2010 IEEE International Conference on Granular Computing

Infobright – Analytic Database Engine using Rough Sets & Granular Computing

Dominik Ślęzak[*†], Piotr Synak[*‡], Jakub Wróblewski[*‡] and Graham Toppin[*]
[*]*Infobright Inc., Toronto, Canada / Warsaw, Poland*
{slezak,synak,jakubw,toppin}@infobright.com
[†]*Institute of Mathematics, University of Warsaw, Warsaw, Poland*
[‡]*Polish-Japanese Institute of Information Technology, Warsaw, Poland*

***Abstract*—We discuss the usage of the paradigms of rough sets and granular computing in the core components of the Infobright's database engine. Having data stored in the form of compressed blocks of attribute values, our query execution methods utilize compact information about those blocks' contents instead of brute-force data decompression. The paper contains examples of algorithms implemented with the aim to minimize the need of accessing the compressed data in such operations as filtering, joining and aggregating.**

I. Introduction

Relational databases provide a useful and popular framework for storing and querying data [1]. Among the current trends, there is a focus on *analytical database engines* that do not require advanced equipment or DBA expertise, and that are able to deal with increasingly large amounts of data and complex SQL workloads. In [2], [3], we reported that our database product, available as *Infobright Community Edition*[1] (*ICE*, open source) and *Infobright Enterprise Edition*[2] (*IEE*, commercial), meets such requirements. Query workload diversity is addressed by replacing standard database indices with incomparably smaller compact information about the blocks of rows being created while loading data. ICE/IEE automatically creates *granulated tables* with their rows, further called *rough rows*, corresponding to the blocks of original rows, and with their attributes corresponding to various forms of compact information. One may look at ICE/IEE as a *granular database engine*, where all operations are designed to work adaptively with two layers:

1) the granulated tables with rough rows and their *rough values* corresponding to information about values of particular attributes within particular blocks of rows
2) the underlying repository of *data packs*, which store, in a highly compressed form, the blocks of values of particular attributes within particular blocks of rows

Infobright's granulated tables can be considered as an example of *non-deterministic information systems* [4], given the fact that rough values may take the form of sets, intervals, or more complex structures. Furthermore, the above two-layered data representation and the methods of its usage while executing SQL queries can be interpreted using the basic notions of *granular computing* [5] and *rough sets* [6]. Combination of this approach with the already-mentioned data compression [7], *columnar data storage* [8], and other techniques widely known in the database industry have led us towards the software-only solution for querying terabytes of data, that is now commercially used in such areas as: online analytics, telecommunications, and others.[3]

Our goal in this paper is to step back to the core internals of the ICE/IEE query execution and to show how it uses rough values, in particular, how the calculations on rough values and individual values interact with each other. The paper is organized as follows: In Section II, we discuss the basics of granulated tables and rough values. In Section III, we outline the overall architecture of our database product. In Section IV, we present several examples of utilizing rough values by the algorithms currently implemented in ICE/IEE. In Section V, we conclude the paper.

II. Granulated Data and Rough Values

Figure 1 illustrates how data packs and rough values are assembled while loading data into ICE/IEE (more generally, during data manipulation operations including load, insert, update and delete) and how rough values may be then used in query execution (here we refer to an example of the data filtering operation, described in detail in Section IV).

We implemented various types of rough values that should be informative enough to minimize data access and, on the other hand, small enough to use them efficiently. The examples of rough values include: minimum/maximum values (interpreted differently for the blocks of numeric and alpha-numeric values), binary histograms (for the blocks of numeric values), and generalizations of character maps (for the blocks of alpha-numeric values). For illustration how rough values may look like, we refer also to our forums.[4] In [9], we began considering the *expert knowledge* about

[1]http://www.infobright.org/
[2]http://www.infobright.com/
[3]http://www.infobright.com/Customers_Partners/Customers/
[4]http://www.infobright.org/forums/viewthread/538/

