Injecting Domain Knowledge into RDBMS – Compression of Alphanumeric Data Attributes

Marcin Kowalski[1,2], Dominik Ślęzak[1,2]
Graham Toppin[2], and Arkadiusz Wojna[1,2]

[1] MIM, University of Warsaw, Poland
[2] Infobright Inc., Canada & Poland

Abstract. We discuss the framework for applying knowledge about internal structure of data values to better handle alphanumeric attributes in one of the analytic RDBMS engines. It enables to improve data storage and access with no changes at the data schema level. We present the first results obtained within the proposed framework with respect to data compression ratios, as well as data (de)compression speeds.

Keywords: Analytic RDBMS, Data Semantics, Data Compression

1 Introduction

Data volumes that one needs to operate on daily bases, as well as the costs of data transfer and management are continually growing [6]. This is also true for analytic databases designed for advanced reports and ad hoc querying [3].

In this study, we discuss how the domain knowledge about data content may be taken into account in an RDBMS solution. By injecting such knowledge into a database engine, we expect influencing data storage and query processing. On the other hand, as already observed in our previous research [8], the method of injecting domain knowledge cannot make a given system too complicated.

As a specific case study, we consider the Infobright's analytic database engine [11,12], which implements a form of adaptive query processing and automates the task of physical database design. It also provides minimized user interface at a configuration level and low storage overhead due to data compression.

We concentrate on alphanumeric data attributes whose values have often rich semantics ignored at the database schema level. The results obtained for appropriately extended above-mentioned RDBMS engine prove that our approach can be useful for more efficient and faster (de)compression of real-life data sets.

The paper is organized as follows: Section 2 introduces the considered analytic database platform. Section 3 describes our motivation for developing the proposed framework for alphanumeric attributes. Sections 4 and 5 outline the main steps of conceptual design and implementation, respectively. Section 6 shows some experimental results. Section 7 summarizes our research in this area.

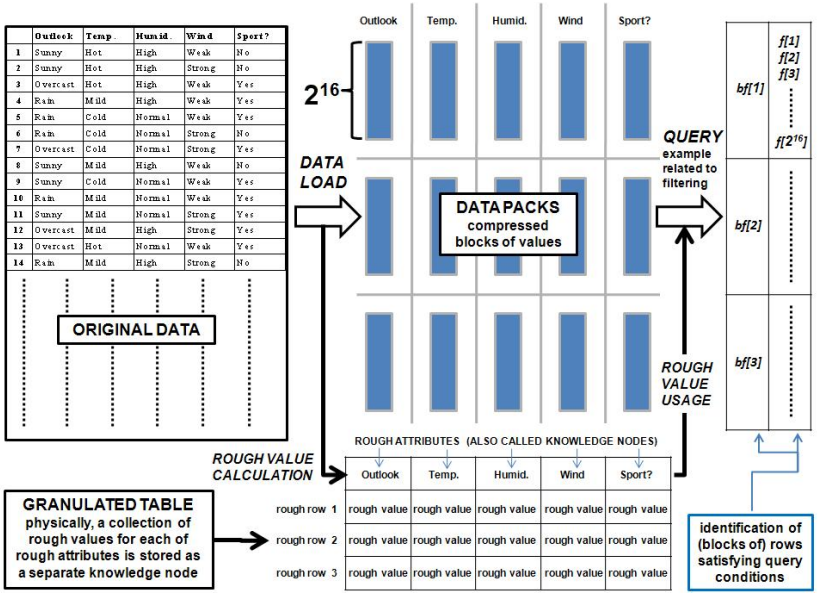

Fig. 1. Loading and querying in ICE/IEE. Rough values can be used, e.g., to exclude data packs that do not satisfy some query conditions.

2 Starting Point

In our opinion, the approach proposed in this paper might be embedded into a number of RDBMS engines that store data in a columnar way, to achieve faster data access while analytic querying [3,10]. *Infobright Community Edition* (*ICE*) and *Infobright Enterprise Edition* (*IEE*) are just two out of many possible database platforms for investigating better usage of domain knowledge about data content. On the other hand, there are some specific features of ICE/IEE architecture that seem to match with the presented ideas particularly well.

