An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

Timofey Rechkalov and Mikhail Zymbler
South Ural State University, Chelyabinsk, Russia
trechkalov@yandex.ru, mzym@susu.ru

This work was financially supported by the Russian Foundation for Basic Research (grant No. 17-07-00463), by Act 211 Government of the Russian Federation (contract No. 02.A03.21.0011) and by the Ministry of education and science of Russian Federation (government order 1.9624.2017/7.8).

Authors thank RSC Group (Moscow, Russia) for the Intel Xeon Phi Knights Landing for experimental evaluation.
Data mining outside vs. inside DBMS

An approach to data mining inside PostgreSQL based on parallel implementation of UDFs
An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

Data mining inside DBMS

<table>
<thead>
<tr>
<th>Points</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>dim=2</td>
<td>k=5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

procedure Clustering (  
  inpTable text, -- Input table name  
  dim int, -- # of columns  
  k int, -- # of clusters  
  Eps real -- Accuracy  
) return text  
begin  
  ...  
end

sql> exec Clustering('Points', 2, 5, 0.001);
Data mining inside PostgreSQL

#include <libpq-fe.h> // API of PostgreSQL
#include "pgmining.h" // API of pgMining library

void main (void)
{
    char *inpTable = "Points"; // Table with data to be mined
    int dim = 3;                // Number of columns
    int k = 5;                  // Number of clusters
    float Eps = 0.001;          // Accuracy
    char *outTable = "Clusters"; // Table to save mining results

    // Connect to server
    char *conninfo = "user=postgres port=5432 host=localhost";
    PGconn *conn = PQconnectdb (conninfo);

    // Call mining UDF
    pgClustering (conn, inpTable, dim, k, Eps, outTable);

    PQexec (conn, strcat ("SELECT * FROM ", outTable)); // Show results
    PQfinish (conn); // Cleanup
}
What mining UDF encapsulates

- Parallel implementation by OpenMP

```c
#pragma omp parallel for
for (i=0; i<n; i++) {
    ...
    ...
}
```

fork

join
An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

## What mining UDF encapsulates

... for Intel MIC (Many Integrated Core) platforms

<table>
<thead>
<tr>
<th>Feature</th>
<th>Device</th>
<th>Intel Xeon X5680</th>
<th>Intel Xeon Phi, Knights Corner SE10X</th>
<th>Intel Xeon Phi, Knights Landing 7250</th>
</tr>
</thead>
<tbody>
<tr>
<td># of physical cores</td>
<td></td>
<td>6</td>
<td>61</td>
<td>68</td>
</tr>
<tr>
<td>Hyper threading factor</td>
<td></td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td># of logical cores</td>
<td></td>
<td>12</td>
<td>244</td>
<td>272</td>
</tr>
<tr>
<td>Frequency, GHz</td>
<td></td>
<td>3.33</td>
<td>1.1</td>
<td>1.4</td>
</tr>
<tr>
<td>Vector processing unit</td>
<td>No</td>
<td>512 bit</td>
<td>512 bit</td>
<td></td>
</tr>
<tr>
<td>Bootable</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Peak performance, TFLOPS</td>
<td>0.371</td>
<td>1.076</td>
<td>3.046</td>
<td></td>
</tr>
</tbody>
</table>
What mining UDF encapsulates

- Using cache of pre-computed mining structures
  - E.g. distance matrix to perform clustering

\[
\begin{array}{cccc}
0 & 1 & \ldots & n-1 \\
0 & \text{dist}(a_0, a_1) & \ldots & \text{dist}(a_0, a_{n-1}) \\
1 & 0 & \text{dist}(a_1, a_j) & \text{dist}(a_1, a_{n-1}) \\
\ldots & \ldots & \ldots & \ldots \\
n-1 & 0 & \text{dist}(a_{n-1}, a_{n-1}) & 0 \\
\end{array}
\]
How mining UDF is designed

- pgClustering
- wrapClustering
- micClustering

End-user library function

Wrapper of PostgreSQL UDF

Parallel mining on Intel MIC (Many Integrated Core)
Module structure

An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

- Library of end-user functions
- PostgreSQL Server Programming Interface
- pgMining
  - Frontend
  - Backend
  - Wrapper
  - Cache mgr
- Wrappers for UDFs
- Cache of pre-computed mining structures
- Parallel mining functions on Intel MIC

