Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries

Dominik Ślęzak
Infobright Inc., Poland
slezak@infobright.com

Jakub Wróblewski
Infobright Inc., Poland
jakubw@infobright.com

Victoria Eastwood
Infobright Inc., Canada
victoriae@infobright.com

Piotr Synak
Infobright Inc., Poland
synak@infobright.com

ABSTRACT

Brighthouse is a column-oriented data warehouse with an automatically tuned, ultra small overhead metadata layer called Knowledge Grid, that is used as an alternative to classical indexes. The advantages of column-oriented data storage, as well as data compression have already been well-documented, especially in the context of analytic, decision support querying. This paper demonstrates additional benefits resulting from Knowledge Grid for compressed, column-oriented databases. In particular, we explain how it assists in query optimization and execution, by minimizing the need of data reads and data decompression.

1. INTRODUCTION

Beginning from the middle 80's in academia [10] and the middle 90's in industry [42], one can observe a permanent growth of interest in column-oriented databases [23, 31, 39, 40]. The differences between the column- and row-oriented architectures show that the first ones are more suitable for analytic data warehousing, with selective access to small subsets of columns and emphasis on data compression, while the second ones seem to be a better choice for OLTP systems (cf. [17, 38]). There are also approaches attempting to take an advantage of both strategies, suggesting mixed horizontal/vertical decomposition, not so strictly column-driven data processing, etc. (cf. [2, 21]).

Regarding logical model, column-oriented and row-oriented architectures provide the same framework, though, e.g., denormalization would be more acceptable in the column-oriented case, if there is a need to apply it at all. Furthermore, the fundamental principles of database tuning and administration remain of a similar kind, with a need of reorganizing the physical model subject to evolution of the query workload and data regularities (cf. [9, 36]). Given that it may be quite a hard task depending on the real-life data and querying specifics, we can observe an increasing number of database solutions that give up the original means for physical model tuning (see e.g. [18, 35]).

In this paper, we discuss Brighthouse [19, 44], which combines the advantages of being column-oriented and storing highly compressed data with an alternative approach to indexing. We follow an idea of automatic creation and usage of higher-level *data about data* (see e.g. [3, 5, 6, 8]), here referred to as *Knowledge Grid*, available while the query optimization and execution with no need of accessing the actual data. A difference with respect to standard approaches to indexing is that the elements of Knowledge Grid, here referred to as *Knowledge Nodes*[1], describe relatively large (combinations of) portions of compressed data, referred to as *Data Packs*, instead of single data items or rows. With this respect, Brighthouse should be compared, e.g., to Netezza's *nearly ordered maps* (better known as *zone maps*) [25] or Sand's compressed segments' metadata [16], as well as to various other, more or less conventional approaches to data block-level indexing. However, we use the data about data in a significantly extended way.

Knowledge Nodes are far smaller than standard indexes, which results in their faster, in-memory processing, as well as in the ability of storing more of their types, for far more (combinations of) tables and columns. This advantage fits well with a growing need of dealing with ad hoc, unpredictable ways of querying [14]. It also helps to eliminate the previously-mentioned requirements of the physical model (re-)tuning. For the same reason, we try to avoid also other "knobs", treating both simplicity and performance of Brighthouse as important factors. In particular, unlike in many other solutions, we do not define Data Packs by using any kind of data partitioning mechanism and, as a result, Brighthouse's Knowledge Grid should not be interpreted by means of standard metadata over the partitioned data sets.

Knowledge Grid serves as a kind of "mediator" between the query optimization/execution and the data storage/(de) compression layers of Brighthouse. The primary objective of Knowledge Nodes is to identify Data Packs that do not require decompression while query resolving, along the lines of the theory of *rough sets* – originally established as a

[1] Our understanding of Knowledge Grid is different than that in grid computing or semantic web [7], though there are some intuitive analogies (see Section 3). In the same way, Knowledge Nodes are not to be confused with any type of *nodes* in grid/distributed/parallel architectures [38].

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212)869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

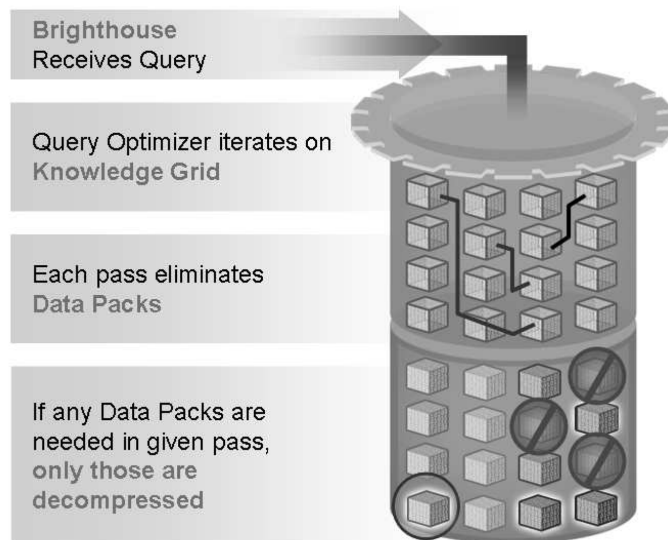

Figure 1: Brighthouse engine. Data stored within compressed Data Packs; Metadata stored within Data Pack Nodes and Knowledge Nodes; Query Optimizer and Executor combined together in an iterative fashion.

methodology of data mining and knowledge discovery [32, 33], though recently finding its applications also in a more database-related research [22, 29]. Knowledge Grid can be also used, e.g., to improve the flow of data being decompressed, as well as to iteratively refine Knowledge Node-based information to better assist with dynamic optimization of query execution. With this respect, one should also refer to the principles of adaptive query processing [4, 12].

As mentioned in the abstract, column orientation and data compression have been already widely studied in the database-related literature. Therefore, we focus on the components of Knowledge Grid and the scenarios of their usage in query optimization/execution. Further, although Brighthouse has been already commercially proven to work efficiently with terabytes of compressed data [19], we regard this paper as the means for introducing only the basics of our approach, with experimental framework limited to minimum. On the other hand, we should emphasize that the overall architecture would not be complete without appropriately adjusted algorithms providing the average of 10:1 data compression ratio (including the overhead of Knowledge Nodes; with decompression speed meeting the performance requirements). We refer to [15, 18, 41, 45] as the state of the art in database compression. The reader may find the details of our compression algorithms in [43].