ICE/IEE creates *granulated tables* with rows (called *rough rows*) corresponding to the groups of 2^{16} original rows and columns corresponding to various forms of compact information. We refer to the layer responsible for maintaining granulated tables as to *Infobright's Knowledge Grid*.[1] Data operations involve two levels: 1) granulated tables with rough rows and their *rough values* corresponding to information about particular data attributes, and 2) the underlying repository of *data packs*, which are compressed collections of 2^{16} values of particular data attributes. Rough values and data packs are stored on disk. Rough values are small enough to keep them at least partially in memory during query sessions. A relatively small fraction of data packs is maintained in memory as well. Data packs are generally accessed on demand. We refer to [11,12] for more details.

[1] Our definition of Knowledge Grid is different than, e.g., in grid computing or semantic web [1], though the framework proposed in this paper exposes some analogies.

3 Challenges

ICE/IEE runs fast when rough values are highly *informative* and when data packs are highly compressed and easy to decompress. As already mentioned, that second aspect can be considered for some other RDBMS platforms as well. Also the first aspect might be analyzed for other databases although their usage of information analogous to rough values is relatively limited (see e.g. [5]).

By informativeness of rough values we mean that they should possibly often classify data packs as fully *irrelevant* or *relevant* at the moment of requesting those packs' status by the query execution modules. In case of full irrelevance or full relevance, the execution modules are usually able to continue with no need of decompression [12]. For the remaining *suspect* packs that require to be accessed value by value, the size of data portions to be taken from disk and the speed of making them processable by the execution modules are critical.

Achieving good quality of rough values and good (de)compression characteristics becomes harder for more complex types of attributes. Given such applications as online, mobile, or machine-generated data analytics, the issues typically arise with long varchar columns that store, e.g., URLs, emails, or texts. Moreover, even for such fields as IP or IMSI numbers that could be easily encoded as integers, the question remains at what stage of database modeling such encodings should be applied and how they may affect the end users' everyday work.

We examined many rough value structures that summarize the collections of long varchars. The criteria included high level of the above-mentioned informativeness and small size comparing to the original data packs. Some of those rough values are implemented in ICE/IEE. However, it is hard to imagine a universal structure representing well enough varchars originating from all specific kinds of applications. The same can be observed for data compression. We adopted and extended quite powerful algorithms compressing alphanumeric data [4,11]. However, such algorithms would work even better when guided by knowledge about the origin or, in other words, the internal semantics of input values.

In [8], we suggested how to use the attribute semantics while building rough values and compressing data packs. We noticed that in some applications the data providers and domain experts may express such semantics by means of the data schema changes. However, in many situations, the data schemas must remain untouched because of high deployment costs implied by any modifications. Moreover, unmodified schemas may provide the end users with conceptually simpler means for querying the data [6]. Finally, the domain experts may prefer *injecting* their knowledge independently from standard database model levels, rather than cooperating with the database architects and administrators.

An additional question is whether the domain experts are really needed to let the system know about the data content semantics, as there are a number of approaches to recognize the data structures automatically [2]. However, it is unlikely that all application specific types of value structures can be detected without a human advise. In any way, the expert knowledge should not be ignored. Thus, an interface at this level may be useful, if it is not overcomplicated.

4 Ideas

When investigating the previously-mentioned machine-generated data sets, one may quickly realize that columns declared as varchars have often heterogenous nature. Let us refer to Figure 2, precisely to a data pack related to MaybeURL column. Within the sequence of 2^{16} values, we can see *sub-collections* of NULLs, integers, strings that can be at least partially parsed along the standard URI structure,[2] as well as outliers that do not follow any kind of meaningful structure. Following [8], our idea is to deliver each of such data packs as the original string sequence to the query execution modules but, internally, store it in form of homogeneous sub-collections compressed separately. The consistency of a data pack can be secured by its *match table*, which encodes membership of each of 2^{16} values to one of sub-collections. This is an extension of our previously implemented method of decomposing data packs onto their NULL and not-NULL portions [11], applied in various forms in other RDBMS approaches as well. The difference is that here we can deal with multiple sub-types of not-NULLs.