- PostgreSQL
  - SPI
- mcMining
  - Clustering
    - PAM
    - k-Means
  - Classification
    - kNN
  - Patterns
    - Apriori
End-user library function

```c
int pgClustering ( PGconn * conn, // ID of PostgreSQL connection char * inpTable, // Name of input table int dim, // Number of coordinates in data point int k, // Number of clusters float Eps, // Accuracy char * outTable) // Name of output table
{
    // Register UDF
    PQexec (conn, "CREATE OR REPLACE FUNCTION wrapClustering (text, integer, integer, real) RETURNS text AS 'pgmining ', 'wrapClustering' LANGUAGE C STRICT;"
    );
    // Create resulting table
    PQexec (conn, "CREATE %s TABLE IF NOT EXISTS (data text)", outTable);
    // Execute UDF
    return PQexec (conn, "INSERT INTO %s
        SELECT wrapClustering (%s, %d, %d, %f);", outTable, inpTable, dim, k, Eps);
}
```
Datum \texttt{wrapClustering}(PG\_FUNCTION\_ARGS)

\{
    // Extract input parameters of clustering \\
    // from the PostgreSQL parameters \\
    char* inpTable = text\_to\_cstring(PG\_GETARG\_TEXT\_P(0));
    int dimension = PG\_GETARG\_INT32(1);
    int k = PG\_GETARG\_INT32(2);
    float Eps = PG\_GETARG\_FLOAT4(3);
    int N;

    // Check if we have pre-calculated mining structures \\
    // in our cache, else calculate (in parallel) and load it \\
    void* distMatr = cache\_getObject(strcat(inpTable, "\_distMatrix"));
    if (distMatr == NULL) {
        // Check if input table is in our cache \\
        void* inpData = cache\_Get(inpTable);
        if (inpData == NULL) {
            // Allocate memory and load input table to cache \\
            inpData = (float4*)palloc(dimension \* sizeof(float4));
            wrap\_Tab\_Read(inpData, inpTable, dim, &N);
            cache\_Put(inpTable, inpData, sizeof(inpData));
        }
        distMatr = mic\_Calc\_Dist\_Matr(inpData, dim, N);
        cache\_PutObject(strcat(inpTable, "\_distMatrix"), distMatr, sizeof(distMatr));
    }

    // Cluster data (in parallel) using mining structures \\
    // and save results in output table \\
    mic\_Clustering\_Res* outData = mic\_Clustering\_Res\_Create();
    mic\_Clustering(N, k, Eps, outData, distMatr);
    PG\_RETURN\_TEXT(data\_2\_String(outData));
\}
Datum `wrapClustering` (PG_FUNCTION_ARGS) {

    // Extract input parameters of clustering
    // from the PostgreSQL parameters
    char * inpTable = text_to_cstring (PG_GETARG_TEXT_P (0));
    int dim = PG_GETARG_INT32 (1);
    int k = PG_GETARG_INT32 (2);
    float Eps = PG_GETARG_FLOAT4 (3);
    int N;

    // Check if we can use pre-calculated mining structures
    void * distMatr = cacheGet (strcat (inpTable, "_distMatr"));
    if (distMatr == NULL) {
        // Check if input table is in our cache
        void * inpData = cacheGet (inpTable);
        if (inpData == NULL) {
            // Allocate memory and load input table to cache
            inpData = (float4 *) palloc (dim * sizeof (float4));
            wrapTabRead (inpData, inpTable, dim, &N);
            cachePut (inpTable, inpData, sizeof (inpData));
        }
        distMatr = micCalcDistMatr (inpData, dim, N);
        cachePut (strcat (inpTable, "_distMatr"), distMatr, sizeof (distMatr));
    }

    micClustering_res * outData = micClustering_resCreate ();
    micClustering (N, k, Eps, outData, distMatr); // Perform clustering
    PG_RETURN_TEXT (data2String(outData)); // Write results to the output table
}
Datum `wrapClustering` (PG_FUNCTION_ARGS) 
{
    char * inpTable = text_to_cstring (PG_GETARG_TEXT_P (0));
    int dim = PG_GETARG_INT32 (1);
    int k = PG_GETARG_INT32 (2);
    float Eps = PG_GETARG_FLOAT4 (3);
    int N;

    // Check if we can use pre-calculated distance matrix
    void * distMatr = cacheGet (strcat (inpTable, "_distMatr"));
    if (distMatr == NULL) {
        // Check if input table is in our cache and load if not
        void * inpData = cacheGet (inpTable);
        if (inpData == NULL) {
            inpData = (float4 *) palloc (dim * sizeof (float4));
            wrapTabRead (inpData, inpTable, dim, &N);
            cachePut (inpTable, inpData, sizeof (inpData));
        }

