Report Vertiefung Thesis, Fall 2012
SQL Implementation of the Centroid Decomposition Method

Alessandro De Carli, 10-751-717

December 10, 2012

1 Introduction

In our digital era the annual amount information we produce is growing every year [3] (Hilbert, Martin and López, Priscila 2011). There's a need for fast and scalable algorithms to analyze this data and understand the correlations between the data. The aim of my Vertiefung thesis was to compare the scalability of different implementations of the centroid decomposition method.

The centroid decomposition decomposes a matrix into centroid factors and loading vectors. As described in *The Centroid Decomposition: Relationships between Discrete Variational Decompositions and SVDs* (Moody T. Chu and Robert E. Funderlich 2002) [1] the centroid decomposition is an approximate singular value decomposition. This approximation allows a faster computation than full singular value decomposition. The singular value decomposition allows us to find correlations inside data matrices.

In section 2 I begin with the formal definition of the centroid decomposition. In section 3, I show how the centroid decomposition can be implemented in SQL using two different approaches. In section 4 I compare the different implementations of the centroid decomposition algorithm concerning their execution times. In section 5 I conclude this report with a discussion about further optimizations and thoughts.

2 Definition

The Centroid Decomposition decomposes a given Matrix X into loading vectors b and the according centroid vectors v. The sum of the multiplication of all loading and centroid vectors will result in the initial matrix X:

$$X = \sum_{i=1}^{c} b_i v_i^T \tag{1}$$

c is the number of centroid vectors to compute. In all here mentioned implementations $c = m$, where m is the number of columns in the matrix X.

3 Implementation

3.1 Algorithm

The algorithm of the centroid decomposition method takes as input a matrix X and returns as output a matrix B containing the loading vectors and a matrix V containing the centroid vectors. The algorithm is described in Figure 1

input : X - input matrix $m \times n$
input : c - number of factors
output: B - loading matrix composed of c loading vectors
output: V - centroid factors matrix composed of c factor vectors

```
1  repeat
2      z := 1; // vector of m ones
3      d := diag(XX^T); // diagonal of covariance matrix XX^T
4      w := X^T z;
5      k := 0;
6      repeat
7          e := d − (z · Xw); // · - stands for dot product
8          y := max(e); // maximum element of e
9          k := idx(e, y); // index of maximum element of e
10         if y > 0 then
11             z(k) := −z(k); // change sign of k-th element
12             if z(k) = 1 then
13                 w = w + row(A, k)^T;
                   // row(A,k) - k-th row of matrix A
14             else
15                 w = w − row(A, k)^T;
16         else
17             k := 0;
18     until (k! = 0) ;
19     v := w/||w||;
20     b := Av;
21     col(V, n − c) := v;
22     col(B, n − c) := b;
23     X := X − bv^T;
24 until (c = 0) ;
```

Figure 1: Algorithm of Centroid Decomposition

3.2 Data structure

The first challenge is to represent those matrices in SQL relations. In order to illustrate the different representations let me take the following matrix shown in figure 2 as an example.

$$\begin{matrix} 4 & -7 & 5 \\ 2 & 3 & 1 \\ -1 & 0 & -2 \end{matrix}$$

Figure 2: A matrix with 3 rows and 3 columns

In 3.2.1 and 3.2.2 the matrix shown in figure 2 is represented into relations using two different approaches.

3.2.1 Relational representation

The first idea was to transform the input matrix X to a relation with n tuples of m values as shown in table 1. Since tuples are stored in arbitrary order inside a relational database, this representation needs an index (row_number) to maintain order. This representation is visualized in table 1.

col1	col2	col3	row_number
4	-7	5	1
2	3	1	2
-1	0	-1	3

Table 1: Relational representation of a matrix as a SQL relation

The pros and contras of this representation are:

- Pros
  - Easy to understand
  - Fast INSERT's
- Contras
  - Order of tuples completely arbitrary → needs an order index
  - Cannot use SELECT statements to perform needed matrix calculations → PL/SQL loops needed

This representation is not usable, because the needed matrix calculations cannot be implemented using SQL queries.

