Rana Faisal Muniry, Sergi Nadal, Oscar Romero, Alberto Abell´o, Petar Jovanovic, Maik Thiele,Wolfgang Lehner

Intermediate Results Materialization Selection and Format for Data-Intensive Flows

Erstveröffentlichung in / First published in:

Fundamenta Informaticae. 2018. 163(2), S. 111-138. IOS Press. ISSN 1875-8681.

DOI: https://doi.org/0.3233/FI-2018-1734

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

Intermediate Results Materialization Selection and Format for Data-Intensive Flows*

Rana Faisal Munir†, Sergi Nadal, Oscar Romero, Alberto Abelló, Petar Jovanovic

Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

{*fmunir,snadal,oromero,aabello,petar*}*@essi.upc.edu*

Maik Thiele, Wolfgang Lehner

Technische Universität Dresden (TUD), Dresden, Germany

{*maik.thiele,wolfgang.lehner*}*@tu-dresden.de*

Abstract. Data-intensive flows deploy a variety of complex data transformations to build information pipelines from data sources to different end users. As data are processed, these workflows generate large intermediate results, typically pipelined from one operator to the following ones. Materializing intermediate results, shared among multiple flows, brings benefits not only in terms of performance but also in resource usage and consistency. Similar ideas have been proposed in the context of data warehouses, which are studied under the materialized view selection problem. With the rise of Big Data systems, new challenges emerge due to new quality metrics captured by service level agreements which must be taken into account. Moreover, the way such results are stored must be reconsidered, as different data layouts can be used to reduce the I/O cost. In this paper, we propose a novel approach for automatic selection of multi-objective materialization of intermediate results in data-intensive flows, which can tackle multiple and conflicting quality objectives. In addition, our approach chooses the optimal storage data format for selected materialized intermediate results based on subsequent access patterns. The experimental results show that our approach provides 40% better average speedup with respect to the current state-of-the-art, as well as an improvement on disk access time of 18% as compared to fixed format solutions.

Keywords: Big Data, Data-Intensive Flows, Intermediate Results, Data Format, HDFS

*This research has been partly funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC) and the GENESIS project, funded by the Spanish Ministerio de Ciencia e Innovación (num. TIN2016-79269-R)

†Address for correspondence: Campus Nord Omega-125, UPC - dept ESSI, C/Jordi Girona 1-3, E-08034 Barcelona, Spain.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

1. Introduction

Nowadays, many organizations are shifting their business strategy towards data analytics in order to guarantee their success. In the past, the vast majority of analyzed data was transactional, however the emergence of Big Data systems allows a new range of data analytics, by replacing traditional extract-transform-load (ETL) process with much richer data-intensive flows (DIFs) [1]. This new range of data analytics is supported by the Hadoop[1] ecosystem which has a distributed storage system (Hadoop Distributed File System - HDFS[2]) to store large scale data and a processing engine (i.e., MapReduce [2]) to execute DIFs. It works on a distributed cluster of commodity hardware which provides competitive advantage to organizations by reducing their hardware costs. In addition, many modern cloud providers offer pay-per-use services to organizations by implementing the big data systems under service level agreements (SLAs).

An in-depth study of analytical workloads, in Big Data systems across seven enterprises, shows that user workloads have high temporal locality, as 80% of them will be reused by different stakeholders on the range of minutes to hours [3]. Thus, providing partial materialization of results in shared flows can clearly bring benefits by saving computational resources. However, the aforementioned study raises two key questions: "*what intermediate results to materialize?*" and "*how to store them?*". The first question boils down to the traditional data management problem of *materialized view selection* [4], which is well-known to be NP-hard [5]. On the other hand, the second question leads to the *storage layout selection problem*, aiming to overcome the problem of fixed storage layout [6, 7, 8] caused by different workloads requiring different layouts to achieve optimal performance.

These questions are not easily addressable, despite the efforts of the research community. Some works[9, 10, 11] have tackled the problem of finding the optimal partial materialization in DIFs, however all of them are specific to the MapReduce framework and only aim at optimizing the system performance-wise by ignoring other relevant SLAs (such as freshness, reliability, scalability, etc.[12]).

Similarly, the existing materialized view solutions in relational databases [13] focus only on improving query execution time without considering multiple generic SLAs. Moreover, the aforementioned solutions do not consider different characteristics associated with different SLAs. For instance, in some organizations, they allow to get results from a stale materialized node (i.e., to allow low freshness) for a certain time period to reduce the loading cost. These characteristics can be expressed separately and the optimal value should be chosen for each materialized node. These shortcomings of existing solutions are addressed by our proposed approach, which is a technology independent materialization solution and can take into consideration generic quantifiable SLAs with their associated characteristics. It should also be noted that none of the existing works focuses on how partial materializations are stored for future re-use, which is important as all Big Data systems store data on HDFS [14], where I/O operations are expensive. Indeed, the I/O operations can be reduced by using different storage formats for materialization based on the workload.

[1] https://hadoop.apache.org

[2] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

1.1. Motivational example

To motivate our work, we present a DIF, shown in Figure 1, which depicts a high-level representation containing relational operations and User Defined Functions (UDFs). It uses five input sources and serves three queries. Each data source and data operator is labeled by its estimated processing cost (i.e., consumed resources, in seconds) and storage cost (in GB). Note that data processing entails extracting and loading data from the sources into the data processing system.

For the sake of this example, let us suppose that all the sources update once per day, except *Source 1* and *Source 3* that have a update frequency of 6 and 4 times per day, respectively. *Query 1*, *Query 2* and *Query 3* have a frequency of 2, 20, and 10 times per day, respectively. In addition, let us assume that we allow stale materialized results and it is provided as a characteristic vector (given as number of updates per time unit $[1, 2, ..., n]$).

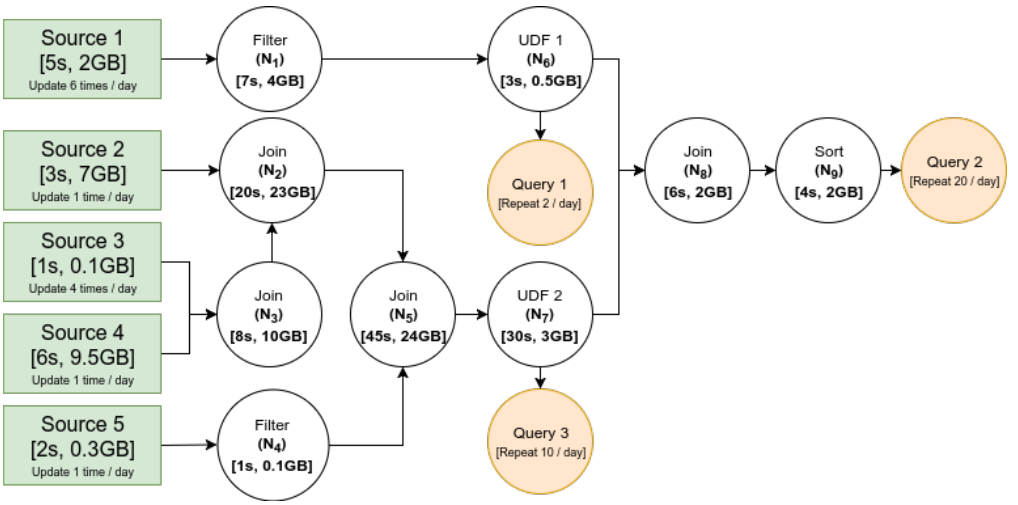

Figure 1: An Example of a DIF

In this example, we focus on optimizing four SLAs (i.e., time to load, time to query, space needed to store intermediate results, and freshness). *Loading time* is measured by the sum of processing cost from the sources to the partial materializations, *query time* is measured by the sum of execution cost from the partial materializations to the users output, *storage space* is measured by the sum of storage cost for the selected partial materializations, and *freshness* is estimated using the cost function presented in Section 3.3.

Several of the existing intermediate result materialization approaches from Big Data Systems can be used, as discussed in Section 7.1. Let us focus on one of them, namely ReStore [10], which uses two kinds of heuristics (i.e., conservative and aggressive) in order to choose DIF nodes for materialization. Conservative heuristics materialize the output of those operators that reduce the input size (i.e., project and filter). The aggressive heuristics materialize the output of those operators which produce large outputs and those known to be computationally expensive (i.e., join, group and cogroup). Table 1 shows the selected nodes and a quadruple with the costs (i.e., loading time, query time, storage space, and

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

freshness). It should be noted that *query time* represents the sum of times of all three queries. The first two columns show both ReStore's heuristics, while the rightmost shows a pareto-optimal solution (i.e., a solution that cannot be improved further in the presence of multiple conflicting SLAs) in boldface.

Table 1: Selected intermediate nodes and cost for the four SLAs (load, query, store, freshness)

	ReStore Cons.	ReStore Agg.	Pareto-optimal
Nodes	N_1, N_4	N_2, N_3, N_5, N_8	$\boldsymbol{N_7}$
Cost	$\langle 75$ s, 257 s, 4.1 GB, $1\rangle$	$\langle 336$ s, 115 s, 59 GB, $1\rangle$	$\langle \mathbf{101}$ **s, 100 s, 3 GB**, $\mathbf{0.78}\rangle$

As shown in Table 1, ReStore conservative heuristics choose N_1 and N_4. They take more time in loading, due to *Source 1*, which updates very frequently and effects the loading cost of N_1. We calculate the total load time of a node by multiplying its load time with the update frequency of its input sources. Furthermore, both N_1 and N_4 do not provide good speedup because they have high query time. ReStore aggressive heuristics choose N_2, N_3, N_5, and N_8. These nodes help to reduce the query time, but require more space to store and more time to load. It should be noted that ReStore does not support freshness. If the input sources change, it deletes all their dependent materialized nodes. This pull-strategy provides a fixed degree of freshness and thus, we set it to 1 in our quadruple.

Finally, the pareto-optimal solution considers four SLAs (i.e., loading time, query time, storage space, and freshness) together by assigning them the same weight, and based on them, it chooses only node N_7, which provides better speedup compared to ReStore's heuristics. Even though, N_7 depends on *Source 3* , which has a high loading cost due to its high update frequency, it is still worth to materialize because it is reused more often by repetitive queries (i.e., Query 2, Query 3). Moreover, the loading cost can be improved by choosing the optimal value of refresh frequency for N_7. The possible values are $[1, 2, 3, 4]$, where 4 will provide the maximum freshness. The pareto-optimal solution chooses 3 as the refresh frequency for N_7, which helps to improve the load time and it also provides a good degree of freshness.