        // Calculate distance matrix in parallel and load it to cache
        distMatr = micCalcDistMatr (inpData, dim, N);
        cachePut (strcat (inpTable, "_distMatr"), distMatr,
                  sizeof (distMatr));
    }

    micClusteringRes * outData = micClusteringResCreate ();
    micClustering (N, k, Eps, outData, distMatr); // Perform clustering
    PG_RETURN_TEXT (data2String(outData)); // Write results to the output table
}
Datum **wrapClustering** (PG_FUNCTION_ARGS)
{
    // Extract input parameters
    char * inpTable = text_to_cstring (PG_GETARG_TEXT_P (0));
    int dim = PG_GETARG_INT32 (1);
    int k = PG_GETARG_INT32 (2);
    float Eps = PG_GETARG_FLOAT4 (3);
    int N;
    // Check if we can use pre-calculated distance matrix
    void * distMatr = cacheGet (strcat (inpTable, "_distMatr"));
    if (distMatr == NULL) {
        // Check if input table is in our cache and load if not
        void * inpData = cacheGet (inpTable);
        if (inpData == NULL) {
            inpData = (float4 *) palloc (dim * sizeof (float4));
            wrapTabRead (inpData, inpTable, dim, &N);
            cachePut (inpTable, inpData, sizeof (inpData));
        }
        // Calculate distance matrix in parallel and load it to cache
        distMatr = micCalcDistMatr (inpData, dim, N);
        cachePut (strcat (inpTable, "_distMatr"), distMatr, sizeof (distMatr));
    }
    // Cluster data (in parallel) using mining structures
    micClusteringRes * outData = micClusteringResCreate ();
    micClustering (N, k, Eps, outData, distMatr);
    // Save results in output table
    PG_RETURN_TEXT (data2String(outData));
}
Methods of Cache manager

- `void * cacheGet(char * objName)`
  
  // Searches for a specified object in the cache.
  // Updates internal statistics of the mined object
  // (number of calls, timestamp of recent call, etc.).

- `int cachePut(char * objName)`
  
  // Loads a specified object into the cache.
  // Pops out a victim object if it is not enough space in the cache
  // (according to a cache management policy,
  // e.g. Least Recently Used, Least Frequently Used, etc.).
PAM: Partition Around Medoids

- **Goal**
  - Organize $n$ objects of data set in $k$ clusters

- **Key idea**
  - Cluster centers are chosen as objects of data set (*medoids*)

- **Method**
  - **BUILD phase**
    - Initially choose cluster centers
  - **SWAP phase**
    - Iteratively move objects across clusters to improve value of an objective function
void calcDistMatr (const float* rowData, const float* colData, float* distances, const int n, const int pointWidth){
    const int vecLen = 32;
    __attribute__((aligned(64)))
    __attribute__((aligned(64)))
    #pragma omp parallel
    {
        float point[pointWidth];
        float result[vecLen];
        #pragma omp for
        for(int i=0; i<n; ++i){
            point[] = rowData[i*pointWidth:pointWidth];
            for(int ii = 0; ii < n; ii += vecLen){
                result[] = 0;
                for(int j=0; j < pointWidth; ++j){
                    const float* restrict point2 = colData+i*pointWidth;
                    result[] += (point[j]-point2[j*vecLen:vecLen])* (point[j]-point2[j*vecLen:vecLen]);
                }
                distances[i*n+ii:vecLen] = sqrtf(result[]);
            }
        }
    }
}

OpenMP
Rewriting loops
Loop tiling
Data alignment
Experimental evaluation

- Clustering algorithm: PAM (Partition Around Medoids)
  - Represents cluster centers as points of input data set (medoids)
  - \textit{CALCULATION} of distance matrix
  - \textit{BUILD phase}: initial clustering by the successive selection of medoids
  - \textit{SWAP phase}: improving clustering according to an objective function

- Datasets

<table>
<thead>
<tr>
<th>Name</th>
<th>dim</th>
<th># of clusters</th>
<th># of points, $\times 2^{10}$</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Census</td>
<td>67</td>
<td>10</td>
<td>35</td>
<td>Population surveys by the US Census Bureau</td>
</tr>
<tr>
<td>MixSim</td>
<td>5</td>
<td>10</td>
<td>35</td>
<td>Generator of synthetic datasets for evaluation of clustering algorithms</td>
</tr>
<tr>
<td>Power</td>
<td>3</td>
<td>10</td>
<td>35</td>
<td>Individual household electricity consumption</td>
</tr>
<tr>
<td>FCS Human</td>
<td>423</td>
<td>10</td>
<td>18</td>
<td>Aggregated human gene information</td>
</tr>
</tbody>
</table>

- Competitor: PAM algorithm of R data mining package
Performance: Census dataset

- Serial R PAM
- Serial pgPAM (2xXeon, 1 thread)
- Parallel pgPAM (2xXeon, 24 threads)
- Parallel pgPAM (2xXeon, 24 threads) + cache
- Parallel pgPAM (KNC, 240 threads) + cache
- Parallel pgPAM (KNL, 272 threads) + cache

- 2xXeon, 24 threads
- Prepare Distance Matrix
- BUILD phase
- SWAP phase

- KNC, 240 threads
- Prepare Distance Matrix
- BUILD phase
- SWAP phase

- KNL, 272 threads
- Prepare Distance Matrix
- BUILD phase
- SWAP phase
Performance: MixSim dataset