Another layer is to handle the ODBC connectors, query parsing, permission management, etc. Brighthouse leverages MySQL's *pluggable storage engine* platform [28] to achieve full database functionality. In particular, our query optimizer takes partially an advantage of MySQL's syntactic rewrites and parsing mechanisms, though the major part at this level is replaced by the Knowledge Node-based methodology. Replacement of the optimizer is one of differentiators when comparing to the other data storage engines, limited to single-table processing [13].

The paper is organized as follows: Section 2 discusses the basics of column-oriented data storage with compression. Section 3 introduces Knowledge Grid and presents some examples of Knowledge Nodes. Section 4 introduces principles of Knowledge Node-based query optimization/execution. Section 5 provides an illustration of the Knowledge Node-based adaptive query processing. Section 6 describes MySQL environment and reports a few examples of performance over real-life data. Section 7 concludes the paper.

2. DATA PACKS

As it has been pointed out already in the middle 80's, data compression aims at minimizing the disk I/O costs while executing queries [11]. Brighthouse is vertically decomposed, which enables to better compress data. Since each column stores a single data type (as opposed to rows that typically contain different types and value ranges), compression can be adjusted to each of columns specifically (cf. [1, 37]). In Brighthouse, the rows' values over each of the columns are stored by the 64K-item[2] groupings – Data Packs, which are illustrated by means of the lowest layer in Fig. 1. We actually split the rows onto 64K-row groupings and, for each of them, we store the values of each of the columns in a separate Data Pack. We refer to such row groupings as *Row Packs* and we suggest comparing them with the minipage-based *PAX* model [2], further investigated, e.g. with regards to compression efficiency, in [41, 45]. The rows in Brighthouse are neither sorted nor partitioned with respect to any column values. It is important for achieving high data load speed, as well as avoiding administrative overheads and higher costs of physical model re-tuning [19, 44].

The choice of the number of rows in Row Packs is an interesting problem, partially referrable to the pagesize tuning in other database frameworks (cf. [42]). Operating with larger collections of items enables the compression routines to take more advantage of data regularities. On the other hand, such regularities may change across the data. Hence, well-designed compression algorithms may adjust better to smaller, potentially more locally homogeneous portions of values. In Brighthouse, compression algorithms vary not only with respect to the data types but also with respect to

[2]By 64K we mean $2^{16} = 65,536$ elements.

regularities automatically observed inside particular Data Packs. As mentioned in Section 1, we provide, on average, 10:1 compression ratio, counted with all possible overheads, simply by comparing the size of input data files with the complete size of Data Packs and Knowledge Nodes.

The number of items put into each Data Pack has also an impact on dynamics of data accessibility. One would like to minimize the number of Data Pack reads while querying. Vertical decomposition helps as analytic queries usually refer to a relatively small percentage of columns (cf. [1, 37]). Further way to limit the data reads is to equip Data Packs with such metadata (*data about data*) that enable to identify and exclude the data portions irrelevant to a query, or to resolve it without the need of accessing particular Data Packs at all, even if they are (partially) relevant. The idea of fast pre-excluding irrelevant data basing on their compacted forms and various types of metadata is quite popular (cf. [3, 22]), easily translatable to the level of larger Data Packs and Row Packs (cf. [16, 25]). On the other hand, applying compacted data representation instead of the actual data portions to answer to a query is assumed to be a domain of *imprecise* rather than *precise* query resolving (cf. [5, 30]). In the foregoing sections, we describe how this idea is used in Brighthouse to handle *precise* querying.

The key features of metadata understood as above should be *quality* and size. Smaller Data Packs would be labeled with more precise metadata, but the resulting larger number of metadata instances needed for larger number of Data Packs would decrease efficiency of their usage. On the other hand, larger packs result in less precise but also less sized metadata structures, though here we should also remember about the above discussion on data compression and a potential overhead related to extracting larger pieces of irrelevant items together with those really needed. In our tests, we found 64K items per Data Pack as a reasonable number to optimize both data accessibility and compression.

3. KNOWLEDGE GRID

There are two metadata layers in Brighthouse. The first one, called *Data Pack Nodes*, is more comparable to other approaches (see e.g. [16, 25]). Each Data Pack has a corresponding Data Pack Node containing such statistics as the minimum/maximum values (interpreted specifically for different data types), the sum of values (only in case summing the values makes sense), the number of null values, and the number of all elements in the Data Pack. We refer to Fig. 2a, which illustrates the Data Packs' Data Pack Nodes.[3]

The second layer, called Knowledge Grid, contains more advanced structures – Knowledge Nodes, which refer to more detailed information about Data Packs, as well as to various kinds of dependencies between Data Packs belonging to different Row Packs, columns and tables. It is important to note that Data Pack Nodes and Knowledge Nodes together take roughly 1% of the compressed data size. All Data Pack Nodes and Knowledge Nodes relevant to the given query can be easily stored in memory, supporting various kinds of query optimization/execution operations independently from accessing and decompressing Data Packs.

The term "Knowledge Grid" often refers to web services and semantic web, as well as to parallel computing and distributed data, to mention just a few areas. In particular, it may relate to synthesizing data-based knowledge and to enabling the search and/or database engines to answer queries and draw conclusions from the masses of data (cf. [7]). We focus on the *Database* Knowledge Grid, interpreted as the means for extracting, representing, and applying high-granularity knowledge about data, at the level of (combinations of) Data Packs. Let us give some basic examples of Knowledge Nodes implemented in Brighthouse:

- *Histograms* (*HISTs*) are built for numeric columns. HIST collects information about each of Data Packs for the given column. For each Data Pack, HIST splits the range between its min and max onto 1024 equal intervals. HIST contains binary information (which makes it far smaller than in case of various versions of widely common histograms). For each Data Pack, each of 1024 intervals is labeled with 1, if there is a value in the Data Pack which drops into the given interval, and 0 otherwise. To summarize, our HISTs are not so comparable to the histograms widely studied in the literature [20], as they provide binary and very local Data Pack-related information. On the other hand, extensions of the currently implemented HISTs are still possible, while keeping in mind the constraint of a relatively small size of the Knowledge Grid elements.