3.2.2 Indexed representation

Each value of the matrix has one corresponding tuple in the relation. The matrix is represented with 2 indexes one for the column and one for the row. This representation is visualized in table 2.

val	row_index	column_index
4	1	1
-7	1	2
5	1	3
2	2	1
3	2	2
1	2	3
-1	3	1
0	3	2
-1	3	3

Table 2: Indexed representation of a matrix as a SQL relation

The pros and contras of this representation are:

- Pros:
  - Ability to perform matrix calculations using SELECT's
  - Indices allow fast and easy access to values
- Contras:
  - Uses more storage
  - More INSERT's needed

The advantage that let me choose this representation is that all needed matrix calculations can be implemented with SQL SELECT statements. Both PL/SQL implementations of the centroid decomposition method use this representation.

3.3 Transforming the given dataset

For the scope of this work the dataset was given. Each entry in the dataset has a timestamp (ts) and an id linking the entry to specific series (series_id). The dataset's relation can be seen in table 3.

Not all series have the exact same amount of rows, but the series have the same granularity of timestamp ($t_{i+1} = t_i + 2$). The dataset was incomplete, some series missed several timestamps. Before the computation could start the dataset had to be transformed into a usable input matrix.

series_id	ts	val
100	11034	1.889232
110	2	1.208321
104	40	2.029293
104	120	0.779081

Table 3: Extract of the given dataset

```
INSERT INTO input_matrix(val,series_id,ts) (
  SELECT val,series_id,ts FROM (
    SELECT o.val,r.series_id,r.ts
    FROM observation o RIGHT OUTER JOIN  (
      SELECT n AS ts,m AS series_id FROM (
        SELECT rownum-1 n FROM dual CONNECT BY level-1 <= 10000
      ),
      (
        SELECT rownum m FROM dual CONNECT BY level <= 3
      )
    WHERE MOD(n,2) = 0 AND m > 0) r
    ON o.series_id = r.series_id AND o.ts = r.ts
  )
);
```

The query above returns us a table with series 1, 2 and 3 each with timestamp 0 to 1000, missing values are filled with a NULL due to the OUTER JOIN. The knowledge of the timestamp's granularity and series_id's incremental step is used to efficiently generate an input relation out of the incomplete dataset. Without this knowledge all distinct values for both indexes need to be retrieved, which is a costly operation. The input now has the in 3.2.2 discussed representation. ts is the index for the rows and series_id is the index for the columns.

3.4 Algorithm calculations

The CDECOMP() SQL implementation had the aim to consist out of just SQL queries and loops. In fact only the maximum value of e and its index is stored as a variable. This first SQL implementation did not perform well because of the SELECT's inside the loop that generate a lot of disk IO, if the needed values are not cached. That's why in the optimized SQL implementation CDECOMP_BULK_COLLECT() the aim was to reduce the disk IO and to cache as many needed values as possible. CDECOMP_BULK_COLLECT() is much faster than its predecessor as can be seen in figure 6. The disadvantage is that it's implemented using Oracle's BULK COLLECT, no other database system provides this operation, making the implementation an Oracle dependent solution. Cursors are not an option because they don't perform as good in loops [2] (Feuerstein 2012).

To show the difference of the two SQL implementations let me compare some parts of the implementations:

- In line 7 (figure 1) we calculate the e vector, which includes a matrix multiplication and a vector subtraction.
  - In the CDECOMP() implementation this calculation is accomplished by the following SQL query:

    ```
    SELECT d.val-zxw.val as val, zxw.ts
    FROM d_vector d,(
      SELECT xw.val*z.val as val,xw.ts as ts
      FROM (
        SELECT sum(w.val*i.val) as val, i.ts
        FROM w_vector w,input_matrix i
        WHERE w.series_id=i.series_id
        GROUP BY i.ts
      ) xw, z_vector z
    WHERE z.ts=xw.ts
    ) zwx
    WHERE d.ts = zxw.ts
    ```

