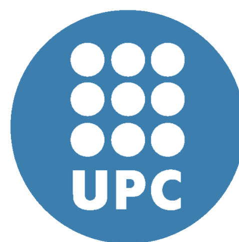
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC



Multi-Objective Materialized View Selection in Data-Intensive Flows

Sergi Nadal

Supervised by: Petar Jovanovic, Oscar Romero and Alberto Abelló

A thesis submitted for the master degree Erasmus Mundus Information Technologies for Business Intelligence

September 2015

Multi-Objective Materialized View Selection in Data-Intensive Flows

Sergi Nadal

Supervised by: Petar Jovanovic, Oscar Romero and Alberto Abelló

Abstract. Data-intensive flows deploy a variety of complex data transformations to build information pipelines from data sources to different end users. Materializing intermediate results of data and transformations shared by multiple users would clearly provide benefits in terms of performance and reusability. The classical materialized view selection problem has been tackled as the trade-off between query evaluation and view maintenance costs, further constrained by space boundaries. However, in modern BI applications a variety of other quality metrics, often captured by service level agreements (SLAs), must be taken into account, like reliability, recoverability, availability, and others.
In this thesis we present *Forge*, a tool for automating multi-objective materialization of intermediate results in data-intensive flows, driven by a set of different quality objectives. We revisit the traditional problem of materialized view selection, extending it for the case of data-intensive flows of BI 2.0. *Forge* can tackle multiple and conflicting quality objectives when searching for the optimal data materialization. Moreover, for assessing different objectives, we deploy efficient cost estimation techniques, leveraging on different data flow statistics. We report initial evaluation results, showing the feasibility and efficiency of our approach.

1 Introduction

Nowadays, it is widely accepted by business intelligence (BI) practitioners that the process of designing a data warehouse (DW) consists of a set of clear and well-defined sequential tasks. Altogether, it comprises a process of converting a set of coarse-grained business requirements into a final DW solution capable of providing answers to such requirements. Two core activities of such design process are multidimensional (MD) design of a DW schema and design of an extract-transform-load (ETL) process, these two design tasks are typically tackled sequentially in that order.

In early DW design approaches, where data warehouses were seen as sets of materialized views, the phase of target schema creation was typically tackled via the materialized view selection problem. For example, [50] presents a framework for materialized view selection based on the minimization of a set of design goals, which can be seen as quality factors of a DW design. Particularly, this framework aims at minimizing the *operational cost*, defined as a linear combination of query evaluation cost and view maintenance cost. Afterwards, ETLs were deployed in a form of refreshment processes, benefiting from efficient view maintenance techniques.

However, such setting soon became unrealistic, as real-world DW applications required more complex transformations over source data [46] (e.g., data cleaning). Lately, the rise of BI 2.0 systems (or self-service BI) [1], that argue for more operational decision making scenarios [11], brought back the idea of selecting the "right" materialization of input data for optimizing user workloads. In such scenarios, the complete DW deployment is usually avoided, and automatic tools are in charge of providing combined data flows to bring the "right" data, at the "right" time to end-users. Moreover, [8] 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. Thus, providing a partial materialization of shared flows, would highly increase the reusability and hence the performance of user workloads.

Some recent tools have been tackling the issue of automating either solely an ETL process design (e.g., Orchid [12]), or both ETL and DW schema designs (e.g., Quarry [22]). However, these approaches overlook the potential gain from providing the partial materialization of data, while leaving parts of user workloads to be executed at runtime.

However, we believe that the theoretical underpinnings for the problem of providing the "right" materialization of data in BI 2.0 systems, should be searched in a traditional materialized view selection framework (e.g., that of Theodoratos and Bouzeghoub [50]). However, notice that such framework is not directly applicable here, as it assumes a relational database environment and is therefore limited to the set of relational algebra operations. Conversely, data-intensive flows considered in BI 2.0 systems go beyond this, and typically require more complex transformations over data, such as data sampling, sentiment analysis, or cluster detection. Moreover, the given framework only considers the operational cost when selecting views for deploying a DW, while overlooking other important non-functional requirements, typically expressed as Service Level Agreements (SLAs) [48].

To tackle this issue, in this thesis we revisit the traditional framework for materialized view selection [50] and analyze its applicability and extensions for the context of data-intensive flows in BI 2.0. As a result, we present *Forge*, our tool for automatically selecting the optimal partial materialization of data, from a data-intensive flow, driven by multiple quality objectives represented as SLAs. We apply well-known multi-objective optimization techniques, proven to efficiently tackle multiple and conflicting objectives.

To assess different objectives considered by the system, we introduce efficient cost estimation techniques, leveraging on different data flow statistics gathered from data sources and propagated over the data-intensive flow. *Forge* currently considers an example set of four quality objectives (i.e., *query time*, *loading time*, *occupied space*, and *multidimensionality*). Besides the first three objectives that are typically used when deciding an optimal materialized view selection, we additionally analyzed *multidimensionality*, which shows to which extent the proposed data materialization fits the target MD schema of a DW. However, notice that *Forge* in general can work agnostic from the target DW schema, and build the partial materialization of data on the fly.

Lastly, we evaluate our approach using the flows based on the data integration benchmark TPC-DI [39]. First experimental results have shown the efficiency of *Forge* in applying the optimization techniques for finding the near optimal data materialization from an input data-intensive flow.

In particular, our main contributions are as follows:

- We revisit the materialized view selection problem for the context of data-intensive flows in BI 2.0 systems.
- We propose a novel system, called *Forge*, for multi-objective selection of the optimal partial materialization of data from data-intensive flows.
- We introduce efficient techniques for assessing quality objectives of data-intensive flows.
- We adapt existing multi-objective optimization algorithms for solving the problem of finding the optimal materialization of data-intensive flows.
- We finally show an initial set of experiments showing the effectiveness and quality of *Forge*.

Outline. In section 2 we formalize our approach and introduce an illustrative running example. In section 3 we describe the building blocks of *Forge*, i.e. statistics, metrics, and cost functions. In section 4 we present our approach in the context of multi-objective optimization. In section 5 we report on our experimental findings, while sections 6 and 7 discuss related work and conclude the document, respectively.

Appendices. In appendix A we provide detail on the measurement of data-intensive flow operators cost estimations to complete section 3. In appendix B we provide an overview of *Quarry*, the system where *Forge* has been implemented in, and some of its implementation details. Finally, appendix C extends the related work from section 6 is provided.

2 Formalization

2.1 Multiquery AND/OR DAGs and Data-Intensive Flows

The general framework for materialized view selection [50] relies on multiquery AND/OR DAGs. Widely used in the field of multiquery optimization, several works have used them to tackle the materialized view selection problem (e.g., [50, 52, 53]). As defined in [54] query AND/OR 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, provided a set of queries Q defined over a set of source relations R, a multiquery AND/OR DAG $\mathcal{G}$ is a query AND/OR DAG which may have multiple sink (query) nodes. Roughly speaking, the view selection problem can be expressed as a search space based problem over the multiquery AND/OR DAG $\mathcal{G}$, where the design goals are conditions over $\mathcal{G}$.

Additionally, [52] formalizes the output of the view selection problem as a data warehouse 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 there exist two special DW configurations: $\langle Q, Q^Q \rangle$ and $\langle R, Q \rangle$. The former represents a materialization of the query set Q, while the latter represents a complete materialization of the source data stores R as data warehouse. Note that a DW configuration $C = \langle V, Q^V \rangle$ is composed of a set of views and a set of queries. For the sake of simplicity, in the context of data-intensive flows, we represent a DW configuration as a set of views to materialize $C = \{v_1, \ldots, v_n\}$. From now on, we will interchangeably refer to both representations of C.

On the other hand, [25] gives a formalization for general data-intensive flows (e.g., ETL processes) where they capture both relational algebra and more complex data transformations. They are defined as parametrized DAGs, where vertices capture information such as: semantics, implementation type and used engine.

Such flows are encoded using their own logical model [57], namely xLM. In order to support the previously introduced specificities, they propose the concept of *agnostic xLM* (*a-xLM*) to encode logical flows, and *specific xLM* (*s-xLM*) to encode physical flows.

Example 1. Figure 1 depicts an example data-intensive flow based on the TPC-DI domain. Source nodes are labeled with S and query nodes (sinks) are labeled with Q. Specifically, such flow outputs aggregated taxes per minute (Q_1) and per hour (Q_2).

Clearly, any multiquery AND/OR DAG $\mathcal{G}$ is representable using data-intensive flow notation from [25]. The opposite does not hold, due to the fact that AND/OR DAGs are solely based on relational operators, while data-intensive flows are extended with more complex operations.

However, having the same problem context (i.e., the same inputs and outputs) as the traditional view selection framework in [50], for generalizing the problem of materialized view selection for data-intensive flows we revisit the given framework and analyze the new components that should adapt the framework for the case of data-intensive flows.

2.2 Components

Data-Intensive Flow Based on the formalization from [25], in our context we define a data-intensive flow DIF, as a DAG (V, E) where its nodes (V) are either data stores (DS) or operations (O), and its edges (E) represent the directed data flow. Operational nodes are defined as $o = \langle \mathbb{I}, \mathbb{O}, \mathbb{S} \rangle$, where:

- $\mathbb{I}$ and $\mathbb{O}$ are the sets of respectively the input and output schemata defined as a finite set of attributes.
- $\mathbb{S}$ expresses the formalization of operator's semantics in the form of a binary search tree.