The above given example shows that the state-of-the-art solutions produce suboptimal results in the case of different SLAs. In addition, the physical storage layout must be chosen for the selected materialized nodes. To address these problems, we revisit the traditional frameworks for materialized view selection [15] and analyze their applicability and extensions in the context of DIFs for Big Data systems. As a result, we present an approach to automatically select the optimal materialization of intermediate results coupled with the optimal layout, driven by multiple quality objectives represented as quantifiable SLAs with their associated characteristics. This is achieved by implementing a multi-objective optimization technique (discuss in Section 4) which efficiently tackles multiple and conflicting objectives to select materialized nodes. Moreover, for each to-be-materialized node, our approach also chooses the most appropriate layout (i.e., horizontal, vertical or hybrid) based on the workload.

Contributions. The main contributions of our work are as follows:

- We propose a novel cost model for multi-objective selection of optimal partial data materialization for DIFs.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

- We present a local search algorithm that, driven by a set of SLAs, probabilistically selects a set of near-optimal intermediate results to materialize.
- We show the performance bottleneck of data formats in different workloads, and define a set of heuristic rules to select the appropriate data format based on their different access characteristics.
- We assess our method and show its performance gain by using the TPC-H benchmarking suit.

Outline. The rest of the paper is structured as follows. Section 2 presents the theoretical building blocks and formalizes our approach. Sections 3 and 4 present the cost model for intermediate result selection and the algorithm to explore the search space. In Section 5, we present our data format selection approach for intermediate results. In Section 6, we present the experimental results. Section 7 discusses the related work, while Section 8 concludes the paper and discusses our future work.

2. Formal building blocks and problem statement

2.1. Multiquery AND/OR DAGs and data-intensive flows

The general framework for materialized view selection [15] relies on multiquery AND/OR Directed Acyclic Graphs (DAGs). As defined in [16], a query DAG is a bipartite graph $\mathcal{G}$, where AND nodes (or *operational nodes*) are labeled by a relational algebra operator, and OR nodes (or *view nodes*) are labeled by a relational algebra expression. Moreover, given a set of queries Q defined over a set of source relations R, a multiquery DAG $\mathcal{G}$ is a query DAG, which may have multiple sink (query) nodes. Roughly speaking, the materialized view selection problem can be expressed as a search space based problem over the multiquery DAG $\mathcal{G}$. Additionally, [17] formalizes the output of such problem as a data warehouse (DW) configuration $C = \langle V, Q^V \rangle$, where Q^V is the set of queries in the query set Q rewritten over the view set V. Note that there exist two special DW configurations: $\langle Q, Q^Q \rangle$ which represents a materialization of the query set Q and $\langle R, Q \rangle$ which represents a complete materialization of the source data stores R.

However, the multiquery AND/OR DAGs fail to capture the complex semantics present in DIFs operators, as they solely rely on relational operators. To this end, in this paper we build upon the ideas from the aforementioned frameworks and adapt them for the case of DIFs, which consider more complex data transformations [18]. It is straightforward to see that any multiquery DAG $\mathcal{G}$ can be represented as a DIF, however the opposite does not hold due to the fact that AND/OR DAGs are solely based on relational operators, while DIFs are extended with more complex operations. Thus, in this paper, we extend the notion of DW configuration to **Big Data system (BDS) configuration** for the case of DIFs. Hereinafter, we will depict a BDS configuration as a set of nodes from the DIF to materialize $B = \{b_1, \ldots, b_n\}$. In the following sections, we describe the specific components for the problem in-hand, and reformulate the materialized view selection problem in the context of BDS.

2.2. Components

Data-Intensive Flow. In this paper, we adopt the notation from [19], hence we define a DIF D as a DAG (V, E) where its nodes (V) are the flows' operational nodes, and its edges (E) represent the

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

directed data flow. Operational nodes are defined as $o = \langle \mathbb{I}, \mathbb{O}, \mathbb{S}, V_{pre} \rangle$, where $\mathbb{I}$ and $\mathbb{O}$ are sets of respectively input and output schemata, where each schema is defined as a finite set of attributes, $\mathbb{S}$ expresses operator's semantics, and V_{pre} a subset of attributes of the input schemata ($V_{pre} \subseteq \bigcup_{I \in \mathbb{I}} I$) whose values are used by o.

Design Goal (DG). $\mathbb{DG}$ represents a set of design goals, where each (DG_i) characterizes an SLA. It can be specified as either a minimization or a maximization of an objective cost function, or alternatively as a boundary that must not be surpassed in such cost function. Formally:

- *Min/Max*: From a set of BDS configurations $\mathbb{B}$, it returns the minimal B by means of evaluating the cost function CF, defined as $DG_{min}(\mathbb{B}) = \min_{B \in \mathbb{B}} (CF(B))$. Note that maximizing the cost function is equivalent to the negation of minimization.
- *Constraints*: For a BDS configuration, it checks whether the evaluation of the cost function CF fulfills the constraint K, formally $DG(B) = [CF(B) \leq K]$. Note that the constraint can express an arbitrary logical predicate (e.g., $<, >, \geq$). It is important to note that DG(C), where C is constraints, in this case is binary true/false and it differs from the above which gets the value obtained from the cost function.

Cost Function (CF). Given a BDS configuration, $\mathbb{CF}$ represents a set of cost functions where each (CF_i) is the estimation of an SLA for B. Hence, we define $CF(B) = \sum_{b \in B} E(b)$ (where $E(b)$ is the cost estimation of an element $b \in B$).

Characteristics Vector (CV). Some costs are determined once a node is chosen, but for SLAs, we can select arbitrary values for the features that impact them. For instance, in some organizations, they allow stale data for some period of time, which can be defined as a refresh frequency for every materialized node. Thus, the refresh frequency should be chosen to maximize the overall benefit. We keep a vector of such choices. Each position of the vector represents a characteristic affecting some SLAs. These characteristics will impact on the associated cost function CF.

Utility Function. In the context of multi-objective optimization (MOO) [20], it is common to aggregate all objectives $DG_1, \ldots, DG_n$ into a single value to obtain the global utility. Such function, known as the *utility function* U as it measures benefit, is formally defined as $U(\mathbb{DG}) = U(CF_1(B), \ldots, CF_n(B))$. Each $CF_i(B)$ provides a quantitative evaluation of B (it can be seen as individual utility functions for each cost function) for its associated DG_i, in the case of min/max design goals, or 0, and $+\infty$ for satisfied and non-satisfied constraints, respectively. Generally in MOO higher utilities are preferred. However, in our context there are some CF where we aim for minimal utilities (e.g., query time).

Rules for Format Selection. $\mathbb{HR}$ represents a set of heuristic rules for data storage format selection. Opposite to the intermediate results selection problem, which is based on a cost model, here we opt for heuristic rules. There are two major reasons for this decision. Firstly, incorporating format selection in the cost model for intermediate result selection, would increase the overall complexity, and thus lower the probability of finding an optimal solution. Secondly, as shown in Section 5.1, the search space for format selection, with few features present, is comparatively smaller than that of intermediate results

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

selection, hence a rule-based approach can cover most representative scenarios. Consequently, we tackle both problems separately. Lightweight approximations have been consistently used in databases before, as they yield a good balance between the performance gain obtained and the extra overhead introduced. Indeed, as shown in Section 6.2, heuristic rules yield a good performance/accuracy ratio for this problem.

2.3. Problem statement

We state the problem of intermediate results materialization selection and format in DIFs as: given a data-intensive flow D, a set of design goals $\mathbb{DG}$, a set of cost functions $\mathbb{CF}$, a characteristics vector $\mathbb{CV}$, a utility function $U(\mathbb{DG})$, a cost model represented by a set of estimators over D calculated by means of statistical information from sources, and a set of heuristic rules for data format selection $\mathbb{HR}$, return a BDS configuration B', such that, $U(\mathbb{DG})$ is minimal, each $b \in B'$ with its optimal characteristics values for $\mathbb{CV}$ and propose a data format to store it, leveraging on $\mathbb{HR}$, where its disk access time is minimal.

3. Cost model for intermediate results materialization selection

In this section, firstly we present our approach to estimate statistics for each operation of a DIF. We assume that the statistics of each input source are available. Secondly, we discuss the metrics (i.e., execution and storage) that we consider in this paper. It should be noted that we choose to ignore the CPU cost, and focus only on the I/O operations. Also, regardless of being executed in parallel, the overall execution cost of the flow will remain the same (only time span would be reduced). Finally, we use the proposed metrics to estimate the cost of SLAs. In this paper, we present the cost functions for four SLAs (loading, query, storage, and freshness).

3.1. Data-intensive flow statistics

As previously mentioned, cost functions are computed from estimators. Every operational node in a data-intensive flow D might have several estimators, each assessing a single SLA (e.g., an execution cost), where they perform a cost based estimation according to the operator's semantics. In order to devise more accurate metrics, some essential statistics must be obtained from the input data stores and propagated across D. By topologically traversing D, we can propagate such statistics at each node, based on the specific semantics of operators. In [21], the authors describe a complete set of statistics which are necessary to perform cost based estimations for DIFs. Here we focus on the following subset: selectivity factor $sel_P(R)$, number of distinct values per attribute $V(R.a)$ and cardinality $T(R)$. R denotes an input data store, while $R.a$ is an attribute of R. Note that statistics only consider logical properties of the flow, hence they are independent of the underlying engine where the flow is executed.