For a given data pack, each of its corresponding sub-collections is potentially easier to compress than when trying to compress all 2^{16} values together. Each of sub-collections can be also described by separate higher-quality statistics that constitute all together the pack's rough value available to the execution modules that do not even need to be aware of its internal complexity. Data compression and rough value construction routines can further take into account that particular sub-collections gather values sharing (almost) the same structure. Going back to Figure 2, each of the values in the URI sub-collection can be decomposed onto particles such as scheme, path, authority, and so on. Sub-collections of specific particles can be compressed and summarized even better than sub-collections of not decomposed values. Again, such decompositions can be kept as transparent to the query execution modules, which refer to rough values via standardly looking functions hiding internal complexity in their implementation and, if necessary, work with data packs as sequences of *recomposed* values.

Surely, the above ideas make sense only if the domain knowledge about data content is appropriately provided to the system. According to the available literature [6] and our own experience, there is a need for interfaces enabling the data providers to inject their domain knowledge directly into a database engine, with no changes to data schemas. This way, the end users are shielded from the complexity of semantic modeling, while reaping most of its benefits. In the next section, we present one of the prototype interfaces that we decided to implement. The language proposed to express the structural complexity of attribute values is a highly simplified version of the regular expressions framework, although in future it may also evolve towards other representations [9]. The choice of representation language is actually very important regardless of whether we acquire data content information via interfaces or learn it (semi)automatically, e.g., by using some algorithms adjusting optimal levels of decomposing the original values according to hierarchical definitions recommended by domain experts.

[2] URI stands for Uniform Resource Identifier (http://www.ietf.org/rfc/rfc3986.txt).

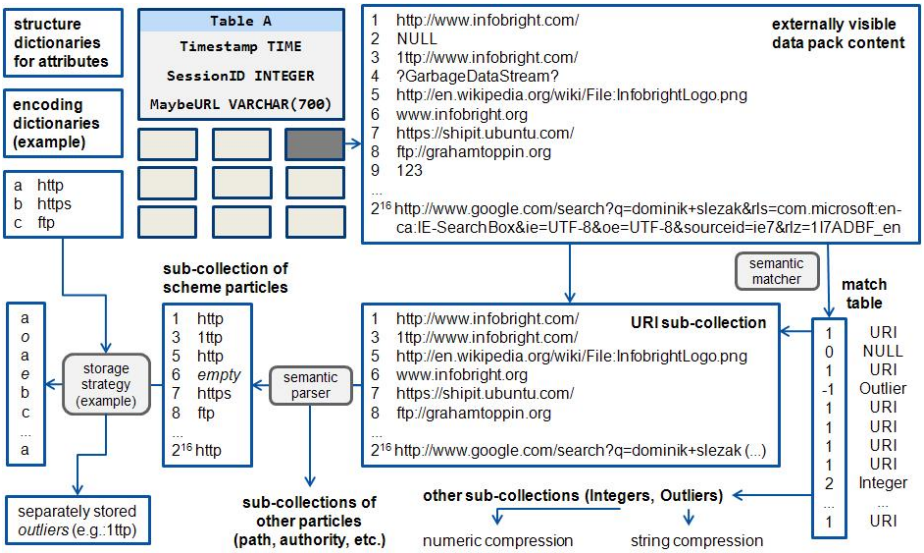

Fig. 2. Storage based on the domain knowledge. Data packs are decomposed onto sub-collections of values corresponding to different structures that can be further decomposed along particular structure specifications. It leads to sequences of more homogeneous (particles of) values that can be better compressed. For example, it is shown how to store the scheme particles of the URI values.

5 Implementation

For practical reasons, the currently implemented framework differs slightly from the previously-discussed ideas. We represent structures occurring for a given attribute (like, e.g., URIs and integers for MaybeURL, Figure 2) within a single *decomposition rule*. Such decomposition rule might be treated as disjunction of possible structures (e.g.: URI or integer or NULL or outlier), although its expressive power may go beyond simple disjunctions. The main proposed components are as follows: 1) dictionary of available decomposition rules, 2) applying decomposition rules to data attributes, and 3) parsing values through decomposition rules. These components are added to the ICE/IEE implementation integrated with MySQL framework for the *pluggable storage engines* (Figure 3).[3]

The first component – the dictionary of available decomposition rules – corresponds to the newly introduced system table *decomposition_dictionary* that holds all available decomposition rules. The table is located in the system database *sys_infobright* and is created at ICE/IEE's installation. The table contains three columns: ID (name of a decomposition rule), RULE (definition of a decomposition rule), and COMMENT (additional comments, if any). The rules can be added and modified with help of the following three stored procedures:

[3] http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html

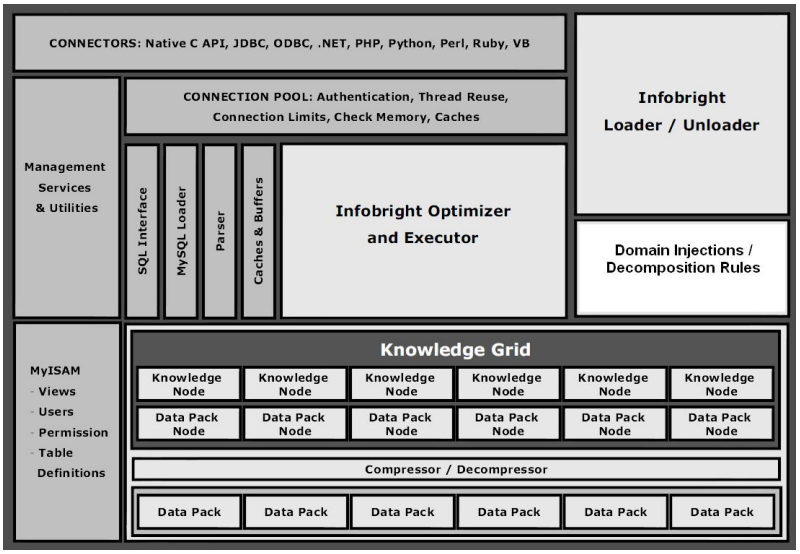

Fig. 3. Conceptual layout of the current ICE/IEE implementation (as of June 2011). The white box represents the components related to domain injections that have significant impact on the load processes and the (meta)data layers. The dark boxes are adapted from MySQL. (MySQL management services are applied to connection pooling; MyISAM engine stores catalogue information; MySQL query rewrite and parsing modules are used too.) MySQL optimization and execution pieces are replaced by the code designed to work with the compressed data packs, as well as their navigation information and rough values stored in *data pack nodes* and *knowledge nodes*, respectively. (Not to be confused with nodes known from MPP architectures [3].)

```
CREATE_RULE(id,rule,comment)
UPDATE_RULE(id,rule)
CHANGE_RULE_COMMENT(id,comment)
```

Currently, the *decomposition_ dictionary* table accepts rules defined in the simplistic language accepting concatenations of three types of primitives:

- Numeric part: Nonnegative integers, denoted as `%d`.
- Alphanumeric part: Arbitrary character sequences, denoted as `%s`.
- Literals: Sequences of characters that have to be matched exactly.

For instance, the IPv4 and email addresses can be expressed as `%d.%d.%d.%d` and `%s@%s`, respectively, where "." and "@" are literals. Obviously, this language requires further extensions, such as composition (disjunction) or Kleene closure (repeating the same pattern). Also, the new types of primitives, such as single characters, may be considered. Nevertheless, the next section reports improvements that could be obtained even within such a limited framework.

The next component – applying decomposition rules to attributes – is realized by the system table *columns* that contains four columns: DATABASE_NAME, TABLE_NAME, COLUMN_NAME, and DECOMPOSITION. This table stores assignments of rules to data attributes identified by its first three columns. The forth column is a foreign key of ID from *decomposition_dictionary*. There are two auxiliary stored procedures provided to handle the rule assignments:

```
SET_DECOMPOSITION_RULE(database,table,column,id)
DELETE_DECOMPOSITION_RULE(database,table,column)
```

For example, the following statement

```
CALL SET_DECOMPOSITION_RULE('NETWORK','CONNECTION','IP','IPv4');
```

means that the column IP in the table CONNECTION will be handled by the decomposition rule IPv4, due to its definition in *decomposition_dictionary*.

If one of the existing rules needs to be revised by a domain expert, there are two possibilities: 1) altering the rule's definition per se if its general pattern is wrong, or 2) linking a specific data attribute to another rule. Once the rule's definition or assignment is changed, new data portions will be processed using new configuration but already existing data packs will remain unmodified.

The last component – parsing values through decomposition rules – should be considered for each data pack separately. Data packs contain in their headers information about the applied rule. Therefore, at this level, the architecture implements the above-mentioned flexibility in modifying decomposition rules – for the given attribute, different packs can be parsed using different rules.