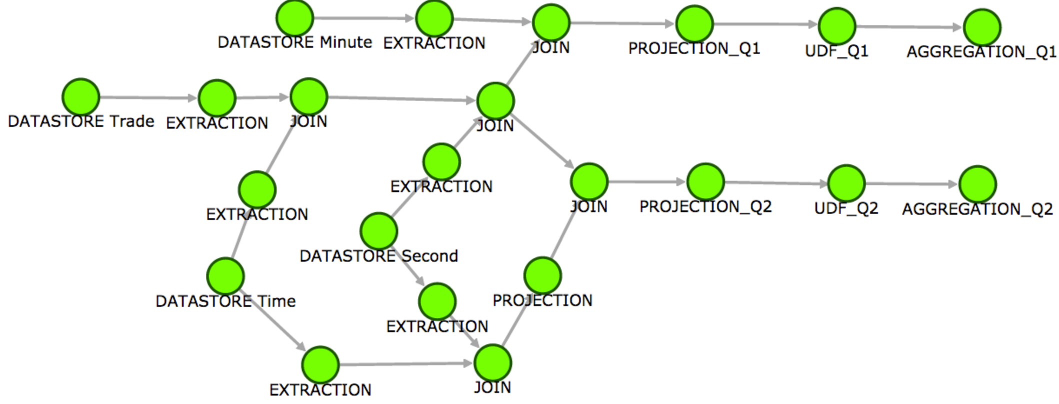


Fig. 1: Example data-intensive flow (based on TPC-DI)

Example 2. For the data-intensive flow depicted in example 1, an example of an operational node in V is the one that performs a JOIN between *Trade (TR)* and *Time (T)*, logically represented as:

- $I = \{\langle TR.tax, TR.timestamp\rangle, \langle T.hour, T.minute\rangle\}$
- $O = \{\langle TR.tax, TR.timestamp, T.hour, T.minute\rangle\}$
- $S = \langle minute(TR.timestamp) = T.minute\rangle$

Design Goal $\mathbb{DG}$ represents a set of design goals, where each (DG_i) characterizes a DW quality objective. 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*:

$$DG_{min}(SC) = \min_{C \in SC}(CF(C))$$

From a set of DW configurations SC, it returns the minimal configuration C by means of evaluating the cost function CF. Note that maximizing the cost function is equivalent to the negation of minimization:

$$DG_{max}(SC) = \max_{C \in SC}(CF(C)) = \min_{C \in SC}(-CF(C))$$

- *Constraints*:

$$DG(C) = [CF(C) \leq K]$$

For a DW configuration C, it checks whether the evaluation of the cost function CF fulfills the constraint K. Note that the constraint can express an arbitrary logical predicate (e.g., $<, >, \geq$).

Example 3. The design goal minimizing query time is $DG_{QT}(SC) = \min_{C \in SC} CF_{QT}(C)$, where CF_{QT} is the cost function that computes the query cost in the DW configuration C. The design goal that sets the space boundary is $DG_S(C) = CF_S(C) \leq S_{max}$, where CF_S is the cost function that computes the space occupied by the views in C, and S_{max} denotes the space threshold.

Cost Function Given a DW configuration $C = \langle V, Q^V\rangle$, $\mathbb{CF}$ represents a set of cost functions where each (CF_i) is the estimation of a DW quality factor from one of the components of C, either $C.V$ or $C.Q^V$ (here characterized as $C.x$). Formally:

$$CF(C) = \sum_{x \in C.x} E(x)$$

where $E(x)$ is the cost estimator of parameter $C.x$.

Example 4. The cost function estimating the storage space needed for materializing the views in V is $CF_S(C) = \sum_{V \in C.V} Space_{estimator}(V)$.

Utility Function In the context of multi-objective optimization (MOO) [34], it is common to aggregate all objectives $DG_1, \ldots, DG_n$ into a single value to obtain the global utility. Such function, is known as the *utility function* U, and is formally defined as $U(\mathbb{DG}) = U(CF_1(C), \ldots, CF_n(C))$. Each $CF_i(C)$ provides a quantitative evaluation of C (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, $+\infty$ for satisfied and non-satisfied constraints, respectively. Generally in MOO higher utilities are always preferred to lower utilities. However, in our context there are some CF where we aim for minimal utilities (i.e., query time).

Example 5. One of the most commonly used utility functions is the weighted sum. Provided a DW configuration C and a set of weighting coefficients $\mathbb{W}$, it is defined as:

$$U(\mathbb{DG}) = \sum_{i=0}^{n} w_i \cdot CF_i(C)$$

2.3 Problem Statement

In terms of the previously presented formalization, we now state the problem of multi-objective materialized view selection in data-intensive flows as:

Input

- A data-intensive flow DIF.
- A set of design goals $\mathbb{DG}$.
- A set of cost functions $\mathbb{CF}$.
- A utility function $U(\mathbb{DG})$.
- A cost model represented by a set of estimators over DIF and calculated by means of statistical information from source relations.

Output

- A DW configuration $C' = \langle V', Q'^V \rangle$ such that $U(\mathbb{DG})$ is minimal.

Note that such approach guarantees that all resulting configurations are self-maintainable, which entails that a given solution $\langle R, Q \rangle$ is a replication of the source relations into a DW, and the data sources are not queried.

3 Metrics and Cost Functions

3.1 Data-Intensive Flow Statistics

As previously mentioned, cost functions are computed from estimators. Every operational node in a DIF might have several estimators, each assessing a single quality factor (e.g. execution cost). They perform a cost based estimation according to the operator semantics and additional characteristics present in the node, i.e. the encoded information in a DIF as described in section 2.1.

In order to devise more accurate metrics, some essential statistics must be obtained from the input data stores and propagated across the DIF. By traversing the DIF in topological

order we are able to properly propagate such statistics at each node, based on specific operator semantics. [16] describes a complete set of statistics necessary to perform cost based estimations for ETL flows. 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, as opposed to metrics, the way in which the values of these statistics are obtained is independent of the underlying engine where the flow is executed.

Example 6. Let us assume a JOIN operator $R' = R \bowtie S$ with semantics $P_{R.a=S.b}$. Inspired by the work in [13], we can measure the above-mentioned statistics as:

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

$$V(R'.att_i) = V(R.att_i) \cdot (1 - sel_P)^{\frac{T(R)}{V(R.att_i)}}$$

$$T(R') = sel'_P(R \bowtie S) \cdot T(R) \cdot T(S)$$

The selectivity factor is obtained as the fraction of the number of shared values in the join attributes and the size of their cartesian product, while the number of distinct values is estimated proportionally to the number of tuples that such selectivity factor sel_p reduces.

3.2 Metrics

Once statistics for a DIF have been calculated, they can act as building blocks to metrics. As previously said, metrics are the engine-dependent estimation of a quality factor for a specific operational node in DIF. Here we focus on estimating both performance-wise ($Execution_{estimator}$) and space-wise ($Space_{estimator}$) metrics, the former measured by means of estimated disk I/O (in blocks) and the latter by number of disk blocks occupied by the materialized view. It is worth noting that in terms of execution, as the CPU cost is negligible as opposed to I/O cost, 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 $Execution_{estimator}$.

In order to devise the aforementioned 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 8 KB, while in Hadoop's HDFS it is 64 or 128 MB. 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 7. Given the join operation from example 6, in a relational DB, one implementation of such operator is based on the block-nested loop algorithm, thus the estimation for execution and space costs is as follows:

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

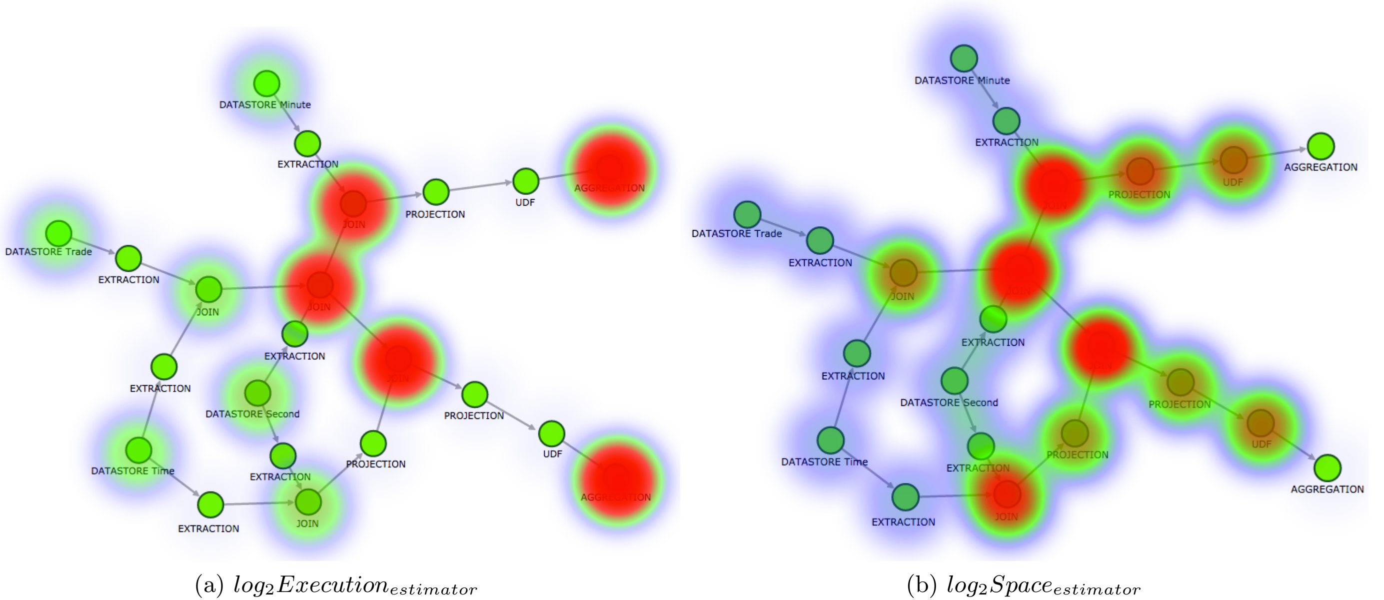
$$Space_{estimator} = B(R')$$

However, in a MapReduce environment, execution cost is dominated by data transfers that occur during the data shuffling phase between mapper and reducer nodes, i.e. communication cost over the network [2]. Assuming a hash join, where the hash function maps keys to k buckets, 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 MapReduce block, the cost estimations of the join are:

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

$$Space_{estimator} = B(R')$$

Note that the presented metrics can be highly variable within a DIF, this fact is depicted in a logarithmic scale through heatmaps in figure 2 based on example 1. For instance, not surprisingly, $JOIN$ nodes are the most consuming operations in both time and space to generate intermediate results. Finally, it is worth noting that other approaches exist to measure such metrics, for example [47] propose a method based on microbenchmarking of hybrid flows. On the contrary, our approach does not require to perform any execution of the flow, however that impacts the quality of the estimation.



