Abstract—We investigate computational complexity of a task of optimizing multi-machine execution of analytical data processing operations. We concentrate on an example of aggregation queries in Infobright’s database engine, which is designed using the paradigms of rough sets and granular computing. The task is to optimize decomposition of data blocks and aggregation groups among machines having access to a shared data storage. The paper includes some examples of optimization functions and constraints, as well as the proof of NP-hardness of the considered task for some of them. It can be treated as a guideline for developing scalable data processing and mining methods based on massive parallelism and approximate computing.

I. INTRODUCTION

An important trend in the database industry is related to solutions optimized for advanced reporting and data exploration. Such solutions are often employed to store and process rapidly growing volumes of machine-generated data, such as network logs, sensor data, call records, transactions and more.

For decades, a great emphasis with respect to scalability of analytical databases has been put on massive parallelism. Over time, some relatively new architectural aspects have emerged. There is nowadays a kind of technological baseline that every analytical database engine refers to. On the other hand, database vendors attempt to develop some new unique features that make their products more competitive. The critical need is to combine such main-stream and innovative computational aspects, so they help, rather than disturb each other.

In this paper, we highlight the above need using an example of Infobright’s engine [1], where query execution is based on interaction between a layer of vertically and horizontally partitioned and compressed data packs, and a layer of their statistics utilized to minimize the data access. This interaction makes Infobright prepared for resolving complex and ad-hoc analytical SQL statements against data tables with billions of rows. It also positions Infobright’s architecture as a methodological reference for both academic and industry projects aiming at intelligent processing of massive data.

As a use case, we consider ad-hoc analytical aggregation queries with potentially huge amounts of groups corresponding to returned tuples. Such aggregations occur very often in academic and commercial implementations of database engines, for the purposes of data staging and monitoring, as well as data mining and knowledge discovery [2]. Using the example of aggregations, we show how Infobright’s principles of approximate computing over granulated data can go along with the standards of multi-machine querying.

The paper is organized as follows: In Section II, we recall the basics of Infobright’s architecture. In Section III, we introduce criteria for optimal split of an aggregation query execution onto multiple machines performing some computational jobs concurrently. In Section IV, we discuss the constraints for valid aggregation decomposition and we interpret the resulting layout in terms of well-known scheduling problems. In Sections V-VIII, we formalize the problem of optimal decomposition by means of minimizing mathematical formulas reflecting an expected cost of producing and merging partial results. In Section IX, we conclude our work.

From the point of view of foundations, our major contribution in this paper refers to the analysis of complexity of optimization problems related to multi-machine execution of aggregation queries. From a more practical perspective, the presented study may hopefully inspire some new granular approaches to scalable data processing and data mining, going beyond the framework of Infobright’s solution.

II. BASICS OF INFOBRIGHT’S DATABASE ENGINE

Infobright combines the principles of information granulation [3] and columnar databases [4] to efficiently store and query relational data. During load to a given data table, the set of input rows is partitioned onto so called row packs. Each row pack is vertically partitioned on data packs – collections of values of a single column for rows belonging to a given row pack. We can see here several levels of granulation: table ⇒ row pack ⇒ data pack ⇒ single value. A table is the most general while a value is the most detailed granule.

Data packs are compressed in a lossless way and stored on disk [5]. Prior to compression, various forms of mathematical descriptions of data packs contents are calculated. For each data pack understood as a sequence of values, they contain information such as its minimal and maximal elements, a total sum of its numeric elements, its number of nulls and so on. We refer to all such descriptions as to rough values. They can be partially compared to generalized decision functions introduced in the theory of rough sets [6], as they are utilized to build approximations of concepts corresponding to particular stages of resolving SQL statements [1].
One of the most common types of analytical queries refers to so called aggregations. Let us use them to illustrate benefits that rough values can bring in. For groups of rows defined by so called aggregating (or grouping) columns there are computed aggregation functions over aggregated (or grouped) columns. For example, in the following query we have groups of transaction detail records determined by IMSI (International Mobile Subscriber Identity) numbers, for which we compute total amount of transmitted data in a given day:

```
SELECT SUM(total_in_kb) FROM tdr WHERE billing_date='2014-07-25' GROUP BY imsi
```

The query returns as many tuples as many different IMSIs logged a data connection in a given day. One can also specify multiple aggregating columns and then each group is defined as corresponding to a unique vector of their values occurring in data. Each output tuple reports a score of aggregation function computed over a subset of input rows corresponding to a given value or a vector of values. From now on, for simplicity, we will focus on the case of a single aggregating column. However, all ideas and results presented in this paper remain valid for multiple aggregating columns as well.