Example 3.1. Let us assume a *JOIN* operator $R' = R \bowtie S$ (e.g., N_6 in Figure 1), with input schemata $R(a, b)$, $S(c, d)$ and semantics $P_{R.a=S.c}$. Inspired by the work in [22], we propose measures for the

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

above-mentioned statistics for this *JOIN* operator (we have done likewise for the rest of operators) as:

$$sel'_P = \begin{cases} \dfrac{1}{V(R.a)}, & \textit{if } domain(S.c) \subseteq domain(R.a) \\ \dfrac{V(R.a \cap S.c)}{V(R.a) \cdot V(S.c)}, & \textit{otherwise} \end{cases}$$

$$V(R'.att_i) = V(R.att_i) \cdot (1 - sel_P)^{\frac{T(R)}{V(R.att_i)}} \qquad T(R') = sel'_P(R \bowtie S) \cdot T(R) \cdot T(S)$$

The selectivity factor is obtained as the fraction of the number of shared values in the *JOIN* attributes, when the domain of the right-hand side is contained in the domain of the left-hand side (i.e., analogously to Primary Key(PK)-Foreign Key(FK) relations), otherwise an estimation is made as a fraction of shared values and its Cartesian product. Regarding the number of distinct values for an attribute, it is estimated as the input number of distinct values, multiplied by the probability that no repetitions of a value are selected. Finally, the cardinality is measured likewise the relational case.

3.2. Metrics

Once statistics for D have been calculated, they can act as building blocks for metrics. Here, we focus on estimating both performance-wise ($Execution_{estimator}$) and space-wise ($Space_{estimator}$) metrics. Performance metrics are measured by means of estimated disk I/O (in blocks) and space metrics by the number of disk blocks occupied by the intermediate results materialization. It is worth noting that in terms of execution, the CPU cost is negligible as opposed to I/O cost [23], hence we ignore CPU cost and focus on the I/O cost of operators. Therefore, non-blocking operational nodes (acting as pipelines) will not incur any cost for such $Execution_{estimator}$.

To devise metrics, certain characteristics of the underlying engine are required. We focus on the following subset: the size of a disk block B, the number of main memory buffers available M, and the size in bytes that each attribute occupies $sizeOf(att_i)$. For instance, in the Oracle relational database the block size is approximately of 8KB, while in Hadoop's HDFS it is 64MB or 128MB. The incurred space of intermediate results is measured by means of the estimated number of blocks generated. However, this will vary according to the underlying schema that such results have and therefore, we need to make this calculation based on the record length, that is $sizeOf(att_i)$ (including the corresponding control information). Thus, the specific number of blocks for an input R is measured as:

$$B(R) = \left\lceil \frac{T(R)}{\left\lfloor \frac{B}{\sum sizeOf(att_i)} \right\rfloor} \right\rceil$$

Example 3.2. Given the *JOIN* operation from example 3.1, one implementation of such operator is based on the block-nested loop algorithm, which scans S for every block of R using $M - 2$ memory buffers (as the remaining two are used to perform tuple comparisons and output results), thus the estimation for execution and space costs is as follows:

$$Execution_{estimator} = B(S) + B(R) \cdot \left\lceil \frac{B(S)}{M-2} \right\rceil \qquad Space_{estimator} = B(R')$$

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

However, in a MapReduce environment, execution cost is dominated by data transfers (i.e., communication cost over the network) that occurs during the data shuffling phase between mapper and reducer nodes [24]. In such case, the natural implementation of a $JOIN$ is using the hash join algorithm, where the hash function maps keys to k buckets and data is shipped to k reducers. Assuming no data skewness, each reducer receives a fraction of $\frac{T(R)}{k}$ and $\frac{T(S)}{k}$. Having c as a constant representing the incurred network overhead per transferred HDFS block, the cost estimations of the *JOIN* are:

$$Execution_{estimator} = \frac{B(R) + B(S)}{k} \cdot c \qquad Space_{estimator} = B(R')$$

Note that the presented metrics can be highly variable within D. For instance, not surprisingly, *JOIN* nodes are the operations that consume the most time and space in order to generate intermediate results. Additionally, modern DIFs make heavy usage of User Defined Functions (UDFs) which consists of ad-hoc operations, difficulting the estimation of their I/O cost. An approach to solve this problem is to rely on static analysis of code to estimate the I/O cost for UDFs [25]. Finally, it is worth noting that other approaches exist to measure the presented metrics, for instance [26] proposes a method based on micro-benchmarking. On the contrary, our approach does not require any execution of the flow which however, impacts the quality of the estimation.

3.3. Cost functions

In this section, we present a set of cost functions to evaluate a BDS configuration, based on metrics from the materialized operational nodes of D. In our experiments, we focus on traditional metrics used in multi-query optimization namely loading cost, query cost, storage cost and freshness. However, note that our approach is extensible to other types of metrics such as monetary aspects [27], energy consumption [28], etc. For instance to estimate monetary cost, the pay-per-use cloud services charge based on the resources used, which can be estimated by our cost model. Further, the estimated resource utilization can be used to calculate the cost of renting machines on different cloud providers. Regarding storage, here we are not concerned with the layout to be used as this is assessed once the selection of nodes to be materialized has been found.

First, we must present some auxiliary methods over BDS configurations in which cost functions are based on. Let $Pre(b)$ and $Suc(b)$ be respectively the input and output subgraphs for a node b, recursively defined as $Pre(b) = b \cup \forall_{b_i \in predecessors(b)} Pre(b_i)$ and $Suc(b) = b \cup \forall_{b_i \in successors(b)} Suc(b_i)$, and respectively ending when $deg^-(b) = 0$ and $deg^+(b) = 0$. Hence, we can define the input and output subgraphs of a BDS configuration B as $I(B) = \bigcup_{b \in B} Pre(v)$ and $O(C) = \bigcup_{v \in C} Suc(v)$. Specifically, the former is a subgraph where its source nodes are the sources in a D and sink nodes are all the elements $b \in B$. The latter is a subgraph where its source nodes are all elements $b \in B$ and sink nodes are the final nodes in D. Additionally, $sources(b)$ gives the input sources of a node b and $sinks(b)$ provides the queries over a node b.

Loading Cost. The cost of loading a set of intermediate results CF_{LT} is the sum of processing source data and propagating them to the intermediate results in B. Our approach is valid for both maintenance and update of intermediate results, as long as source statistics are properly updated. From a BDS configuration B, the estimated loading cost is intuitively the cost of executing the operations

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

of D, loading the intermediate results for each node $b \in B$ (i.e., $I(B)$), and the cost of writing such results to the disk. Thus, we define $CF_{LT} = \sum_{b \in B}[\sum_{b_i \in I(b)} Execution_{estimator}(b_i) * RF(b_i)] + \sum_{b \in B} Space_{estimator}(b)$. Here, RF represents the refresh frequency of materialized nodes which is fixed in the characteristics vector CV of each node. The unit of refresh frequency is *total number of updates per time unit*. It should be noted that we are talking about sequential files that do not provide random access, so only full update is possible (no incremental).

Query Cost. The query cost CF_{QT} is the sum of querying the intermediate results, transform the data and deliver results to the user. From a BDS configuration B, the estimated query cost is computed as the sum of execution costs of successor nodes for each node $b \in B$ (i.e., $O(C)$). However, note that the cost of processing the operations of the nodes in B should not be taken into account as it is already evaluated in CF_{LT}. Therefore, it is necessary to consider only nodes in the set $O(B) \setminus B$, denoted $O^+(B)$. Finally, it is necessary to consider the cost of reading such intermediate results from the disk. Hence, we define $CF_{QT} = \sum_{b \in B}[\sum_{b_i \in O^+(b)}(Execution_{estimator}(b_i) * (\sum_{s_i \in sinks(bi)} QF(s_i))] + \sum_{b \in B} Space_{estimator}(b)$. Here, QF represents the frequency of queries. The query frequency can be expressed per day, hour or minute. QF helps to select the most reused node. Queries with high frequencies benefit more from the re-usage. Hence, a node which is used in highly repetitive queries will be given more weight during selection.

Storage Cost. The storage cost function CF_S concerns the storage space needed to store intermediate results. It is computed as the sum of estimated space for storing the results of each node in B, and it can be seen as the estimated space require to accommodate the deployed BDS configuration. It is defined as $CF_S = \sum_{b \in B} Space_{estimator}(b)$. Notice that $Space_{estimator}$ can be used for estimating the costs of reading and loading intermediate results, showed in CF_{LT} and CF_{QT}, as well as to estimate the occupied space for the case of minimizing or constraining its value.

Freshness. The freshness cost function estimates the freshness of the results of a query, which are obtained using materialized nodes, denoted as B'. For the freshness function, we build on the formula from [29]. The variable age tells how old data are in a materialized result with regard to the current data in the input sources. In our case, age cannot be known as it is not possible to foresee when materialized results are going to be used. It should be noted that update frequency of a node is calculated based on its input sources. Whereas, the refresh frequency is given in the characteristics vector CV of each node. We calculate the update frequency of a materialized node b as an average of the input frequencies of its input sources $UF(b) = Average_{s_i \in sources(b)} UF(s_i)$. Then, we can approximate age of b as the mid point between two refreshments $Age(b) = RF(b)^{-1}/2$. With such, we can measure the freshness of a node b as $Freshness(b) = (1 + UF(b) * Age(b))^{-1}$. Furthermore, $Freshness(b)$ is used to calculate the freshness of the results of a query Q as $CF_{Freshness}(Q_{results}) = Average_{b' \in B'} Freshness(b')$. This cost function helps to choose a node for materialization which provides up-to-date results to the queries.

4. State space search algorithm

As previously mentioned, the problem of intermediate results materialization in DIFs can be reduced to the general materialized view selection problem, known to be NP-hard. Hence, we must avoid

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

exhaustive algorithms and rely on informed search algorithms. Furthermore, in this particular case, purely greedy algorithms will not provide near-optimal results as the proposed cost functions are not monotonic. In classic artificial intelligence, a state space search problem is usually represented with the following components: *(i) initial state* where to start the search; *(ii) set of actions* available from a particular state; *(iii) transition model* describing what each action does and what are the derived results from it; *(iv) goal test* which determines whether the evaluated state is the goal state (i.e., the optimal state); and *(v) path cost* function to assign cost to the actions path.

In our context, we see a state as any BDS configuration B over which action functions are applied. It is noteworthy to mention that in such problem we are not interested in the set of actions that have led to a solution, but in the solution itself, which is initially unknown. Additionally, as any state B is a valid solution, we drop the component of *goal state*. Furthermore, the path cost is substituted by the definition of a *heuristic function*, which will guide the search. In the following subsections, we present the particularities of our specific problem for the remaining components.