978-0-7695-4161-7/10 $26.00 © 2010 IEEE
DOI 10.1109/GrC.2010.177

IEEE computer society

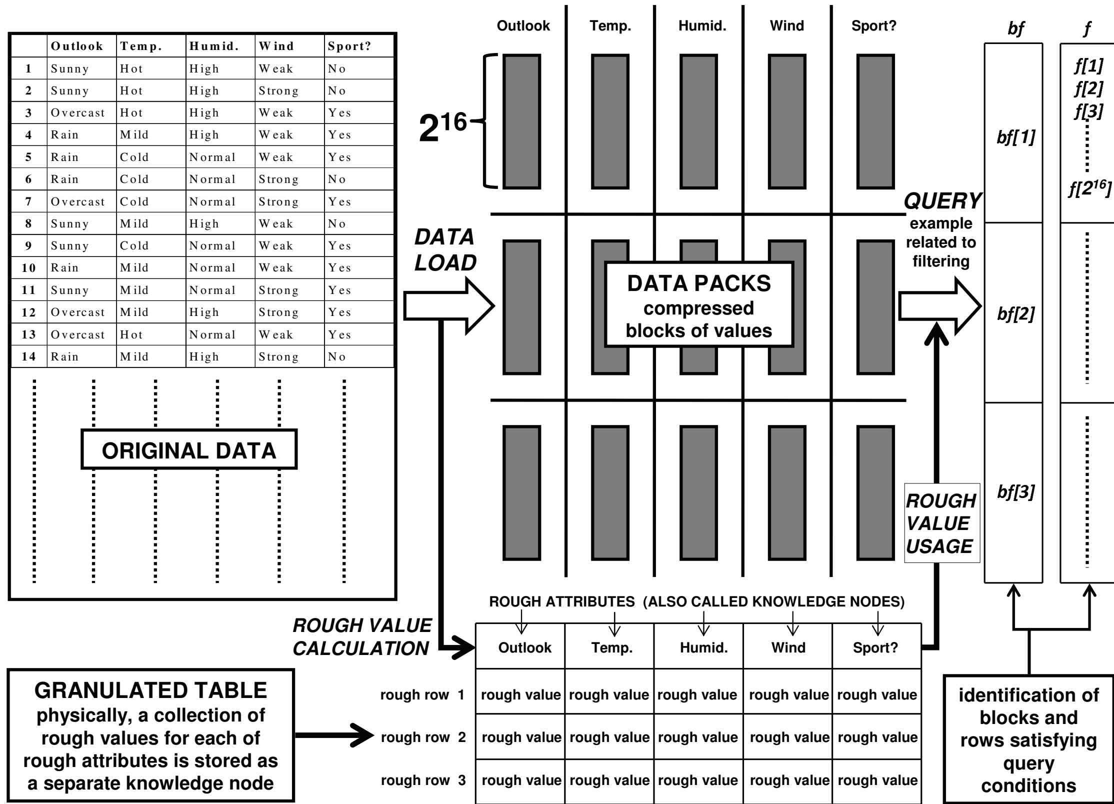

Figure 1. Loading and querying in ICE/IEE. When loading data from an external file to a previously declared table, the rows are decomposed onto their particular attributes' values. After gathering 2^{16} of new rows, ICE/IEE automatically computes rough values and compresses the blocks of 2^{16} of values corresponding to each of the table's attributes. Rough values and compressed blocks of values are stored on disk. When querying, all potentially applicable rough values are put into memory. In case of, e.g., data filtering (see Section IV for more details), rough values are applied to quickly exclude the blocks of values that do not satisfy a given SQL condition. Only not excluded blocks need to be decompressed and examined value by value. The filtering results obtained at the level of blocks of values and single values are stored in the *BlockFilter* and *Filter* arrays denoted as bf and f, respectively.

semantics of data content as very useful for both data compression and rough value formation, especially in case of the blocks of long string values. From this perspective, it is also worth mentioning that we often refer to the framework for handling granulated tables as Infobright's *knowledge grid*, which is quite different interpretation of this concept than in, e.g., the area of *semantic web* [10], but with some analogies in the way our rough values mediate between the query execution and the data storage layers.