There are various strategies of computing aggregations. For example, one may think about data compression and column scans aimed at acceleration of data access and processing in columnar databases [4]. As another example, the most basic algorithm currently implemented in Infobright’s database engine works with a dynamically created hash table\(^1\) where one entry corresponds to one group. When a new row is analyzed during a data scan, it is matched against tuples in the hash table. If a given group already exists, then appropriate aggregations are updated. Otherwise, a new group is added to the hash table and initial values of aggregation functions for this group are specified. Thus, the size of hash table depends on the number of different values of an aggregating column occurring in data subject to filtering conditions.

For large numbers of distinct groups, hash tables may not fit pre-allocated memory and, even if they do, operating with them may become a bottleneck. In such cases, Infobright’s algorithms can compute aggregations in multiple passes. Denote by \(w\) the maximum number of distinct tuples that is allowed to be maintained by a hash table in the same time. Once the number of tuples reaches \(w\) during a data scan, no more groups can be added – we continue scanning until the end of a given table but only groups already included into the hash table are taken into account in order to update their aggregates. After the scan is finished, we are sure that the partial result related to groups currently present in the hash table is completed. We can then empty the hash table and start the whole procedure from the very beginning, omitting rows which have been already taken into account in earlier table scan passes.

\(^1\) By a hash table we mean a memory-based structure requiring random (hash) access to the values that it stores.

The above algorithm takes advantage of rough values in many ways. First of all, it operates only on row packs which have a chance to contain rows satisfying filtering criteria. Such a chance is evaluated by comparing rough values with WHERE or JOIN conditions. It also lets us avoid row packs which do not have a chance to match with already completed hash table contents in the multiple pass scenario.

Furthermore, following inspiration related to rough set positive regions, if rough values indicate that all elements of a given row pack drop into a single group, then its corresponding scores of aggregation functions can be often updated without accessing data. Moreover, rough values are used to sort row packs in order to limit a need of multiple data passes, estimate the number of resulting groups, optimize encoding of values of aggregating columns in hash tables and so on.

### III. Multi-Machine Aggregation

In the approach described in the previous section, disjoint sets of groups (which correspond to values of an aggregating column) are determined dynamically while scanning through data and, in the same time, the scanning itself is optimized by utilizing rough values. Another possibility is to define such disjoint sets of groups in advance and, basing on them, plan computational jobs which can be processed concurrently, assuming that all of them have an access to a kind of shared data framework and, therefore, can analyze the same row packs with respect to different groups. Such optimization might be beneficial in both concurrent and serialized scenarios, where the aim would be to minimize accessing the same row packs many times, during multiple data scan passes.

An “orthogonal” strategy is to schedule jobs operating on disjoint sets of row packs and, if they include rows corresponding to any common groups, do a result merge after all jobs are finished. Such a merge might resemble a reduce step in the popular MapReduce methodology [7]. In Infobright’s case, one can utilize rough values in order to intelligently plan how to decompose aggregation with respect to input row packs and output groups, so the effort to be spent on merging partial results is minimized. One could say that the map step corresponds to aggregation decomposition driven by rough values, where data subsets required to finish particular jobs are available in a (possibly distributed) shared environment, ready to be accessed if necessary.

In multi-machine environment, one of possible realizations of decomposed aggregating is to consider one selected machine as so called master, which can use other machines (workers) to compute dedicated jobs. The master is responsible for defining and dispatching jobs, as well as collecting and assembling partial results. As discussed above, jobs can be defined with respect to both subsets of input row packs and subsets of output groups. The key observation is that the master can specify jobs using rough values, which are orders of magnitude smaller than the underlying data.