- *Character Maps* (*CMAPs*) are built for alpha-numeric columns. Like above, CMAP collects information about each of Data Packs. For each Data Pack, CMAP stores binary information about occurrence of particular characters at particular positions of the string values stored in the Data Pack. For example, if there is no string value with character "b" at the third position in the whole Data Pack, then the corresponding CMAP's box for this Data Pack is 0. As a summary, CMAPs, like HISTs, should be also compared to the already existing methodologies.

- *Pack-to-Packs* are built for the pairs of data tables. Each Pack-to-Pack is related to a specific join relation based on equality of two columns in two tables. It is kept in Knowledge Grid only if the given join relation has already occurred in one of the queries. Pack-to-Pack is a binary matrix labeled by the pairs of identification numbers of Row Packs from two tables. We refer to such numbers as *Row Pack IDs.* For the pair of Row Packs from two data tables, the matrix value is 1, if there is at least one pair of rows belonging to these Row Packs which satisfy the join relation, and 0 otherwise. Pack-to-Packs should be certainly compared to more traditional indexes. They are, actually, a simple example of the previously mentioned general ability to translate widely known index structures to the language of Data Packs and Row Packs.

The simplest case of using Knowledge Nodes is fast, metadata-only-based exclusion of data irrelevant to a query. For example, given a query with a BETWEEN condition over a numeric column, we can use its Data Pack Nodes (specifically the min/max values) and its HIST to identify as many Data Packs with no items satisfying the condition as possible. In the same way, Data Pack Nodes and CMAP can help in exclusion of Data Packs with no items satisfying, e.g., a LIKE condition. All above Knowledge Nodes can be

[3]In the example related to Fig. 2, we are interested only in the min and max values, so illustration is simplified.

also useful in elimination of collections of tuples not satisfying join relations, etc. This most straightforward application has been followed by other methodologies described in the literature, though, as already mentioned, only relatively simple statistics were used and only relatively simple query execution steps were considered so far (see e.g. [16, 25]).

In Section 6, we experimentally show that operating with advanced Knowledge Nodes on top of simple Data Pack Nodes can provide significant improvements. Beforehand, in Section 4, we show that Data Pack Nodes and Knowledge Nodes can be applied not only to data exclusion, but also to other steps of query optimization and execution. Here, let us finish with just a few more examples, illustrating a variety of data-driven structures that may be useful (cf. [44]):

- Distribution of numeric values within the given Data Pack's range may be biased. There may be large subranges with no values occurring. Also, some 1's in HIST can occur just because of a single item, laying almost at the edge of a given interval. A more intelligent way of cutting the ranges may maximize the overall length of intervals labeled with 0's, using as small number of cuts as possible. This task partially resembles some of machine learning problems [27] and may be also compared to inducing multi-dimensional histogram structures [6, 20].
- Data Pack Nodes and Knowledge Nodes created independently for each of columns do not take advantage of column correlations or dependencies. Being of recently growing interest in the area of query optimization [12, 24], such inter-column value relationships can be expressed as Knowledge Nodes too. For example, within each Row Pack, the Data Pack Node / HIST / CMAP statistics can be calculated over subsets of only those rows which satisfy some conditions on other columns. Again, it looks like a task close to machine learning, as we want to search for column conditions affecting statistics of other columns to the largest degree.
- Analytic queries are often aggregations over a table T1 subject to conditions defined over another table T2. Statistics related to T2 can be expressed as new Knowledge Nodes of T1. For example, we can keep the minimum and maximum values of a column T2.a over all rows in T2, which are in a T1.b=T2.c relation with the rows belonging to every given Row Pack in T1. It may be interpreted as a kind of *virtual* data denormalization where, however, only Knowledge Nodes are created. Potentially, it has application not only in case of simple one-to-many relations, but also in case of more complex joins.

One should not exaggerate with the number of different Knowledge Nodes. In many cases, combined application of more standard Knowledge Nodes (HIST, CMAP, Pack-to-Pack) enables to obtain almost the same performance characteristics as in case of more complicated structures described above. Nevertheless, it is required to conduct further research also on automatic optimization and maintenance of the whole Knowledge Node groups, to keep the particular Knowledge Nodes' functionalities complementary to each other, and to keep the Knowledge Node group overall size at a level around 1% of the compressed data size. Again, it partially resembles some tasks known from data mining and, as an optimization problem, it may be addressed by adapting some well-known heuristics. Certainly, it can be also discussed in the context of physical data model tuning (cf. [9]), though we would like to stress that relatively small sizes of Knowledge Nodes, when compared with the structures used in other solutions, make the whole Brighthouse framework more flexible.

4. OPTIMIZATION AND EXECUTION

Given Knowledge Nodes and most simple examples of their usage introduced, let us proceed with a description of the top layer of the Brighthouse core solution. As symbolically illustrated by Fig. 1, we regard the modules of query optimization and execution as entirely combined. The first reason for this is that, although our design of the query execution plan as a result of query optimization is quite standard, the query optimizer works on the basis of simulating the query execution against Knowledge Grid. Starting with (logical combinations of) single-table conditions, the optimizer uses available Data Pack Nodes and Knowledge Nodes to classify Data Packs into three categories:

- *Irrelevant* Data Packs, which have no data elements relevant for further resolving the given query
- *Relevant* Data Packs, where all of data elements are relevant for further resolving the given query
- *Suspect* Data Packs, which cannot be classified as Relevant or Irrelevant on the basis of available Data Pack Nodes and Knowledge Nodes

In Section 3, while talking about Data Pack Node and Knowledge Node-based processes of excluding as many Data Packs as possible, we referred only to the first above category. Using all three categories provides far more possibilities. For example, optimization of multi-join query execution plans usually starts with estimation of selectivity of single-table condition components. This is also the case in Brighthouse, where, given no indexes, the condition components are analyzed against Knowledge Grid. In such a case, identifying Relevant Data Packs provides valuable information. As another example, consider an analytic query with some after-SELECT statistics to be calculated. Such statistics can be partially reconstructed based on Data Pack Nodes of Relevant Data Packs. If we are, e.g., interested in calculating the total of some column over the rows satisfying some condition, if we know that a given Data Pack is Relevant with respect to that condition, then we can use the total value stored in the corresponding Data Pack's Data Pack Node to contribute to the final result, avoiding decompression.