In order to build a robust and extendable technology, we surely needed to think about appropriate internal interfaces between granulated tables and other components of ICE/IEE. In particular, it is important to have a common functionality of all types of rough values that can be used while querying and manipulating the data, regardless of whether these are simple minimum/maximum values of advanced structures based on the expert knowledge. Some aspects of such uniform functionality will be discussed in more detail in further sections, with no need of referring to any specific types of rough values. For now, let us briefly itemize what each type of rough value should be capable of:

- Ability to compute during data load (and manipulation) or querying. Computations should not decrease the load/query speed. Rough value specifications should include default settings (lack of information) in case some of rough values were not yet computed [2].
- Ability to use in *Rough* functions classifying blocks as *irrelevant* ($\mathfrak{I}$), *relevant* ($\mathfrak{R}$), and *suspect* ($\mathfrak{S}$) with respect to (adaptively refined) query constraints. $\mathfrak{I}/\mathfrak{R}/\mathfrak{S}$-states correspond to *negative*, *positive*, and *boundary* regions in rough sets [6]. $\mathfrak{I}$ means that the block does not contain any rows matching the constraints. $\mathfrak{R}$ means that all rows match them, so the partial query results can be assembled using the block's rough values. Thus, only *suspect* blocks need to be decompressed [3].
- Ability to utilize in functions estimating the degrees of satisfaction of query constraints and prioritizing the blocks whose decompression may decrease a need of accessing other blocks after refining the constraints.
- Finally, ability to assist in efficient value encoding for the purposes of intermediate operations. (One may refer it to the idea of querying compressed data [7].)