In order to balance computational effort of particular workers, the following criteria apply:
#1 All workers should receive similar amounts of row packs, so they process similar amount of data.

#2 All workers should be responsible for similar amounts of groups, so they generate similar processing cost and memory consumption.

Assuming that the same groups can be partially handled by many workers, the merge process may be difficult. A given group has to be searched in all results obtained from workers. Once the groups are located, the aggregations have to be updated/combined. For aggregation queries with truly large amounts of resulting tuples, the process of merging takes a significant part in overall execution time.

Let us also note that in some cases, e.g., when computing average, we cannot assemble local aggregations in a direct way. The original aggregation has to be transformed to more basic ones so the assembling partial results is valid. For example, AVG(a) has to be transformed to SUM(a) and COUNT(a). After all sums and counts are computed locally, the master has to sum them up and compute the final outputs.

To address the problem of costly merges, let us observe that if a given worker is assigned with a group that is not dedicated to any other workers, then such a group can be appended to the final result much faster. The more such private groups are handled by workers, the simpler the merge process is. Hence, let us specify one more criterion for work balancing:

#3 On average, the amount of workers handling each single group should be minimal.

In the ideal situation, each worker should process private groups only. However, if there are no private groups but each group is handled by relatively few workers, it may significantly reduce the cost of merging anyway. Thus, when specifying jobs for particular workers, it may be useful to look at clusters of row packs with possibly non-overlapping subsets of values on aggregating columns. For this purpose, one can leverage information stored in rough values as well.

### IV. Constraints for Decomposing Aggregations

Let us consider a set of row packs $P = \{p_1, p_2, \ldots\}$ and a set of groups $G = \{g_1, g_2, \ldots\}$ corresponding to values of an aggregating column. Consider $X \subseteq P \times G$ as a set of all pairs $(p, g)$ such that row pack $p$ contains at least one row which takes a value corresponding to $g$. The set $X$ can be represented by crosses on the plane $P \times G$.

In practice, the crosses are approximated using rough values of aggregating columns. We put a cross for $(p, g)$, if comparison of rough values of $p$ and description of $g$ does not exclude a chance that $p$ and $g$ should be considered together. Note that because row pack $p$ is a collection of rows, it is likely to contain rows from different groups, and hence there can be multiple crosses for the same $p$.

It may turn out that a given $(p, g)$ should not have been crossed. This happens if, e.g., the value of $g$ falls between the minimum and maximum values of the grouping column over the rows in $p$, but – after accessing data – it turns out that there is no element exactly equal to $g$. On the other hand, it is fully certain that no cross for $(p, g)$ means that there is truly no relationship between $p$ and $g$. Thus, it is guaranteed that query outcomes are correct, even if such “false crosses” influence the efficiency of aggregation decomposition.

Our problem of finding a schedule for workers can be now stated as follows: For a given number of workers $k$, we want to find subsets $P_1, \ldots, P_k \subseteq P$ and $G_1, \ldots, G_k \subseteq G$ such that $X \subseteq \bigcup_{i=1}^{k} P_i \times G_i$ and $\forall_{i, j} X_i \cap X_j = \emptyset$, where $X_i = X \cap (P_i \times G_i)$. The last condition assures that rows in any $p$ with values corresponding to any $g$ are processed by exactly one worker. This is necessary as we do not want to count a given row more than once, in order to avoid incorrect results.

Fig. 1 shows how schedules look like. Rows and columns represent row packs and groups, respectively. A schedule is a set of “rectangles” determined by row packs and groups. $X_1, X_2, X_3$ are sets of crosses corresponding to particular jobs. Rectangles are disjoint with respect to crosses. Otherwise some of rows might be wrongly “over-aggregated”.

Jobs can be indeed graphically represented as rectangles, subject to permutations of row packs and groups. One might try to consider also other “shapes”. However, specifications in form of $P_i \times G_i$ have significant advantages. Whatever subset $Y \subseteq X$ is considered, the corresponding worker would need to deal with all related row packs (as input) and groups (as output) anyway. Thus, operating with the above Cartesian products seems to be simplest and most natural. Such representation is also convenient for already mentioned scenarios, where workers receive disjoint subsets of row packs or are supposed to handle disjoint subsets of groups. In the above convention it means that subsets $P_1, \ldots, P_k$ or $G_1, \ldots, G_k$ are supposed to partition $P$ or $G$, respectively.

