Matrix representations in Scylla database

Introduction

The problem of storing matrices in Scylla database had us face the inevitable question: what is the most effective way to represent our data? Among the most popular representations of sparse matrices used in numeric calculations are:

- dictionary of keys (**DOK**)
- list of lists (**LIL**)
- coordinate list (**COO**)
- compressed sparse row (**CSR**)

Each of the representations has its own advantages and disadvantages. Scylla, as a data storage system, has its limitations – we expect overhead caused by collection of data through queries instead of direct access, and need to accomodate for the peculiar partitioning feature (which, to some degree, can also be taken advantage of). This is why we had to treat the aforementioned formats as an inspiration rather than direct specification. This document aims to summarise our representations based on each of the models listed above, describing all their advantages and disadvantages found to-date. The summary will serve as a basis on which the most suitable representation will be picked for future experiments.

Overview

The representations were implemented in the C++ programming language and used by subclasses of an abstract **multiplicator** class (`multiplicator.hh`). The intended application for each of the specialisations was thus:

- the method `load_matrix()` would be called twice, each time with a matrix generator object as an argument, so that each implementation of the multiplicator would read and save two (possibly different) matrices and store it using its respective representation;
- the method `multiply()` would then be called, and the two matrices loaded earlier (the first we will call A, the latter we will call B) would be multiplied in order to obtain a matrix C as a result (satisfying $C = AB$), that would be stored in the database in the format used by the multiplier
- the values of the result matrix could be obtained upon the completion of the `multiply()` function by calls to `get_result()`, which takes the pair of coordinates (i, j) of the requested cell as arguments.

The signatures are as follows:

```
1    virtual void load_matrix(matrix_value_generator<T>&& gen) = 0;
2    virtual void multiply() = 0;
3    virtual T get_result(std::pair<size_t, size_t> pos) = 0;
```

The implementations were cross-tested with an aid of Boost Unit Test framework. The tests were implemented in the `simple_test.cc` file and can be executed by running the `simple_test` program. As of now, the tests are slow due to the inherent slowness of our implementations, which have been designed as a proof-of-concept rather than ready-to-use, and lack the necessary optimizations of table creation, value retrieval, and even the multiplication itself.

WARNING

The tests may fail due to a 'lack of permissions' or 'memory access violation error', both of unknown origin. This may be an issue either of the testing mechanism or of our implementations. In case it is the latter, further investigation is needed.

1 Dictionary of keys

2 List of lists

3 Coordinate list

Blah blah some text. [WIP]

I used blocks, and each block is represented with a list. This format is pretty meh. With blocks you need to do lots of data collection and multiplications to get the result for a single cell (admittedly, though, the overhead should not be too high – for this, however, we'd need to tweak the database extraction queries. It's quite easy to build result matrices. We should be able to query portions of data so large that we can ignore the cost of queries. Block operations works rather intuitively.

We can transpose blocks easily in RAM – efficient storage. Currently storing values in Scylla's $set < tuple < int, int, int >>$ type – may not be the best idea. It is, however, true to the idea of COO. Scylla's *lists* are allegedly even worse.

Perhaps it would be better to combine blocks with dictionary of keys? Wouldn't it be worse efficiency-wise, though, with data scattered all over a partition, instead of keeping it close together? IDK.

4 Compressed sparse row

```
#include <stdio.h>
int main(void)
{
    printf("Hello World!");
}
```