Our inspiration to consider three categories of Data Packs grew from the theory of rough sets [32, 33], where the data is split among positive, negative, and boundary regions with respect to their membership to the concepts of interest, often called decisions. In the theory of rough sets, the data rows can be analyzed only via their values on available columns. The rows can, e.g., correspond to some bank's customers and the concept of interest – to information whether each given customer had any payment problems. The customers are organized into groups with the similar values of available demographic and account history attributes. Positive region consists of the groups of rows completely included into the decision; negative region consists of the groups with no intersection with the decision; and boundary region consists of

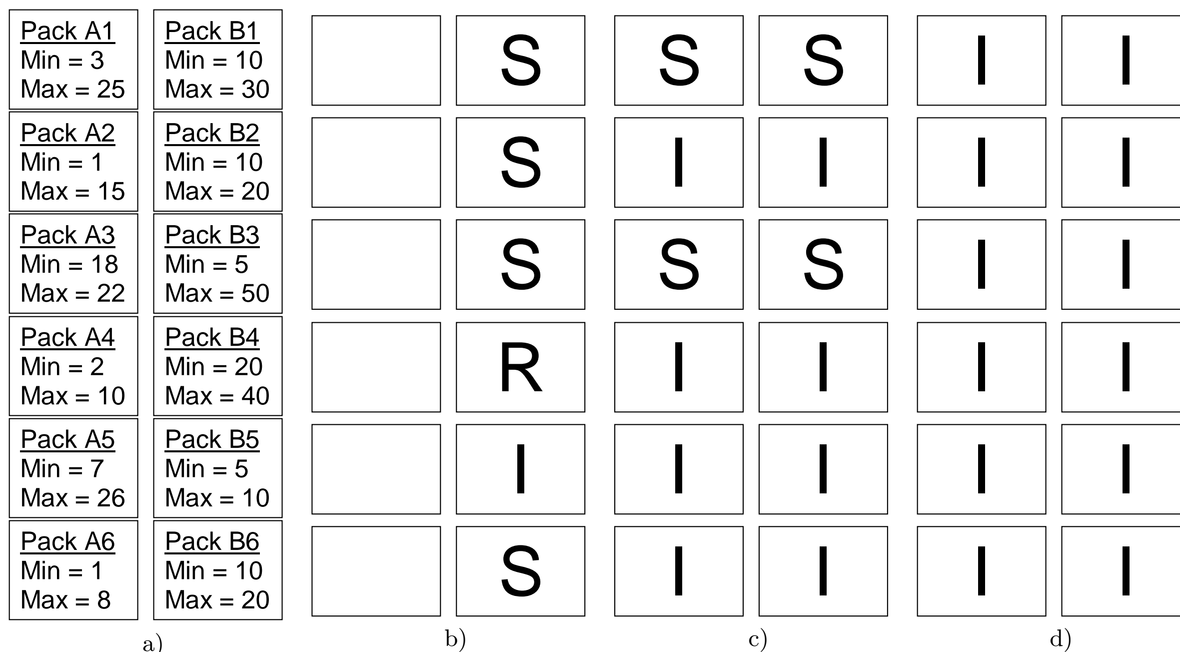

Figure 2: (a) Simplified Data Pack Nodes for the table T considered in Section 5; (b,c,d) Status of Data Packs at particular described stages of query processing. Symbols R/S/I denote Relevant/Suspect/Irrelevant Data Packs, respectively.

groups of the third kind, where the available attributes are not sufficient to tell whether the given group of rows fully supports or negates the decision. Although the ways of understanding the groups of rows in Brighthouse and in the original rough set framework are different, the Data Packs being Relevant, Irrelevant, and Suspect with respect to a given query intuitively correspond to the positive, negative, and boundary regions with respect to a given decision.

The three-mode classification can be conducted also for the Row Packs or their combinations called *Tuple Packs*. In case of Row Packs, we use Knowledge Grid to classify their Data Pack components with respect to, e.g., conjunctions and disjunctions of multi-column filters. Single-column results are then combined using a three-valued logic. For example, if a Row Pack is evaluated against disjunction of conditions and one of them turns out to have the Relevant status, then the Row Pack becomes Relevant automatically. Relevant Tuple Packs occur relatively rarely in the case of joins based on equalities – it would mean that, for a pair of Row Packs from two different tables, the given equality relation is satisfied for all pairs of their rows. However, it is more likely for joins based on inequalities – it may often happen that the minimum of one Data Pack is greater than the maximum of another Data Pack and, hence, all pairs of values stored in those two Data Packs satisfy the inequality. Multi-table and single-table conditions defined over Tuple Packs can be combined using logical operators or they can get mixed with Data Pack Node-based estimations of SELECT expressions coming from subqueries. As a summary, Brighthouse is able to apply Knowledge Grid to thoroughly analyze Tuple Packs in a way corresponding to the given query, prior to (or during) accessing the actual data.

Further reason to treat the Brighthouse optimization and execution modules as so closely related to each other is that Data Pack Nodes and Knowledge Nodes accompany and potentially modify the query execution through all its steps, by themselves or in combination with information obtained from decompressed Data Packs, along the lines of adaptive query processing (cf. [4, 12]). It also finally clarifies that our way of using data about data is more advanced than just a single-step phase of filtering out the irrelevant blocks (cf. [3, 16, 22, 25]). The next section contains a very simplified case study to clarify the basics of such a Data Pack Node and Knowledge Node-based adaptive approach. The reader is, however, encouraged to consider more advanced possible scenarios. For example, the "ROUGH ORDER BY" procedure mentioned below may be applied to optimization of the order of decompressing Data Packs to avoid large intermediate structures while resolving ORDER BY, GROUP BY, or multiple JOIN operations.