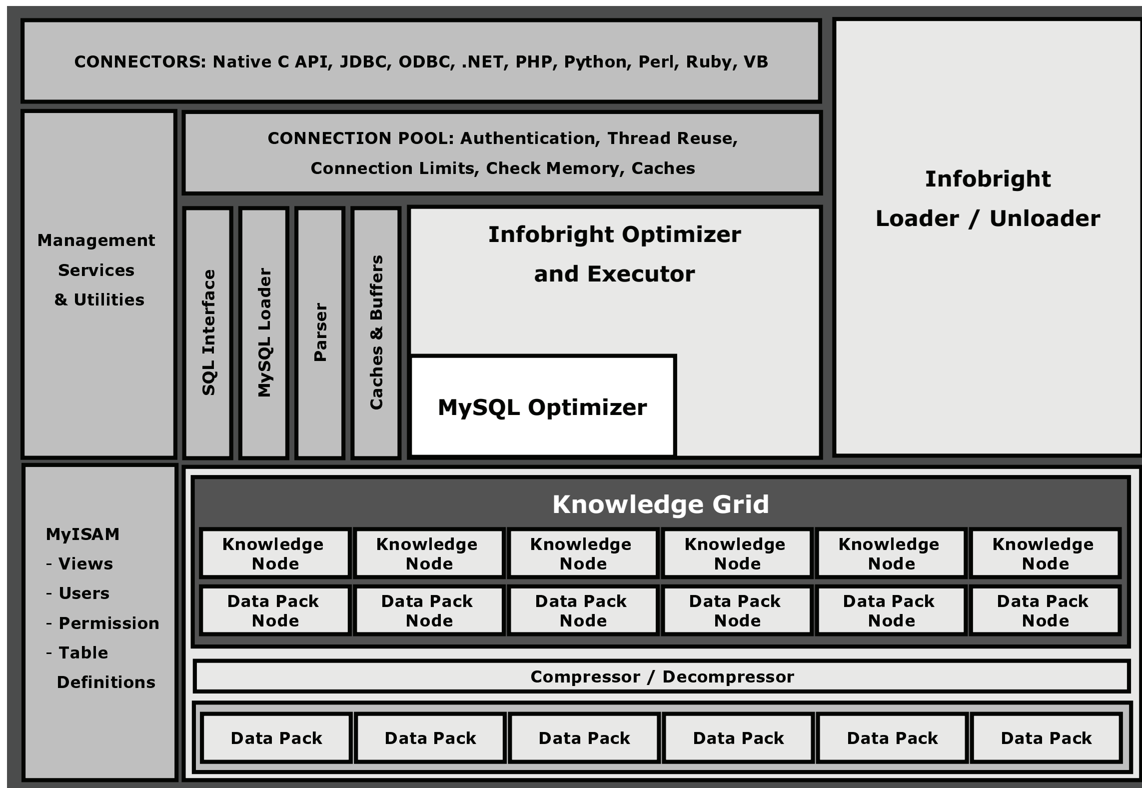

Figure 2. Integration of the core ICE/IEE components with MySQL code base (connectors, permissions, etc.). Original MySQL optimizer dedicated to the pluggable storage engines is reconsidered as a part of Infobright's optimizer and executor based on the rough/granular algorithms like these described in Section IV. Rough values discussed in Section II are stored in knowledge nodes and data pack nodes (see [3] for details). Infobright's knowledge grid provides interfaces enabling other components of the ICE/IEE architecture to work with various types of rough values in a unified fashion.

III. Infobright's Architecture

Figure 2 illustrates conceptual architecture of ICE/IEE. Besides such already-mentioned aspects as widely known columnar-oriented data access or a novel methodology of employing granulated tables in data processing, we take an advantage of the MySQL framework for *storage engines*.[5] Such a careful combination of the existing and brand new techniques makes our database software fully functional and easy enough to deploy. For example, MySQL management services are used for connection pooling. Furthermore, the MyISAM[6] engine stores catalogue information such as table definitions, views and user permissions. MySQL query rewriting and parsing modules are preserved as well. On the other hand, original MySQL optimization and execution pieces are replaced by our own code that is designed to work specifically with the compressed data packs and their corresponding rough values. Thus, ICE/IEE is the database engine with external functionality resembling MySQL, but without the need of tuning database indices, and with totally different performance characteristics and storage capabilities.

[5]http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html
[6]http://dev.mysql.com/doc/refman/5.1/en/myisam-storage-engine.html

IV. Rough/Granular Query Execution

ICE/IEE would not differ much from other technologies, if not the algorithms like those sketched in the foregoing subsections. Columnar data stores have been present in the analytical database industry for years [8]. Data compression is used in multiple ways [7]. Data operations based on the blocks of rows/values become a standard too [11]. However, all above-cited approaches assume that one has to scan the whole tables (or columns), i.e., that each block, page, etc., needs to be sooner or later accessed to resolve a query. As a result, their goal is to speed up the data scans, instead of minimizing their intensity, which is our main focus.

On the other hand, we are certainly not the first ones to notice that *data about data* may be used to decrease data access [12], [13]. Other approaches, however, concentrate on excluding the biggest possible portions of data prior to query execution. In ICE/IEE, rough values are utilized during all execution stages. Moreover, rough values are dynamically refined using information being gathered during execution. These features, nicely interpretable in a language of granular computing [5] and adaptive query processing [14], make ICE/IEE truly suitable for complex, analytical SQL.

Algorithm 1 TableCheck

Input: $Tables$, $Conditions$ $\circledast_T$
Result: $BlockFilters$, $Filters$

```
for all T ∈ Tables do
   for all block ∈ Blocks_T do
      bf_T[block] := 1
      if Rough(block, ⊛_T) = ℑ then
         bf_T[block] := 0
      end if
   end for
end for
RoughProjection(BlockFilters, Conditions)
for all T ∈ Tables do
   changed := FALSE
   for all block ∈ Blocks_T, bf_T[block] = 1 do
      bf_T[block] := 0
      for all row ∈ block do
         f_T[row] := 1
         if Exact(row, ⊛_T) = ℑ then
            f_T[row] := 0
         else
            bf_T[block] := 1
         end if
      end for
      if bf_T[block] = 0 then
         changed := TRUE
      end if
   end for
   if changed then
      RoughProjection(BlockFilters, Conditions)
   end if
end for
```

