Integrating DBMS and Parallel Data Mining Algorithms for Modern Many-Core Processors

Timofey Rechkalov and Mikhail Zymbler(✉)

South Ural State University, Chelyabinsk, Russia
trechkalov@yandex.ru, mzym@susu.ru

Abstract. Relational DBMSs (RDBMSs) remain the most popular tool for processing structured data in data intensive domains. However, most of stand-alone data mining packages process flat files outside a RDBMS. In-database data mining avoids export-import data/results bottleneck as opposed to use stand-alone mining packages and keeps all the benefits provided by a RDBMS. The paper presents an approach to data mining inside a RDBMS based on a parallel implementation of user-defined functions (UDFs). Such an approach is implemented for PostgreSQL and modern Intel MIC (Many Integrated Core) architecture. The UDF performs a single mining task on data from the specified table and produces a resulting table. The UDF is organized as a wrapper of an appropriate mining algorithm, which is implemented in C language and is parallelized by the OpenMP technology and thread-level parallelism. The heavy-weight parts of the algorithm are additionally parallelized by intrinsic functions for MIC platforms to reach the optimal loop vectorization manually. The library of such UDFs supports a cache of precomputed mining structures to reduce costs of further computations. In the experiments, the proposed approach shows good scalability and overtakes *R* data mining package.

Keywords: Data mining · In-database analytics · PostgreSQL
Clustering · Partition Around Medoids (PAM) · Thread-level parallelism
OpenMP · Intel Xeon Phi

1 Introduction

Nowadays relational DBMSs (RDBMSs) remain the most widely used tool for processing structured data in data intensive domains (e.g. finance, medicine, physics, etc.). Meanwhile, most of data mining algorithms deal with flat files. In data intensive domains, exporting of data sets and importing of mining results inhibit analysis of large databases outside a RDBMS [18]. Data mining inside a RDBMS avoids export-import bottleneck and provides the end-user with all the built-in RDBMS's services (query optimization, data consistency and security, etc.).

Approaches to integrating data mining with RDBMSs include special data mining languages and SQL extensions, SQL implementation of mining algorithms and user-defined functions (UDFs) implemented in high-level programming language. In order to increase performance of data analysis, the latter could serve as a subject of applying parallel processing on modern many-core platforms.

L. Kalinichenko et al. (Eds.): DAMDID/RCDL 2017, CCIS 822, pp. 230–245, 2018.
https://doi.org/10.1007/978-3-319-96553-6_17

In [25], we presented an approach to data mining inside the PostgreSQL open-source DBMS exploiting capabilities of modern Intel MIC (Many Integrated Core) [1] platform. The mining UDF is organized as a wrapper of an appropriate algorithm, which is implemented in C language and is parallelized for Intel MIC platform by OpenMP technology and thread-level parallelism. We took Partition Around Medoids (PAM) clustering algorithm [9] and wrapped its parallel implementation for Intel MIC proposed in [24]. Our experiments on the platforms of Intel Xeon CPU and Intel Xeon Phi, Knights Corner (KNC) generation, showed an efficiency of the proposed approach.

In this paper, we give a more detailed description of our approach and extend the study mentioned above as follows. We enhance parallel PAM by accelerating its step of distance matrix computation and conduct additional experiments on Intel Xeon Phi, Knights Landing (KNL), which is the second-generation MIC architecture product from Intel.

The rest of the paper is organized as follows. In Sect. 2, we discuss related works. The proposed approach is described in Sect. 3. We give the results of experimental evaluation of our approach in Sect. 4. Section 5 concludes the paper.

2 Related Work

The problem of integrating data analytics with relational DBMSs has been studied since data mining research originates. Early developments considered data mining query languages [5, 7] and implementation of data mining functionality in SQL, e.g. clustering algorithms [15, 17], association rules [23, 27], classification [20, 26], and graph mining [4, 21].

In [18], authors proposed integration of correlation, linear regression, PCA and clustering into the Teradata DBMS based on UDFs. In [19], it was shown that UDFs implementing common vector operations are as efficient as automatically generated SQL queries with arithmetic expressions, and queries calling scalar UDFs are significantly more efficient than equivalent queries using SQL aggregations. In [8], a technique for execution of aggregate UDFs based on data parallelism was proposed, which will be embodied later in Teradata DBMS.

The MADlib library [6] provides many methods for supervised learning, unsupervised learning and descriptive statistics for PostgreSQL. The MADlib exploits UDAs, UDFs, and a sparse matrix C library to provide efficient representations on disk and in memory. Since many statistical methods are iterative, authors wrote a driver UDF in Python to control iteration in such a way that all large data movement is done within the database engine and its buffer pool.

The Bismarck system [3] provides a unified architecture for in-database analytics, facilitating UDFs as a convenient interface for the analyst to describe their desired analytics models. This development is based on incremental gradient descent (IGD), which is a general technique to solve a large class of analytical models expressed as convex optimization problem (e.g. logistic regression, support vector machines, etc.). Authors showed that IGD has a data access pattern identical to the UDA access pattern and provided a UDA-based implementation for the analytical problems mentioned above.