Currently, decomposition rules affect only data storage. There are no changes at the level of rough values, i.e., they are created as if there was no domain knowledge available. Internal structure of data packs follows Figure 2. In the match table, given the above-described language limitations, there is a unique code for all values successfully parsed through the decomposition rule, with additional codes for NULLs and outliers. In future, after enriching our language with disjunctions, the codes will become less trivial and match tables will reoccur at various decomposition levels. Sub-collections to be compressed correspond to the `%d` and `%s` primitives of parsed values. A separate sub-collection contains alphanumeric outliers. At this stage, we apply our previously-developed algorithms for compressing sequences of numeric, alphanumeric, and binary values [11]. The original data packs can be reassembled by putting decompressed sub-collections together, using the match tables and decomposition rules' specifications.

As reported in the next section, the proposed framework has potential impact on data load, data size, and data access. On the other hand, it also yields some new types of design tasks. For example, the domain injections will eventually lead towards higher complexity of Infobright's Knowledge Grid, raising some interesting challenges with respect to rough values' storage and usage. One needs to remember that the significant advantage of rough values lays in their relatively small size. However, in case of long, richly structured varchar attributes, we should not expect over-simplistic rough values to be informative enough.

6 Experiments

We tested the described framework against alphanumeric columns in the real-world tables provided by the ICE/IEE users. Decomposition rules were chosen according to preliminary analysis of data samples. The rules are evaluated with respect to the three following aspects that are crucial for the users:

- **Load time**: It includes parsing input files and compressing data packs. With a decomposition rule in place, the parsing stage includes also matching the values in each of data packs against the rule's structure. For more complex rules it takes more time. On the other hand, more complex rules lead to higher number of simpler sub-collections that may be all together compressed faster than collections of original varchar values.
- **Query time**: Currently, it is related to decompression speed and to the cost of composing separately stored particles into original values. Decompression and compression speeds are not necessarily correlated. For example, for sub-collections of numeric particles our decompression routines are much faster than corresponding compression [11]. In future, query time will be reduced due to higher informativeness of rough values, yielding less frequent data pack accesses. We expect it to be more significant than any overheads related to storing and using more compound rough values.
- **Disk size**: It is primarily related to data compression ratios. The rules decompose values into particles, whose sub-collections are compressed independently by better adjusted algorithms. For example, it may happen that some parts of complex varchars are integers. Then, numeric compression may result in smaller output, even though there is an overhead related to representing packs by means of multiple sub-blocks.

Table 1 illustrates load time, query time, and disk size for the corresponding decomposition rules, measured relatively to the situation with no domain knowledge in use. We examined data tables containing single alphanumeric attributes. Results were averaged over 10 runs. Load time is likely to increase when decomposition rules are applied. However, we can also see some promising query speedups. Compression ratios are better as well, although there are counterexamples. For instance, decomposition rule `%s://%s.%s.%s/%s` did not lead to possibility of applying compression algorithms that would be adjusted significantly better to particular URI components. On the other hand, URI decomposition paid off by means of query time. In this case, shorter strings turned out to be far easier to process, which overpowered the overhead related to a need of concatenating them into original values after decompression.

Besides the data set containing web sites parsed with the above-mentioned URI decomposition rule, we considered also IPv4 addresses and some identifiers originating from the telecommunication and biogenetic applications. Such cases represent mixtures of all types of the currently implemented primitives: numerics (`%d`), strings (`%s`), and literals (such as `AA` or `gi` in Table 1).

Table 1. Experiments with three aspects of ICE/IEE efficiency: load time, query time, and disk size. Query times are reported for SELECT * FROM table INTO OUTFILE. Results are compared with domain-unaware case. For example, query time 50.1% in the first row means that the given query runs almost two times faster when the corresponding decomposition rule is used. Five data sets with single alphanumeric attributes are considered, each of them treated with at least one decomposition rule. There are five rules studied for the last set.