(a) $log_2 Execution_{estimator}$ (b) $log_2 Space_{estimator}$

Fig. 2: Heatmaps for DIF metrics, blue-green-red in that order depict lower to higher values

3.3 Cost Functions

At this point, we are able to provide cost functions to evaluate a DW configuration $C = \langle V, Q^V \rangle$, based on metrics from materialized operational nodes, i.e. those specified in V, for a data-intensive flow DIF. First we must present some helper methods over DW configurations on which cost functions are based on. Let $I(C) = \bigcup_{v \in C} Pre(v)$ and $O(C) = \bigcup_{v \in C} Suc(v)$ be respectively the input and output subgraphs of C. Specifically, the former is a subgraph where its source nodes are the sources in DIF and sink nodes all elements v in C, similarly

the latter is a subgraph where its source nodes are all elements v in C and sink nodes the sinks in DIF. Formally, they are computed as follows:

$$Pre(v) = v \cup \forall_{v_i \in predecessors(v)} Pre(v)$$

$$Suc(v) = v \cup \forall_{v_i \in successors(v)} Suc(v)$$

where $Pre(v)$ and $Post(v)$ are recurrences, ending when $deg^{-}(v) = 0$ and $deg^{+}(v) = 0$, respectively.

Loading Cost The cost of loading a set of views, i.e., view maintenance cost, is the sum of transforming source data and propagating them to the views in C. In a DW, this can be seen as a cost of ETL processes to load such DW. Our approach is valid for both refreshment and update of materialized views, as long as source statistics are properly updated.

From a DW configuration C, the estimated loading cost is intuitively the cost of executing the parts of the DIF loading the materialized views for each node $v \in C$, i.e. $I(C)$ and the cost of writing such materialized views to a disk.

$$CF_{LT} = \sum_{v \in I(C)} Execution_{estimator}(v) + \sum_{v \in C} Space_{estimator}(v)$$

Query Cost The query cost is the sum of querying the materialized views in the DW, transform the data and deliver results to the user. From a DW configuration C, the estimated query cost is computed as the sum of execution costs of successor nodes for each node $v \in C$, i.e. $O(C)$. However, note that the cost of processing the operations of the nodes in C 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 $O(C) \setminus C$, denoted $O^{+}(C)$. Finally, it is necessary to consider the cost of reading such views from a disk.

$$CF_{QT} = \sum_{v \in O^{+}(C)} Execution_{estimator}(v) + \sum_{v \in C} Space_{estimator}(v)$$

Space This cost function concerns the storage space needed to store materialized views. It is computed as the sum of estimated space for storing the results of each node in C, and it can be seen as the estimated space necessary to accomodate the deployed data warehouse.

$$CF_{S} = \sum_{v \in C} Space_{estimator}(v)$$

Notice that $Space_{estimator}$ can be used for estimating the costs of reading and loading materialized view, showed in CF_{LT} and CF_{QT}, as well as for estimating the occupied space for the case of minimizing or constraining its value.

Multidimensionality In a typical DW environment, a target multidimensional schema is generated in order to provide a conceptualization of the data warehouse in terms of dimensions, facts and their relations. In this work, we assume the existence of a reference MD schema (MD_{ref}) generated from user requirements (queries). It is possible to evaluate the multidimensionality of non-dimensional materialized views by comparing the output schema of an operational node to the schema of factual and dimensional nodes in MD_{ref}.

To this end, we use *Jaccard index*, which is a simple, but powerful similarity measure between two sets. Hence, multidimensionality is measured as the sum of *Jaccard* indices for the most similar nodes in MD_{ref}. Note that we are interested in maximizing multidimensionality as a design goal, thus in order to be combined with other (minimizing) objectives, it should be converted to minimizing design goal, as explained in section 2.2.

$$CF_{MD} = \sum_{v \in C} [max\left(Jaccard(v, MD_{ref})\right)]$$

4 Proposed approach: Forge

As previously mentioned, the problem of materialized view selection in data-intensive flows can be reduced to the general materialized view selection problem. In [15] it is shown that such problem is NP-hard, hence we have to avoid 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 (e.g., see heatmaps in figure 2). In classical AI, a state space search problem is usually represented with the following five components:

1. *Initial state* where to start the search.
2. *Set of actions* available from a particular state.
3. *Transition model* describing what each action does and what are the derived results from it.
4. *Goal test* which determines whether the evaluated state is the goal state, i.e. the optimal state.
5. *Path cost* function to assign cost to the actions path.

In this work we follow such approach, in the following subsection we present the particularities of our specific problem in the context of a search state problem.

4.1 Multi-Objective Materialized View Selection as a State Space Search Problem

In our context, we see a state as any DW configuration $C = \{v_1, \ldots, v_n\}$ over which action functions are applied. It is noteworhy 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 C is a valid solution, therefore 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.

Actions For a DW configuration C, we can compute actions (navigations over the graph), which will yield new DW configurations C'. First, we define the generic navigation operation $C' = Nav(v_{origin}, v_{target})$, with $v_{origin}, v_{target} \in DIF$ and semantics defined as:

$$Nav(v_{origin}, v_{target}) = (C \setminus v_{origin}) \bigcup (C \cup v_{target})$$

We then define three specific operators, applied over nodes in C:

1. *Materialize* ($M(v, v') = Nav(v, v')$): characterizing a forward movement from v to v' in the DIF, applicable when $v' \in successors(v)$.
2. *Unmaterialize* ($U(v, v') = Nav(v', v)$): characterizing a backwards movement from v' to v in the DIF, applicable when $v' \in predecessors(v)$.
3. *Stay* ($N(v) = \emptyset$): always applicable, as it does not perform any movement. Such operator is only useful when furtherly combined with operations M or U.

From the previous definitions, for each node v we define the set *Actions*(v) as the following union:

$$\forall_{v_i \in successors(v)} M(v, v_i) \cup \forall_{v_i \in predecessors(v)} U(v, v_i) \cup N(v)$$

Finally, we are able to obtain all possible actions from a DW configuration $C = \{v_1, \ldots, v_n\}$ by computing the cartesian product of the power set of each $Actions(v_i)$ (for the sake of readability it is not shown, however empty sets should be removed from each power set).

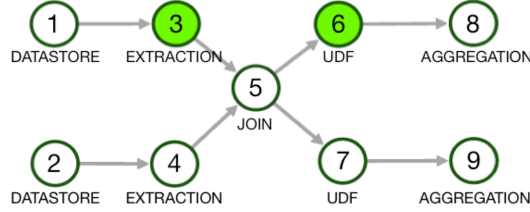
$$\mathcal{P}(Actions(v_1)) \times \ldots \times \mathcal{P}(Actions(v_n))$$

Note that such set can be extremely large for complex DIFs, but it can be clearly seen that many of those actions do generate configurations that can be considered invalid. We define the two essential conditions that a DW configuration must fulfil in order to be valid:

1. *Answearability of all queries*: this condition ensures that all queries (sink nodes) can be answered from the materialized views. This can be checked for each path from source to sink nodes, ensuring there is at least one materialized node.

$$\forall_{v \in sources(DIF)} \forall_{p_i \in Paths_{v,sinks(DIF)}} \exists_{node \in p_i} : node \in C$$

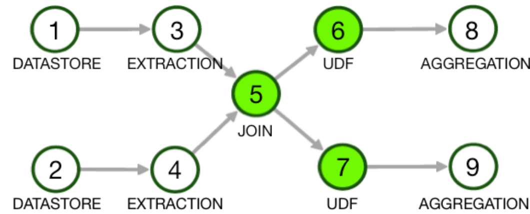
Example 8. The green-colored configuration does not satisfy answearability as the path from 2 to 9 does not contain any materialized node.



2. *Non-dominance of nodes*: the purpose of our approach is to minimize the number of nodes to materialize avoiding all unnecessary materializations, for instance if it is decided to materialize all sink nodes there is no point in materializing 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 path from a materialized node v to sink nodes, there is at least one containing only v.

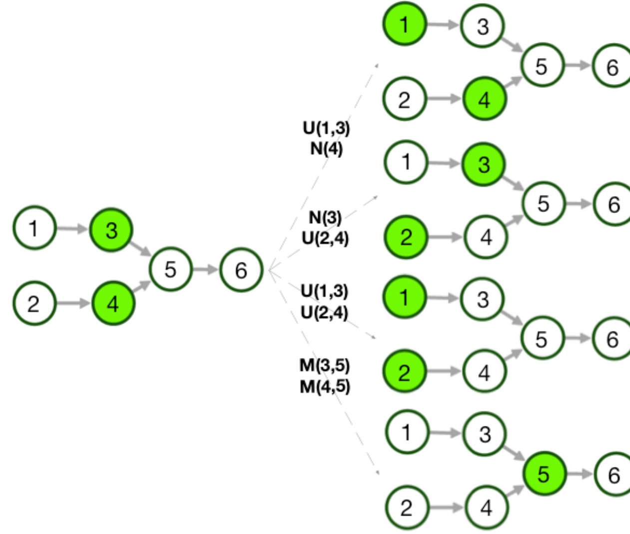
$$\forall_{v \in C} \exists_{p_i \in Paths_{v,sinks(DIF)}} : |\{\forall_{node \in p_i} : node \in C\}| = 1$$

Example 9. The green-colored configuration does not satisfy non-dominance as nodes 6 and 7 dominate 5.