The DAnA [12] system automatically maps a high-level specification of in-database analytics queries to the FPGA accelerator. The accelerator implementation is generated from an UDF, expressed as part of a SQL query in a Python-embedded Domain-Specific Language. In order to implement efficient in-database integration, DAnA-generated accelerators contain a special hardware structure, Striders, that directly interface with the buffer pool of the database. The Striders extract, cleanse, and process the training data tuples, which are consumed by a multi-threaded FPGA engine that executes the analytics algorithm.

In this paper, we suggest an approach to embedding data mining functions into PostgreSQL. As methods mentioned above, our approach exploits UDFs. The implementation of those systems, however, involves a combination of SQL, UDFs, and driver programs written in other languages, so the systems could become obscure and relatively difficult to maintain. Our approach assumes parallelization of UDFs for Intel many-core platform that current RDBMS is running on and hiding the parallelization details from the RDBMS. This make it possible to port our approach to some other open-source RDBMS with possible non-trivial but mechanical software development effort. Additionally, there is a special module in our approach, which provides a cache of precomputed mining structures and allows UDF to reuse these structures in order to reduce costs of computations.

3 Data Mining Inside PostgreSQL Using Intel MIC

3.1 Key Ideas

The goal of our approach is to provide a database application programmer with the library of data mining functions, which could be run inside a DBMS as it is shown in Fig. 1. In this example, the *pgPAM* function applies the Partition Around Medoids (PAM) clustering algorithm [9] to data points from the specified input table and saves results in the output table (for the specified number of the input table columns and number of clusters). An application programmer is not obliged to export input data

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

void main (void)
{
  char * inpTab = "points";
  char * outTab = "clusters";
  int dim = 3;
  int k = 5;
  char * conninfo="user=postgres port=5432 host=localhost";

  PGconn * conn = PQconnectdb(conninfo);
  pgPAM(conn, inpTab, dim, k, outTab);
  PQexec(conn, strcat("SELECT * FROM ", outTab));
  PQfinish(conn);
}
```

Fig. 1. An example of using data mining function inside PostgreSQL

from PostgreSQL and import mining results back. At the same time, PAM encapsulates parallel implementation [24] based on OpenMP and thread-level parallelism.

Implementation of such an approach is based on the following ideas. A data mining algorithm is implemented with C language and parallelized for Intel MIC by OpenMP technology. Next, we cover the parallel mining function by two wrappers, namely a system-level wrapper and a user-level wrapper. The *user-level wrapper* registers the system-level wrapper in the database schema, connects to the database server and calls system-level wrapper. The *system-level wrapper* is an UDF, which parses parameters of the user-level wrapper, calls the parallel mining function and saves the results in the table(s).

3.2 Component Structure

Figure 2 depicts the component structure of our approach being applied to PostgreSQL.

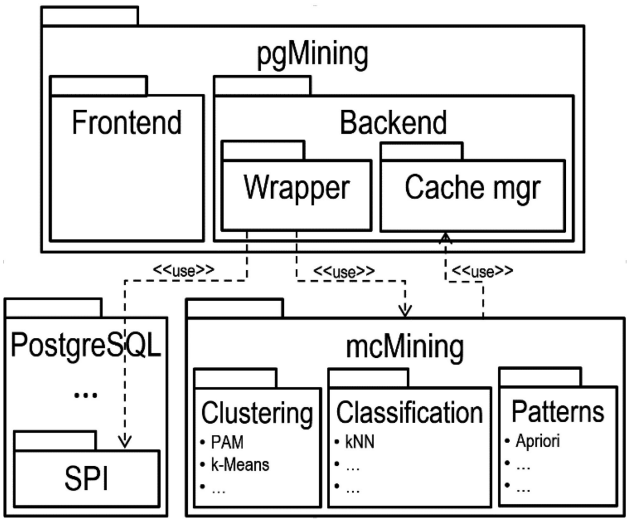


Fig. 2. Component structure of the proposed approach

The *pgMining* is a library of functions for data mining inside PostgreSQL. The *mcMining* is a library that exports data mining functions, which are parallelized for Intel MIC systems and are subject of wrapping by the respective functions from *pgMining* library. Implementation of the *pgMining* library uses the PostgreSQL SPI (Server Programming Interface), which provides the low-level functions for data access. The *pgMining* library consists of two following subsystems, namely *Frontend* and *Backend*, where the former provides presentation layer and the latter provides data access layer of concerns for an application programmer.

3.3 Frontend

The *Frontend* implements a user-level wrapper. A *Frontend* function wraps the respective UDF from *Backend*, which is loaded into PostgreSQL and executed as "`INSERT INTO … SELECT …`" query to save mining results in the specified table. An example of the *Frontend* function is given in Fig. 3.