    The multiplication of the X with w is performed by grouping by the row index (ts) and joining by the column index (series_id). A join and a value-by-value subtraction make the vector's subtraction. The result is a relation representing a vector, each row is indexed with a series_id.
  - CDECOMP_BULK_COLLECT() solves the same problem with this PL/SQL snippet:

    ```
    max_value := -999999999;
    max_value_index := -1;

    FOR k in 0..(xw_var.count-1) LOOP
      xw_var(k+1):=0;
    END LOOP;
    FOR k in 0..(values_count-1) LOOP
      xw_var(TRUNC(k/(column_count)+1))
        := xw_var(TRUNC(k/(column_count))+1)
          + (x_var(k+1)*w_var(MOD(k,column_count)+1));
    END LOOP;
    FOR k in 0..(xw_var.count-1) LOOP
      IF max_value < (d_var(k+1) - (z_var(k+1)*xw_var(k+1))) THEN
        max_value := d_var(k+1) - (z_var(k+1)*xw_var(k+1));
        max_value_index := k+1;
      END IF;
    END LOOP;
    ```

The CDECOMP_BULK_COLLECT() implementation performs the multiplication using MOD(k,column_count)+1) and TRUNC(k/(column_count)+1) to get the appropriate values while looping through the values of X. The values of X are bulk collected inside x_var in a previous part of the implementation. The vector subtraction is made while searching the maximum value. Since we are only interested in the maximum value of e there is no need to store the whole e vector.

- In line 13 and 15 (figure 1) we calculate the w vector needed for the next iteration.
  - The CDECOMP() implementation performs this calculation with following SQL query:

    ```
    SELECT w.val+z.val*ir.val AS val, w.series_id AS series_id
    FROM w_vector w,(
      SELECT val, series_id
      FROM input_matrix
      WHERE ts = ‘‘index of maximum e_vector value’’
    ) ir, z_vector z
    WHERE w.series_id = ir.series_id
    AND z.ts = ‘‘index of maximum e_vector value’’
    ```

    X's row holding the index of maximum e_vector value is added or subtracted to w's value depending on z's value on that index. A join on series_id allows us to calculate the new value row-by-row.
  - CDECOMP_BULK_COLLECT() solves the same problem with this PL/SQL snippet:

    ```
    FOR k in 0..(column_count-1) LOOP
      w_var(k+1)
        := w_var(k+1)+(2*z_var(max_value_index)
          *x_var(((max_value_index-1)*column_count)+1+k));
    END LOOP;
    ```

    The multiplication of max_value_index with column_count allow us to navigate directly to the first value of the needed row of X.

As we can see in the first implementation the data representation in indexes is a great support. Matrix multiplications can be easily implemented with JOIN's and GROUP BY's. The second implementation exploits the data representation, in the way that it stores only an ordered list of the values. By knowing the number of columns and rows we can treat this 1-dimensional list as a 2-dimensional matrix.

4 Comparison of implementations

This section provides an empirical comparison between the 3 different implementations. The Java implementation was provided by the supervisor of this Vertiefung and computes the calculations in memory. In both SQL implementations the Oracle server handles memory management.

4.1 Runtime complexity

The centroid decomposition method has a complexity of $O(kn^2)$ for a rank-k approximation (Chu, Moody T. and Funderlic, Robert E. 2002) [1]. The here mentioned implementations set k to be the number of columns $(m) \rightarrow O(mn^2)$. This means that the execution time should grow linearly when increasing m and quadraticaly when increasing n.

4.2 Running time

All results were measured on *horatio* as SQL server and *econbase* as client. This means *econbase* computed the Java implementation and *horatio* both SQL implementations. Both machines run in the same (UZH) network, which allowed me to reduce the network delay. The *econbase* machine has more computing power than the *horatio* machine. Both machines are running on the same amount of memory (4 GB).

In the algorithm (figure 1) there are 3 variable points to run an empirical comparison:

- The number of rows X has (n)
- The number of columns X has (m)
- The number of centroid vectors we want to calculate (c)

The implementations set the number of centroid vectors to be equal to the number of columns the X has $(c = m)$. In the following graphs the measured time is plotted in milliseconds. Each point represents the average of at least 4 observations taken from independent executions.