Certainly, the above framework is still a kind of simplification. For example, for huge, practically unknown amounts of groups, it is impossible to consider the above model at the level of single values of aggregating columns. In such cases, the set of all possible groups can be partitioned into some predefined or dynamically derived classes described by their own rough values. The optimization problem discussed in further sections is then handled for such classes instead of single values. However, for the sake of clarity, we will operate with groups $g$ corresponding to single values.
Let us consider the following example with two machines, four row packs \(p_1, p_2, p_3, p_4\) and three groups corresponding to \(imsi_i\): \(imsi_1, imsi_2, imsi_3\). Let us assume for a while that we ask both machines to deal with potentially all three groups and focus on splitting the set of row packs in \(P\) with tuples corresponding to groups in \(G_i\) stored in a dynamically appended hash table.

Let us consider the following example with two machines, four row packs \(p_1, p_2, p_3, p_4\) and three groups corresponding to \(imsi_i\): \(imsi_1, imsi_2, imsi_3\). Let us assume for a while that we ask both machines to deal with potentially all three groups and focus on splitting the set of row packs in \(P\) with tuples corresponding to groups in \(G_i\) stored in a dynamically appended hash table.

Formally, we should consider the job scheduling problem with respect to a particular optimization function. Let \(\mathbb{G}\) be a family of such functions over \(h_1, w_1, \ldots, h_k, w_k\), where \(h_1\) and \(w_i\) are defined as above, and where \(k\) is the number of workers. For a given \(f \in \mathbb{G}\), we say that decomposition characterized by quantities \(h_1, w_1, \ldots, h_k, w_k\) is optimal with respect to \(f\), if \(f(h_1, w_1, \ldots, h_k, w_k)\) is minimal among all possible choices of valid schedules \(\{P_i \times G_i : i = 1, \ldots, k\}\).

In particular, one may define \(f \in \mathbb{G}\) over costs \(c_1, \ldots, c_k\). The following is an example of such function:

\[
    f_{\text{prod}}^{\max}(h_1, w_1, \ldots, h_k, w_k) = \max\{h_i \cdot w_i\} \quad (1)
\]

It is based on intuitive observation that minimization of maximal local cost is implicitly followed by making jobs more balanced and disjoint from each other.

Minimization of maximum cost is not the only possibility. It is reasonable, if we assume that the master needs to wait for all workers to finish their tasks prior to final merging. However, it is also possible to design the multi-machine aggregation process in such a way that the master does not need to wait for the slowest worker. Instead, it can keep merging partial results as soon as they arrive. In more advanced implementations, it can even redistribute some yet unfinished jobs among workers which have already finished their previous tasks. In such scenarios, one should measure how the workload is balanced among all of workers. For this purpose, as an example, we can reformulate function \(f_{\text{prod}}^{\max}\) as follows:

\[
    f_{\text{sum}}^{\max}(h_1, w_1, \ldots, h_k, w_k) = \sum_{i=1}^{k} (h_i \cdot w_i) \quad (2)
\]

This function has an intuitive graphical interpretation. Basically, minimization of \(f_{\text{sum}}^{\max}\) assures that rectangles encoding jobs should have overall minimal number of “holes”, i.e., areas with no crosses. In an ideal case, the optimal schedule with respect to \(f_{\text{sum}}^{\max}\) would include no holes at all.

VI. OTHER FUNCTIONS

Relying on any mathematical formula in order to model a real data processing cost is always a kind of simplification. The following example can be treated as a more generic alternative comparing to functions introduced so far:

\[
    f_{\text{dev}}^{\max}(h_1, w_1, \ldots, h_k, w_k) = \max\{|h_i - h| + |w_i - w|\} \quad (3)
\]

Function \(f_{\text{dev}}^{\max}\) represents a general and simple case where we minimize the maximal size of either components of job size for the most overloaded worker. By using \(f_{\text{dev}}^{\max}\), we assure that there is no worker getting exceptionally many row packs or groups to take care of. We also do not make any assumptions about modelling computational cost of local jobs.