```
// PAM clustering inside PostgreSQL
// Returns 0 in case of success or negative error code.
int pgPAM (
  PGconn * conn, // ID of PostgreSQL connection
  char * inpTab, // Name of input table
  int dim,       // Number of coordinates in data point
  int k,         // Number of clusters
  char * outTab) // Name of output table
{
  PQexec(conn, "CREATE FUNCTION
    wrap_pgPAM(text, integer, integer, real) RETURNS text AS
    'pgmining', 'wrap_pgPAM' LANGUAGE C STRICT;");
  PQexec(conn, "CREATE %s TABLE (data text)", outTab);
  return PQexec(conn, "INSERT INTO %s
   SELECT wrap_pgPAM(%s, %d, %d);", outTab, inpTab, dim, k);
}
```

Fig. 3. Interface and implementation schema of function from *Frontend*

Such a function connects to PostgreSQL, carries out some mining task and returns exit code. The function mandatory parameters are PostgreSQL connection ID, names of input and output tables, and a number of first left columns in the input table containing data to be mined. The rest parameters are specific to the mining task. As a side effect, the function creates a table with mining results, which are stored as a text in JSON format. Notation of such a text allows to define various results specific to the mining algorithm, e.g. for a clustering algorithm this text could describe output data points with associated numbers of clusters, centroids of resulting clusters, etc. Application programmer is then in charge of parse and extract the results, and save it in relation table(s) if necessary.

3.4 Backend

The *Wrapper* of the *Backend* implements a system-level wrapper. Figure 4 depicts an example of *Wrapper*'s function. Such a function is an UDF, which wraps a parallelized mining function from *mcMining* and performs as follows. Firstly, the function parses its input to form the parameters to call the *mcMining* function. After that, the function checks if input table(s) and/or supplementary mining structure(s) are in the cache maintained by *Cache manager* and then loads them if not. Finally, a call of the *mcMining* function with appropriate parameters is performed. The *Cache manager* provides buffer pool to store precomputed mining structures.

```
// Wrapper for PAM clustering inside PostgreSQL
// Returns clustering result as JSON string.
Datum wrap_pgPAM(PG_FUNCTION_ARGS)
{
  // Extract parameters of the algorithm
  char * inpTab = text_to_cstring(PG_GETARG_TEXT_P(0));
  int dim = PG_GETARG_INT32(1);
  int k = PG_GETARG_INT32(2);
  int N;
  // Check if mining structure is in the cache
  void * distMatr = cache_getObject(
    strcat(inpTab, "_distMatr"));
  if (distMatr == NULL) {
    // Check if input table is in the cache
    void * inpData = cache_getObject(inpTab);
    if (inpData == NULL) {
      // Allocate memory and load input table to the cache
      inpData = (float *) malloc(dim * N * sizeof(float));
      wrap_tabRead(inpData, inpTab, dim, &N);
      cache_putObject(inpTab, inpData, sizeof(inpData));
    }
    distMatr = mcCalcMatrix(inpData, dim, N);
    cache_putObject(strcat(inpTab, "_distMatr"),
      distMatr, sizeof(distMatr));
  }
  // Perform clustering and save results to the output ta-
ble
  mcPAM_res * outData = mcPAM_resCreate();
  mcPAM(N, k, outData, distMatr);
  PG_RETURN_TEXT(data2String(outData));
}
```

Fig. 4. Interface and implementation schema of function from *Backend*

Distance matrix $d_{ij} = dist(a_i, a_j)$ stores distances between each pair of a_i and a_j elements of input data set and is a typical example of mining structure to be cached. Being precomputed once, distance matrix could be used many times for clustering or kNN-based classification with various parameters (e.g. the number of clusters, the number of neighbors, etc.).

The *Cache manager* exports two basic functions depicted in Fig. 5. The *putObject* function loads a mining structure specified by its ID, buffer pointer and size into the cache. The *getObject* searches in the cache for an object with the given ID. An ID of mining structure is a string, which is made as a concatenation of an input table name and object informational string (e.g. "_distMatr"). In order to handle a situation when there is not enough space in the buffer pool to put a new object, *Cache manager* implements one of the replacement strategies, e.g. LRU-K [16], LFU-K [28], etc.

```
// Load an object to the cache.
// Returns 0 in case of success or negative error code.
int cache_putObject(
  char * objID,// An object's ID
  void * data, // Pointer to data buffer
  int size);   // Data size
// Search for an object by the given ID in cache.
// Returns pointer to the object in case of success or NULL.
void * cache_getObject(char * objID);
```

Fig. 5. Interface of *Cache manager* module

3.5 Library of Parallel Algorithms for Intel MIC

Figure 6 gives an example of the *mcMining* library function. Such a function encapsulates parallel implementation for Intel many-core systems based on OpenMP. In this example, we use Partition Around Medoids (PAM) [9] clustering algorithm, which is applicable when minimal sensitivity to noise data is required. The PAM provides such a property since it represents cluster centers by points of the input data set (*medoids*). Firstly, PAM computes distance matrix for the given data points. Then, the algorithm carries out an initial clustering by the successive selection of medoids until the required number of clusters have been found. Finally, the algorithm iteratively improves clustering in accordance with an objective function.