Besides the two essential conditions, it is necessary to maintain a set of visited nodes to check whether a state C has not been already visited, and thus avoid unnecessary expansions in the search space. Experimenting with the example DIF depicted in 1 it has been observed that on average eliminating states that do not fulfil such conditions make a reduction on the search space by 88%.

Example 10. The following figure depicts all valid actions from $C = \{3, 4\}$, after pruning those that do not fulfill the conditions.



Initial States As shown in figure 2, the search space contains many local optimum points due to non-monotonicity of cost functions, therefore the solution might be highly different depending on the initial state. Four possible initial states have been found trying to cover all search space varieties:

1. *Sources*: considering as materialized nodes all source nodes, it represents the DW configuration $C = \langle R, Q \rangle$.
2. *Queries*: as opposite, here all sink nodes are materialized and represents the configuration $C = \langle Q, Q^Q \rangle$.
3. *Random*: widely used in the field of artificial intelligence, random states surprisingly yield decent results. We compute such states as shown in algorithm 1.

Algorithm 1 Random Initial State

Input DIF ▷ Data-Intensive Flow
Output C ▷ DW Configuration

1: ▷ For each path p from source to sink nodes get a random node
2: $C = \emptyset$
3: $r = \emptyset$
4: **for** $p \in Paths_{sources(DIF),sinks(DIF)}$ **do**
5: $\quad r$.`add(randomNode(p))`
6: ▷ Check if any element in $\mathcal{P}(r)$ is a valid configuration (non-dominance)
7: **for** $C \in \mathcal{P}(r)$ **do**
8: $\quad$ **if** `nonDominance(`C`)` **then return** C
9: ▷ Restart if none
10: `restart`

One might quickly see that the generation of a random state entails an exponential cost (given that a power set is computed). However given the nature of a data-intensive flow, it is highly likely that the same intermediate node will appear in many different paths, and in accordance to the Central Limit Theorem (CLT) it is highly unlikely that such power set outgrows the computational limits. Following the same idea, the random state is guaranteed to be a balanced configuration betwixt both extremes $\langle R, Q \rangle$ and $\langle Q, Q^Q \rangle$.

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 which normalizes all objectives. There exist a vast number of different normalization strategies [14]. For our purpose, and given the nature of the problem, we make use of the *normalized weighted sum* as utility function:

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

$CF_i^{trans}(C)$ stands for the evaluation of the transformed cost function for DW configuration C, defined as:

$$CF_i^{trans}(C) = \frac{CF_i(C) - CF_i^o}{CF_i^{max} - CF_i^o}, where :$$

- $CF_i(C)$: evaluation of cost function CF_i (section 2.2).
- CF_i^o: *utopia* point for CF_i (a.k.a., ideal point), which refers to the DW configuration where CF_i is minimal.
- CF_i^{max}: maximum point for CF_i refers to the maximal obtain value of CF_i for given DW configurations.

Such approach generally 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 the exact values and we have to rely on estimations. To achieve so, we compute estimations of utopian configurations 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 stating constraints are present, it is possible to use such constraint value K as maximum point dismissing the need of estimations.

4.2 Searching The Solution Space

Local search algorithms consist on the systematical modification of a given state, by means of *Action* functions, in order to derive an improved solution state. Many complex techniques do exist for such approach, e.g. *simulated annealing* or *genetic algorithms*. The intricacy of these algorithms consists on their parametrization, which is at the same time their key performance aspect.

We thus focus on *hill-climbing*, a non-parametrized search algorithm, which can be seen as a greedy local search, always following the path that yields higher heuristic values.

Since the cost functions we are using 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 2). After certain number of iterations, we can obtain the most converging solution.

Such approach of hill-climbing with restarts is surprisingly effective, specially when considering random initial states. In the next section we scrutinize the performance and quality of the algorithm with the components defined in section 4.1.

5 Evaluation

5.1 Experimental Scenario

The evaluation of our approach is based on the data-intensive flow depicted in figure 1. In this section, we first provide a performance evaluation of the approach (i.e. the overhead

Algorithm 2 Shotgun Hill-Climbing

```
Input i:                                              ▷ Number of Iterations
Output C′                                             ▷ Solution DWConfiguration
 1: solutions = ∅                                     ▷ Here a multiset (i.e. bag)
 2: do
 3:     C′ = initialState()
 4:     finished = false
 5:     while !finished do
 6:         neighbors = ResultsFromActions(C)
 7:         h′ = highestHeuristic(neighbors)
 8:         if h(C′) < h′_value then
 9:             C′ = h′_state
10:         else
11:             finished = true
12:     solutions = solutions ∪ C′
13:     − − i
14: while i > 0
15: return mostFrequentElementInMultiset(solutions)
```

of the multi-objective view selection), and then we evaluate the convergence and quality of obtained solutions for the algorithm in section 4.

We experiment with 15 different combinations of weights for 4 considered objectives (see table 1). These combinations aim at covering different cases of objective dominance, going from a single objective weighted 100%, through all 4 objectives equally weighted 25%. To evaluate the convergence of solutions, for each of these 15 combination, we systematically execute independent executions of i iterations, with i in the range $[1, 75]$, i.e. a total of 2850 for each combination. The appropriate number of iterations (i.e., 75) has been chosen experimentally to guarantee that, for any combination of weights, the solution converges.

Each individual execution of the algorithm is measured in terms of execution time. Additionally, for each batch of i executions, the obtained configurations are stored in a list with associated heuristic values, and ordered by the number of value occurrences (i.e., *frequencies*).

5.2 Performance Evaluation

One common problem in multi-objective algorithms is their scalability in terms of number of objectives. Here we want to see how our approach scales, in other words how it behaves for objectives with a given weight greater than 0%.

By evaluating all 15 combinations with the 4 considered objectives, we can observe if there is any objective penalizing (dominating) the overall evaluation. Table 1 depicts, for each combination of weights, the average time to execute the algorithm over a total number of 2850 executions, resulted from the chosen number of iterations (see section 5.1).

For considering only one objective (i.e., weight being 100%; first 4 rows of table 1), the average time goes between 1,82s for multidimensionality and 3,8s for query time. Notice that this difference is cased by the manner in which the given objective is assessed, i.e., multidimensionality checks the *Jaccard* index only for the output of the considered operation ($\Theta(|C|)$), while load and query time check the costs of all the preceding ($\Theta(|I(C)|)$) and succeeding ($\Theta(|O^+(C)|)$) operations, respectively. Furthermore, the execution time raises to 6.82s, for equally weighed objectives (each weight being 25%). This is due to the fact that more objectives require more cost function evaluations in order to calculate heuristic values, plus the initial overhead for computing utopian and maximum points.

Such overhead is still tractable, especially taking into account that the materialization is typically decided at the design time. However, further experiments with more objectives should be performed to confirm that the times stay in tractable boundaries.

Table 1: Average time with different weighted objectives

CF_{QT}	CF_{LT}	CF_S	CF_{MD}	Average Time (s)
100%	0%	0%	0%	3,80
0%	100%	0%	0%	3,16
0%	0%	100%	0%	2,50
0%	0%	0%	100%	1,82
50%	50%	0%	0%	2,66
0%	50%	50%	0%	2,77
50%	0%	50%	0%	2,70
50%	0%	0%	50%	2,94
0%	50%	0%	50%	4,26
0%	0%	50%	50%	3,26
33%	33%	33%	0%	2,59
33%	33%	0%	33%	3,06
33%	0%	33%	33%	3,21
0%	33%	33%	33%	3,44
25%	25%	25%	25%	6,82

5.3 Obtained Solutions

As shown in section 4, our approach relies on repeatedly generating random initial states and executing the hill-climbing algorithm. Here we show an overview of the obtained solutions for different combinations of objectives. In this subsection, we aim to study the quality of the solutions obtained and how the number of iterations impact on the quality of such solutions. In this subsection, we focus on the combination of 4 objectives all equally weighted to 25%.

Figure 3 depicts in the form of stacked barcharts the resulting DW configurations C. For each batch of i iterations (x-axis) we count the frequency of resulting configurations (y-axis), where each color depicts one solution. From such figure, we can see that as the number of iterations increases few solutions tend to appear most of the times, however we cannot conclude that the most appearing solution is the optimal one.

