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

{ Should we write something about how we generated our matrices? Also: matrix_value type }

The representations were implemented in the C++ programming language and used by subclasses of an abstract template **multiplicator<T>** class (contained in `multiplicator.hh`), parametrised by the type of values stored in the matrices (most likely `float` or `double`). 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

The **coordinate list** format assumes that the entire matrix is stored as a continuous, lexicographically sorted array of records of the type `(coord_i, coord_j, value)`. There are a few reasons as to why it was impossible to implement this approach in its purest form:

- We want to cooperate with Scylla's clustering by splitting the matrix' data into smaller chunks, residing in different partitions of the database (so as to split it roughly evenly among all clusters).
- The Scylla's native type for continuous lists is highly inefficient.

In order to satisfy the condition that the data be split into multiple chunks, we split the stored matrix into multiple blocks of the size (`_block_size` x `_block_size`), numbered $1..m$, (growing from left to right, then from the top to the bottom) where m is the total number of blocks (full or partial) that make up the original matrix.

Each block belongs in its own partition, and is represented as a single database record containing its unique key (the block's number, being also a partition number), the identifier of its original matrix (for now simply a number), and a (relatively short) list of the coordinates that the block contains.

```
CREATE TABLE zpp.coo_matrices (
    block_id int,
    matrix_id int,
    vals set<frozen<tuple<int, int, double>>>,
    PRIMARY KEY (block_id, matrix_id)
) WITH CLUSTERING ORDER BY (matrix_id ASC);
```

(Please note that the current implementation of the **coordinate list**-based multiplicator creates a new table in its constructor – this means that this kind of multiplicator must be a singleton at most, and that the creation of every multiplicator takes a good bit of time).

The multiplication is done block-wise: for each result blocks, the blocks of multiplicated matrices are loaded blockwise and their results are added as the result is computed in the local memory. Then, the block is submitted to the database, and the next one is computed. This makes for an intuitive multiplication algorithm, albeit with some flaws – for instance, it is difficult to obtain a single call of the result quickly.

Note that:

- In order to perform the multiplication slightly more effectively, every block loaded from one of the matrices is transposed before the multiplication. This part uses sorting, which in fact can be done quicker, in linear time, at the cost of a more difficult implementation.
- Perhaps the multiplication process itself could be sped up with some minor optimisations. For one thing, we probably don't even need to transpose any of the blocks.

For now, querying a result value loads the entire block that the value is located in, and looks for it in the list loaded from the database. It is likely that we could use a refined query to retrieve a single value from the database instead of requesting an entire block.

Remarks:

- The representation is at least a bit uncomfortable for use.
- Most of the benefits of using blocks could be obtained with any other representation, although **coordinate lists** seem to be among the most natural tools for iteration over the values of a matrix. On the other hand, the block-based representation makes construction of the actual coordinate list of an entire matrix slightly more difficult than we would like it.
- There could be better partitioning keys for blocks. Perhaps their diagonal, instead of serial, numbers?
- Scylla's native `set`s are said not to be the as efficient as we would like it. Perhaps we would be better off without them. That would probably bring us closer to the **dictionary of keys** representation, though.

In summary, the **coordinate list** representation seems to combine both advantages and weaknesses of other representations, and might not be the best candidate for our experiments with Scylla, given that there are likely better options, with more positives than negatives.

It is, however, far from useless, as it looks like a fairly convenient way of data storage in the C++ context (the other being a set or a map, which, again, brings about the question of effectiveness that would have to be confirmed in tests). Quite possibly, we could combine its application in C++ with some other internal representation in Scylla to get the best of both worlds and to obtain optimal results. Still, I would not recommend it as first choice for our further work, and endorse it only if all other options prove to have numerous difficulties of their own.

4 Compressed sparse row

Rubbish bin

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