```
// PAM clustering parallelized for Intel MIC platform.
// Returns 0 in case of success or negative error code.
int mcPAM(
    int N,             // Number of data points
    int k,             // Number of clusters
    void * outData,    // Array of output centroids
    void * distMatr); // Precomputed distance matrix
```

Fig. 6. Interface of function from *mcMining* library

In our previous work [24], a parallel version of the PAM algorithm exploits auto-vectorization of distance matrix computation. Auto-vectorization relies on a compiler's ability to transform the loops into sequences of vector operations and utilize vector processor units. Thus, auto-vectorization does not guarantee the optimal vectorization and, in turn, the best performance.

3.6 Advanced Vectorization of Parallel Algorithms for Intel MIC

In this study, we take a step forward and accelerate parallel PAM by speeding up its first phase, keeping in mind that according to experiments in our previous work, distance matrix computation takes up to 80% of overall runtime.

We implemented distance matrix computation phase by intrinsics instead of auto-vectorization. Intrinsics are assembly-coded functions that wrap processor specific instruction sets and allow us to use the C function calls and variables in place of assembly instructions. Intrinsics are expanded inline eliminating function call

overhead. Thus, intrinsics allow make it possible to reach the optimal vectorization manually, at the expense of a programmer efforts and the code maintainability.

Moreover, we implemented sophisticated SoAoS (Structure of Arrays of Structures) data layout [22] to organize data points in memory as follows. Suppose, there are N data points where each point comprises of *dim* float coordinates and there is a platform-dependent parameter, namely $size_{vector}$. Then, data points are represented by an array comprising of N div $size_{vector}$ elements. Each element of the array is a structure comprising of *dim* attributes where each attribute is an array of $size_{vector}$ float coordinates. The $size_{vector}$ parameter is chosen with respect to a number of floats that could be processed in one vector operation for the given platform's instruction set (e.g. for Intel Xeon and its SSE instruction set we took $size_{vector}$= 4, and for Intel Xeon Phi and its AVX-512 instruction set we took $size_{vector}$= 16).

Figure 7 depicts implementation scheme of Euclidean distance matrix computation for Intel MIC platform based on intrinsics and the SoAoS data layout. The algorithm performs as follows.