4.1. Actions

For a BDS configuration B, we can compute actions (navigations over the graph), yielding new BDSs B'. First, we define the generic navigation operation $B' = Nav(b_{origin}, b_{destination})$, with $b_{origin}, b_{destination} \in D$ and semantics $Nav(b_{origin}, b_{destination}) = (B \setminus \{b_{origin}\}) \cup \{b_{destination}\}$. We then define three specific actions applied over nodes in B:

1. *Forward* ($F(b, b') = Nav(b, b')$): characterizing a forward movement from b to b' in D, applicable when $b' \in successors(v)$.

2. *Backward* ($B(b, b') = Nav(b', b)$): characterizing a backward movement from b' to b in D, applicable when $b' \in predecessors(b)$.

3. *Stay* ($N(b) = \emptyset$): always applicable, as it does not perform any movement. Such operator is only useful when other nodes b' are combined with M or U, so that a new state is generated where b remains selected.

4. *Increment* ($Inc(b, i)$): Increases the value of a characteristic (identified by position i_{th} of the vector CV) for a node b.

5. *Decrement* ($Dec(b, i)$): On the contrary, it helps to decrease the value of a characteristic for node b at i_{th} position of the vector CV.

From the previous definitions, for each node b, we define the set $Actions(b)$ as:

$$\bigcup_{b_i \in successors(b)} \{F(b, b_i)\} \cup \bigcup_{b_i \in predecessors(b)} \{B(b, b_i)\} \cup \{N(b)\} \cup \{Inc(b, i)\} \cup \{Dec(b, i)\}$$

Finally, we obtain all possible actions from a BDS configuration B by computing the Cartesian product of the power set of each $Actions(b_i)$ (note, empty sets are removed from each power set but this is not depicted for readability) as $\mathcal{P}(Actions(b_1)) \times \ldots \times \mathcal{P}(Actions(b_n))$. The rationale behind this operation is to generate, for each node b, all combinations of movements. The usage of a power set

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

is relevant for the cases when the input or output schemata of a node is not unary (e.g., a $JOIN$). Then, such different combinations are furtherer combined with the rest of nodes via a Cartesian product. Note that such set can be extremely large for complex Ds, however it is easy to see that many combinations generate invalid BDS configurations. To this end, we define the two essential conditions that a BDS configuration must fulfill in order to be valid, namely answerability and non-dominance.

Answerability of all queries. Ensuring that all queries (sink nodes) can be answered from materialized results. It can be checked by guaranteeing there is at least one materialized node for each path in D. Formally, $\forall b \in sources(D) \forall p_i \in Paths_{b,sinks(D)} \exists node \in p_i : node \in B$. For instance, in Figure 2a, we can see that the green-colored BDS configuration does not satisfy answerability as the path from N_2 to N_9 does not contain any materialized node.

Non-dominance of nodes. The purpose of our approach is to minimize the number of nodes to materialize by avoiding unnecessary materializations. For instance, if it is decided to materialize all sink nodes then it is unnecessary to materialize any intermediate node. In graph theory, a node m *dominates* n if all paths from the source node to n must pass through m. We extend this definition for the case of multiple nodes, and thus we test non-dominance of a set of nodes by checking that, for each node b there is at least one path from b to sink nodes where b is the only selected node. This is formally defined as $\forall b \in B \, \exists p_i \in Paths_{b,sinks(D)} : |\{\forall node \in p_i : node \in B\}| = 1$. For instance, Figure 2b, shows that the green-colored BDS configuration does not satisfy non-dominance, as nodes N_6 and N_7 dominate N_5.

Besides the two essential conditions, it is necessary to maintain a set of visited nodes to check whether a state B has not been already visited, and thus avoid unnecessary expansions in the search space. Figure 2c, depicts the valid BDS configurations obtained by applying actions to the BDS configuration $\{3,4\}$. Experimenting with the DIF in Figure 1, it has been observed that on average eliminating states that do not fulfill such conditions makes a reduction on the search space by 88%.

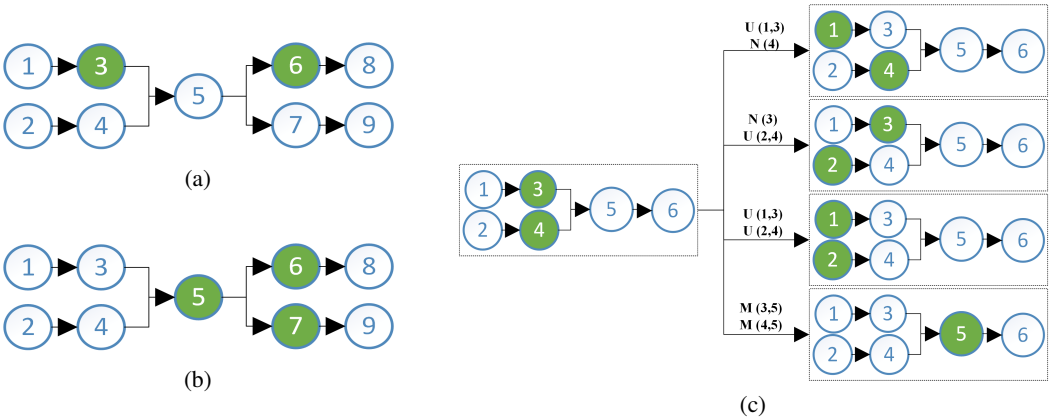

Figure 2: (a) depicts a BDS configuration where answerability is not satisfied, (b) depicts a configuration where non-dominance is not satisfied, and (c) depicts the valid actions for configuration $\{3,4\}$

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

Next, we generate increment and decrement actions for the current node to move vertically by using different index positions of CV. This helps to find the best possible values for given characteristics vector CV for each to-be-materialized node.

4.2. Initial state

As previously mentioned, the search space contains many local optimum points due to the non-monotonicity of cost functions, therefore the obtained solution might vary depending on the initial state. Three possible initial states have been devised aiming to cover all search space varieties:

- Materialize all source nodes, representing the BDS configuration $B = sources(D)$.
- Materialize all sink nodes, representing the BDS configuration $B = sinks(D)$.
- Random selection of nodes, guaranteeing a valid initial state where answerability and non-dominance are satisfied. Further, for the random selected nodes, we also randomly choose values in all positions of the characteristics vector CV.

Note that the two former are special cases of the third, thus this is the strategy that has been chosen to generate initial states (we provide a more thorough discussion on this in Section 4.4).

4.3. Heuristic

Provided that values of the different objectives lay in very different ranges, and in order to provide a consistent comparison, it is necessary to make use of a non-dimensional utility function normalizing all objectives. There exist a vast number of different normalization strategies [30]. For our purpose, and given the nature of the problem, we make use of the *normalized weighted sum* as utility function, defined as:

$$h(B) = U(\{CF_1, \ldots, CF_n\}, B) = \sum_{i=1}^{n} w_i \cdot CF_i^{trans}(B)$$

$CF_i^{trans}(B)$ stands for the evaluation of the transformed cost function for B, being $CF_i(B)$ is evaluation CF_i (see Section 2.2), CF_i^o the *utopia* point (i.e., minimal BDS for CF_i), and CF_i^{max} the maximal BDS:

$$CF_i^{trans}(B) = \frac{CF_i(B) - CF_i^o}{CF_i^{max} - CF_i^o}$$

Such approach yields values between zero and one, depending on the accuracy of both CF_i^o and CF_i^{max} computation. However, it is mostly unattainable to get their exact values, and for that we have to rely on estimations. To achieve this, we compute estimations of utopian BDSs for all cost functions as the union of all minimum nodes for each path from source to sink nodes. Maximum points are obtained by following the similar approach, in this case obtaining maximum nodes for each path from source to sink nodes. Note that, if design goals with constraints are presented, then it is possible to use such constraint value K as maximum point by dismissing the need of estimations.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

4.4. Searching the solution space

Local search algorithms consist of the systematic modification of a given state, by means of *Action* functions, in order to derive an improved state. Many complex techniques do exist for such approach (e.g., simulated annealing or genetic algorithms). The intricacy of these algorithms consists of their parametrization, which is also their key performance aspect at the same time . In this paper, we focus on *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search by always following the path that yields higher heuristic values. Since the used cost functions are highly variable due to their non-monotonicity, hill-climbing might provide different outputs depending on the initial state. In order to overcome such problem, we adopt a variant named *Shotgun hill-climbing* which consists of a hill-climbing with restarts (see Algorithm 1). After certain number of iterations, we can keep the best solution. Such approach of hill-climbing with restarts is surprisingly effective, specially when considering random initial states.

Algorithm 1 Shotgun Hill-Climbing

INPUT D, i: ▷ DIF, number of iterations
OUTPUT $solution$ ▷ Solution BDS configuration

```
1: solution = null
2: do
3:     B = randomInitialState(D); finished = false
4:     while !finished do
5:         neighbors = ResultsFromActions(B)
6:         B' = stateWithSmallestHeuristic(neighbors)
7:         if h(B') < h(B) then
8:             B = B'
9:         else
10:            finished = true
11:        end if
12:    end while
13:    if h(B) < h(solution) then
14:        solution = B
15:    end if
16:    − − i
17: while i > 0
18: return solution
```

5. Data format selection

In this section, we discuss how we choose the storage format for a materialized node. We compare the existing data formats based on their features. In this paper, we focus on the following set of data formats, which vary from each other in term of the implemented layout: *horizontal*: SequenceFile[3]

[3]https://wiki.apache.org/hadoop/SequenceFile

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

and Avro[4]; *vertical*: Zebra[5]; and *hybrid*: Parquet[6]. Next, we present our heuristic rules based on the devised features of such data formats.

5.1. Comparison of data formats

In Table 2, a comparison of all the layouts and their representative formats is given. This allows to look at their features side by side. As it can be noted from the table, all formats except SequenceFile, store the schemas of data. The schema information helps during the data serialization and de-serialization phases by avoiding the need to cast the data at the application level - which is a costly operation. Moreover, the table shows that both vertical and hybrid layouts provide support for column pruning. It means, they only read the required columns and do not perform unnecessary reads. Hybrid layouts can also push down the selection predicates into the storage layer because they store indexing information that helps in filtering the records while reading. Furthermore, since hybrid layouts store statistical information for each column, they enable easier computation of aggregates. Additionally, vertical and hybrid layouts have support for nested records which helps in storing bag, map and custom user data types. It can also be noted that hybrid layouts have additional features but they also have a significant overhead when writing and therefore when reading the same amount of data (due to the amount of metadata stored with data). Moreover, nowadays vertical layouts have been subsumed into hybrid, as they support all their features.

