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:

```
    virtual void load_matrix(matrix_value_generator<T>&& gen) = 0;
    virtual void multiply() = 0;
    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

The **dictionary of keys** format benefits from database's data indexing, making both random access via key, and access to sorted rows possible. Below is the table used for this representation:

```
CREATE TABLE zpp.dok_matrix (
 matrix_id int,
 pos_y     bigint,
 pos_x     bigint,
 val       double,
PRIMARY KEY (matrix_id, pos_y, pos_x)
 );
```

Like in other representations, another table is necessary if user wants to store matrix's dimensions as zeroes are not stored in this representation.

Matrix's values are sorted and thus returned first by row number and then by column number. This allows easy matrix multiplication row by row of $A \cdot B$, given that A and B^T is stored in the table.

With this representation, querying subsequent rows is quick. It is possible to chunk the queries into both big parts containing multiple rows and small parts containing only parts of a row. On the other hand, querying subsequent columns is usually slow as iterating through a single column may mean making multiple database queries.

However, it is possible to bypass this problem easily, if, for a given matrix, only one way of iterating is needed – we can transpose the matrix or upload it to a different table that is sorted column number first, row number second and iterate.

2 List of lists

This representation uses quite different approach than others. Each matrix is stored in 2 copies - once as rows list of lists, once as columns list of lists. Both tables work the same way, so let's just talk about one storing rows.

```
CREATE TABLE zpp.lil_rows (
    matrix_id int,
    row bigint,
    part bigint,
    filled int,
    i_0 bigint,
    v_0 double,
    i_1 bigint,
    v_1 double,
    <more columns following same pattern>,
    PRIMARY KEY (matrix_id, row, part)
) WITH CLUSTERING ORDER BY (row ASC, part ASC);
```

Partitioning key definitely should be different (maybe (matrix_id, row)?), but it doesn't really matter now.

To avoid using built-in sets or lists, we treat each database row as constant-size array of pairs (i_x - column index, v_x - value in this column). Each such row in database represents part of matrix row. Column "filled" tells us how much of this database row is used.

Second table works exactly the same way, but stores columns, not rows - it is used to perform multiplication more effectively. Unfortunately, preparing matrix in second table (so pretty much transposing matrix) is slow.

Currently, multiplication is performed without any optimizations, and in very slow way. For each row, we iterate trough columns and calculate single cell of result. After completing whole row, we submit it to database. After that, we create a column matrix of the result matrix in the second table. First optimization would be to calculate blocks of result. For each block of size NxN, we could fetch parts of N rows and parts of N columns, start multiplying them, and fetch new parts (while forgetting old ones) as necessary. Second optimization would be fetching data using different thread than used to count result, to maximize throughput.

Advantages:

- Low overhead of fetching row / part of row. Small amount of database rows need to be retrieved (because many values can be stored per row), also, much bigger % of fetched data is actual data, not matrix_id / row_id etc, compared to DOK and CSR.
- Potentially very fast multiplication and other algorithms.
- No lists or sets, so no performance problems related to them (using them might be a problem with COO).
- Doesn't need the O(n) space for storing row index array, which is the case with CSR - so it can easily store matrices with really big dimensions.

- Creating matrix in the rows table is fast and easy - we can create whole part of a row locally, and insert it with one prepared query, which isn't the case with DOK/CSR.

Disadvantages:

- Fetching single cell at given coordinates is not straightforward. We need to locate right part of the row, fetch the whole part, and find right column. Logarithmic complexity, but with very small constant. COO/CSR have similiar problems, DOK seems best in that regard.
- Creating columnar representation is slow - we need a query for each value in matrix, and using preapred queries is problematic (we would need to prepare a lot of them). I don't really have any idea how to speed up this process. However, DOK and CSR both need inserting one value per query when creating matrix, so this issue may not be disqualifying.

Overall, I think it may be pretty decent representation and I'd consider using it in further work.

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's data into smaller chunks that would reside 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 matrix being stored into multiple blocks sized (`_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 with a unique key (the block's number, being also a partition number), the identifier of the matrix of origin (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 creation of each new multiplicator takes a good portion of time).

The multiplication is done block-wise: for each result block, the blocks of multiplied matrices are loaded and multiplied pairwise. The results are aggregated in local memory. Then, the result 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 is performed by loading the entire block that the value is located in, and browsing the coordinate list obtained this way. 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 a bit awkward and uncomfortable to use. (But seems more intuitive and user-friendly than the LIL and CSR implementations).
- 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.
- Perhaps better partitioning keys for blocks could be found. Maybe their diagonal, instead of serial, numbers?
- Scylla's native `set`s are said not to be the as efficient as we would like it. We would likely be better off without them. This again brings us back to DOK.

In summary, **coordinate list** may not be the best representation for internal storage in the database, yet it seems like a fairly good extension (or one of them) for **dictionary of keys** for RAM-based calculations. It could easily be simulated e.g. with **std::set** or **std::map**, combining random access with an iterable ordering, or with **std::list**, if it allows us to reduce the overall overhead.

4 Compressed sparse row

In the **compressed sparse row** representation (also known as **Yale format**) the matrix is represented by three arrays - holding column indices, values and beginnings of consecutive rows respectively.

Currently the Scylla implementation of this representation uses the two following tables:

```
CREATE TABLE zpp.csr_test_matrix_values (
        matrix_id int,
        idx int,
        column int,
        value float,
        PRIMARY KEY (matrix_id, idx)
) WITH CLUSTERING ORDER BY (idx ASC);
```

```
CREATE TABLE zpp.csr_test_matrix_rows (
        matrix_id int,
        row int,
        idx int,
        PRIMARY KEY (matrix_id, row)
) WITH CLUSTERING ORDER BY (row ASC);
```

The first of those tables corresponds to the first two arrays of our representation, the second table corresponds to the row index array. It should be noted that if we want to store matrices of different sizes we would also have to keep track of their dimensions, possibly in a separate table.

Possibly the main advantage of CSR representation is the fact that we can easily retrieve a single row of the matrix. In order to do that we first look up the beginning and the end of a given row in `csr_test_matrix_rows` table and then retrieve the values from `csr_test_matrix_values` table.

The current implementation contains a separate entry for each non-zero element. Another possibility would be grouping the elements into blocks and holding those as values in the table (similarly to what was done in the implementations of other representations).

In the presented implementation the multiplication is done line by line. In order to construct the ith line of the result matrix AB we first load the ith row of matrix A. Then, for every element a_{ij} of said row we load the corresponding jth row of matrix B, multiply said row by a_{ij} and add the result to the row we are constructing. After the entire row is constructed its values are inserted into the database.

Here are some of the possible issues with CSR representation:

- Inserting a new value in the middle of an already constructed matrix would require updating several entries in both of the tables which makes it very ineffective. For this reason, when constructing a new CSR matrix incrementally the values should be passed row by row.

- The representation always takes up $\Theta(n)$ space for the row index array where n is the number of rows. This might be inefficient for matrices in which the number of non-zero entries is relatively small compared to n. Using another representation such as **coordinate list** or **dictionary of keys** might be preferable in this case.

As already mentioned the main feature of CSR representation is the ability of easily retriving a single row. For this reason the representation is mostly useful in algorithms and applications that naturally use such queries (one possible example could be representing large, immutable sparse graphs as matrices - a single row corresponds to the neighbours of a given vertex). It might be worth comparing this representation with the **list of lists** representation in such uses.

Rubbish bin

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