```
// Computation of Euclidean distance matrix.
// Returns pointer to the matrix.
float * mcCalcMatrix (float * inpData, int dim, int N)
{
  const int vecSize = 16;
  float * distMatr = ALLOC_ALIGNED_FLOAT_ARRAY (N*N);
  float * SoAoS = ALLOC_ALIGNED_FLOAT_ARRAY (dim*N);
  SoAoS_permute(inpData, SoAoS, N, dim, vecSize);
  #pragma omp parallel for
  for (int i=0; i<N; i++) {
    for (int k=0; k<N; k+=vecSize) {
      VECTOR res = GET_ZERO_VEC ();
      for (int j=0; j < dim; j++) {
        VECTOR p1 = FILL_VEC (inpData + i*dim + j);
        VECTOR p2 = LOAD_VEC (SoAoS + k*dim + j* vecSize);
        VECTOR diff = SUB_VEC (p1, p2);
        res = FMADD_VEC (diff, diff, res);
      }
      STORE_VEC (distMatr + i*N + k, res);
    }
  }
  free(SoAoS);
  return distMatr;
}
```

Fig. 7. Parallel implementation of Euclidean distance matrix computation

Before the computation, we permute an array of input data points to represent them as a SoAoS layout. Computation is organized as three nested loops where the outer loop runs along the input data points and parallelized by the OpenMP `#pragma` compiler directive. The second inner loop runs along the SoAoS blocks of data points and initializes a vector register in order to store temporary results. The innermost loop runs along the coordinates of a data point and carries out the following actions by intrinsic functions. Vector register is filled by the j-th coordinate of an input data point.

Then, we read *j*-th coordinates of the sixteen data points from a SoAoS block. Next, we calculate the difference of the vectors mentioned above, square the difference and add the result to the vector register. At the end of the second loop, data from the vector register are moved to the resulting matrix. In the end, resulting matrix will comprise of squared Euclidean distances.

4 Experimental Evaluation

4.1 Background of the Experiments

In the experiments, we firstly evaluated how intrinsics and new data layout affect the performance of distance matrix computation. We also investigated the scalability of the modified *mcPAM* version on Intel MIC platforms depending on the number of threads employed. Here, we mean speedup and parallel efficiency as basic characteristics of a parallel algorithm, which are defined as follows. Speedup and parallel efficiency of a parallel algorithm being ran on k threads are calculated as

$$s(k) = \frac{t_1}{t_k}, \; e(k) = \frac{t_1}{k \cdot t_k} \cdot 100\%$$

where t_1 and t_k are run times of the algorithm on one and k threads, respectively. A parallel algorithm with speedup closer to one and parallel efficiency at least 50% is considered to have good scalability.

We took the *pgPAM* function with the modified *mcPAM* function and compared the *pgPAM* performance with the PAM from the *R* package [14] as well.

We performed the evaluation on the Tornado SUSU supercomputer [10] and the RSC[1] cluster node. The former provides a node with two Intel MIC platforms, namely Intel Xeon and Intel Xeon Phi (KNC generation) and the latter provides a node with Intel Xeon Phi (KNL generation) platform (cf. Table 1 for the specifications).

Table 1. Specifications of hardware

Specifications	Host CPU	Coprocessor	CPU system
Model, Intel Xeon	2 × X5680	Phi (KNC), SE10X	Phi (KNL), 7250
Physical cores	2 × 6	61	68
Hyper threading factor	2	4	4
Logical cores	24	244	272
Frequency, GHz	3.33	1.1	1.4
Vector processing unit size, bit	128	512	512
Boot ability	Yes	No	Yes
Peak performance, TFLOPS	0.371	1.076	3.046

[1] http://www.rscgroup.ru/en/.

In the experiments, we used the datasets depicted in Table 2.

Table 2. Datasets used in experiments