Table 2: Comparison of data formats

Features	**Horizontal**		**Vertical**	**Hybrid**	
	SequenceFile	**Avro**	**Zebra**	**ORC**	**Parquet**
Schema	No	Yes	Yes	Yes	Yes
Column Pruning	No	No	Yes	Yes	Yes
Predicate Pushdown	No	No	No	Yes	Yes
Metadata	No	No	No	Yes	Yes
Nested Records	No	No	Yes	Yes	Yes

In summary, each format provides a different set of features that will affect the overall performance when retrieving the intermediate results from the disk. Generally, hybrid layouts perform well if a subset of data is read. Alternatively, horizontal layouts perform well if all, or most of the data is read.

5.2. Selecting the appropriate format

In this section, we introduce the set of heuristic rules that aid on the decision of what data format to choose for each $b \in B$, which derive from well-known properties of horizontal, vertical and hybrid layouts and their features presented in Table 2. Note that more than one rule may apply when deciding

[4]https://avro.apache.org
[5]https://pig.apache.org/docs/r0.7.0/zebra_pig.html
[6]http://parquet.apache.org

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

for a given to-be-materialized node. In case two contradictory rules apply (e.g., selecting Avro and Parquet), we prioritize the selection based on the data format's features. Hence, we give the highest priority to Parquet owing to the fact that Parquet has more features and a more flexible behavior. The second highest priority is assigned to Avro because it stores schema information about the data which speeds up the reading. Finally, the lowest belongs to SequenceFile, which is only chosen for key-value data. Next, leveraging on the formalization presented in Section 2.2, as well as the operations presented in Section 4.1, we introduce the following set of rules:

$\texttt{rule1} : v \rightarrow$ SequenceFile, $\texttt{IF}\ |v.\mathbb{O}| = 2$

$\texttt{rule2} : v \rightarrow$ Parquet, $\texttt{IF}\ \exists x \in successors(v), \texttt{WHERE}\ x.\mathbb{S} \in \{AggregationOps\}$

$\texttt{rule3} : v \rightarrow$ Parquet, $\texttt{IF}\ \exists x \in successors(v), \texttt{WHERE}\ x.v_{pre} \subsetneq v.\mathbb{O}$

$\texttt{rule4} : v \rightarrow$ Avro, $\texttt{IF}\ \forall x \in successors(v), \texttt{WHERE}\ x.v_{pre} = v.\mathbb{O}$

$\texttt{rule5} : v \rightarrow$ Avro, $\texttt{IF}\ \exists x \in successors(v), \texttt{WHERE}\ x.\mathbb{S} \in \{Join, CartesianProduct, GroupALL, Distinct\}$

`Rule1` chooses SequenceFile for the materialization of nodes that have exactly two attributes. This is an immediate application of the SequenceFile format (which stores data as key-value pairs). Otherwise, several columns would need to be combined (e.g., with a separator marker such as "-" or ";") either in the key or the value and parsed at the application level. `Rule2` chooses Parquet when performing aggregations on data. Since Parquet stores statistical information for each column, it is the most efficient when computing aggregates. Also, Parquet's hybrid layout is also the best choice when it comes to read subsets of data, or when operators apply on subsets of columns (see Table 2). This rationale is behind `rule3`. Oppositely, Avro is chosen when all the data is read or when the operator does not apply on a certain subset of columns. This is a consequence of Avro implementing a horizontal layout. Hence, both `rule4` (the operator affects all columns) and `rule5` (the operator requires to read the whole data without filtering) recommend Avro. It is noteworthy to mention that our rules do not consider vertical layouts as they are subsumed by hybrid layouts. Furthermore, leveraging on the presented formalization other formats can be easily added. Finally, our algorithm for data format selection takes a materialized node b as input and finds the best storage format for it. Then, it iteratively applies all the heuristic rules and gets their suggested formats. Finally, it gets the best format among the ones that were suggested by using the previously presented prioritization.

6. Experiments

In this section, we report our experimental findings. Firstly, we report on the evaluation of our approach for intermediate results selection, and secondly on the format selection. Our experiments are performed on an 8-machine cluster. Each machine has a Xeon E5-2630L v2 @2.40GHz CPU, 128GB of main memory and 1TB SATA-3 of hard disk. Each machine runs Hadoop 2.6.0 and Pig 0.15.0[7] on Ubuntu 14.04 (64 bit). We have dedicated one machine for the name node and the remaining seven machines

[7]https://pig.apache.org

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

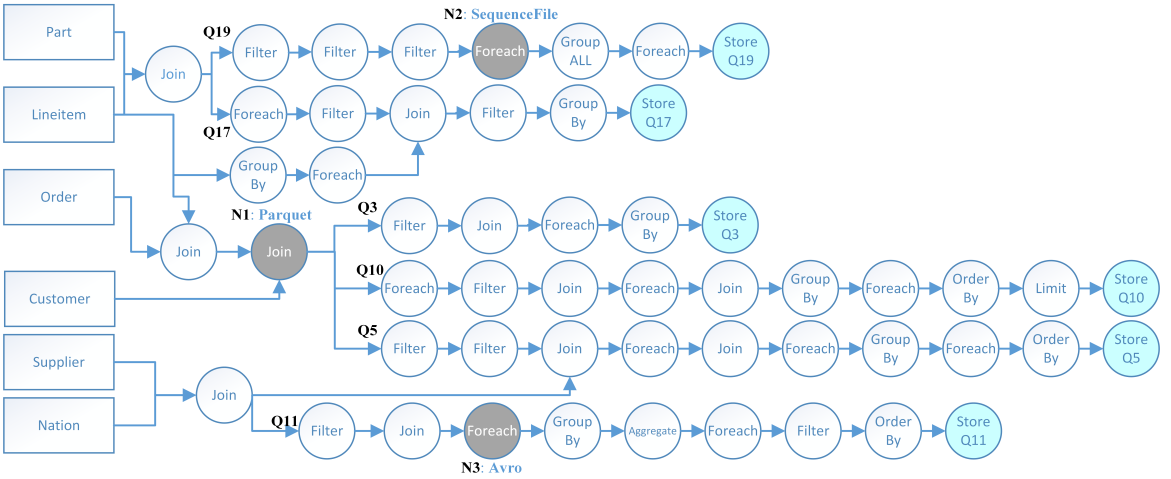

Figure 3: DIF of six TPC-H queries

for data nodes. We use TPC-H[8] benchmarking tool to generate datasets and queries. These queries have been converted to Apache Pig which is a procedural language of the big data systems. It has support for user defined functions which is a key feature of modern DIFs. In order to create a complex DIF, we use CoAl [31], which in this case, combines six TPC-H queries into one integrated DIF as shown in Figure 3. The DIF size is chosen with the goal of representing a realistic data pipeline, however being still tractable for validation with an exhaustive search.

6.1. Intermediate results selection evaluation

In this section, we evaluate our approach to validate its two properties: one is convergence and second is the quality of the obtained solutions. We also compare our approach with an existing state of the art solution to show its effectiveness.

6.1.1. Evaluation of convergence and quality of the obtained solutions

The goal of this experiment is to evaluate convergence and quality of the obtained solutions in Algorithm 1. For the sake of experiments, we assign update frequency to each table of TPC-H as shown in Table 3. We assume that *supplier* and *nation* tables never update and hence, they have 0 update frequency. Further, *part* and *customer* tables do not update very often and their changes can be applied every 6 hours. That is why, we assign them 4 per day update frequency. Finally, *orders* and *lineitem* tables are frequently updated and they update together whenever there is a new order. We assume that their changes are synchronized every 1 hour and thus, their update frequencies are 24 per day.

To evaluate the convergence of solutions, we systematically executed single shots of our approach (i.e., one iteration) until the number of obtained solutions converged and no new solutions were

[8]http://www.tpc.org/tpch/

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

Table 3: Update Frequency (UF) of TPC-H tables

Table Name	UF
Part	4 / day
Lineitem	24 / day
Orders	24 / day
Customer	4 / day
Supplier	0 / day
Nation	0 / day

obtained. With such information, and using the different frequencies, we can provide an estimation of the probability to obtain a solution B_K, formally defined as:

$$P(B_K) = \frac{freq(B_K)}{\sum_{j=1}^{n} freq(B_j)} \tag{1}$$

We aim to provide an estimation of the probability of the running Algorithm 1 with i iterations, a solution B_K will be found. To this end, we introduce Equation (2) measuring the probability to obtain such solution at position K ($1 \leq K \leq n$) by running i iterations. It should be noted that B_1 has been confirmed to be the optimal after performing a breadth first search.

$$P(B_K, i) = P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1) + P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1) \tag{2}$$

The above mentioned formula is a recursive formula where the base case (i.e., $P(B_K, 1)$) corresponds to the previously defined $P(B_K)$ (i.e., the probability to find solution B_K in one iteration). The rationale behind the recursive case is that after each iteration the one with the lowest heuristic value is kept. Thus, we measure the probability that the solution at position K (i.e., B_K) remains chosen after i iterations. This is achieved by adding (a) the probability that in the previous $i-1$ iterations, B_K was chosen and in the ith iteration an equal or worst solution is chosen (i.e., $P(B_K, i-1) \sum_{j=K}^{n} P(B_j, 1)$); and (b) the probability that in the previous $i-1$ iterations a worst solution was chosen and in the ith iteration B_K is chosen (i.e., $P(B_K, 1) \sum_{j=K+1}^{n} P(B_j, i-1)$). Intuitively, increasing the number of iterations, those with smallest heuristics will have a higher probability to be found regardless of the initial probability being low. With such basis, we can provide an estimation of the evolution of the probability to find a solution B_K by applying the aforementioned formula.