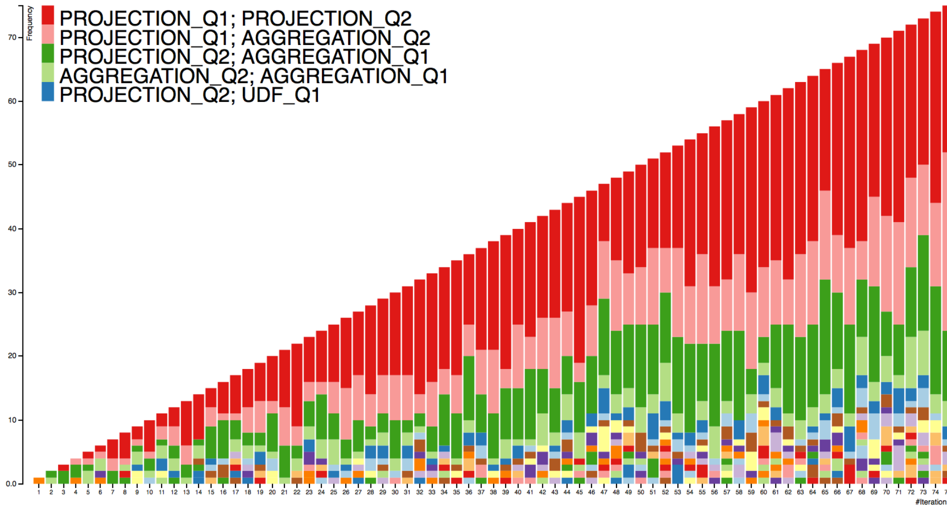


Fig. 3: Solutions obtained across 75 iterations for $CF_{QT} = 25\%$, $CF_{LT} = 25\%$, $CF_S = 25\%$, $CF_{MD} = 25\%$

In order to study the evolution of the optimal configuration, i.e., that with the lowest heuristic value with respect to the number of iterations, we depict the set of solutions

$C_1, \ldots, C_n$ ordered by heuristic value, i.e., C_1 being the optimal. Then, equation (1) measures the probability to obtain solution at position K $(1 \leq K \leq n)$ running i iterations.

$$P(K,i) = P(K,i-1) \cdot P(K,1) + P(K,i-1) \sum_{j=K+1}^{n} P(j,1) + P(K,1) \sum_{j=K+1}^{n} P(j,i-1) \quad (1)$$

Such formula, which can be easily implemented using dynamic programming techniques, considers the results obtained in previous iterations and combines them taking into account that after each iteration the solution with the lowest heuristic value is kept. Note that such formula is recursive, and therefore it needs a base case, i.e., $i = 1$. It is defined as the probability to obtain the solution at position K running 1 iterations, it is computed as the frequency of K over the total, as depicted in equation (2).

$$P(K,1) = \frac{freq(K)}{\sum_{j=1}^{n} freq(j)} \quad (2)$$

In table 2 we show, for the complete set of 15 solutions (obtained across 2850 executions), the probabilities to obtain each configuration C_K ordered by heuristic value $h(C_K)$. Note that the optimal configuration C_1 has been confirmed to be the optimal after performing a full BFS search. Furthermore, figure 4 depicts the evolution of such probabilities applying the aforementioned formula.

C	$h(C)$	$P(K,1)$	$P(K,2)$	$P(K,3)$	...	$P(K,74)$	$P(K,75)$
C_1	0,038	1.18×10^{-2}	2.35×10^{-2}	3.50×10^{-2}	...	5.85×10^{-1}	5.90×10^{-1}
C_2	0,046	4.53×10^{-2}	8.74×10^{-2}	1.27×10^{-1}	...	4.02×10^{-1}	3.98×10^{-1}
C_3	0,046	7.48×10^{-2}	1.35×10^{-1}	1.84×10^{-1}	...	1.29×10^{-2}	1.21×10^{-2}
C_4	0,049	1.06×10^{-1}	1.73×10^{-1}	2.12×10^{-1}	...	2.85×10^{-5}	2.47×10^{-5}
C_5	0,051	2.95×10^{-2}	4.41×10^{-2}	4.94×10^{-2}	...	1.71×10^{-9}	1.30×10^{-9}
C_6	0,052	5.91×10^{-3}	8.61×10^{-3}	9.42×10^{-3}	...	4.37×10^{-11}	3.23×10^{-11}
C_7	0,053	2.60×10^{-1}	3.10×10^{-1}	2.82×10^{-1}	...	5.32×10^{-11}	3.87×10^{-11}
C_8	0,053	3.74×10^{-2}	3.35×10^{-2}	2.25×10^{-2}	...	3.14×10^{-25}	1.46×10^{-25}
C_9	0,053	3.46×10^{-1}	1.77×10^{-1}	7.85×10^{-2}	...	6.48×10^{-28}	2.78×10^{-28}
C_{10}	0,059	4.33×10^{-2}	5.29×10^{-3}	5.04×10^{-4}	...	7.70×10^{-81}	6.37×10^{-82}
C_{11}	0,059	1.97×10^{-2}	1.16×10^{-3}	5.34×10^{-5}	...	1.10×10^{-104}	4.34×10^{-106}
C_{12}	0,06	3.94×10^{-3}	1.40×10^{-4}	3.72×10^{-6}	...	5.84×10^{-127}	1.15×10^{-128}
C_{13}	0,063	1.97×10^{-3}	5.81×10^{-5}	1.29×10^{-6}	...	3.93×10^{-134}	6.19×10^{-136}
C_{14}	0,065	3.94×10^{-3}	9.30×10^{-5}	1.66×10^{-6}	...	2.01×10^{-138}	2.77×10^{-140}
C_{15}	0,07	9.84×10^{-3}	9.69×10^{-5}	9.53×10^{-7}	...	3.09×10^{-149}	3.04×10^{-151}
$\sum$	-	1	1	1	...	1	1

Table 2: Probability to obtain each solution C_k per number of iterations. C_1 depicts the optimal configuration.

From such results, we conclude that the problem of finding optimal solutions 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 (see section 4.2). However, the approach of shotgun hill-climbing, i.e., with restarts, quickly yields near-optimal results after few iterations (i.e., around 10) with high probability, and furthermore, as shown in table 4 with about 50 iterations with 50% of probability, the optimal solution will be obtained. This is clearly, due to the fact that in every iteration the one with the lowest value is kept, so configurations with high probability to appear with few iterations (see C_7 in table 2) rapidly have a decrease in such probability.

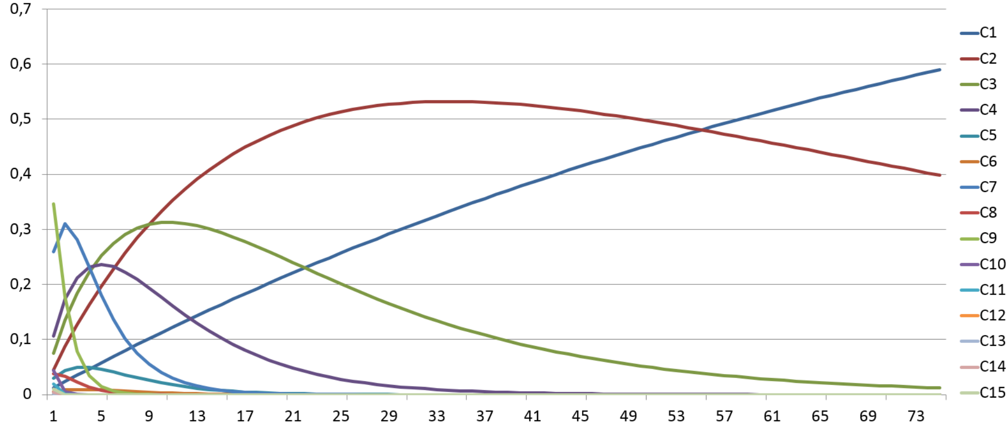


Fig. 4: Evolution of probabilities per number of iterations for each different configuration

Finally, we aim to study the evolution of the error associated to the probability $P(K, i)$, i.e., $E(K, i)$ as defined in equation (3). The base case is computed as the normalized error for each configuration C_K. Note that C_w depicts the worst possible configuration, i.e., that with the greatest heuristic value, obtained with a full BFS search.

$$E(K, i) = \begin{cases} \dfrac{h(C_K) - h(C_1)}{h(C_w) - h(C_1)}, & i = 1 \\ \\ P(K, i) \cdot E(K, 1), & \textit{otherwise} \end{cases} \tag{3}$$

Figure 5, in blue and left axis, depicts the evolution of $E(K, i)$. In green and right axis it is depicted the average error obtained calculating the heuristic of random configurations, i.e., average hill-depth, as before with respect to the worst configuration C_w. We can conclude that our approach improves the solution in one order of magnitude to that randomly obtained, and that the error decreases exponentially w.r.t. i. To improve such validation we should implement some existing approach, e.g., the one in [17] and make the comparison w.r.t. their associated heuristics instead of $h(C_w)$.

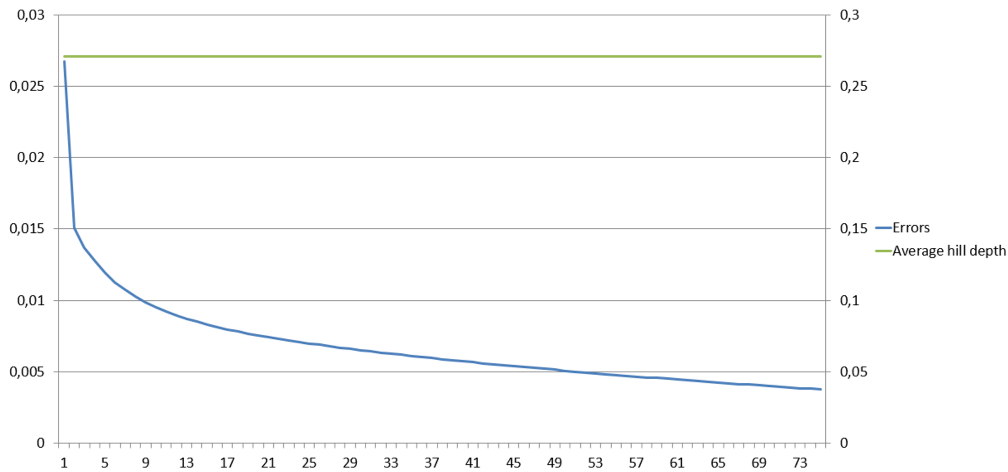


Fig. 5: Evolution of total error per number of iteration

6 Related Work [1]

The materialized view selection problem has been widely studied in the context of data warehouses. According to the survey in [32], most view selection methods follow the approach of balancing the trade-off between query processing and view maintenance cost. Additionally, and even though the problem has been extensively studied, most approaches rely on few frameworks, namely multiquery DAG, syntactical analysis of the workload or query rewriting.