Because I was not the only user of the Oracle server, every computation was executed multiple times, by doing this the probability of getting “wrong” times because of the load can be reduced.

4.2.1 Changing the number of rows

The number of columns is fixed to 2. The number of rows is doubled on every step $(t_{i+1} = 2t_i)$ until reaching a maximum of 32768 rows.

Figure 3 shows us that with 4096 rows the CDECOMP() SQL implementation takes around 4 minutes. Because of this long time the CDECOMP() SQL implementation has been excluded in the next plots.

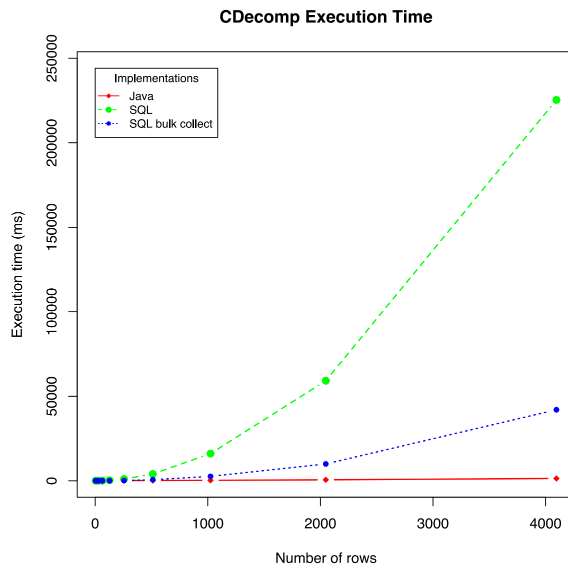

Figure 3: An overview of all implementation's execution times

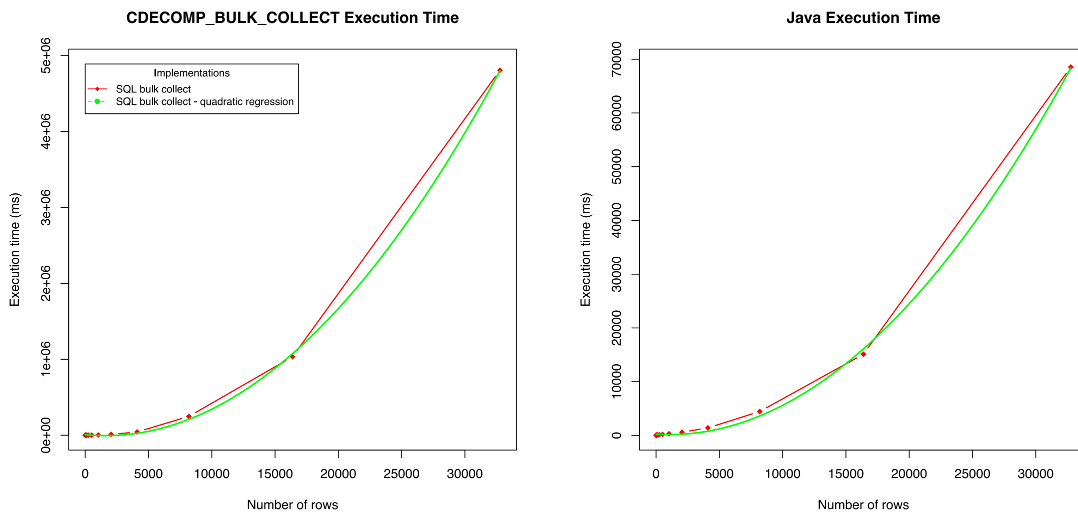

Figure 4: Execution times in detail with quadratic regression

The Java implementation seemed to be linear in the first plots (figure 3), after adapting the axis we can see that it is quadratic as well (right graph figure 4). Both graphs show that the algorithms fit nicely to a quadratic regression, the significance measured

by R^2 is almost 1 (table 4).

implementation	function	adjusted R^2
Java	$y = 54.32 + 0.1617x + 0.00004533x^2$	0.9999
SQL bulk collect	$y = 256 - 3.452x + 0.004086x^2$	0.9999

Table 4: Significance of the regression functions shown in figure 4

The regression functions shown in table 4 allow us to estimate how long the calculation takes for any given row number (n). The runtime complexity $O(mn^2)$ mentioned in 4.1 indicates a quadratic function of n, this has been confirmed by the results shown in table 4.