Based on the above mentioned formula, we experiment with the trade-off between different SLAs. We perform evaluation with the following settings: (1) two SLAs (i.e., load time, query time), equally weighted to 50%, (2) three SLAs (i.e., load time, query time, storage space), equally weighted to 33%, and (3) four SLAs (i.e., load time, query time, storage space, freshness), equally weighted to 25%. Our experiments show that the number of iterations to *converge* gradually increases with the number of considered SLAs. As shown in Figure 4, our approach takes 11 iterations, 26 iterations, and 39

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

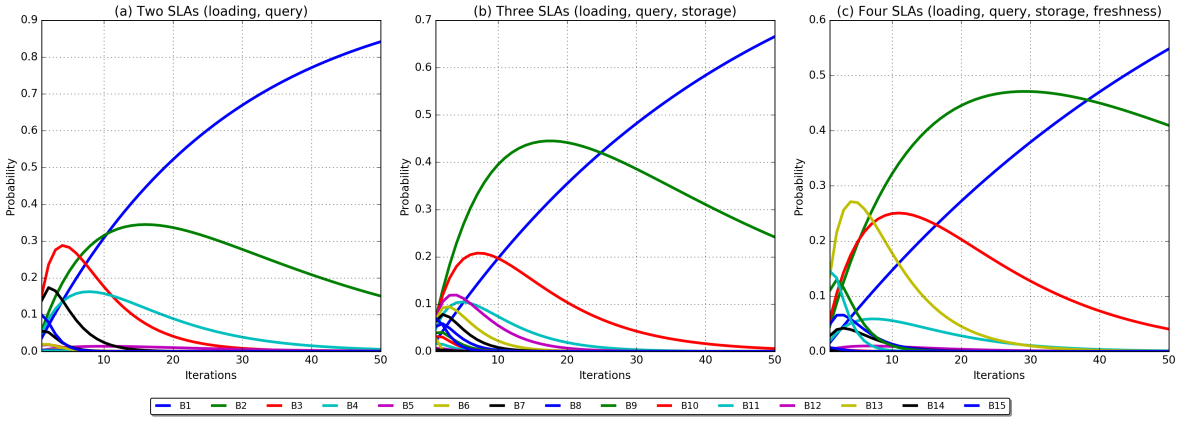

Figure 4: Evolution of probabilities per number of iterations for each different solution

iterations to converge (i.e., to be certain with a probability of 80%, that the obtained solutions will be one in the top three) for two, three, and four considered SLAs, respectively. In addition, we measure the average execution time of an iteration in different settings. Our approach takes 55.45 seconds, 58.68 seconds, and 183.34 seconds for two, three and four SLAs, respectively. For four SLAs, it takes more time because it has larger search space, due to the conflicting SLAs and the characteristics vector (i.e., refresh frequency). As all considered scenarios follow the same convergence trend as shown in Figure 4, let us focus on the most complex scenario involving the trade-off of four SLAs. For four SLAs, we obtained $n = 22$ different solutions across 90 executions. It can be seen that after 39 iterations, it is almost certain (i.e., $>$90% probability) that the obtained solutions will be one in the top three.

From such results, we conclude that the problem of finding optimal solutions by using hill-climbing indicates the issues with local optimums, known for greedy multi-objective optimization algorithms, and opens the challenge of applying more complex (i.e., parametrized solutions). However, the approach of shotgun hill-climbing, quickly yields near-optimal results after few iterations with high probability.

6.1.2. Comparison with an existing solution

Several intermediate materialization approaches for Big Data systems can be found in the literature, as discussed in Section 7.1. However, all of them focus on improving the query execution time without considering others SLAs (such as freshness). In order to show the effectiveness of our approach, we compare against the best representative solution (i.e., ReStore).

As mentioned in [3], 80% of queries are repeated in the range of minutes to hours. Thus, we created a sample workload by utilizing six TPC-H queries, based on the aforementioned work. We set four out of six queries as repetitive and two out of six as non-repetitive queries. In addition, we set their query frequencies in the range of minutes to hours as shown in Table 4. Moreover, ReStore has a configuration parameter for applying its eviction policies (to delete unused materialized nodes). For experiments, we chose different configuration values such as 9, 29, 55, and 70 in minutes to compare with all the possible behaviors of ReStore.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

Table 4: Sample workload based on TPC-H

Query Name	Start Time	Repeated	When to Repeat
Q3	0	Yes	6 / hour
Q5	1	No	-
Q10	2	Yes	2 / hour
Q11	3	Yes	1 / hour
Q17	0	Yes	14 / hour
Q19	2	No	-

Figure 5 depicts three charts to show different metrics for comparison. In Figure 5a, we compare *the total number of materialized nodes*, in Figure 5b, we show the total space required, and in Figure 5c, we show the average speedup gain in the repetitive queries. When executing the queries for the first time as shown in Figure 5a and Figure 5b, ReStore materializes each operator matching the heuristics and thus, it materializes more nodes and takes more space. Whereas, our approach uses the cost model to materialize only those nodes which satisfy all the four objectives (i.e., loading time, query time, storage space, and freshness).

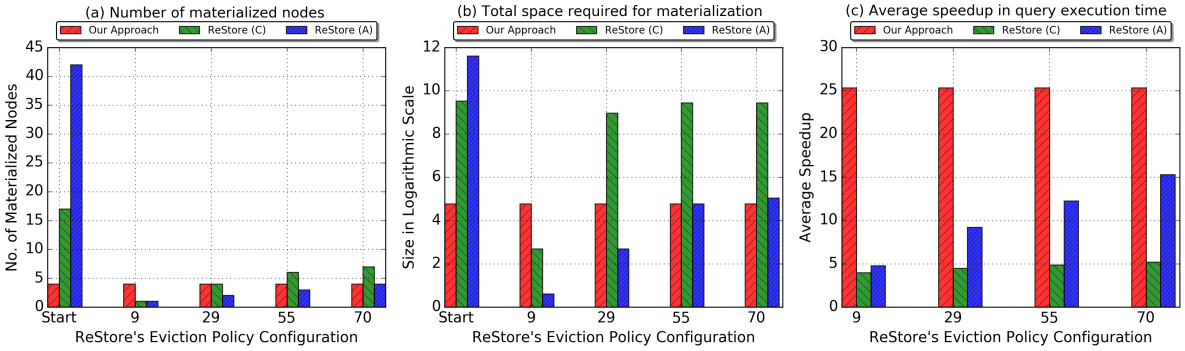

Figure 5: Comparison of our approach, ReStore (C) conservative heuristics and ReStore (A) aggressive heuristics

When we configured *9 minutes* for applying ReStore's eviction policies, it perceives only Q17 as a repetitive query because it is repeated before applying the eviction policies. Hence, it deletes all the materialized nodes except those which are used in Q17. Similarly, when we chose *29 minutes*, now it assumes that Q3 and Q17 are repetitive queries and keeps only their materialized nodes. This decision helps to reduce the occupied space but it also decreases the average speedup as shown in Figure 5c. Likewise, when we configured *55 minutes*, ReStore notices three queries (i.e., Q3, Q10, and Q17) as repetitive and keeps only the associated materialized nodes. As a consequence, it deletes all other materialized nodes which also reduces the average speedup. Finally, when we configured *70 minutes*, now it detects all possible repetitive queries and manages to keep all the required materialized nodes.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

However, still ReStore (C) keeps more nodes and takes more space compared to our approach as shown in Figure 5a and Figure 5b. On the other hand, ReStore (A) keeps a similar number of materialized nodes to our approach, but provides less average speedup. In general, our approach considers query frequency which helps to choose only the required materialized nodes from the start and provides better speedup than ReStore's heuristics.

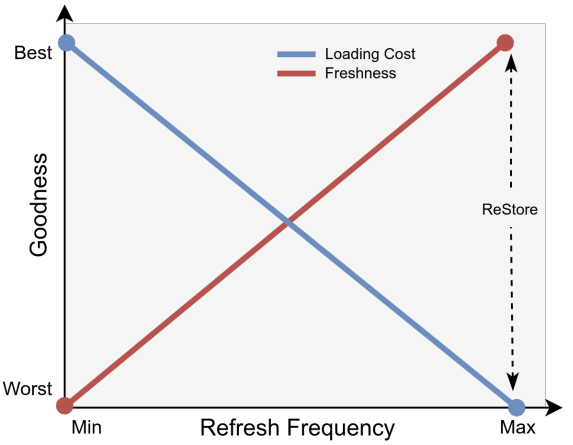

Figure 6: Effect of Refresh Frequency on Loading Cost and Freshness

In our experiments, we also evaluated our approach based on the characteristics vector (i.e. refresh frequency) to find the trade-off between loading cost and freshness. As shown in Figure 6, ReStore does not have support to balance them. It always deletes a materialized node as soon as any of its input source is updated. Thus, it always provides maximum freshness (only affected by the time to materialize new nodes). Consequently, it worsens the loading cost for materialized nodes, which may have highly variable input sources. Oppositely, our approach takes refresh frequency as an input and based on this, it tries to balance loading cost and freshness, by choosing the optimal value for each to-be-materialized node.

From these experiments, we conclude that our approach provides better solutions for materialization than ReStore. In addition, it can consider different SLAs as discussed in Section 3.3, which are not an option in the existing materialization solutions. Moreover, our example shows that we can also accept refresh frequency as a characteristic to find the balance between freshness and loading cost which is not possible in the existing materialized solutions.

6.2. Data format selection evaluation

In this section, we discuss the data format selection for chosen materialized nodes. Out of our experiments, we have selected three nodes which cover the three formats, as shown in Figure 3. Thus, we apply the heuristic rules in Section 5.2 for their format selection, which chose Parquet for N_1 (i.e., $JOIN$ operation), SequenceFile for N_2 (i.e., $FOREACH$ operation), and Avro for N_3 (i.e., also a $FOREACH$ operation), respectively. We use two metrics to analyze our approach, namely write and

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

read time which are measured for each materialized node by using the following formulas.

$$Time(Write) = Count_{stored}(Chunks) * Cost_{write}(Chunk)$$
$$Time(Read) = Count_{read}(Chunks) * Cost_{read}(Chunk) + Cost(FirstOP)$$

We only consider the first operator in `Time(Read)`, because the subsequent operators are executed in memory and hence they read from memory instead of HDFS. Figure 7, shows the $JOIN$ node materialized using Parquet. It should be noted that Parquet spends more time in writing (since it writes more metadata) but performs much better in reading (since it pushes down the predicates to the storage layer). Figure 7 (b, c and d) show the reading time for the intermediate results in different queries. The metadata writing overhead (e.g., schema) proves beneficial when reading is performed.