data type	decomposition rule	load time	query time	disk size
IPv4	`%d.%d.%d.%d`	105.8%	50.1%	105.9%
id_1	`00%d%sAA%s%d-%d-%d`	156.4%	96.1%	87.6%
id_2	`gi%d-%s_%s%d%s`	92.7%	61.8%	85.1%
URI	`%s://%s.%s.%s/%s`	135.3%	89.7%	152.6%
logs	notice 1	113.3%	88.1%	67.5%
	notice 2	113.2%	105.4%	97.0%
	notice 3	113.1%	82.2%	61.5%
	notices 1,3 generalized	103.6%	71.2%	40.9%
	notices 1,2,3 generalized	132.2%	100.4%	82.2%

The last case (denoted as logs) refers to the data set, where each value follows one of three, roughly equinumerous distinct structures (denoted as notices 1, 2, and 3) related to three subsystem sources. Given that the currently implemented language of domain injections does not support disjunction, our first idea was to adjust the decomposition rule to notice 1, 2, or 3. Unfortunately, fixing the rule for one of notices results in 66% of values treated as outliers. Nevertheless, Table 1 shows that for notices 1 and 3 it yields quite surprising improvements. We also investigated more general rules addressing multiple notices but not going so deeply into some of their details. (This means that some parts that could be finer decomposed are now compressed as longer substrings.) When using such a rule for notices 1 and 3, with 33% of outliers (for notice 2) and slightly courser way of compressing 66% of values (for notices 1 and 3), we obtained the best outcome with respect to load speed, query speed, and compression ratio. However, further rule generalization aiming at grasping also notice 2 led us towards losing too much with respect to values corresponding to structures 1 and 3.

The above example illustrates deficiencies of our current decomposition language. It also shows that the same columns can be assigned with different rules and that it is hard to predict their benefits without monitoring data and queries. It emphasizes the necessity of evolution of the domain knowledge and the need for adaptive methods of adjusting that knowledge to the data problems, which can evolve as well. Regardless of whether the optimal approach to understanding and conducting such evolution is manual or automatic, it requires gathering the feedback related to various database efficiency characteristics and attempting to translate it towards, e.g., the decomposition rule recommendations.

7 Conclusions

Allowing domain experts to describe the content of their data leads to substantial opportunities. In this paper, we discussed how to embed such descriptions into an RDBMS engine. Experiments with data compression show significant potential of the proposed approach. They also expose that more work shall be done with respect to human-computer interfaces and the engine internals.

One of our future research directions is to use domain knowledge not only for better data compression but also for better data representation. For the specific database solution discussed in this article, one may design domain-driven rough values summarizing sub-collections of decomposed data packs.

Another idea is to avoid accessing all sub-collections of required data packs. In future, we may try to use data pack content information to resolve operations such as filter or function computation, keeping a clear border between domain-unaware execution modules and domain-aware data processing.

References

1. M. Cannataro and D. Talia: The Knowledge Grid. Commun. ACM 46(1): 89–93 (2003).
2. D. Chen and X. Cheng (Eds.): Pattern Recognition and String Matching. Kluwer Academic Publishers (2002).
3. J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton: Architecture of a Database System. Foundations and Trends in Databases 1(2): 141–259 (2007).
4. S. Inenaga, et al.: On-line Construction of Compact Directed Acyclic Word Graphs. Discrete Applied Mathematics (DAM) 146(2): 156–179 (2005).
5. J. K. Metzger, B. M. Zane, and F. D. Hinshaw: Limiting Scans of Loosely Ordered and/or Grouped Relations Using Nearly Ordered Maps. US Patent 6,973,452 (2005).
6. L. T. Moss and S. Atre: Business Intelligence Roadmap: The Complete Project Lifecycle for Decision-support Applications. Addison-Wesley (2003).
7. W. Pedrycz, A. Skowron, and V. Kreinovich (Eds.): Handbook of Granular Computing. Wiley (2008).
8. D. Ślęzak and G. Toppin: Injecting Domain Knowledge into a Granular Database Engine: A Position Paper. In Proc. of CIKM, ACM (2010) pp. 1913–1916.
9. J. F. Sowa: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole Publishing (2000).
10. P. W. White and C. D. French: Database System with Methodology for Storing a Database Table by Vertically Partitioning All Columns of the Table. US Patent 5,794,229 (1998).
11. M. Wojnarski, et al.: Method and System for Data Compression in a Relational Database. US Patent Application, 2008/0071818 A1.
12. J. Wróblewski, et al.: Method and System for Storing, Organizing and Processing Data in a Relational Database. US Patent Application, 2008/0071748 A1.