Algorithm 2 RoughProjection

Input: $BlockFilters$, $Conditions$ $\circledast_{T,T'}$
Result: $BlockFilters$ further refined

```
done := FALSE
while !done do
   done := TRUE
   for all T, T' ∈ Tables, T' ≠ T do
      for all block ∈ Blocks_T, bf_T[block] = 1 do
         bf_T[block] := 0
         for all block' ∈ Blocks_T', bf_T[block'] = 1 do
            if Rough(block, block', ⊛_T,T') ≠ ℑ then
               bf_T[block] := 1
               break
            end if
         end for
         if bf_T[block] = 0 then
            done := FALSE
         end if
      end for
   end for
end while
```

A. Case Study: Filtering

Consider SQL query involving $Tables$ joined by conditions $\circledast_{T,T'}$, with additional single-table conditions $\circledast_T$, all of them further referred as $Conditions$. Algorithm 1 shows how rough values can be used to filter data, with a minimized need of examining single values. In particular:

- Function $Rough(block, \circledast_T)$ uses available rough values to classify a given $block \in Blocks_T$[7] as $\mathfrak{I}$, $\mathfrak{R}$, or $\mathfrak{S}$ (see Section II). If $Rough(block, \circledast_T) = \mathfrak{I}$, then $block$ does not need to be further processed, as captured by the $BlockFilter$ array bf_T (see Figure 1).
- Function $Exact(row, \circledast_T)$ does the same but with single rows. $Exact$'s outcomes are $\mathfrak{I}$ or $\mathfrak{R}$, as we can say for each single row whether it satisfies a given $\circledast_T$ or not. If $Exact(row, \circledast_T) = \mathfrak{I}$, then row is labeled with 0 in the $Filter$ array f_T (see Figure 1 again).

[7] In Algorithms 1,2,3,4, *block* stands for a block of rows of T. However, whenever we write about the *block* access, we mean only those of its components that are related to the attributes involved in a query

Function $Rough(block, block', \circledast_{T,T'})$ in Algorithm 2 compares rough values of the elements of $Blocks_T$ and $Blocks_{T'}$ with respect to join condition $\circledast_{T,T'}$. For example, if $\circledast_{T,T'}$ takes the form of $T.a = T'.a'$ for attributes a and a', if rough values of $block$ and $block'$ include their minimum/maximum values over a and a', and their comparison implies that there are no $row \in block$ and $row' \in block'$ such that $a(row) = a'(row')$, we get $Rough(block, block', \circledast_{T,T'}) = \mathfrak{I}$. If it happens for each $block' \in Blocks_{T'}$ satisfying $bf_{T'}[block'] \neq 0$, there is no need to process $block \in Blocks_T$ any longer.

Algorithm 1 applies $RoughProjection$ iteratively. Each of its reoccurrences is preceded by the $Exact$ checking, whether any blocks change their status from $\mathfrak{R}$/$\mathfrak{S}$ to $\mathfrak{I}$. The arrays bf and f that result from Algorithms 1 and 2 can be then taken as an input to further steps of query execution, e.g., joining (Subsection B) or aggregating (Subsection C). Indeed, having the values of different attributes accessible independently, it is worth passing intermediate results of data filtering in the form of $Filters$ and, in case of Infobright, $BlockFilters$, with materialization of rows satisfying query conditions pushed to further execution stages [15].

B. Case Study: Joining

Let us continue with the example of join condition $\circledast_{T,T'}$ taking the form of $T.a = T'.a'$. Algorithm 3 shows how we compute the set X of pairs of rows that satisfy $\circledast_{T,T'}$. Actually, this is just one of the join techniques implemented in ICE/IEE for various types of conditions and relations between tables. It loads the portions of T into the $hash$

Algorithm 3 HashBlockJoin

Input: $bf_T, bf_{T'}$, $\circledast_{T,T'} \equiv (T.a = T'.a')$
Result: X – pairs of rows satisfying $\circledast_{T,T'}$

```
Initialize(H)
for all block ∈ Blocks_T, bf_T[block] = 1 do
   for all row ∈ block, f_T[row] = 1 do
      if !Full(H) then
         Add((row, a(row)), H)
      else
         for all block' ∈ Blocks_T', bf_T'[block'] = 1 do
            if Rough(H, block', ⊛_T,T') ≠ ℑ then
               for all row' ∈ block', f_T'[row'] = 1 do
                  Add({(row, row') : (row, a'(row')) ∈ H}, X)
               end for
            end if
         end for
         Empty(H)
      end if
   end for
end for
```