5. ILLUSTRATIVE EXAMPLE

Consider table T with 350,000 rows and two columns A and B. We have six Row Packs: (A1,B1) corresponds to the rows 1-65,536 with Data Packs A1 and B1 containing their values on A and B, respectively. (A2,B2) corresponds to the rows 65,537-131,072, etc., until the last Row Pack (A6,B6) corresponding to the rows 327,681-350,000. The minimum and maximum values available in Data Pack Nodes of corresponding Data Packs are displayed in Fig. 2a. For simplicity, assume there are no nulls in T. Let us also remind that Knowledge Grid may contain more information about Data Packs, though we do not use it in this simple case study. The query of our interest is the following:

```
SELECT MAX(A) FROM T WHERE B>15;
```

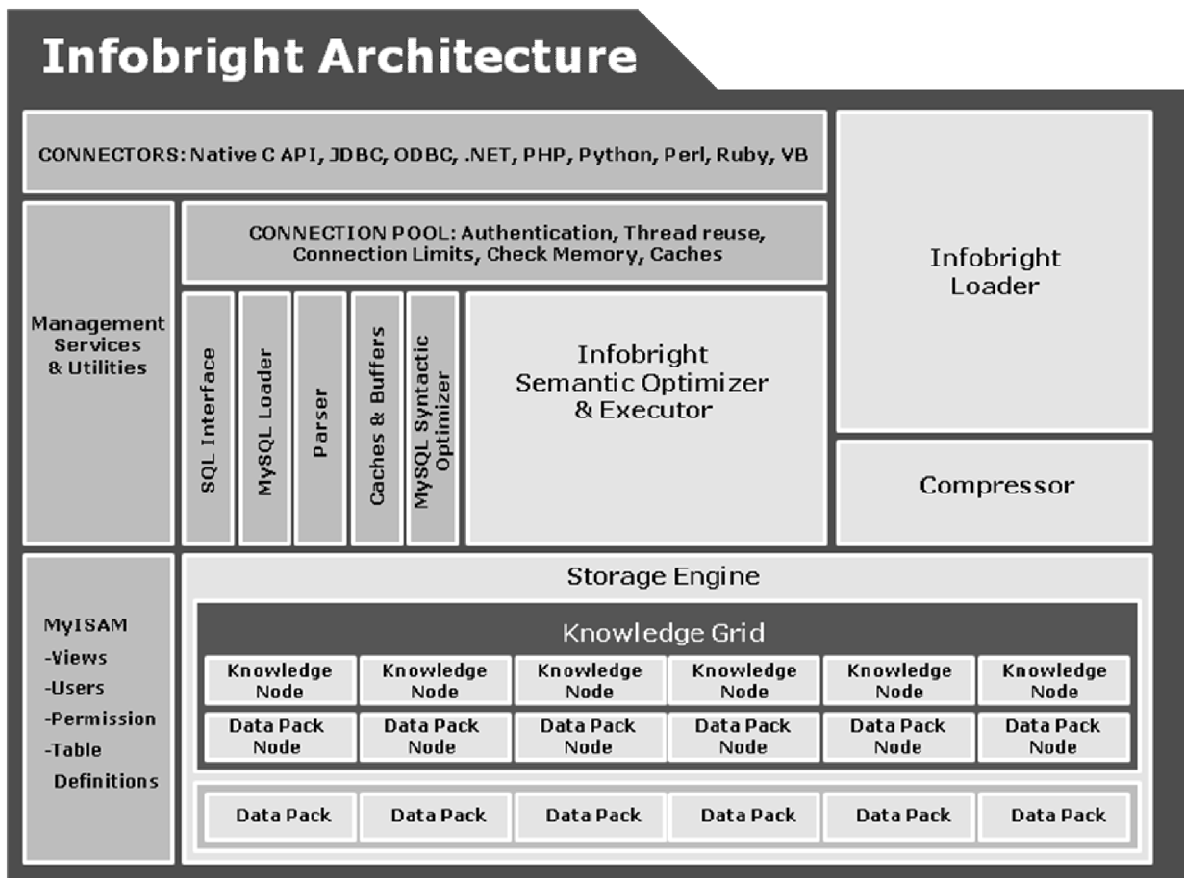

Figure 3: Integration of the core Brighthouse components with MySQL. Brighthouse Semantic Optimizer & Executor is the module replacing the original MySQL optimizer in the pluggable storage engine architecture.

For illustration purposes, we use so called *approximate query mechanism*, already mentioned in the previous section. One can consider its potential applications and correspondences to the *imprecise* query methodologies applied in various areas. However, we consider it only *inside* the Brighthouse engine, and we purposely use the word "mechanism" instead of "language". It is simply a convenient way to talk about how Knowledge Nodes assist in query processing, but with no strict relationship to the actual implementation. Having this in our minds, we can start as follows:

```
ROUGH MAX(A) FROM T WHERE B>15;
```

By using the word "ROUGH" we emphasize that we do not base on exact data, but only on Data Pack Nodes and Knowledge Nodes. The result of such operation should be treated as approximation, by a kind of analogy to the rough set regions discussed in the previous section. Using only the minimum and maximum values over the column B's Data Packs, we can see that B1, B2, B3, and B6 are Suspect, B4 is Relevant, and B5 is Irrelevant with respect to condition B>15, as illustrated by Fig. 2b. While approximating MAX(A), we do not consider A5 at all. Looking at the maximum value in A4, we obtain that MAX(A) is at least 10. Further, given 18 as the minimum value in A3, we get MAX(A) equal at least to 18. Further, we check whether there is any possibility that the result is higher than 18. This can be modeled as follows:

```
ROUGH ID FROM T WHERE B>15 AND A>18;
```

ROUGH ID is a command that returns the Row Pack IDs, together with its Relevant/Suspect/Irrelevant status. Condition A>18 is added to B>15 to let us focus on only those portions of data, which might put the value of MAX(A) higher than 18. Row Packs (A2,B2), (A4,B4), (A5,B5), as well as (A6,B6) become Irrelevant and the two remaining ones are Suspect, as in Fig. 2c. The above operation could be enriched by an attempt to sort Row Packs with respect to a chance of containing the actual maximum. This may be carried out using a heuristic algorithm based on available Data Pack Nodes and Knowledge Nodes, informally employed by command ORDER BY:

```
ROUGH ID FROM T WHERE B>15 AND A>18 ORDER BY A;
```