A drawback of \(f_{\text{dev}}^{\max}\) comparing to \(f_{\text{prod}}^{\max}\) and \(f_{\text{sum}}^{\max}\) is that if the number of groups is significantly higher than the number of row packs, then \(f_{\text{dev}}^{\max}\) ignores the latter factor. Analogously, if there are many row packs and relatively few groups, then only a number of row packs counts. In both cases, a solution may be to introduce additional weights making the quantities of \(h_i\) and \(w_i\) more comparable to each other.

We can also compare an evaluated schedule with a hypothetically optimal decomposition. Ideally, each worker should process the same number of row packs and groups, where all groups are private (see Section III). In such a case, optimal numbers of row packs and groups are \(h = n/k\) and \(w = m/k\), where \(n = \text{card}(P)\) and \(m = \text{card}(G)\). The optimal job would be of size \(h \cdot w\) (see Fig. 3). Minimization of the following formula assures that the most overloaded worker gets amounts of row packs and groups close to such optimum:

\[
    f_{\text{dev}}^{\text{dev}}(h_1, w_1, \ldots, h_k, w_k) = \max_i\{|h_i - h| + |w_i - w|\} \quad (4)
\]
Function $f_{\text{dev}}$ reflects the largest deviation $d_i = |h_i - h| + |w_i - w|$ of scheduled jobs $i = 1, \ldots, k$ from the “golden” setting. Analogously to the discussion at the end of Section V, we could also modify $f_{\text{dev}}$ as follows:

$$f_{\text{dev}}^h(h_1, w_1, \ldots, h_k, w_k) = \sum_{i=1}^{k} (|h_i - h| + |w_i - w|)$$

Function $f_{\text{dev}}^h$ can be interpreted as a kind of Manhattan distance of scheduled jobs from the optimal scenario. One could also consider other distance measures, adopt other concepts of divergence and so on. Nevertheless, the fundamental idea of comparing candidate schedules with hypothetically perfect decompositions would remain the same.

Let us also remind that minimization of considered functions is supposed to decrease the amounts of row packs and groups handled by multiple workers. In particular, it means an increase of private groups, for which the merge process is trivial. This can be achieved by embedding more explicit constraints into definitions of optimization criteria, or simply by assuming that solutions minimizing the above formulas usually yield relatively many private groups, at least for data sets for which such groups can be specified.

VII. ILLUSTRATIVE EXAMPLE

This section aims at re-emphasizing that there may be indeed no “optimal” optimization function, i.e., a function accurately modelling actual benefits of multi-machine aggregation processing in all possible scenarios.

Let us consider an example in Fig. 4. It shows that $f_{\text{prod}}^h$ fails when there is a row pack corresponding to significantly more groups than other row packs. Fig. 4a displays decomposition, which is optimal by means of minimization of $f_{\text{prod}}^h$. Machine 1 is responsible for one row pack and six groups, hence $c_1 = 6$. Machine 2 is responsible for five row packs and one group, hence $c_2 = 5$. However, this schedule is not efficient in practice. This is because one of workers is forced to process almost the whole data set. Following enhancements described in the end of Section II, one might say that if a given row pack corresponds to a single value of an aggregating column, then partial aggregation results may be potentially extracted from its rough values. However, if any non-trivial WHERE condition is involved, the contents of all such row packs need to be usually accessed anyway. Thus, in this case, a better solution would be the one in Fig. 4b. However, it would not be selected by $f_{\text{prod}}^h$ because $c_1 = 18$ and $c_2 = 3$.

For comparison, let us consider function $f_{\text{max}}$. The optimal number of row packs $h$ and groups $w$ is equal to three. Hence, for the schedule shown in Fig. 4b, we have $d_1 = |3 - 3| + |6 - 3| = 3$ and $d_2 = |3 - 3| + |1 - 3| = 2$. In case of Fig. 4a, there is $d_1 = |1 - 3| + |6 - 3| = 5$ and $d_2 = |5 - 3| + |1 - 3| = 4$. Thus, $f_{\text{dev}}^h$ puts in favor decomposition in Fig. 4b.