Approaches based on multiquery DAGs are the most popular and they can be categorized in:

- *Multiquery AND/OR DAGs*: introduced in [42] as *expression DAGs*, all possible execution plans are coded in the form of an AND/OR DAG.
- *Multi-view processing plan (MVPP)*: proposed in [58], a MVPP is very similar to a multiquery AND/OR DAG but restricted to have only operational nodes in the intermediate nodes
- *Data cube lattice*: proposed in [17], this approach relies on dimensional attributes and OLAP queries. Each node in the DAG represents a particular group-by set in the roll-up lattice.

Approaches based on query rewriting do not start from a multiquery DAG but from the query definitions. Conversion rules might be applied to the queries in order to find common subexpressions [51]. More complex approaches do exist, such as: syntactical analysis of the workload [3], which analyzes candidate materialized views on basis of the workload in the database and interesting subsets of tables in terms of query time; [33] where a method is proposed based on resource constraints (e.g. space); or [5] where clusters of similar queries are merged to generate a set of candidate views to materialize. [30] proposes the use of multi-objective genetic algorithms (MOEAs) for the view selection problem dismissing the need of aggregating multiple objectives.

Finally, it is worth mentioning that there exist similar lines of work to ours such as [7] and [40]. The former approach is purely based on workload queries, while we benefit from more inputs, such as data-intensive flows and a reference MD schema. The latter one benefits from having as input the data flows, but focuses on a performance-oriented approach aimed to cloud environments.

7 Conclusions and Future Work

In this thesis we have presented an approach for the selection of materialized views from data-intensive flows. We have built upon the general framework for materialized view selection giving it additionally a multi-objective perspective. Moreover we have provided a set of 4 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 show that our approach is highly efficient in terms of performance, while providing near-optimal results.

As future work, we plan to explore more complex multi-objective algorithms, for instance multi-objective evolutionary algorithms as done in [30], with the goal of improving the reliability of the approach in terms of its quality. Precisely, more exhaustive theoretical and experimental validation should be addressed, for instance by comparing the proposed approach to those presented in related work (section 6) in terms of heuristic values.

We want to additionally provide a comparison study of the solutions provided by *Forge*. Comparing the results obtained in an automatical manner with those generated from domain experts, we will be able to get a subjective evaluation in the quality of the approach.

[1] Appendix C presents an extended version of this section with further research areas.

Finally, more objectives should be taken into account besides the ones presented. Economical cost models for cloud environments are presented in [26, 36], which could be integrated in our approach.

References

1. A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J.-N. Mazón, F. Naumann, T. Pedersen, S. B. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen. Fusion Cubes. *International Journal of Data Warehousing and Mining*, 9(2):66–88, 2013.
2. F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.
3. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.
4. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, pages 939–964, 2014.
5. K. Aouiche, P.-E. Jouve, and J. Darmont. Clustering-Based Materialized View Selection in Data Warehouses. (1), 2007.
6. V. R. Basili. Software modeling and measurement: the Goal/Question/Metric paradigm, 1992.
7. M. Bouzeghoub and Z. Kedad. A Quality-Based Framework for Physical Data Warehouse Design. *Management*, 2000:1–12, 2000.
8. Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *VLDB*, 5(12):1802–1813, 2012.
9. L. Chung. Non-Functional Requirements. *IEEE definition development and evaluate for the customer prior to NFRs : NFRs :*, pages 1–26, 2000.
10. L. Chung and J. C. S. do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.
11. U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data integration flows for business intelligence. In *EDBT*, pages 1–11, 2009.
12. S. Dessloch, M. a. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating schema mapping and ETL. *Proceedings - International Conference on Data Engineering*, 00:1307–1316, 2008.
13. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.
14. O. Grodzevich and O. Romanko. Normalization and other topics in multi-objective optimization. In *FMIPW*, pages 89–101, 2006.
15. H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17:24–43, 2005.
16. R. Halasipuram. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. (c):307–318, 2014.
17. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently, 1996.
18. J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.
19. F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, 5:1256–1267, 2012.
20. M. Jarke, M. A. Jeusfeld, C. Quix, and P. Vassiliadis. Architecture and quality in data warehouses: an extended repository approach. *Information Systems*, 24:229–253, 1999.
21. P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló. Integrating ETL processes from information requirements. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7448 LNCS:65–80, 2012.
22. P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, H. Candón, and S. Nadal. Quarry: Digging Up the Gems of Your Data Treasury. pages 549–552, 2015.
23. P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, H. Candón, and S. Nadal. Quarry: Digging up the gems of your data treasury. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 549–552, 2015.

24. P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, and D. Mayorova. A requirement-driven approach to the design and evolution of data warehouses. *Information Systems*, 44:94–119, 2014.
25. P. Jovanovic, A. Simitsis, and K. Wilkinson. Engine independence for logical analytic flows. *Proceedings - International Conference on Data Engineering*, pages 1060–1071, 2014.
26. Z. Karampaglis. A Bi-objective Cost Model for Database Queries in a Multi-cloud Environment.
27. G. Kougka and A. Gounaris. Declarative expression and optimization of data-intensive flows. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8057 LNCS:13–25, 2013.
28. G. Kougka and A. Gounaris. Optimization of Data-intensive Flows: Is it Needed? Is it Solved? *Proceedings of the 17th International Workshop on Data Warehousing and OLAP - DOLAP '14*, pages 95–98, 2014.
29. G. Kougka, A. Gounaris, and K. Tsichlas. Practical algorithms for execution engine selection in data flows. *Future Generation Computer Systems*, 45:133–148, 2015.
30. M. Lawrence. Multiobjective genetic algorithms for materialized view selection in OLAP data warehouses. *Proceedings of the 8th annual conference on Genetic and evolutionary computation - GECCO '06*, page 699, 2006.
31. A. Löser, F. Hueske, and V. Markl. Situational business intelligence. *Lecture Notes in Business Information Processing*, 27 LNBIP:1–11, 2009.
32. I. Mami and Z. Bellahsene. A survey of view selection methods. *ACM SIGMOD Record*, 41(1):20, 2012.
33. I. Mami, R. Coletta, and Z. Bellahsene. Modeling view selection as a constraint satisfaction problem. In *DEXA*, pages 396–410, 2011.
34. R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.
35. E. Nakuçi. Data Generation for the Simulation of Artifact-Centric Processes. (IT4BI Master Thesis), 2014.
36. T.-V.-A. Nguyen, S. Bimonte, L. D'Orazio, and J. Darmont. Cost Models for View Materialization in the Cloud. *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 47–54, 2012.
37. F. R. S. Paim and J. Castro. Enhancing Data Warehouse Design with the NFR Framework. *Proceedings of the 5th Workshop on Requirements Engineering, WER 2002*, pages 40–57, 2002.
38. G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Metrics for the Prediction of Evolution Impact in ETL Ecosystems: A Case Study. *Journal on Data Semantics*, 1:75–97, 2012.
39. M. Poess, T. Rabl, B. Caufield, and I. Datastage. TPC-DI : The First Industry Benchmark for Data Integration. 7:1367–1378, 2014.
40. W. Qu and S. Dessloch. A Real-time Materialized View Approach for Analytic Flows in Hybrid Cloud Environments. *Datenbank-Spektrum*, 14(2):97–106, 2014.
41. O. Romero, A. Simitsis, and A. Abelló. GEM: Requirement-driven generation of ETL and multidimensional conceptual designs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6862 LNCS:80–95, 2011.
42. K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, pages 447–458, 1996.
43. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition, 2003.
44. A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziovara. Benchmarking etl workflows. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5895 LNCS:199–220, 2009.
45. A. Simitsis, P. Vassiliadis, and T. Sellis. State-space optimization of ETL workflows. *IEEE Transactions on Knowledge and Data Engineering*, 17(10):1404–1419, 2005.
46. A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL processes in data warehouses. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 564–575, 2005.
47. A. Simitsis and K. Wilkinson. Revisiting ETL benchmarking: The case for hybrid flows. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7755 LNCS:75–91, 2013.

48. A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In *SIGMOD*, pages 953–960, 2009.
49. A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing ETL workflows for fault-tolerance. *Proceedings - International Conference on Data Engineering*, pages 385–396, 2010.
50. D. Theodoratos and M. Bouzeghoub. A general framework for the view selection problem for data warehouse design and evolution. *Proceedings of the 3rd ACM international workshop on Data warehousing and OLAP*, pages 1–8, 2000.
51. D. Theodoratos, S. Ligoudistianos, and T. Sellis. View selection for designing the global data warehouse. *Data and Knowledge Engineering*, 39:219–240, 2001.
52. D. Theodoratos and T. Sellis. Data warehouse configuration. *Proceedings of the Twenty-Third International Conference on Very Large Databases*, (22469):126–135, 1997.
53. D. Theodoratos and T. Sellis. Designing data warehouses. *Data and Knowledge Engineering*, 31:279–301, 1999.
54. D. Theodoratos and T. Sellis. Dynamic data warehouse design. In M. Mohania and A. Tjoa, editors, *DataWarehousing and Knowledge Discovery*, volume 1676 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 1999.
55. V. Theodorou, A. Abelló, and W. Lehner. Quality Measures for ETL Processes. In *Data Warehousing and Knowledge Discovery - 16th International Conference, DaWaK 2014, Munich, Germany, September 2-4, 2014. Proceedings*, volume 8646, pages 9–22, 2014.
56. V. Tziovara, P. Vassiliadis, and A. Simitsis. Deciding the physical implementation of ETL workflows. *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP - DOLAP '07*, page 49, 2007.
57. K. Wilkinson, A. Simitsis, M. Castellanos, and U. Dayal. Leveraging Business Process Models for ETL Design. *Business*, 6412:15–30, 2010.
58. J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *VLDB'97*, pages 136–145, 1997.