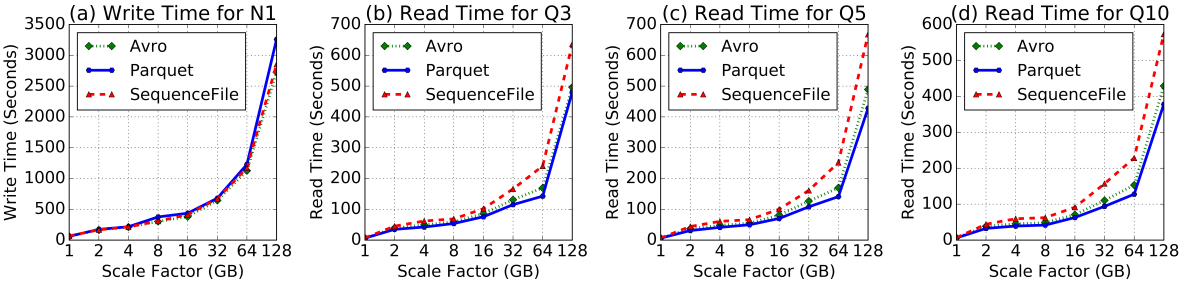

Figure 7: Experiment results for N_1 (Q_3, Q_5 & Q_{10})

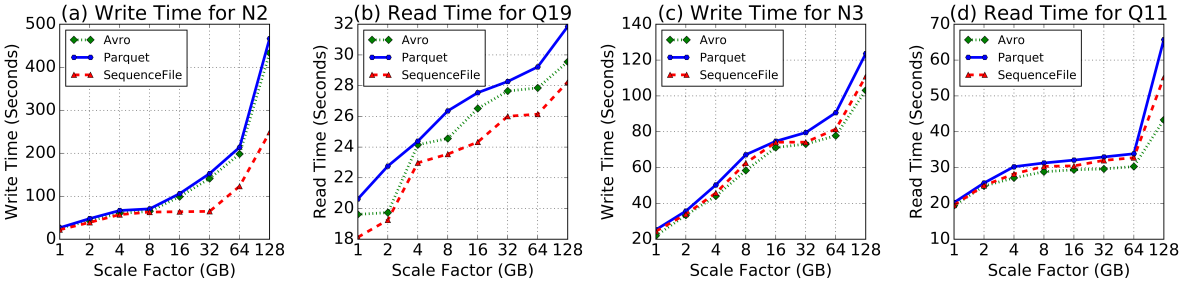

Figure 8: Experiment results for N_2 (Q_{19}) and N_3 (Q_{11})

Furthermore, it can be noted from Figure 8a and 8b, SequenceFile is a better choice for N_2. For all the other nodes, SequenceFile takes more time in writing than Avro because it stores data as key-value pairs and to store the columns in the value section, it needs markers to separate them. However, in N_2 SequenceFile writes less data since only two columns are written (one as key, the other as value) and no marker is needed. In N_2, SequenceFile performs also good when reading because it does not need to convert key-value pairs back to tuples. In Figure 8c and 8d, we show the performance for N_3, which chooses Avro. Note that Avro writes less data for all nodes except for N_2. This is the reason, why Avro performs well in N_3 because all the data needs to be read. However, in the other nodes, Avro does not perform that well because of the column pruning / predicate pushdown applied by Parquet.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

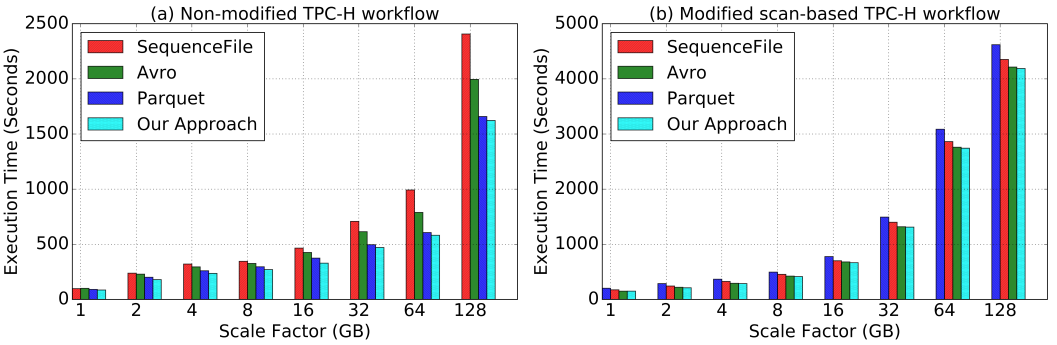

Figure 9: Single Fixed Format vs Our Approach

The experiments show that our rules work well in realistic scenarios such that of TPC-H. In Figure 9, we compare our approach with, when a single fixed format is chosen by depicting the overall execution time of a DIF. Figure 9a shows the overall performances in the TPC-H queries. For these queries, our approach on average provides 32% speedup over fixed SequenceFile, 19% speedup over fixed Avro, 4% speedup over fixed Parquet and overall, it provides 18% speedup. However, these queries have a very low selectivity factor (i.e., the median is 0.8%), which benefits Parquet. To exemplify a scenario where full-scans would dominate (e.g., for computing data mining algorithms), we modified the TPC-H queries to transform them into scan-based ones (i.e., 100% selectivity factor). In such scenario, our solution chooses Avro for N_1 instead of Parquet. Figure 9b shows the overall performances for the modified queries of TPC-H. In average, our approach provides a 9% speedup over fixed SequenceFile, 1.5% speedup over fixed Avro, 21% speedup over fixed Parquet and overall, it provides 10% speedup.

7. Related work

In this paper, we focus on two research problems: first is selecting intermediate results for materialization and second is choosing their storage layout. To the best of our knowledge, there is no existing solution which handles both. Thus, in this section, we independently discuss their respective related work.

7.1. Intermediate results selection

Our intermediate results selection approach encompasses four different research lines, namely: *materialized view selection in relational databases*, *materializing intermediate results in BDS*, *multi-query optimization*, and *sharing computation in BDS*. In this section, we separately present their related work.

Materialized Views Selection. The materialized view selection problem has been extensively studied in the context of data warehouses [13]. Most commercial database systems now include a physical design advisor that automatically recommends materialized views by analyzing a sample workload of queries (e.g. [32]). According to the survey [13], most view selection methods follow the approach of balancing the trade-off between query processing and view maintenance cost, and dismiss other relevant SLAs (such as freshness, etc.). Whereas, our approach is generic, thus it can consider

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

any type of SLAs that are quantifiable. In addition, our approach takes a characteristics vector (such as refresh frequency) as an input for different SLAs and based on it, it chooses the optimal characteristic value for each to-be-materialized node. This feature is not an option in the existing materialized views solutions.

Materializing Intermediate Results. There exist several approaches that aim to identify potential materialization of intermediate results for future reuse in Big Data systems. ReStore [10] is a heuristic based materialized solution which is implemented for Apache Pig. It has two heuristics (i.e., conservative and aggressive) to decide which operator's output to materialize. Similarly, m2r2 [33] also helps in choosing output of different operators for materialization. However, both solutions are tightly coupled with the technology and in addition, they do not consider different SLAs. In [34], a similar approach to ours is presented, however it focuses on a performance-oriented approach aimed to cloud environments, while we tackle any generic SLAs that can be represented with cost functions.

Multi-query Optimization. Similar to materialized view selection, there have been detailed work done on multi-query optimization in relational databases [35, 36]. Multi-query optimization focuses on improving the performance of concurrent running queries, while our approach focuses on recurrent queries which have redundant parts. They keep the redundant parts in the memory to use them in the currently running queries. On the contrary, we materialize the redundant parts to use them in the future queries.

Sharing Computations. Similar to multi-query optimization, there are few techniques proposed for Big Data systems. MRShare [9] proposes a cost model to group similar queries and optimizes re-usage of redundant parts in Hadoop. Similarly, [11] presents an approach for concurrent running jobs in Hadoop, by reusing existing multi-query optimization techniques. In general, their goal is to avoid redundant work of concurrent running jobs, whereas our work focuses on recurrent jobs.

7.2. Data format selection

The fixed format problem has been identified by the research community and several solutions have been proposed. Many proposals allow combining multiple layouts, for instance SAP HANA [37] uses horizontal and vertical layouts for OLTP and OLAP workloads, respectively. This is similarly found in DB2 [38], where horizontal and vertical layouts can be used for the same tablespace. These solutions, however, rely on fixed layouts which cannot be modified at runtime. To overcome the aforementioned issue, Polybase [39] relies on a multi-database environment where, based on workloads, it dynamically decides the most suitable solution and performs data transfers. Similarly, [40] proposes a hybrid adaptive store which reads raw files and chooses the layout based on the input workload. However, they propose to keep multiple copies of the same data in different formats, which might be unattainable for very large datasets. H2O [6] can dynamically decide the layout of the data based on the current workload. However, it only considers vertical layouts by creating different column groups, an NP-hard problem. Similarly, Trojan [41] is an adaptable column store for Hadoop, which takes advantage of data replication, that analyzes workload patterns and stores different column groups per replica.

The same problem has been extensively investigated under different contexts, such contexts include the selection between different implementations of a query operator, choosing among multiple candidate (equivalent) Web Service instances in flows with Web services, and choosing the most

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

appropriate execution engine in DIFs. In [42], they investigate the problem of allocating nodes of data-intensive flows to concrete executing engines. They prove that this problem is NP-hard and cannot be approximated within a small constant. An optimal polynomial time dynamic programming solution is proposed for the specific case of flows that are linear, i.e., a chain of activities.

In [43], they propose a brute force approach to find the optimal physical implementation of each operator before its execution has appeared. They model the problem as a state space search and based on the characteristics of the algorithm and dataset, they choose the optimal one. These kind of parameters cannot be applied in our approach because our approach focuses on reducing I/O cost in DIFs which are more data intensive rather than computation intensive. [44] is a web-service selection approach that considers the three objectives, namely performance, monetary cost, and reliability in terms of successful execution. They use max-min heuristic that aims to select a services based on its smallest normalized value. These parameters are not interesting in our context because they do not help in reducing I/O cost which is an important factor in DIFs.

