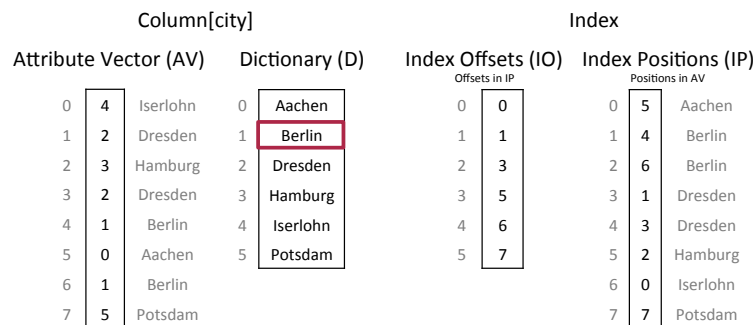


Fig. 18.4: Query Processing Using Indices: Step 1

position of *Berlin* is 1, so the corresponding index offset vector position is 1.

- Since the attribute of the respective search criterion is not necessarily a primary key, it is possible that the same value is used by many records. Consequently, more than one attribute vector entry can be filled with that value. As explained in the beginning of this chapter, the index position vector represents a sorted list of the values in the attribute vector. To determine the range of values to read from the index position vector, we simply read the value of the index offset vector at the position, we determined in Step 1, and the value of the next higher position (see Figure 18.6).
- Since 3 is already the offset of the next value in the index positions vector, we only need to read positions 1 and 2. As shown in Figure 18.7, IP vector position 1 contains the value 4 and position 2 contains the value 6, which

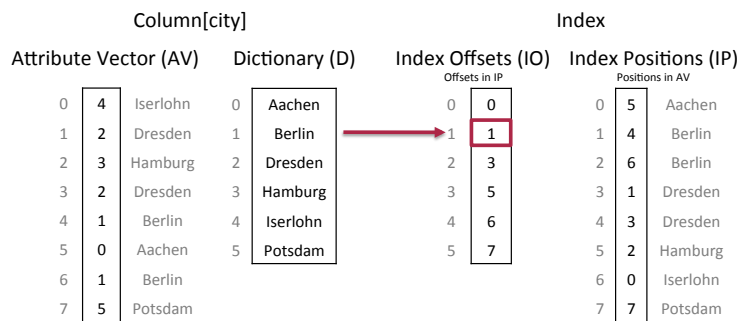


Fig. 18.5: Query Processing Using Indices: Step 2

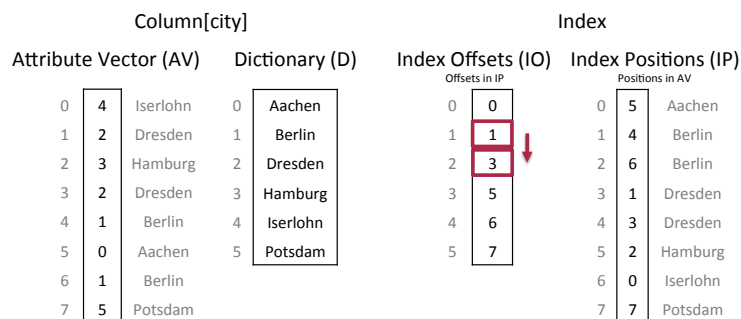


Fig. 18.6: Query Processing Using Indices: Step 3

are the exact positions of the dictionary code for *Berlin* in the attribute vector. By retrieving the offsets of *Berlin* and the next entry, in our case *Dresden*, from the IO vector, we are able to determine the exact range of all values we need to read in order to resolve the respective attribute vector positions of *Berlin*.

5. With the positions resolved in Step 4, we are able to jump directly to the respective attribute vector positions of all other columns of that table in order to materialize the complete records of all the people that live in Berlin.

Using this approach, we reduce the data volume read by a CPU from the main memory by providing a data structure that does not require the scan of the entire attribute vector. Investigations regarding the influence of using indices on memory traffic and performance are shown in Section 18.4.