Design of good heuristics of this type is yet another large research topic remaining beyond the scope of this particular paper. For illustration purposes, assume that the applied heuristic algorithm finds (A1,B1) to more likely have MAX(A) greater than (A3,B3). Now, as there is no more information applicable at the level of Knowledge Grid (at least in this simplified case study), this is the time to access the data. We begin with Data Packs A1 and B1, according to the above "ROUGH ORDER BY" operation:

```
EXACT MAX(A) FROM T WHERE B>15 AND ID=1;
```

The word "EXACT" means that we go down into the data. Additional condition ID=1 means that we are interested in Row Pack (A1,B1). For the sake of illustration, assume the result of the above operation equals to 23, i.e., the maximum value of A for the rows in (A1,B1) satisfying condition B>15 is 23. Given information about possible MAX(A) updated, we iteratively come back to the "rough" level and formulate the next operation along the following lines:[4]

```
ROUGH ID FROM T WHERE B>15 AND A>23;
```

Now, all the Row Packs become irrelevant, as in Fig. 2d. There is no data left to update MAX(A) as being higher than 23. The value of 23 is reported as the result of the SQL statement. Let us note that only 2 out of 12 Data Packs needed to be accessed during all above operations.

6. BRIGHTHOUSE IN PRACTICE

Let us first briefly describe how it is possible to use Brighthouse in a real-life scenario. Clearly, the ideas presented in Sections 2-5 would not be applicable without ODBC connectors, etc., as mentioned in Section 1. Fig. 3 presents how we leverage MySQL's pluggable storage engine architecture [28] to achieve full database functionality. MySQL management services and utilities are used as the technology around connection pooling. As in the case of other MySQL storage engines (cf. [13]), MyISAM is used to store catalogue information such as table definitions, views, users, and their permissions. Brighthouse uses its own load/unload utilities, as this stage is related to data (de)compression and the updates of metadata stored in Knowledge Nodes. Brighthouse's query optimizer takes partially an advantage of MySQL's syntactic rewrites and parsing mechanisms. However, the major part at this level is replaced by the Knowledge Node-based methodology, with the MySQL's standard index support turned off. As also mentioned in Section 1, the other currently known MySQL storage engines are limited to single-table processing, using MySQL's optimizer/executor every time multi-table operations are involved.

We conducted experiments on benchmarks and databases we acquired in the proof-of-concept projects. The choice of reliable benchmarks is important though not easy (cf. [26, 34]). Every approach has its own preferences regarding how a database is designed or what data properties should be expected. In this paper, we limit ourselves to the results over real-life data, with more benchmark-oriented analysis to be provided in future. Given confidentiality constraints, we disclose the results only for the data samples, with the columns' names changed. More than in the answers to SQL queries or in the time required to obtain those answers with various hardware settings, in this paper, we are interested in comparison of the intensity of Data Pack reads with and without Data Pack Nodes and Knowledge Nodes applied. Obviously, one may claim that a more fair comparative experiment should involve the data equipped with Data Pack Nodes and Knowledge Nodes and, on the other hand, the same data equipped with standard-like indexes. The reason we did not follow this path is that, in our experiments, we did not assume any hints which indexes should be created. We did not assume any kind of database tuning, provided that Knowledge Grid is fully automatized, focusing on addressing the ad hoc query workload.

[4]One may claim that we should also add the NOT(ID=1) condition to prevent going back to (A1,B1). We omit it here because in the actual Brighthouse implementation we store information about already visited Row Packs anyway.

The first sample is a single table – five days of data of one of our customers. We call it FACT_TABLE. It has 676 million rows and 66 columns. There are 10,320 Data Packs for every column. The 0.25TB of raw data is compressed to 35.5GB, including Knowledge Grid. It gives us 7:1 compression ratio (comparing to the other data sets we analyzed, this one was relatively hard to compress). We conducted the tests on a machine with 3.2GHz Xeon processor and 6GB RAM. Let us start with the following:

```
SELECT COUNT(*) FROM FACT_TABLE
    WHERE CUSTOMER_ID BETWEEN 13000 AND 14000;
```

With Knowledge Grid turned off, we have to scan through all of the Data Packs of CUSTOMER_ID, even given the fact that the above BETWEEN range is expected to be very selective. With Knowledge Grid turned on, although the values of CUSTOMER_ID are quite unordered across the table and therefore the minimum and maximum values stored in Data Pack Nodes cannot help, about 90% of intervals in HIST of this column turn out to be labeled with 0. In case of this particular query, HIST narrows us down to only 7 Suspect Data Packs, with the rest of them classified as Irrelevant. Hence, the above query is completed nearly immediately, just a fraction of a second. The next query allows localizing customers from another range, who used a particular advertising banner:

```
SELECT COUNT(DISTINCT CUSTOMER_ID) FROM FACT_TABLE
    WHERE CUSTOMER_ID BETWEEN 8000 AND 9000
    AND BANNER_ID = 10891;
```

The query took 23.8 seconds. With Data Pack Nodes and Knowledge Nodes turned off, we would have to scan through 20,640 Data Packs. With Data Pack Nodes and Knowledge Nodes turned on, in the case of this particular BETWEEN range, the CUSTOMER_ID's HIST is not so efficient. BANNER_ID's HIST has 75% of 0 labels and quite useless Data Pack Nodes. If applied separately, those two HISTs result with 10,400 Suspect Data Packs. If combined, given that the conjunction of Irrelevant and Suspect Data Packs within the same Row Pack results with its Irrelevant status, the final number of Suspect Data Packs goes down to 6,136. Although, after all, this is rather a disappointing example, it illustrates potential benefits of the three-valued logic applied at the level of Data Pack classification. The next example's complexity is comparable to the above ones, though now we deal with alpha-numeric data types:

```
SELECT COUNT(*) FROM FACT_TABLE
    WHERE REFERRER LIKE "http://ozzyozbourn%";
```