Dataset	dim	# clusters	# points, $\times 2^{10}$	Semantic
FCS Human	423	10	18	Aggregated human gene information [2]
MixSim	5	10	35	Generator of synthetic datasets for evaluation of clustering algorithms [14]
Census	67	10	35	US Census Bureau population surveys [13]
Power	3	10	35	Household electricity consumption [11]

4.2 Results and Discussion

Figure 7 shows the results of the first series of experiments. We can see that auto-vectorization provides at least 1.5× and 5× faster run time of distance matrix computation for the Intel Xeon and Intel MIC platform, respectively, in comparison with scalar version. The use of intrinsics, in turn, provides up to 2× faster run time in comparison with auto-vectorization. Thus, we can conclude that intrinsics combined with proper data layout significantly increase performance of the most heavy-weight part of the parallel PAM algorithm.

The results of the experiments on the *mcPAM* scalability are depicted in Fig. 8. Both speedup and parallel efficiency are closer to linear when the number of threads matches the number of physical cores the algorithm is running on, for all the platforms (i.e. 12 cores for Intel Xeon, 60 cores for Intel Xeon Phi KNC and 68 cores for Intel Xeon Phi KNL, respectively). Speedup and parallel efficiency become sub-linear when the algorithm uses more than one thread per physical core. We can conclude that after acceleration of the phase of distance matrix computation, the algorithm still demonstrates good scalability for all the considered Intel MIC platforms.

Figure 9 shows the results of the experiments on the *pgPAM* performance on different datasets. We compared serial PAM from the *R* package with our both serial and parallel *pgPAM* where a distance matrix was precomputed or not. Parallel versions ran on the following Intel platforms, namely 2× Xeon CPU (24 threads), Xeon Phi KNC (240 threads), and Xeon Phi KNL (272 threads). We can see that the parallel and serial *pgPAM* versions outperform the *R* PAM for the given platforms and datasets. Next, caching the precomputed distance matrix, we improve the performance, especially in case of high-dimensional data. By using the scalable *mcPAM* algorithm, the *pgPAM* shows the better results on the Intel Xeon MIC systems than on the Intel Xeon CPU and performs best on Intel Xeon Phi KNL (Fig. 10).

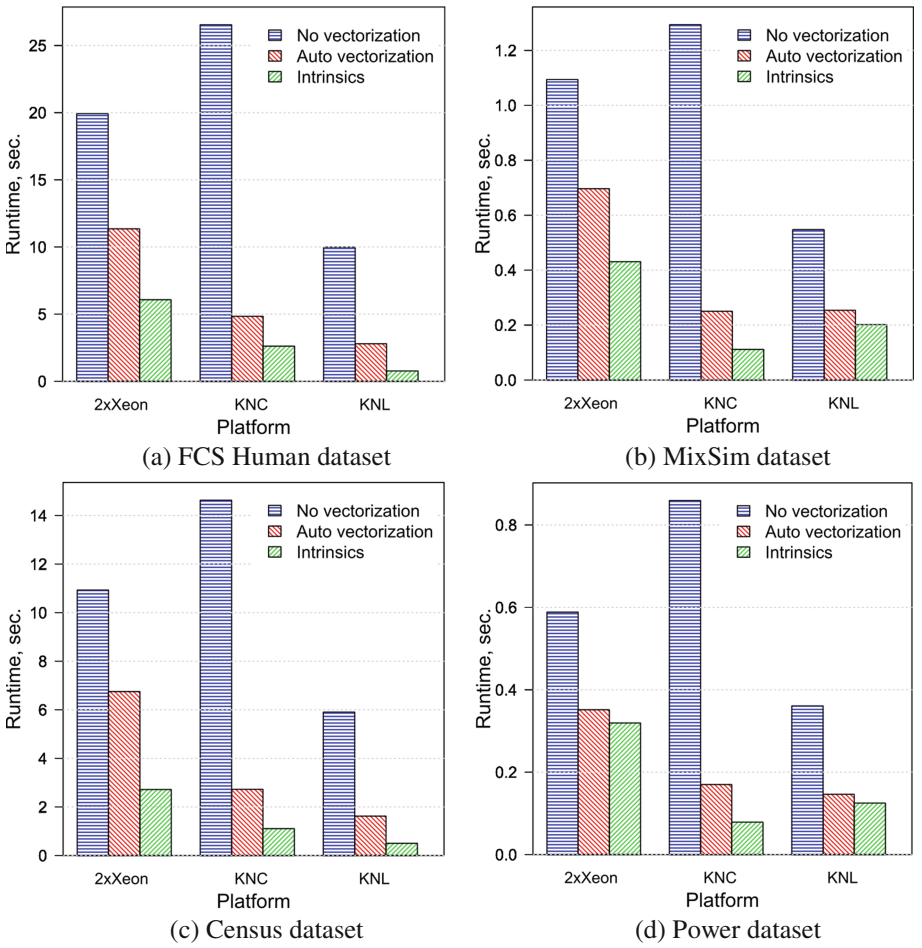
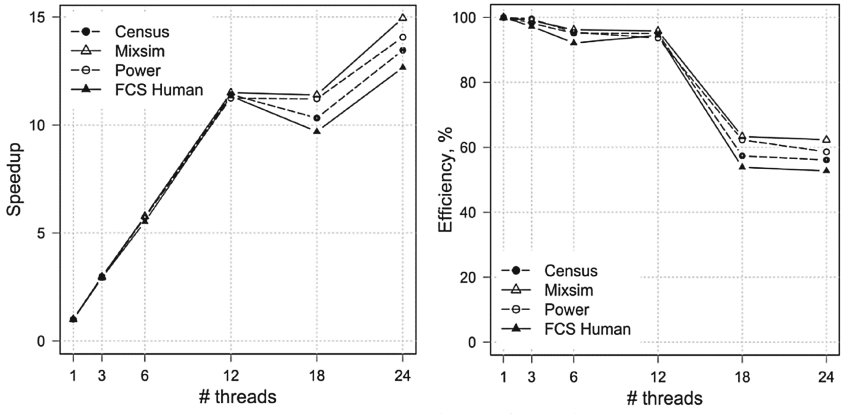
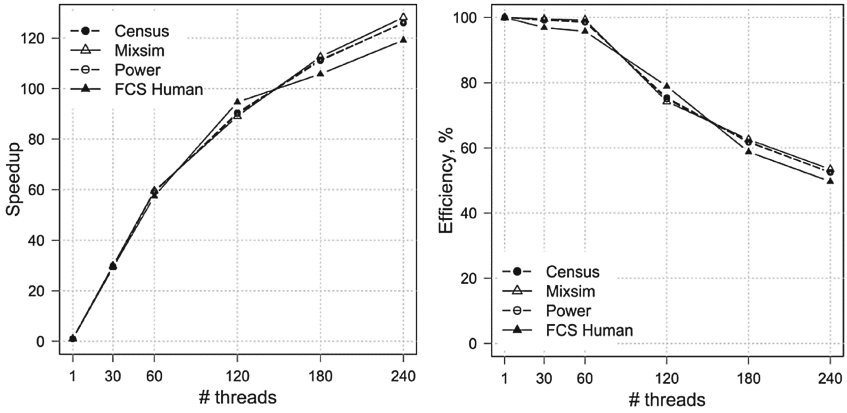


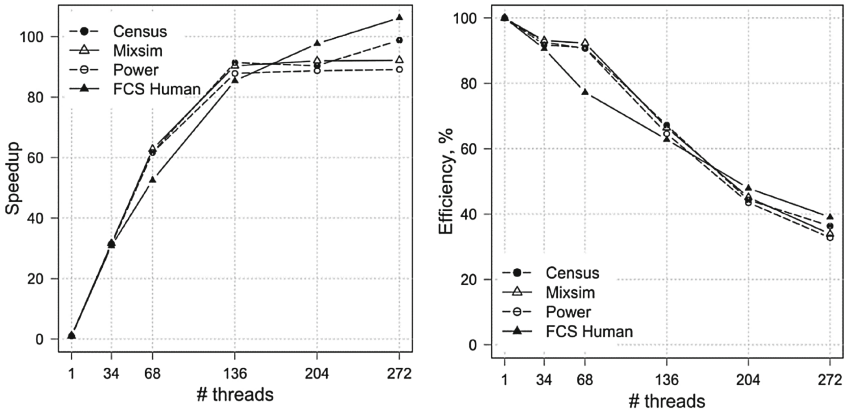
Fig. 8. Impact of vectorization to performance



(a) Host CPU (2×Intel Xeon)

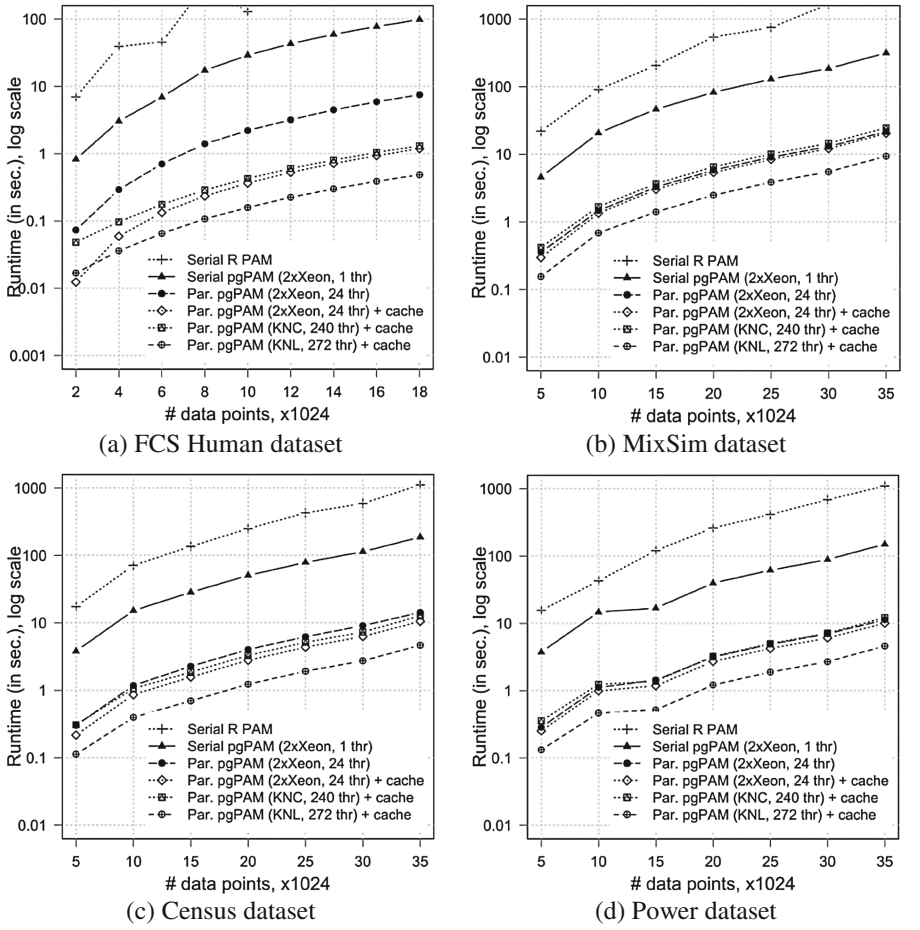


(b) Intel Xeon Phi (Knights Corner)



(c) Intel Xeon Phi (Knights Landing)

Fig. 9. Speedup and parallel efficiency of *mcPAM*



(a) FCS Human dataset (b) MixSim dataset

(c) Census dataset (d) Power dataset

Fig. 10. Performance of *pgPAM*

5 Conclusion

In this paper, we touch upon the problem of integrating data mining algorithms and a relational DBMS in data intensive domains. We presented an approach to implementation of in-database analytics that exploits capabilities of modern many-core platforms to improve performance of analytics. We implemented such an approach for PostgreSQL and Intel Many Integrated Core (MIC) architecture.

Our approach exploits the following key ideas. A data mining algorithm is implemented with C language and parallelized by the OpenMP technology for Intel MIC platforms. The parallel mining function is covered by two wrappers, namely a system-level wrapper and a user-level wrapper. The user-level wrapper registers the system-level wrapper in the database schema, connects to the database server and calls system-level wrapper. The system-level wrapper is an UDF, which parses parameters