Appendix A Data-Intensive Flow Operators Cost

In this appendix we provide the complete set of statistics and metrics for data-intensive flows, introduced in section 3, considered in this thesis. The considered operators are depicted in table 3, and they are based on those presented in [35], where the formalized semantics can be found. Furthermore, throughout this appendix we assume an underlying relational engine.

Data-Intensive Flow Operators	
Aggregation	Intersect
Attribute Addition	Join
Attribute Alteration	Left Outer Join
Attribute Renaming	Projection
Cross Join	Right Outer Join
Dataset Copy	Sampling
Datatype Conversion	Sort
Difference	Union
Duplicate Removal	Union All
Filter	-

Table 3: Considered data-intensive flow operators

The statistics and metrics presented in the following subsections are inspired by [13]. The used notation is akin to the relational algebra, although it has been extended for the non present operators. Note that initially the statistics must be retrieved from each input source, however those will be propagated after the evaluation of each operator's semantics in a topological order manner. Intermediate operators' input are depicted with capital letters, i.e. R, while outputs are depicted with a primed letter, i.e. R'. Lower capitals depict attributes within the schema of R, i.e. $a_1, \ldots, a_n$.

A.1 Statistics

As described in section 3.1, statistics are properties of the data and do not depend on the underlying execution engine where the operator is being executed. The following list presents the required statistics on which the calculation of metrics rely on.

- $V(R.a)$: number of distinct values for attribute a in relation R.
- $T(R)$: cardinality of relation R, i.e. the number of tuples.
- $sel_P(R)$: selectivity factor of filter predicate P in relation R.

Distinct values is generally estimated as the proportion of the number of tuples that the selectivity factor specifided by the operator sel_p reduces. As presented in example 6, it is measured as $V(R'.att_i) = V(R.att_i) \cdot (1 - sel_P)^{\frac{T(R)}{V(R.att_i)}}$

For the sake of simplicity, cardinality of relations per operator is depicted in subsection A.2. It is only required to consider such values without applying the function that performs the conversion to blocks. The following subsection presents the details on how selectivity factors are computed.

Selectivity Factors The estimation of the selectivity factor of a selection will vary depending on the selection predicate. It is important to precise that the following major assumptions are made: *a*) equi-probability of values; *b*) statistical independence of attributes; and *c*) $R.a \in [min, max]$.

Additionally, as the required statistics for some cases would be too difficult to maintain, some values are provided by rule of thumb.

- $sel_{R.a=c}(R) = \dfrac{1}{V(R.a)}$
- $sel_{R.a>c}(R) = \begin{cases} 0, & \text{if } c \geq max \\ 1, & \text{if } c < min \\ \dfrac{max - c}{max - min}, & \text{otherwise} \end{cases}$
- $sel_{R.a>v}(R)$[2] $= \frac{1}{2}$
- $sel_{R.a<c}(R) = \begin{cases} 1, & \text{if } c > max \\ 0, & \text{if } c \leq min \\ \dfrac{c - min}{max - min}, & \text{otherwise} \end{cases}$
- $sel_{R.a<v}(R) = \frac{1}{2}$
- $sel_{R.a\leq c}(R) \simeq sel_{R.a<c}(R)$, assuming $V(R.a)$ is big enough.
- $sel_{R.a\geq c}(R) \simeq sel_{R.a>c}(R)$, assuming $V(R.a)$ is big enough.
- $sel_{P\wedge Q}(R) = sel_P(R) \cdot sel_Q(R)$, assuming P and Q are statistically independent.
- $sel_{P\vee Q}(R) = sel_P(R) + sel_Q(R) - sel_P(R) \cdot sel_Q(R)$, assuming P and Q are statistically independent.
- $sel_{\text{NOT } P}(R) = 1 - sel_P(R)$
- $sel_{R.a \text{ IN } (c_1,\ldots,c_n)}(R) = \min(1, \dfrac{n}{V(R.a)})$
- $sel_{R.a \text{ BETWEEN } (c_1,c_2)}(R) = \dfrac{\min(c_2, max) - \max(c_1, min)}{max - min}$
- $sel_{R.a \text{ BETWEEN } (v_1,v_2)}(R) = \frac{1}{4}$
- $sel_{R.a \text{ BETWEEN } (c_1,v_2)}(R) = \frac{1}{2} \cdot sel_{R.a>c_1}(R)$
- $sel_{R.a \text{ BETWEEN } (v_1,c_2)}(R) = \frac{1}{2} \cdot sel_{R.a<c_2}(R)$

The case of joins When computing selectivity factors for the join of two relations $R[A\theta B]S$, it is necessary to distinguish the used comparison operator:

- $sel(R \times S) = 1$
- $sel_{R.a\neq S.b}(R \bowtie S) = 1$
- $sel_{R.a=S.b}(R \bowtie S) = \begin{cases} \dfrac{1}{V(R.a)}, & \textit{if } dom(S.b) \subseteq dom(R.a) \\ \dfrac{V(R.a \cap S.b)}{V(R.a) \cdot V(S.b)}, & \textit{otherwise} \end{cases}$
- $sel_{R.a<S.b}(R \bowtie S) = \frac{1}{2}$
- $sel_{R.a\leq S.b}(R \bowtie S) = \frac{1}{2}$

A.2 Metrics

In section 3.2 we described metrics, the engine-dependent estimation of quality factors for operational nodes in a data-intensive flow. In this thesis, we focus in performance-wise ($Execution_{estimator}$) and space-wise ($Space_{estimator}$) metrics, the former measured by means

[2] Note the difference between c constant value and v variable value, i.e. coming from intermediate results or a parametrized statement.

of estimated disk I/O (in blocks) and the latter by number of disk blocks occupied by the materialized view. We thus need to keep track of the following properties of the underlying engine:

- $B(R)$: number of blocks that R occupies on disk.
- M: number of main memory buffers available.
- B: size of a disk block.

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, therefore we need to make this calculation based on the record length, that is $\sum sizeof(att_i)$ (including the corresponding control information).

From the previous definition, we define the number of blocks of an intermediate result R as $B(R) = \left\lceil \frac{T(R)}{\left\lfloor \frac{B}{\sum sizeof(att_i)} \right\rfloor} \right\rceil$

Families of Algorithms Most of the operators presented in table 3 can be implemented as a set of patterns, those are:

One-pass Algorithms When one of the operator's components can entirely fit into the memory buffers (M), then data just needs to be read once which yields linear cost for such algorithms. There is a clear restriction on the size of M which varies according to the semantics of each operator.

Sorting-based Algorithms Those algorithms in most cases rely on the same structure, this is (for relations R and S):

1. Sort R
2. Sort S
3. Iterate synchronously over R and S outputting the appropiate value depending on the semantics.

Sorting can be done by any suitable algorithm, but it is usually performed by *Multiway Merge-Sort*, sometimes referred as the *two-pass algorithm*, as the size of the generated sublists grows exponentially in each iteration, and in many cases sublists of M^2 blocks suffice. It is important to remark that such algorithms provide a sorted output, therefore this can be an additional optimization aspect on which consequent operators can leverage.

Hash-based Algorithms As for the case of sorting-based, hash-based algorithms also follow an essential structure (again for relations R and S), however as contrary to the previous algorithms, the output of hash-based algorithms is unsorted.

- Partition R in k buckets of at most $M - 1$ blocks each by means of hash function h.
- Partition S in k buckets by means of hash function h.
- Perform the *one-pass algorithm* for every bucket in R and S.

Note that index-based methods have not been taken into account, as we are mainly focusing on computing the cost for intermediate results and therefore we do not assume that such access structures exist.

Metrics per Operator Metrics per operator are categorized according to the aforementioned performance-wise or space-wise aspects. The provided formulas do not consider the worst case scenario, as those would be usually be too costly and a poor heuristic, but a conservative trade-off for the average case is used.

Aggregation ($\gamma_{a_1,\ldots,a_n,f(a_{n+1})\rightarrow b}(R)$) Groups by attributes $a_1,\ldots,a_n$ and applies function f (e.g., SUM) to attribute a_{n+1} which is renamed to b. This operator requires accessing the complete input relation and thus cannot be pipelined.

- $Execution_{estimator} = \begin{cases} B\left(T\left(R\right)\right), & \text{one-pass} \\ 3B\left(T\left(R\right)\right), & \text{sort-based} \\ 3B\left(T\left(R\right)\right), & \text{hash-based} \end{cases}$
- $Space_{estimator} = B\left(\min\left(\frac{T(R)}{2}, \prod V\left(R, a_i\right)\right)\right)$

Attribute Addition ($\mathbb{AA}_{a_1,\ldots,a_n,f(a_1,\ldots,a_n)\rightarrow b}(R)$) Adds a new attribute b by means of applying function f over a subset of the attributes in the input schema $a_1,\ldots,a_n$. Such operation modifies the schema for each tuple, therefore it can be pipelined[3].

- $Execution_{estimator} = 0$
- $Space_{estimator} = B\left(T\left(R_{a_1,\ldots,a_n,f(a_1,\ldots,a_n)}\right)\right)$

Attribute Alteration ($\mathbb{AM}_{a_1,\ldots,a_n,f(a_1,\ldots,a_n)\rightarrow a_i}(R)$) Similar to attribute addition however instead of adding to the schema a new attribute it modifies an existing one a_i.

- $Execution_{estimator} = 0$
- $Space_{estimator} = B\left(T\left(R_{a_1,\ldots,a_n}\right)\right)$

Attribute Renaming ($\rho_{a_1\rightarrow b_1,\ldots,a_n\rightarrow b_n}(R)$) Renames attributes $a_1,\ldots,a_n$ in the schema to $b_1,\ldots,b_n$. This operation works at the schema level, therefore does not need to access the input data and incurs no execution cost.