Although we create Data Pack Nodes also for such cases (with min and max values interpreted lexicographically), this query is optimized mainly by CMAPs, which enable to determine 99 Suspect Data Packs out of 10,320. The query took 19.6 seconds. When comparing with the previous queries, we can see that the time required to access REFERRER's values is much longer than in the case of CUSTOMER_ID and BANNER_ID, given Data Packs' larger size. The following is a case of analytic query counting rows in different time intervals and action groups:

```
SELECT HIT_DATE, ACTIONS, AVG(CLICK) FROM FACT_
    TABLE GROUP BY HIT_DATE, ACTIONS ORDER BY 1, 2;
```

In our FACT_TABLE, there are only 10 combinations of the values of columns HIT_DATE and ACTIONS, resulting with 10 groups to be considered. Moreover, the values are quite regularly distributed. By using available Data Pack Nodes we identify a significant number of Data Packs as Relevant, i.e., fully belonging to one of the 10 HIT_DATE/ACTIONS groups. As Data Pack Nodes provide sufficient information to contribute to AVG(CLICK) in the case of Relevant Data Packs, it eventually turns out that we need to decompress only 5,261 out of $3\times10{,}320=30{,}960$ Data Packs being under consideration. It is interesting given that the above case is a typical "full scan" query. It took 118 seconds. The following one is a different example of large aggregation:

```
SELECT HIT_DATE, CAMPAIGN_ID, SUM(IMPRESSIONS),
    SUM(CLICKS), SUM(ACTIONS), SUM(VIEWS),
    SUM(INCOME), SUM(EXPENSE) FROM FACT_TABLE
    WHERE CAMPAIGN_ID > 0 GROUP BY 1, 2;
```

With ACTIONS instead of CAMPAIGN_ID as the grouping column, the query processing dynamics would be quite the same as before because the SUMs are resolvable by Data Pack Nodes of Relevant Data Packs like in the case of COUNT(*). However, with CAMPAIGN_ID we get 3,762 groups instead of 10. The three-mode classification based on Knowledge Grid restricts the number of Suspect Data Packs down to 74,113. However, given the overall number of involved Data Packs equal to $8\times10{,}320=82{,}560$, it is not too much of the gain. Moreover, with specific memory settings, it may be impossible to create an intermediate structure enabling to calculate the statistics for all groups during a single scan through the data. Here, given 6GB of RAM available, we were able to avoid multiple reads of the same Data Packs from disk to a large extent. Still, it turned out that smart ordering (a kind of "ROUGH ORDER BY" mentioned in previous sections) of the Data Pack reads saves 10% of time required for multiple scans. Namely, the Brighthouse engine is able to put in order the reads of Data Packs, which are likely to correspond to the same sets of HIT_DATE/CAMPAIGN_ID value combinations, basing on Data Pack Nodes and Knowledge Nodes. Such additional metadata-based optimization is going to give more savings if the proportion of data cardinalities to available RAM increases. It is also an additional illustration confirming that the future performance tests of Brighthouse should be far more advanced than it is presented in this paper.

We finish with an example of multi-table query. The above-discussed data set was not challenging enough with this respect because dimension tables contained just several thousands of rows. Let us consider the data coming from another project. There are two tables: SERVER_CONTENTS (112 million rows) and SERVER_USERS (18.6 million rows). As in the previous case, the names of tables and columns are not original. We choose to report the following query:

```
SELECT T1.X_CONTENT, T2.VISIT_TIME,
    COUNT(DISTINCT T2.URS_ID),
    COUNT(DISTINCT T2.SITE_ID), COUNT(*) FROM
    SERVER_CONTENTS T1, SERVER_USERS T2 WHERE
    T1.SITE_ID = T2.SITE_ID AND
    T1.ADD_KEY = T2.ADD_KEY GROUP BY
    T1.X_CONTENT, T2.VISIT_TIME;
```

Although Data Pack Nodes, HISTs, and CMAPs can partially assist in identifying Irrelevant pairs of Row Packs, the main benefit for the above query comes from Pack-to-Packs corresponding to the conditions T1.SITE_ID=T2.SITE_ID and T1.SITE_ID=T2.SITE_ID. When combined together along the lines of conjunction of two join relations, these two Pack-to-Packs allow for elimination of over 90% of Tuple Packs as being Irrelevant. One can note that operating with such a sparse matrix of pairs of Row Packs that need to be analyzed together opens various possibilities including, e.g., *decomposing* the JOIN execution step onto several reasonably sized substeps that can be merged later. Here, we restrict ourselves to reporting that conjunction of the above Pack-to-Packs resulted in complete irrelevance of over 66% of Row Packs from table SERVER_USERS. This means that over 66% of Row Packs in one of the tables did not contain any rows, which could match both join equalities with at least a single row in another table. Having such information available in memory, without accessing Data Packs, enabled to speed up this particular query three times.

7. CONCLUSION

We discussed the Brighthouse's [19, 44] approach to data organization based on compressed Data Packs (which should be compared to other known techniques of data storage and compression, see e.g. [15, 18, 41, 45]), the Brighthouse's Knowledge Grid layer (which should be compared to other, both academically and commercially developed metadata strategies [3, 16, 22, 25]), as well as the Knowledge Grid-based query optimization/execution principles seeking their origins in the theory of rough sets [32, 33] and remaining in general analogy with widely known techniques adaptive query processing [4, 12]. We wrapped up with reporting the experiments emphasizing the promising performance characteristics with respect to ad-hoc analytic queries, as well as with outlining the overall framework based on the MySQL pluggable storage engine platform [13, 28].

Although Brighthouse has been already proven to be a reliable and efficient commercial product, the performance analysis reported in this paper needs to be further developed, by carefully reconsidering the most popular database benchmarks (though we should also keep in mind the arguments presented in Section 6, cf. [26, 34]), as well as by more thoroughly investigating the most fundamental scaling parameters considered in the literature (cf. [2, 17]). Only such an extended framework will enable us to fully compare our approach with the others, especially those based on the column orientation (cf. [31, 37, 39, 40]).