Algorithm 4 Aggregation

Input: bf_T, aggregates A^*, groupings B
Result: X – the result of aggregation

```
Initialize(H)
done := FALSE
while !done do
   done := TRUE
   for all block ∈ Blocks_T, bf_T[block] = 1 do
      if Full(H) ∧ Rough(block, H, B) = ℑ then
         done := FALSE
      else if Uniform(block, B) then
         if Full(H) ∧ ∀_{h∈H} B(block) ≠ B(h) then
            done := FALSE
         else
            bf_T[block] := 0
            if ∃_{h∈H} B(block) = B(h) then
               for all a* ∈ A* do
                  if Rough(block, a*(h)) ≠ ℜ then
                     Refresh(block, a*(h))
                  end if
               end for
            else
               Add((B(block), A*(block)), H)
            end if
         end if
      else
         for all row ∈ block, f_T[row] = 1 do
            if Full(H) ∧ ∀_{h∈H} B(row) ≠ B(h) then
               done := FALSE
            else
               f_T[row] := 0
               if ∃_{h∈H} B(row) = B(h) then
                  Refresh(row, A*(h))
               else
                  Add((B(row), A*(row)), H)
               end if
            end if
         end for
         if ∀_{row∈block} f_T[row] = 0 then
            bf_T[block] := 0
         end if
      end if
   end for
   Add(H, X)
   Empty(H)
end while
```

table[8] H and running through T' to find all matches with the current content of H. The structure of H assures efficient computation of sets $\{(row, row') : (row, a'(row')) \in H\}$ that are added to the join result X. The meaning of function *Rough* is exactly the same as in Algorithm 2. Now, however, it works with rough values of H, which are computed as a side effect of $Add((row, a(row)), H)$ and annotate the sets of values $a(row)$ stored in H at the moment of executing $Rough(H, block', \circledast_{T,T'})$. Such obtained $HashBlockJoin$ turns out to be very efficient in practice and, in view of the main goals of this paper, illustrates that rough values are worth considering not only for the blocks of original data but also for dynamic structures such as H.

C. Case Study: Aggregating

Algorithm 4 presents the method of computing queries of the form *SELECT B, A^* FROM T GROUP BY B*, wherein B stands for the set of attributes referred as *groupings*, and A^* is the set of aggregate functions ($count(*)$, $max(a)$, $avg(a)$, etc.) referred as *aggregates*. The query can be additionally filtered (which is addressed by bf_T and f_T), it might take as an input the join of several tables, or might contain more complex aggregates (such as e.g. $count(distinct\ a)$).

Algorithm 4 uses H like in Subsection B, but now filled with the elements of the form $(B(h), A^*(h))$ corresponding to the tuples of the final query result X. Herein, $B(h)$ stands for a vector of values over groupings B, $A^*(h)$ stands for a vector of its corresponding aggregate values, and h stands for an internal identifier of entry $(B(h), A^*(h))$ in H.

The amount of distinct vectors on B may cause exceeding memory available for H. In such cases, we continue computations for the current content of H and then scan T again, until there are no rows with yet not considered vectors on B, characterized by dynamically updated f_T and bf_T.

[8]Like in other database technologies, by a hash table we mean a memory-based structure requiring random (hash) access to the values that it stores.

The role of $Rough(block, H, B)$ is to check whether *block* contains any vectors on B that are currently in H. Going further, $Uniform(block, B)$ means that the whole *block*'s content matches the same vector on B, denoted as $B(block)$. Depending on whether $B(block)$ is in H or not, we should add to H a new entry $(B(block), A^*(block))$ or refresh one of its existing elements $h \in H$. In both cases, the algorithm attempts to use rough values corresponding to the attributes being aggregated. For instance, if $a^* \in A^*$ takes the form of $min(a)$ for attribute a, then $Rough(block, a^*(h)) = \mathfrak{R}$ means that the content of *block* with respect to a cannot change the current quantity of $min(a)$ for $h \in H$. This shows the importance of $\mathfrak{R}$-state, which takes us back to the paradigms of rough sets and their usage when judging the relevance of data blocks during particular steps of query execution.