of the user-level wrapper, calls the parallel mining function and saves results in the table(s). The system-level wrapper is accompanied by a cache of precomputed mining structures (e.g. distance matrix) to reduce costs of computations. Since our approach assumes encapsulation of parallel implementation from PostgreSQL, it could be ported to some other open-source RDBMS, with possible non-trivial but mechanical software development effort.

In this study, in order to increase performance of in-database clustering, we additionally implemented distance matrix computation (which is the heaviest part of the clustering algorithm) using advanced data layout and intrinsic functions for MIC platforms. We evaluated our approach on modern Intel MIC platforms (Intel Xeon and Intel Xeon Phi with both Knights Corner and Knights Landing generations) using real datasets where our solution showed good scalability and performance and overtook the *R* data mining package.

Acknowledgments. 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 2.7905.2017/8.9). Authors thank RSC Group (Moscow, Russia) for the provided computational resources.

References

1. Duran, A., Klemm, M.: The Intel Many Integrated Core architecture. In: Smari, W.W., Zeljkovic, V. (eds.) HPCS, pp. 365–366. IEEE (2012)
2. Engreitz, J.M., Daigle Jr., B.J., Marshall, J.J., Altman, R.B.: Independent component analysis: mining microarray data for fundamental human gene expression modules. J. Biomed. Inform. **43**(6), 932–944 (2010)
3. Feng, X., Kumar, A., Recht, B., Re, C.: Towards a unified architecture for in-RDBMS analytics. In: Candan, K.S., Chen, Y., Snodgrass, R.T., Gravano, L., Fuxman, A. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, 20–24 May 2012, pp. 325–336. ACM (2012)
4. Garcia, W., Ordonez, C., Zhao, K., Chen, P.: Efficient algorithms based on relational queries to mine frequent graphs. In: Nica, A., Varde, A.S. (eds.) Proceedings of the Third Ph.D. Workshop on Information and Knowledge Management, PIKM 2010, Toronto, Ontario, Canada, pp. 17–24. ACM, 30 October 2010
5. Han, J., Fu, Y., Wang, W., Chiang, J., Gong, W., Koperski, K., Li, D., Lu, Y., Rajan, A., Stefanovic, N., Xia, B., Zaiane, O.R.: Dbminer: a system for mining knowledge in large relational databases. In: Simoudis, E., Han, J., Fayyad, U.M. (eds.) Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA, pp. 250–255. AAAI Press (1996)
6. Hellerstein, J.M., Re, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The MADlib analytics library or MAD skills, the SQL. PVLDB **5**(12), 1700–1711 (2012)
7. Imielinski, T., Virmani, A.: MSQL: a query language for database mining. Data Min. Knowl. Discov. **3**(4), 373–408 (1999)

8. Jaedicke, M., Mitschang, B.: On parallel processing of aggregate and scalar functions in object-relational DBMS. In: Haas, L.M., Tiwary, A. (eds.) SIGMOD 1998, Proceedings of the ACM SIGMOD International Conference on Management of Data, 2–4 June, 1998, Seattle, Washington, USA, pp. 379–389. ACM Press (1998)
9. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis. Wiley, New York (1990)
10. Kostenetskiy, P., Safonov, A.: SUSU supercomputer resources. In: Sokolinsky, L., Starodubov, I. (eds.) PCT 2016, International Scientific Conference on Parallel Computational Technologies, Arkhangelsk, Russia, 29–31 March 2016, CEUR Workshop Proceedings, vol. 1576, pp. 561–573. CEUR-WS.org (2016)
11. Lichman, M.: UCI machine learning repository. Irvine, CA: University of California, School of Information and Computer Science (2013). http://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption
12. Mahajan, D., Kim, J.K., Sacks, J., Ardalan, A., Kumar, A., Esmaeilzadeh, H.: In-RDBMS Hardware Acceleration of Advanced Analytics. CoRR abs/1801.06027 (2018)
13. Meek, C., Thiesson, B., Heckerman, D.: The learning-curve sampling method applied to model-based clustering. J. Mach. Learn. Res. **2**, 397–418 (2002)
14. Melnykov, V., Chen, W.C., Maitra, R.: MixSim: an R package for simulating data to study performance of clustering algorithms. J. Stat. Softw. Artic. **51**(12), 1–25 (2012)
15. Miniakhmetov, R., Zymbler, M.: Integration of fuzzy c-means clustering algorithm with PostgreSQL database management system. Numer. Methods Programm. **13**(2(26)), 46–52 (2012)
16. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., 26–28 May 1993, pp. 297–306. ACM Press (1993)
17. Ordonez, C.: Integrating k-means clustering with a relational DBMS using SQL. IEEE Trans. Knowl. Data Eng. **18**(2), 188–201 (2006)
18. Ordonez, C.: Building statistical models and scoring with UDFs. In: Chan, C.Y., Ooi, B.C., Zhou, A. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, 12–14 June 2007, pp. 1005–1016. ACM (2007)
19. Ordonez, C., Garcia-Garcia, J.: Vector and matrix operations programmed with UDFs in a relational DBMS. In: Yu, P.S., Tsotras, V.J., Fox, E.A., Liu, B. (eds.) Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, 6–11 November 2006, pp. 503–512. ACM (2006)
20. Ordonez, C., Pitchaimalai, S.K.: Bayesian classifiers programmed in SQL. IEEE Trans. Knowl. Data Eng. **22**(1), 139–144 (2010)
21. Pan, C.S., Zymbler, M.L.: Very large graph partitioning by means of parallel DBMS. In: Catania, B., Guerrini, G., Pokorný, J. (eds.) ADBIS 2013. LNCS, vol. 8133, pp. 388–399. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40683-6_29
22. Peng, Y., Grossman, M., Sarkar, V.: Static cost estimation for data layout selection on GPUs. In: 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2016, Salt Lake, UT, USA, 14 November 2016, pp. 76–86. IEEE (2016)
23. Rantzau, R.: Frequent itemset discovery with SQL using universal quantification. In: Meo, R., Lanzi, P.L., Klemettinen, M. (eds.) Database Support for Data Mining Applications. LNCS (LNAI), vol. 2682, pp. 194–213. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-44497-8_10

24. Rechkalov, T., Zymbler, M.: Accelerating medoids-based clustering with the Intel Many Integrated Core architecture. In: 9th International Conference on Application of Information and Communication Technologies, AICT 2015, 14–16 October 2015, Rostov-on-Don, Russia - Proceedings, pp. 413–417 (2015)
25. Rechkalov, T., Zymbler, M.: An approach to data mining inside PostgreSQL based on parallel implementation of UDFs. In: Kalinichenko, L.A., Manolopoulos, Y., Kuznetsov, S. O. (eds.) Selected Papers of the XIX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2017), Moscow, Russia, 9–13 October 2017, CEUR Workshop Proceedings, vol. 2022, pp. 114–121. CEUR-WS.org (2017)
26. Sattler, K., Dunemann, O.: SQL database primitives for decision tree classifiers. In: Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, 5–10 November 2001, pp. 379–386. ACM (2001)
27. Shang, X., Sattler, K.-U., Geist, I.: SQL Based Frequent Pattern Mining with FP-Growth. In: Seipel, D., Hanus, M., Geske, U., Bartenstein, O. (eds.) INAP/WLP -2004. LNCS (LNAI), vol. 3392, pp. 32–46. Springer, Heidelberg (2005). https://doi.org/10.1007/11415763_3
28. Sokolinsky, L.B.: LFU-*K*: an effective buffer management replacement algorithm. In: Lee, Y., Li, J., Whang, K.-Y., Lee, D. (eds.) DASFAA 2004. LNCS, vol. 2973, pp. 670–681. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24571-1_60