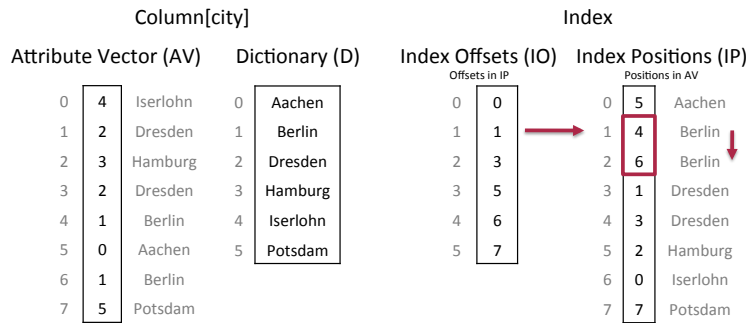


Fig. 18.7: Query Processing Using Indices: Step 4

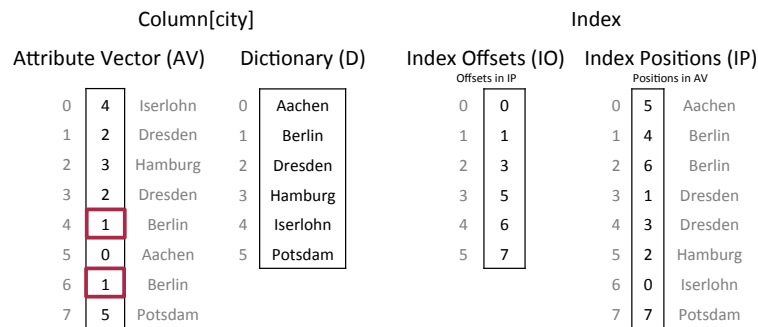


Fig. 18.8: Query Processing Using Indices: Step 5

18.4 Discussion

In the previous section, we explained the idea of using an inverted index on a dictionary-encoded column to improve response time for lookup requests. An index increases the memory consumption per column. In this section we compare data lookup using full table scan against an index, regarding memory consumption and lookup performance. Table 18.1 introduces the symbols that we use.

18.4.1 Memory Consumption

In the beginning of this chapter, we explained that an index consists of an IO vector and an IP vector. To determine the overall size of the index, we need

Description	Unit	Symbol
memory consumption of the index	bits	I_m
length of the index offset vector	-	IO_l
width of the index offset vector	bits	IO_w
memory consumption of index offset vector	bits	IO_m
length of the index positions vector	-	IP_l
width of the index positions vector	bits	IP_w
memory consumption of index positions vector	bits	IP_m
length of dictionary (number of distinct values in column)	-	D_l
length of attribute vector	-	AV_l
width of attribute vector	bits	AV_w

Table 18.1: Symbols used for calculating memory consumption

to calculate the size of these two structures.

$$I_m = IO_m + IP_m$$

The allocated memory of a vector can simply be calculated by multiplying its length (number of entries) with its width (size of a single entry).

$$IO_m = IO_l \cdot IO_w$$

$$IP_m = IP_l \cdot IP_w$$

The length of the index position vector IP directly corresponds to the length of the attribute vector AV_l , since it is basically a sorted version of the corresponding attribute vector. The width of IP is determined by the bit-encoded length of the attribute vector, since it contains direct positions to the values in the attribute vector.

$$IP_l = AV_l$$

$$IP_w = \lceil \log_2(AV_l) \rceil \text{ bits}$$

The length of the index offset vector IO directly corresponds to the length of the dictionary D_l , which in turn is determined by the number of distinct values in the respective column. The width of IO is derived from the biggest offset into IP , because IO contains the bit-encoded offsets used to determine the position ranges in IP . As the maximum offset stored in IO can be the length of IP , the resulting width of IO is $\lceil \log_2(IP_l) \rceil$ bits.

$$IO_l = D_l$$

$$IO_w = \lceil \log_2(IP_l) \rceil \text{ bits}$$

Summarizing, we combine the mentioned formulas to a single equation for calculating the size of an index structure.

$$I_m = D_l \cdot \lceil \log_2(IP_l) \rceil + AV_l \cdot \lceil \log_2(AV_l) \rceil \text{ bits}$$

$$I_m = (D_l + AV_l) \cdot (\lceil \log_2(AV_l) \rceil) \text{ bits}$$

Let us now calculate the actual size of an index for the *city* column of our *world_population* table from Figure 18.1. We need to determine D_l , IP_l , and AV_l . Based on the assumption that there are about 1 million cities around the world and that the world population is 8 billion, we just need to insert these numbers into our formula.

$$I_m = 10^6 \cdot \lceil \log_2(8 \cdot 10^9) \rceil + 8 \cdot 10^9 \cdot \lceil \log_2(8 \cdot 10^9) \rceil \text{ bits}$$

So from this formula, we get an index size of about 31 GB for the *city* column.