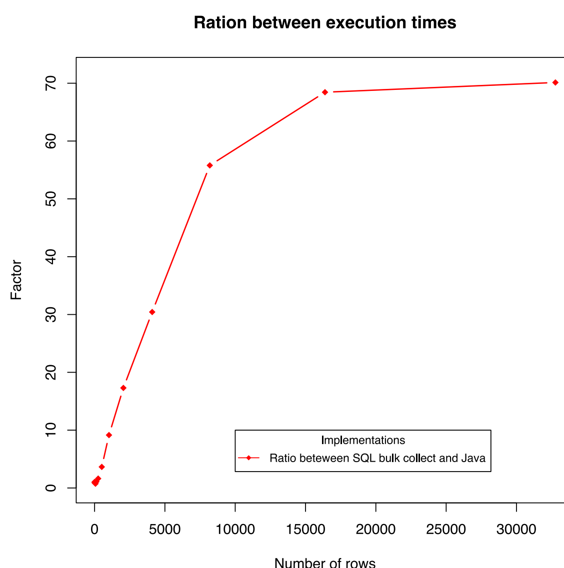

Figure 5: Ratio of java over bulk collect's execution time (e.g. factor 3 means the java implementation is 3 times faster)

One way to compare the implementation's execution times is by computing the ratio between the two. Figure 5 shows us by which factor the Java implementation is faster than the bulk collect SQL implementation.

4.2.2 Changing the number of columns

The number of rows is fixed to 1024. The number of columns is increased by 1 on every step ($t_{i+1} = t_i + 1$) until reaching a maximum of 100 columns.

The CDECOMP() implementation has been excluded from these results, because increasing the number of columns increases the number of iterations (from line 1 to line 24)

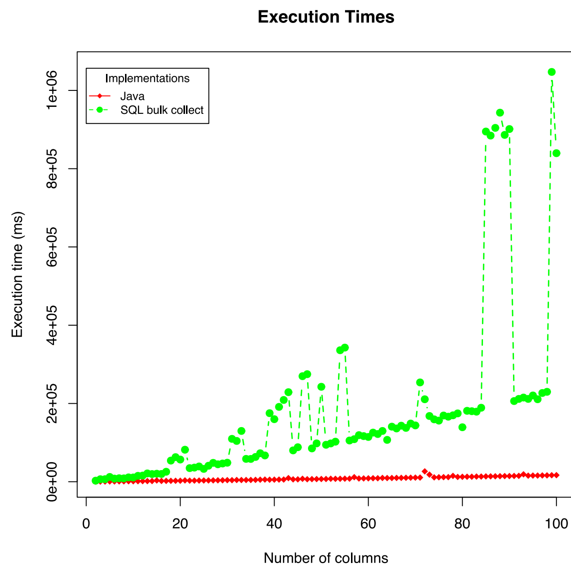

Figure 6: An overview of both implementation's execution times.

and takes to much time to execute. Figure 6 shows that the Java implementation performs a lot better, even when increasing the number of columns. Both implementations show linear trends.

