Exercise 7

Released: June 5, 2018 · Discussion: June 11, 2018

In this exercise we want to show the usage and efficiency of bitmap indices using the TPC-H benchmark.

In order to solve the assignments you need to download the archive code.zip and lineitem.tbl.zip. During the course of the exercise you shall use and complement the C++ source code given in code.zip. For testing you might use the archive lineitem.tbl.zip.

1 Queries

As a start we want to evaluate simple queries on the table lineitem of the benchmark.

For this the provided source code declares four functions query1(), query2(), query3(), query4 which are partially implemented.

1. Use the data generator to generate a file lineitem.tbl and compile the provided source code.

Now you can start the program rawDB. It reads in the file lineitem.tbl and represent it as table in memory. Then it will execute the four queries while measuring their processing time.

2. Add the missing implementations for the functions query2(), query3() and query4(). The following four queries are going to be examined.

Query 1:

```sql
SELECT COUNT(*)
FROM lineitem
WHERE linenumber = 3 OR linenumber = 5
```

Query 2:

```sql
SELECT COUNT(*)
FROM lineitem
WHERE linenumber = 5 AND shipmode = 'RAIL' AND quantity = 10
```

Query 3:

```sql
SELECT COUNT(*)
FROM lineitem
WHERE linenumber IN (2, 4) AND shipmode IN ('SHIP', 'TRUCK')
```
Query 4:

\[
\text{SELECT } \sum (\text{extendedprice} \cdot (1.0 - \text{discount} / 100.0)) \\
\text{FROM } \text{lineitem} \\
\text{WHERE } \text{returnflag} = 'R' \text{ AND } \text{shipmode} \text{ IN ('REG AIR', 'AIR')} \\
\text{AND } \text{quantity} \text{ BETWEEN 5 AND 10}
\]

2 Bitmap Indexing

A framework for using bitmap indices is already provided in the code archive. The entry point for the framework is the class `BitmapIndex<T>`, which is parameterized by the type `T` of the column to be indexed.

```cpp
class BitmapIndex<T> {
    BitmapIndex (FieldType<T> fieldType, uint32_t numrows);
    uint32_t buildIndex (lineitem_t *inputtable);
    const BitVector & getBitVector (const T & fieldValue);
};
```

The function `buildIndex()` is used to read a column and create a bitmap index on it. It returns the number of used 32 bit words. The function `getBitVector()` returns the corresponding bit vector for a given attribute value.

A new bitmap index can be created as follows (e.g. for a column of type `char[11]`):

```cpp
FieldType<char[11]> field_shipmode (10, offsetof (lineitem_t, shipmode));
BitmapIndex<char[11]> bitmap_idx_shipmode (field_shipmode, numrows);
uint32_t numWords = bitmap_idx_shipmode.buildIndex (inputtable);
```

Here 10 denotes the cardinality of the corresponding column.  

---

1 The column `shipmode` only contains fixed values. For that the code maps the corresponding strings to unique numbers between 0 and 9.
Afterwards the query 1 for example can be rewritten using a bitmap index:

```
uint32_t
query1_with_index (...) {
    const BitVector & vec_ln3 = bitmap_idx_linenumber.getBitVector (3);
    const BitVector & vec_ln5 = bitmap_idx_linenumber.getBitVector (5);
    BitVector res = vec_ln3 | vec_ln5;
    return res.count ();
}
```

3. Add the missing parts of the implementation of the bitmap index in the files PlainBitVector.cpp and PlainBitVector.h. The lines where code has to be added are marked with /* TODO: ... */.

Following classes of the C++ standard library may be helpful for the implementation.

`std::vector<T>` This class provides a more flexible and powerful alternative to arrays.
It is parameterized by the type of its elements. An object of `std::vector<T>` can be used like an ordinary array with the difference that the array can grow in size on demand.

`std::bitset<N>` This class allows the comfortable manipulation of bit fields. So for example it allows to directly access a bit in an array-like fashion instead of needing shift operators.

4. After implementing the class `PlainBitVector<T>` now rewrite the queries two to four to use a bitmap index. Examine the difference in efficiency when using a bitmap index compared to not using one.

3 WAH compressed bitmap indices

The provided code is designed to provide to alternative implementations of a bitmap index. So compiling your code with

```
make WAH=1
```

will include the class `WAHBitVector` instead of `PlainBitVector`. Former shall implement bit vectors using the WAH compression technique.

5. Add the missing implementations in the files `WAHBitVector.h` and `WAHBitVector.cpp`. The lines where code has to be added are annotated like in the previous assignment. Do not forget to compile your code with `make WAH=1` so that your code will be executed actually.

6. Compare the measured numbers (Memory consumption and performance) with and without using bitmap indices as well as with and without compression.
Hints

The program also prints the result of queries. They are not relevant for this exercise but might help to check the correctness of your implementation.

Logical operators in particular belong to the implementation of bitmap indices. They can be executed very efficiently on WAH compressed bit vectors. The respective algorithms can be found in [1].

The program `dbgen` allows to create huge data sets. So a table with scaling factor 2 (2 GB in size) can be created with `dbgen -vf -s 2 -T L`.

Literatur