In our MEDI'16 paper[7], we presented a problem of a fixed data format while materializing intermediate results. We proposed heuristic rules to decide the optimal data format for storing intermediate results. We also used our approach in an existing materialization solution named ReStore to show the effectiveness of our approach over a fixed data format. In this paper, we replace ReStore with our own materialized solution which considers different SLAs in selecting materialized nodes. Moreover, our solution is more generic and technology independent, whereas ReStore depends on physical operations of a language (i.e., Apache Pig).

8. Conclusions and future work

In this paper, we have presented an approach for the selection of intermediate results from data-intensive flows. We have built upon the general framework for materialized view selection by giving it additionally a multi-objective perspective. Moreover, we have provided a set of three cost functions with its building blocks (i.e., engine-independent statistics and engine-dependent metrics), and a representation of the approach as a state space search problem. Experimental results have showed that our approach is highly efficient in terms of performance, while providing near-optimal results. Moreover, we have also discussed the problems of a fixed data format for storing intermediate results. A fixed data format does not give the best performance for all types of workloads. Thus, we have discussed the need to introduce flexibility in the data format by deciding it based on the characteristics of the subsequent operators accessing the data. Specifically, we have shown for the Hadoop ecosystem that selecting the data format according to the access patterns helps in reducing the load time of the intermediate results. As a future work, we plan to extend our format selection rules with a cost-based approach.

References

[1] Jovanovic P, Romero O, Abelló A. A Unified View of Data-Intensive Flows in Business Intelligence Systems: A Survey. *T. Large-Scale Data- and Knowledge-Centered Systems*, 2016. **29**:66–107. doi: 10.1007/978-3-662-54037-4_3.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

[2] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008. **51**(1):107–113. doi:10.1145/1327452.1327492.

[3] Chen Y, Alspaugh S, Katz RH. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *PVLDB*, 2012. **5**(12):1802–1813. doi:10.14778/2367502.2367519.

[4] Harinarayan V, Rajaraman A, Ullman JD. Implementing Data Cubes Efficiently. In: SIGMOD. 1996 pp. 205–216. doi:10.1145/233269.233333.

[5] Gupta H, Mumick IS. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.*, 2005. **17**(1):24–43. doi:10.1109/TKDE.2005.16.

[6] Alagiannis I, Idreos S, Ailamaki A. H2O: a hands-free adaptive store. In: SIGMOD. 2014 pp. 1103–1114. doi:10.1145/2588555.2610502.

[7] Munir RF, Romero O, Abelló A, Bilalli B, Thiele M, Lehner W. ResilientStore: A Heuristic-Based Data Format Selector for Intermediate Results. In: MEDI. 2016 pp. 42–56. doi:10.1007/978-3-319-45547-1_4.

[8] Azim T, Karpathiotakis M, Ailamaki A. ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data. *PVLDB*, 2017. **11**(3):324–337. doi:10.14778/3157794.3157801.

[9] Nykiel T, Potamias M, Mishra C, Kollios G, Koudas N. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 2010. **3**(1):494–505. doi:10.14778/1920841.1920906.

[10] Elghandour I, Aboulnaga A. ReStore: Reusing Results of MapReduce Jobs. *PVLDB*, 2012. **5**(6):586–597. doi:10.14778/2168651.2168659.

[11] Wang G, Chan C. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 2013. **7**(3):145–156. doi:10.14778/2732232.2732234.

[12] Simitsis A, Wilkinson K, Castellanos M, Dayal U. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In: SIGMOD. 2009 pp. 953–960. doi:10.1145/1559845.1559954.

[13] Halevy AY. Answering queries using views: A survey. *VLDB J.*, 2001. **10**(4):270–294. doi:10.1007/s007780100054.

[14] Bian H, Yan Y, Tao W, Chen LJ, Chen Y, Du X, Moscibroda T. Wide Table Layout Optimization based on Column Ordering and Duplication. In: SIGMOD. 2017 pp. 299–314. doi:10.1145/3035918.3035930.

[15] Theodoratos D, Bouzeghoub M. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In: DOLAP. 2000 pp. 1–8. doi:10.1145/355068.355309.

[16] Theodoratos D, Sellis TK. Dynamic Data Warehouse Design. In: DaWaK. 1999 pp. 1–10. doi:10.1007/3-540-48298-9_1.

[17] Theodoratos D, Sellis TK. Data Warehouse Configuration. In: VLDB. 1997 pp. 126–135. `http://www.vldb.org/conf/1997/P126.PDF`.

[18] Jovanovic P, Simitsis A, Wilkinson K. Engine independence for logical analytic flows. In: ICDE. 2014 pp. 1060–1071. doi:10.1109/ICDE.2014.6816723.

[19] Jovanovic P, Romero O, Simitsis A, Abelló A. Incremental Consolidation of Data-Intensive Multi-Flows. *IEEE Trans. Knowl. Data Eng.*, 2016. **28**(5):1203–1216. doi:10.1109/TKDE.2016.2515609.

[20] Marler RT, Arora JS. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 2004. **26**(6):369–395. doi:10.1007/s00158-003-0368-6.

Final edited form was published in "Fundamenta Informaticae", 163(2), S. 111-138. ISSN 1875-8681.
https://doi.org/0.3233/FI-2018-1734

[21] Halasipuram R, Deshpande PM, Padmanabhan S. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. In: EDBT. 2014 pp. 307–318. doi:10.5441/002/edbt.2014.29.

[22] Garcia-Molina H, Ullman JD, Widom J. Database systems - the complete book (2. ed.). Pearson Education, 2009. ISBN 978-0-13-187325-4.

[23] Aggarwal A, Vitter JS. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 1988. **31**(9):1116–1127. doi:10.1145/48529.48535.

[24] Afrati FN, Ullman JD. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 2011. **23**(9):1282–1298. doi:10.1109/TKDE.2011.47.

[25] Hueske F, Peters M, Sax M, Rheinländer A, Bergmann R, Krettek A, Tzoumas K. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 2012. **5**(11):1256–1267. doi:10.14778/2350229.2350244.

[26] Simitsis A, Wilkinson K. Revisiting ETL Benchmarking: The Case for Hybrid Flows. In: TPCTC. 2012 pp. 75–91. doi:10.1007/978-3-642-36727-4_6.

[27] Nguyen T, Bimonte S, d'Orazio L, Darmont J. Cost models for view materialization in the cloud. In: EDBT/ICDT Workshops. 2012 pp. 47–54. doi:10.1145/2320765.2320788.

[28] Roukh A, Bellatreche L, Boukorca A, Bouarar S. Eco-DMW: Eco-Design Methodology for Data warehouses. In: DOLAP. 2015 pp. 1–10. doi:10.1145/2811222.2811230.

[29] Encyclopedia of Database Systems. chapter Data Quality Dimensions, pp. 612–615. Springer. ISBN 978-0-387-35544-3, 2009.

[30] Grodzevich O, Romanko O. Normalization and other topics in multi-objective optimization. In: FMIPW. 2006 pp. 42–56. `http://www.maths-in-industry.org/miis/233/`.

[31] Jovanovic P, Romero O, Simitsis A, Abelló A. Incremental Consolidation of Data-Intensive Multi-flows. In: TKDE. 2016 . doi:10.1109/TKDE.2016.2515609.

[32] Agrawal S, Chaudhuri S, Kollár L, Marathe AP, Narasayya VR, Syamala M. Database Tuning Advisor for Microsoft SQL Server 2005. In: VLDB. 2004 pp. 1110–1121. ISBN 0-12-088469-0.

[33] Kalavri V, Shang H, Vlassov V. m2r2: A Framework for Results Materialization and Reuse in High-Level Dataflow Systems for Big Data. In: CSE. 2013 pp. 894–901. doi:10.1109/CSE.2013.134.

[34] Qu W, Dessloch S. A Real-time Materialized View Approach for Analytic Flows in Hybrid Cloud Environments. *Datenbank-Spektrum*, 2014. **14**(2):97–106. doi:10.1007/s13222-014-0155-0.

[35] Roy P, Seshadri S, Sudarshan S, Bhobe S. Efficient and Extensible Algorithms for Multi Query Optimization. In: SIGMOD. 2000 pp. 249–260. doi:10.1145/342009.335419.

[36] Sellis TK. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 1988. **13**(1):23–52. doi:10.1145/42201.42203.

[37] Färber F, Cha SK, Primsch J, Bornhövd C, Sigg S, Lehner W. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 2011. **40**(4):45–51. doi:10.1145/2094114.2094126.

[38] Raman V, Attaluri GK, Barber R, Chainani N, Kalmuk D, KulandaiSamy V, Leenstra J, Lightstone S, Liu S, Lohman GM, Malkemus T, Müller R, Pandis I, Schiefer B, Sharpe D, Sidle R, Storm AJ, Zhang L. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 2013. **6**(11):1080–1091. doi:10.14778/2536222.2536233.

[39] DeWitt DJ, Halverson A, Nehme RV, Shankar S, Aguilar-Saborit J, Avanes A, Flasza M, Gramling J. Split query processing in polybase. In: SIGMOD. 2013 pp. 1255–1266. doi:10.1145/2463676.2463709.

[40] Idreos S, Alagiannis I, Johnson R, Ailamaki A. Here are my Data Files. Here are my Queries. Where are my Results? In: CIDR. 2011 pp. 57–68. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper7.pdf.

[41] Jindal A, Quiané-Ruiz J, Dittrich J. Trojan data layouts: right shoes for a running elephant. In: SOCC. 2011 p. 21. doi:10.1145/2038916.2038937.

[42] Kougka G, Gounaris A, Tsichlas K. Practical algorithms for execution engine selection in data flows. *Future Generation Comp. Syst.*, 2015. **45**:133–148. doi:10.1016/j.future.2014.11.011.

[43] Tziovara V, Vassiliadis P, Simitsis A. Deciding the physical implementation of ETL workflows. In: DOLAP. 2007 pp. 49–56. doi:10.1145/1317331.1317341.

[44] Tan W, Sun Y, Lu G, Tang A, Cui L. Trust Services-Oriented Multi-Objects Workflow Scheduling Model for Cloud Computing. In: ICPCA/SWS. 2012 pp. 617–630. doi:10.1007/978-3-642-37015-1_54.