Our research so far indicates that $f_{\text{max}}$ indeed seems to reflect the real scope of optimizing multi-machine aggregations. However, further investigation is needed. The same can be said about optimization under additional constraints for $P_1, \ldots, P_k$ and $G_1, \ldots, G_k$, which is our next topic below.

VIII. COMPLEXITY AND HEURISTICS

We discussed several examples of functions whose minimization reflects decomposition of aggregation queries in such a way that the cost of completing local jobs and merging their partial results is possibly low. An important aspect is surely the computational complexity of generating optimal splits with respect to particular functions. There can be a number of optimization and decision problems formulated. Below, let us discuss one of them as a starting point.

As before, let $k$ be the number of workers, $P$ be a set of row packs and $G$ be a set of groups corresponding to values of an aggregating column. We define $k$-AggregateSchedule as the problem of finding a schedule $\{P_i \times G_i : i = 1, \ldots, k\}$ such that $P_1, \ldots, P_k$ is a partition of $P$, cardinalities of $P_1, \ldots, P_k$ differ by only up to one unit (i.e., $|h_i - h_j| \leq 1$), and the sum of cardinalities of $G_1, \ldots, G_k$ is minimal, i.e.:

$$\sum_{i=1}^{k} w_i \rightarrow \min$$

It means that we try to find a partition of input row packs such that each worker gets almost the same amount of rows to process and the total number of groups (with repetitions) handled by all workers is minimal. In the best case, when all workers turn out to take care of only private groups, this sum is equal to $m = \text{card}(G)$. In the worst case, it is $m \cdot k$. In that latter situation, there may be still some benefit resulting from processing data in parallel. However, that benefit would be destroyed by a cost of merging partial results.
The $k$-AggregateSchedule problem is NP-hard. In order to show this, let us recall the Optimal Division Into Packs (ODIP) problem, which was proved to be NP-hard in [8]. ODIP is actually related to the analysis of Infobright’s methods as well. It refers to the task of clustering original rows into row packs in such a way that the quality of rough values describing the corresponding data packs is maximized.

An instance of ODIP is a set $U$ of $n$ binary sequences of length $m$, where $1$ corresponds to an outlier and $0$ – to a regular value in an original record. The task is to find a partition of $U$ into $k$ row packs of equal size $n/k$ such that the sum of data packs containing outliers is minimal. This way, outliers are optimally grouped making it possible to describe the rest of data packs by high-quality rough values.

An instance of ODIP can be directly encoded as an instance of $k$-AggregateSchedule. For each sequence, we create a row pack containing groups corresponding to $1$s, as illustrated by Fig. 5. Finding a schedule minimizing $\sum_{i=1}^{k} w_i$ is then the same as minimizing the sum of data packs with outliers.

The above sketch also shows that $k$-AggregateSchedule can be tackled by heuristics similar to those applied in Infobright to cluster incoming rows [10]. This analogy is even more useful for the problem of finding a schedule such that $G_1, \ldots, G_k$ is a partition of $G$ onto (almost) equally-sized blocks and the sum of cardinalities of $P_1, \ldots, P_k$ is minimized, so different workers access the same row packs as rarely as possible:

$$\sum_{i=1}^{k} h_i \rightarrow \text{min}$$

NP-hardness of the above task can be proved just like for $k$-AggregateSchedule. Similar results can be formulated also for criteria analogous to those in previous sections, now with additional constraints. Still, in practice partitioning groups is more difficult than partitioning row packs. As mentioned earlier, we often have no complete information about distinct groups, so we need to identify them while scanning data. It resembles a scenario of assembling row packs from a stream of loaded rows. This resemblance can inspire new multi-pass methods (see Section II), for $k = m/w$, $m = \text{card}(G)$, and $w$ equal to the allowed number of hash table entries.

In [8], polynomial reduction of 3DMatching [9] to ODIP is shown. In its construction, $n$ is dividable by $k$ and the inequality $n \geq 3k$ is assumed.

References