V. Conclusion

We presented methodology of using rough values (which stand for compact information about compressed blocks of data values) in Infobright's analytical database engine, by referring to basic notions of granular computing and rough sets. For illustration purposes, we provided simplified versions of implemented algorithms. Some of the aspects of rough values' functionality were skipped. For instance, ability to prioritize blocks of data with respect to their relevance to the query constraints turns out to be very important for ordering blocks in the main loop of Algorithm 4. As another example, ability to use rough values to produce more efficient value encodings enables to optimize the contents of hash tables in Algorithms 3 and 4. Nevertheless, we believe that the paper clearly documents the benefits of using rough values in our approach to SQL query execution.

There are several items related to the presented framework on the Infobright's R&D roadmap. First of all, one may think about new types of rough values obtained by redesigning the existing metadata and index structures widely applied to, e.g., data filtering [16] or cardinality estimation [17], or by utilizing the domain knowledge about data content [9]. On the other hand, we need a clear interface between the query execution modules and the repository of rough values, so the algorithms outlined in Section IV can refer to all types of rough values in a reasonably uniform fashion.

As an example of other considered future enhancements, let us mention about a possibility of introducing approximate SQL framework based on modifications of *Rough* functions discussed in Sections II and IV [18]. This idea has received an interesting feedback from the community.[9] In such areas as, e.g., online analytics, there is an ongoing debate whether the answers to SQL statements have to be always fully exact. Needless to say, rough sets and granular computing may be a source of valuable inspiration when extending the standard database functionality towards this direction.

[9] http://www.infobright.org/forums/viewthread/454/

References

[1] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*. Prentice Hall, 2008.

[2] D. Ślęzak and V. Eastwood, "Data warehouse technology by Infobright," in *SIGMOD'2009*, pp. 841–846.

[3] D. Ślęzak, J. Wróblewski, V. Eastwood, and P. Synak, "Brighthouse: an analytic data warehouse for ad-hoc queries," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1337–1345, 2008.

[4] S. Demri and E. Orłowska, *Incomplete Information: Structure, Inference, Complexity*. Springer-Verlag, 2002.

[5] A. Bargiela and W. Pedrycz, *Granular Computing. An Introduction*. Kluwer Academic Publishers, 2002.

[6] Z. Pawlak and A. Skowron, "Rudiments of rough sets," *Information Sciences*, vol. 177, no. 1, pp. 3–27, 2007.

[7] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt, "How to barter bits for chronons: compression and bandwidth trade offs for database scans," in *SIGMOD'2007*, pp. 389–400.

[8] P. White and C. French, "Database system with methodology for storing a database table by vertically partitioning all columns of the table," US Patent 5,794,229, 1998.

[9] D. Ślęzak and G. Toppin, "Injecting domain knowledge into a granular database engine - a position paper," in preparation, 2010.

[10] M. Cannataro and D. Talia, "The knowledge grid," *Commun. ACM*, vol. 46, no. 1, pp. 89–93, 2003.

[11] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," *VLDB J.*, vol. 11, no. 3, pp. 198–215, 2002.

[12] V. Z. R. Grondin, E. Fadeitchev, "Searchable archive," US Patent 7,243,110, 2007.

[13] J. Metzger, B. Zane, and F. Hinshaw, "Limiting scans of loosely ordered and/or grouped relations using nearly ordered maps," US Patent 6,973,452, 2005.

[14] A. Deshpande, Z. G. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[15] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *ICDE'2007*, pp. 466–475.

[16] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in *ICDE'2009*, pp. 604–615.

[17] Y. Ioannidis, "The history of histograms (abridged)," in *VLDB'2003*, pp. 19–30.

[18] D. Ślęzak and M. Kowalski, "Towards approximate SQL - Infobright's approach," in *RSCTC'2010*, pp. 630–639.