- **Serial R PAM**
- **Serial pgPAM (2xXeon, 1 thread)**
- **Parallel pgPAM (2xXeon, 24 threads)**
- **Parallel pgPAM (2xXeon, 24 threads) + cache**
- **Parallel pgPAM (KNC, 240 threads) + cache**
- **Parallel pgPAM (KNL, 272 threads) + cache**

- **2xXeon, 24 threads**
  - Prepare Distance Matrix
  - BUILD phase
  - SWAP phase

- **KNC, 240 threads**
  - Prepare Distance Matrix
  - BUILD phase
  - SWAP phase

- **KNL, 272 threads**
  - Prepare Distance Matrix
  - BUILD phase
  - SWAP phase
Performance: Power dataset

The graphs illustrate the runtime (in sec.) of different implementations of UDFs for the Power dataset. The x-axis represents the number of data points (x1024), and the y-axis represents the runtime (log scale). The graphs compare serial and parallel implementations of R-PAM and pgPAM on different hardware configurations.

- Serial R-PAM
- Serial pgPAM (2xXeon, 1 thread)
- Parallel pgPAM (2xXeon, 24 threads)
- Parallel pgPAM (2xXeon, 24 threads) + cache
- Parallel pgPAM (KNC, 240 threads) + cache
- Parallel pgPAM (KNL, 272 threads) + cache

The graphs show that parallel implementations generally have lower runtimes compared to serial implementations, especially as the number of data points increases. The 2xXeon configuration with 24 threads consistently has the lowest runtime across all data points, indicating efficient parallelization.

The graphs also highlight the importance of optimizing cache usage, as indicated by the comparison between parallel implementations with and without cache optimization.
Performance: FCS Human dataset

![Diagram showing performance comparison for different data points and configurations.]
Speedup and Efficiency: $2 \times$ Xeon, 24 threads

![Graph showing speedup and efficiency](image-url)
An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

Speedup and Efficiency: KNC, 240 threads
An approach to data mining inside PostgreSQL based on parallel implementation of UDFs

Speedup and Efficiency: KNL, 272 threads

![Graph showing speedup and efficiency for different thread counts and workloads. The graph compares physical and logical cores for four datasets: Census, Mixsim, Power, and FCS Human.](image-url)

- **Physical cores** vs. **Logical cores**
  - **Speedup** on the left y-axis
  - **Efficiency, %** on the right y-axis

Each dataset has a distinct line representing its performance across different thread counts (1, 34, 68, 136, 204, 272 threads).
Conclusion

- We have proposed an approach to data mining inside DBMS based on parallel implementation of UDFs for many-core platforms.
- We have implemented the approach for PostgreSQL, Intel Xeon Phi (KNC and KNL), and PAM clustering algorithm.
- We have conducted experiments on synthetic and real data sets that show good scalability of our approach and overtaking analogue (R data mining package).

Thank you for paying attention! Questions?

Mikhail Zymbler
mzym@susu.ru
Impact of vectorization

Census

MixSim

Power

FCS Human
Performance of load data into RAM

- C CSV loading
- PostgreSQL data loading
- R CSV loading

Runtime, sec.

# data points, x1024

An approach to data mining inside PostgreSQL based on parallel implementation of UDFs
PAM: Partition Around Medoids

- **Objective function**
  \[ E = \sum_{j=1}^{n} \min_{1 \leq i \leq k} \text{dist}(c_i, o_j) \]

  where \( c_i \) is the medoid, \( o_j \) is the clustered object, \( \text{dist} \) is the distance metric

- **BUILD phase:** complexity is \( O(kn^2) \)

- **SWAP phase:** complexity is \( O(k(n - k)^2) \) per iteration
PAM: pseudocode

Input: set of objects \( O \), # of clusters \( k \)
Output: set of clusters \( C \)

// Calculation of distance matrix
distMatr ← calcDistMatr (O)

// BUILD phase
C ← BuildMedoids (distMatr)

repeat // SWAP phase
    \( T_{\text{min}} \) ← findBestSwap (M, C)
    Swap \( (c_{\text{min}}, o_{\text{min}}) \) for \( T_{\text{min}} \)
until \( T_{\text{min}} < 0 \)
Rewriting loops: an example

- Vector processor unit is of 512 bits
  - 16 float elements or 8 double elements
- Rewriting loops provides vectorization of computations (SIMD, Single Instruction Multiple Data)

Scalar loop

```c
for(i = 0; i < n; ++i)
a[i] = b[i] + c[i];
```

SIMD loop

```c
#ifdef DOUBLE_PRECISION
    #define LEN 8
#else
    #define LEN 16
#endif
for(i = 0; i < n; i += LEN)
a[i:LEN] = b[i:LEN] + c[i:LEN];
```
Loop tiling: an example

Without tiling

With tiling