18.4.2 Lookup Performance

Independent of using an index or not, we need to perform a binary search on the dictionary to determine the encoded value for the respective search term. Let us assume that we need to read $\log_2(D_l)$ entries to perform the binary search. Since the binary search on the dictionary has to be done for both access methods we can ignore it, when we compare them.

In case of a full column scan, we need to traverse the attribute vector sequentially, by reading AV_l entries, each with a size of $\lceil \log_2(D_l) \rceil$ bits. Again, assuming 8 billion rows in the table and 1 million cities, we need to read

$$8 \cdot 10^9 \cdot \lceil \log_2(10^6) \rceil \text{ bits} = 160.000.000.000 \text{ bits}$$

for a full attribute vector scan.

Now, when using an index the situation is different. After the dictionary lookup, we directly read the upper and lower limit from the index offset vector (see Figure 18.6). We neglect this step for our performance consideration since this constant effort does not impact the overall performance. Having determined the upper and lower limit for the index positions vector, we need to traverse through it (see Figure 18.7). The number of entries to read from IP depends on the distribution of values in the column. Reading an attribute value that is used more frequently, we need to read more entries, on less frequently used values we need to read less. Assuming a uniform distribution of values we need to read $AV_l \div D_l$ entries. The width of the entries is $\lceil \log_2(AV_l) \rceil$ bits. Combining both equations, we get

$$IndexPositions = \frac{AV_l \cdot \lceil \log_2(AV_l) \rceil}{D_l}$$

for the number of bits to read from the index positions vector. Taking our world population example, where we look for all people living in Berlin, we

come up with

$$\frac{8 \cdot 10^9 \cdot \lceil \log_2(8 \cdot 10^9) \rceil \text{ bits}}{10^6} = 264.000 \text{ bits}$$

to read using an index. Assuming a CPU scan speed of 4MB/ms/core, a single core needs about 5 seconds to scan the complete attribute vector. Accessing the column using an index, the CPU needs about 0.008 ms to read the attribute vector positions from *IP* of the people living in Berlin. Thus, in this example, we can improve the lookup performance drastically with the index, compared to a sequential attribute vector scan.

We compare the theoretical memory traffic for the attribute vector scan and the position read in Figure 18.9 for different dictionary sizes on a column with 30 million entries. With a uniform distribution, the index leads to less memory traffic, if at least 8 distinct values are present.

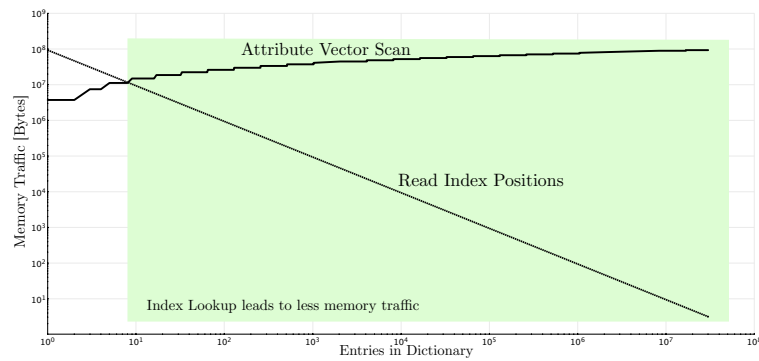


Fig. 18.9: Attribute Vector Scan vs. Index Position List Read for a Column with 30 Million Entries (Note the Log-Log Scale)

18.5 References

- [FSKP12] M. Faust, D. Schwalb, J. Krueger, and H. Plattner. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *ADMS '12: Proceedings of the 3rd international workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures at VLDB'12*, 2012.