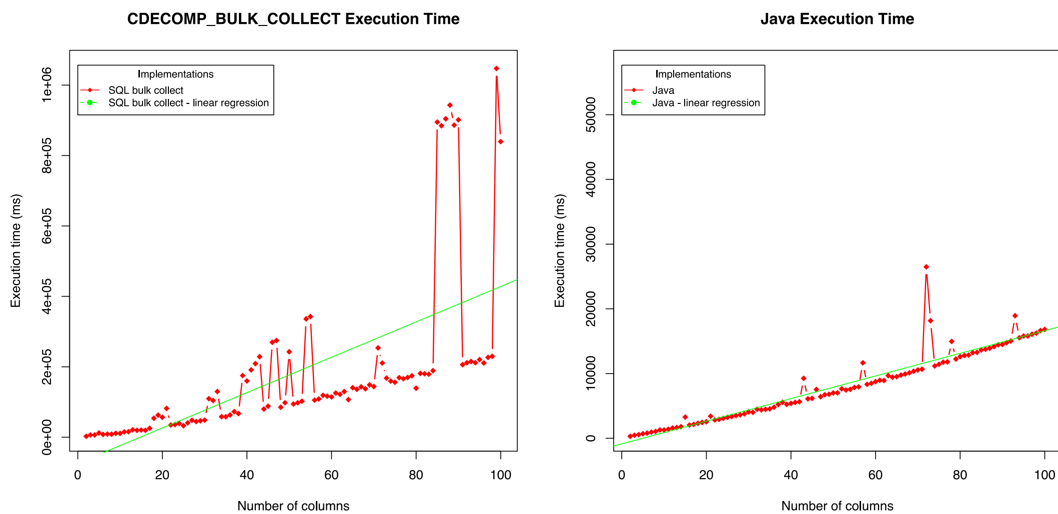

Figure 7: Execution times in detail with linear regression

In figure 7 both graphs show some points not fitting the linear trend. When comparing table 5 with table 4 we see that we have less significant regression functions. This makes the execution times less predictable.

implementation	function	adjusted R^2
Java	$y = -870.310 - 174.935x$	0.8835
SQL bulk collect	$y = -74116 + 5020x$	0.3866

Table 5: Significance of the regression functions shown in figure 7

As already mentioned some points do not fit the linear trend. When changing from the average execution time to the minimal execution time those points persist. Even with those points a linear trend can be seen. Since we increase m this corresponds to the runtime complexity of $O(mn^2)$.

5 Conclusion

Even though the optimized SQL implementation is a lot faster than its predecessor, it's not scalable. Having to wait 81 minutes for the computation of 32768 rows $\times$ 2 columns is not acceptable. The Java implementation performs a lot better in this case with just 1 minute for the same computation. All implementations fit to a quadratic regression, which does not play in favor of scalability. It's nice to see how the theoretical complexity shows up in the results.

The reason for the slow SQL performance is not due to the queries per-se, but because the queries are executed inside loops. This generates a lot of Disk IO as was shown by the results of comparing the optimized SQL implementation with the first one. It's unclear whether BULK COLLECT without LIMIT's stores the whole data in the main memory, but it certainly allows us to access the whole collected dataset faster than when retrieving it via SQL query (figure 3).

The Java implementation computes everything directly in the main memory. This makes it run out of memory quickly. On the other hand this approach reduces the number of IO's and makes it very fast. The speed is acceptable even for larger datasets. One way to improve this algorithm's scalability would be to reduce memory consumption. This is definitely possible as could be noticed in the SQL implementation.

I assume that a very good way to improve this algorithm's performance would be to run all matrix computations on hardware that is designed for matrix computations. More specifically on graphic cards. Graphic cards deal very efficiently with matrix calculations [4] (Larsen, E. Scott and McAllister, David 2001). It would be very interesting to run the same empirical comparisons with a graphic card based algorithm.

Another optimization approach would be to distribute the computation (between line 6 and 18), which is not a trivial task because each iteration is dependent from the previous one.

Bibliography

[1] Moody T. Chu and Robert E. Funderlic. The centroid decomposition: Relationships between discrete variational decompositions and SVDs. *SIAM Journal on Matrix Analysis and Applications*, 23(4):1025–1044, January 2002.

[2] Steven Feuerstein. On cursor FOR loops. http://www.oracle.com/technetwork/issue-archive/2008/08-nov/o68plsql-088608.html.

[3] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, April 2011.

[4] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, page 55–55, New York, NY, USA, 2001. ACM.