Our objective in this introductory paper was mainly to present the concept of our *data about data* framework, Knowledge Grid, as one of the most innovative, key components of the whole architecture. Certainly, the results that we obtained with the particular components of Knowledge Grid being turned on and off require a deeper understanding in comparison other adequately indexed solutions like, e.g., the open source column-oriented databases [21, 23] or the data storage engines available within the above-mentioned MySQL platform. On the other hand, one should remember that the whole concept of Knowledge Grid was designed to cope with highly ad hoc querying scenarios, where the ability to appropriately index the data or, more generally, to appropriately tune the physical database model is highly questionable in its traditional meaning (cf. [14, 35]).

8. REFERENCES

[1] D.J. Abadi. Column Stores For Wide and Sparse Data. CIDR 2007: 292-297

[2] A. Ailamaki, D.J. DeWitt, M.D. Hill: Data page layouts for relational databases on deep memory hierarchies. VLDB J. 11(3): 198-215 (2002)

[3] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, A.S. Tosun: Approximate Encoding for Direct Access and Query Processing over Compressed Bitmaps. VLDB 2006: 846-857

[4] S. Babu, P. Bizarro. Adaptive Query Processing in the Looking Glass. CIDR 2005: 238-249

[5] K.S. Beyer, P.J. Haas, B. Reinwald, Y. Sismanis, R. Gemulla: On synopses for distinct-value estimation under multiset operations. SIGMOD 2007: 199-210

[6] N. Bruno, S. Chaudhuri, L. Gravano: STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD 2001: 211-222

[7] M. Cannataro, D. Talia. The knowledge grid. Commun. ACM 46(1): 89-93, 2003

[8] S. Chakkappen, T. Cruanes, B. Dageville, L. Jiang, U. Shaft, H. Su, M. Zait: Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g. SIGMOD 2008: 1053-1063

[9] S. Chaudhuri, V.R. Narasayya: Self-Tuning Database Systems: A Decade of Progress. VLDB 2007: 3-14

[10] G.P. Copeland, S. Khoshafian: A Decomposition Storage Model. SIGMOD 1985: 268-279

[11] G.V. Cormack: Data Compression on a Database System. Commun. ACM 28(12): 1336-1342 (1985)

[12] A. Deshpande, Z.G. Ives, V. Raman: Adaptive Query Processing. Foundations and Trends in Databases 1(1): 1-140 (2007)

[13] Enterprise Data Warehousing with MySQL. MySQL Business White Paper, 2007

[14] D. Feinberg, M.A. Beyer. Magic Quadrant for Data Warehouse Database Management Systems. Gartner RAS Core Research Note G00151490, 2007

[15] P. Ferragina, R. Grossi, A. Gupta, R. Shah, J.S. Vitter: On Searching Compressed String Collections Cache-Obliviously. PODS 2008: 181-190

[16] R. Grondin, E. Fadeitchev, V. Zarouba. Searchable archive. US Patent 7,243,110, July 10, 2007

[17] J.M. Hellerstein, M. Stonebraker, J.R. Hamilton: Architecture of a Database System. Foundations and Trends in Databases 1(2): 141-259 (2007)

[18] A.L. Holloway, V. Raman, G. Swart, D.J. DeWitt: How to barter bits for chronons: compression and bandwidth trade offs for database scans. SIGMOD 2007: 389-400

[19] www.infobright.com

[20] Y.E. Ioannidis: The History of Histograms (abridged). VLDB 2003: 19-30

[21] M.L. Kersten: The Database Architecture Jigsaw Puzzle. ICDE 2008: 3-4

[22] N. Kerdprasop, K. Kerdprasop: Semantic Knowledge Integration to Support Inductive Query Optimization. DaWaK 2007: 157-169

[23] www.luciddb.org

[24] V. Markl, P.J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, T.M. Tran. Consistent selectivity estimation via maximum entropy. VLDB J. 16(1): 55-76 (2007)

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

[26] C. Mishra, N. Koudas, C. Zuzarte: Generating Targeted Queries for Database Testing. SIGMOD 2008: 499-510

[27] T. Mitchell. Machine Learning. McGraw Hill, 1997

[28] MySQL 5.1 Reference Manual: Storage Engines. http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html

[29] S. Naouali, R. Missaoui: Flexible Query Answering in Data Cubes. DaWaK 2005: 221-232

[30] D. Narayanan, A. Donnelly, R. Mortier, A. Rowstron: Delay Aware Querying with Seaweed. VLDB 2006: 727-738

[31] www.paraccel.com

[32] Z. Pawlak. Rough sets: Theoretical aspects of reasoning about data. Kluwer, 1991

[33] Z. Pawlak, A. Skowron. Rudiments of rough sets. Information Sciences 177(1): 3-27, 2007

[34] M. Poess, R.O. Nambiar, D. Walrath: Why You Should Run TPC-DS: A Workload Analysis. VLDB 2007: 1138-1149

[35] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, R. Sidle: Constant-Time Query Processing. ICDE 2008: 60-69

[36] A. Rasin, S. Zdonik, O. Trajman, S. Lawande: Automatic Vertical-Database Design. WO Patent Application, 2008/016877 A2

[37] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik. CStore: A Column Oriented DBMS. VLDB 2005: 553-564

[38] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, P. Helland: The End of an Architectural Era (It's Time for a Complete Rewrite). VLDB 2007: 1150-1160

[39] www.sybase.com/products/datawarehousing/sybaseiq

[40] www.vertica.com

[41] B. Vo, G.S. Manku: RadixZip: Linear-Time Compression of Token Streams. VLDB 2007: 1162-1172

[42] P.W. White, 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, August 11, 1998

[43] M. Wojnarski, C. Apanowicz, V. Eastwood, D. Ślęzak, P. Synak, A. Wojna, J. Wróblewski: Method and System for Data Compression in a Relational Database. US Patent Application, 2008/0071818 A1

[44] J. Wróblewski, C. Apanowicz, V. Eastwood, D. Ślęzak, P. Synak, A. Wojna, M. Wojnarski: Method and System for Storing, Organizing and Processing Data in a Relational Database. US Patent Application, 2008/0071748 A1

[45] M. Zukowski, S. Heman, N. Nes, P.A. Boncz: Super-Scalar RAM-CPU Cache Compression. ICDE 2006: 59