- $Execution_{estimator} = 0$
- $Space_{estimator} = B\left(T\left(R\right)\right)$

Cross Join ($R \times S$) Generates all combinations of elements by applying cartesian product over the input relations R and S. Although it incurs a very high cost in both performance and space, it is useful for instance to generate a *date* dimension by applying $Years \times Months \times Days$. This operation cannot be pipelined as it requires the complete input relations to be present.

- $Execution_{estimator} = \begin{cases} B\left(T\left(R\right)\right), & \text{one-pass} \\ B(S) + T(S) \cdot B(R), & \text{tuple-based nested loop} \\ B(S) + B(R) \cdot \lceil \frac{B(S)}{M-1} \rceil, & \text{block-based nested loop} \end{cases}$
- $Space_{estimator} = B\left(T\left(R\right) \cdot T\left(S\right)\right)$

Dataset Copy ($\mathbb{D}(R)$) Generates a duplicate dataset for input R, it can be pipelined.

- $Execution_{estimator} = 0$
- $Space_{estimator} = B\left(2T\left(R\right)\right)$

Datatype Conversion ($\mathbb{C}_{a_{1_t}\rightarrow a_{1_{t'}},\ldots,a_{n_t}\rightarrow a_{n_{t'}}}(R)$) Modifies the datatype for input attributes $a_1,\ldots,a_n$. This operation works at the schema level, therefore does not need to access the input data and incurs no execution cost.

- $Execution_{estimator} = 0$
- $Space_{estimator} = B\left(T\left(R_{a_{1_{t'}},\ldots,a_{n_{t'}}}\right)\right)$

[3] Remember that pipeline operators are considered to have $Execution_{estimator} = 0$

Difference $(R \setminus S)$ Performs the set difference for input relations R and S.

- $Execution_{estimator}$[4]
  - One-pass: $B\left(T\left(R\right)+\left(S\right)\right)$
  - Sort-based: $2B\left(T\left(R\right)\right)\lceil log_M B\left(T\left(R\right)\right)\rceil + 2B\left(T\left(S\right)\right)\lceil log_M B\left(T\left(S\right)\right)\rceil + B\left(T\left(R\right)\right) + B\left(T\left(S\right)\right)$
  - Hash-based: $2B(T(R))\lceil log_{M-1} B(T(R))-1\rceil + 2B(T(S))\lceil log_{M-1} B(R)-1\rceil + B(R) + B(S)$
- $Space_{estimator} = B\left(T(R) - sel_{r=s}(R) \cdot T(R) \cdot T(S)\right)$

Duplicate Removal $(\delta(R_{a_1,\ldots,a_n}))$ Eliminates all duplicate tuples by means of grouping attributes $a_1, \ldots, a_n$. Its implementation is the same as for the *Aggregation* operation.

- $Execution_{estimator} = \begin{cases} B\left(T\left(R\right)\right), & \text{one-pass} \\ 3B\left(T\left(R\right)\right), & \text{sort-based} \\ 3B\left(T\left(R\right)\right), & \text{hash-based} \end{cases}$
- $Space_{estimator} = B\left(\min\left(\frac{T(R)}{2}, \prod V\left(R, a_i\right)\right)\right)$

Filter $(\sigma_P(R))$ Outputs a subset of tuples from R, precisely those that satisfy the selection predicate P.

- $Execution_{estimator} = B(T(R))$
- $Space_{estimator} = B(sel_P(R) \cdot T(R))$

Intersect $(R \cap S)$ Performs the set intersection for input relations R and S.

- $Execution_{estimator}$
  - One-pass: $B\left(T\left(R\right)+\left(T(S)\right)\right)$
  - Sort-based: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil + B(T(R)) + B(T(S))$
  - Hash-based: $2B(T(R))\lceil log_{M-1} B(T(R)) - 1\rceil + 2B(T(S))\lceil log_{M-1} B(T(R)) - 1\rceil + B(T(R)) + B(T(S))$
- $Space_{estimator} = B\left(\frac{\max(T(R), T(S))}{2}\right)$

Join $(R \bowtie_P S)$ Natural join of the two relations R and S, which outputs those tuples where both relations agree by means of selection predicate P.

- $Execution_{estimator}$
  - One-pass: $B(T(R)) + B(T(S))$
  - Tuple-based Nested Loop: $B(T(S)) + T(S) \cdot B(T(R))$
  - Block-based Nested Loop: $B(T(S)) + B(T(R)) \cdot \lceil \frac{B(T(S))}{M-1} \rceil$
  - Sort-merge join: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil - B(T(R)) - B(T(S))$
  - Hash-Join: $2B(T(R))\lceil log_{M-1} B(T(S))-1\rceil + 2B(T(S))\lceil log_{M-1} B(T(S))-1\rceil + B(T(R)) + B(T(S))$
- $Space_{estimator} = B\left(sel_P(R \bowtie S) \cdot T(R) \cdot T(S)\right)$

[4] For the sake of readability, format has been changed in the case of long formulas

Left Outer Join ($R ⟕ S$) Natural join of the two relations R and S, which outputs those tuples from R where both relations agree by means of selection predicate P.

– $Execution_{estimator}$

- One-pass: $B(T(R)) + B(T(S))$
- Tuple-based Nested Loop: $B(T(S)) + T(S) \cdot B(T(R))$
- Block-based Nested Loop: $B(T(S)) + B(T(R)) \cdot \lceil \frac{B(T(S))}{M-1} \rceil$
- Sort-merge join: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil - B(T(R)) - B(T(S))$
- Hash-Join: $2B(T(R))\lceil log_{M-1} B(T(S)) - 1\rceil + 2B(T(S))\lceil log_{M-1} B(T(S)) - 1\rceil + B(T(R)) + B(T(S))$

– $Space_{estimator} = B\left(sel_P(R \bowtie S) \cdot T(R)\right)$

Projection ($\pi_{a_1,\ldots,a_n}(R)$) Outputs only the attibutes specified in $a_1, \ldots, a_n$. It can be pipelined.

– $Execution_{estimator} = 0$
– $Space_{estimator} = B\left(T\left(\pi_{a_1,\ldots,a_m}\left(R\right)\right)\right)$

Right Outer Join ($R ⟖ S$) Natural join of the two relations R and S, which outputs those tuples from S where both relations agree by means of selection predicate P.

– $Execution_{estimator}$

- One-pass: $B(T(R)) + B(T(S))$
- Tuple-based Nested Loop: $B(T(S)) + T(S) \cdot B(T(R))$
- Block-based Nested Loop: $B(T(S)) + B(T(R)) \cdot \lceil \frac{B(T(S))}{M-1} \rceil$
- Sort-merge join: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil - B(T(R)) - B(T(S))$
- Hash-Join: $2B(T(R))\lceil log_{M-1} B(T(S)) - 1\rceil + 2B(T(S))\lceil log_{M-1} B(T(S)) - 1\rceil + B(T(R)) + B(T(S))$

– $Space_{estimator} = B\left(sel_P(R \bowtie S) \cdot T(S)\right)$

Sampling $s_{\%}(R)$ Reduces the input relation R to the specified percentage by means of sampling techniques. Streaming sampling techniques can be applied here such as sticky sampling and thus it acts as a pipeline.

– $Execution_{estimator} = 0$
– $Space_{estimator} = B\left(\%T(R)\right)$

Sort ($\tau(R_{a_1,\ldots,a_n})$) Sorts relation of R by means of attributes $a_1, \ldots, a_n$. It requires the complete relation as input and thus it cannot be pipelined.

– $Execution_{estimator} = \begin{cases} B\left(T\left(R\right)\right), & \text{one-pass} \\ 3B\left(T\left(R\right)\right), & \text{sort-based} \end{cases}$
– $Space_{estimator} = B(T(R))$

Union $(R \cup_S S)$ Set-based union (i.e., without repetitions) for input relations R and S.

– $Execution_{estimator}$
  - One-pass: $B(T(R)) + B(T(S))$
  - Sort-based: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil + B(T(R)) + B(T(S))$
  - Hash-based: $2B(T(R))\lceil log_{M-1} B(T(R)) - 1\rceil + 2B(T(S))\lceil log_{M-1} B(T(R)) - 1\rceil + B(T(R)) + B(T(S))$

– $Space_{estimator} = B\left(\max\left(T\left(R\right), T\left(S\right)\right) + \frac{\min\left(T\left(R\right), T\left(S\right)\right)}{2}\right)$

Union All $(R \cup S)$ Bag-based union (i.e., with repetitions) for input relations R and S.

– $Execution_{estimator}$
  - One-pass: $B(T(R)) + B(T(S))$
  - Sort-based: $2B(T(R))\lceil log_M B(T(R))\rceil + 2B(T(S))\lceil log_M B(T(S))\rceil + B(T(R)) + B(T(S))$
  - Hash-based: $2B(T(R))\lceil log_{M-1} B(T(R)) - 1\rceil + 2B(T(S))\lceil log_{M-1} B(T(R)) - 1\rceil + B(T(R)) + B(T(S))$

– $Space_{estimator} = B(T(R)) + B(T(S))$

Appendix B Forge in Quarry

B.1 Introduction to Quarry

Next generation BI systems (or self-service BI) [1] are devoted to enable non-expert users to search, extract and integrate situational data [31], all this achieved via a direct interaction with the system, dismissing the need of additional support or intervention. An example of such system is Quarry, a self-service BI platform developed by the MPI research group at UPC. The core of Quarry is a semi-automatic tool that for a set of input information requirements generates a multidimensional (MD) schema and an ETL process design (data-intensive flow). This thesis revolves around the Quarry platform and thus, Forge has been integrated as a new module of it.

Besides implementing Forge, the beginning of this thesis was devoted to design and implement a robust and extensible architecture for Quarry, by making it service-oriented, see figure 7. Besides that, the front-end had to be completely rewritten which helped to complete the work presented in [23].

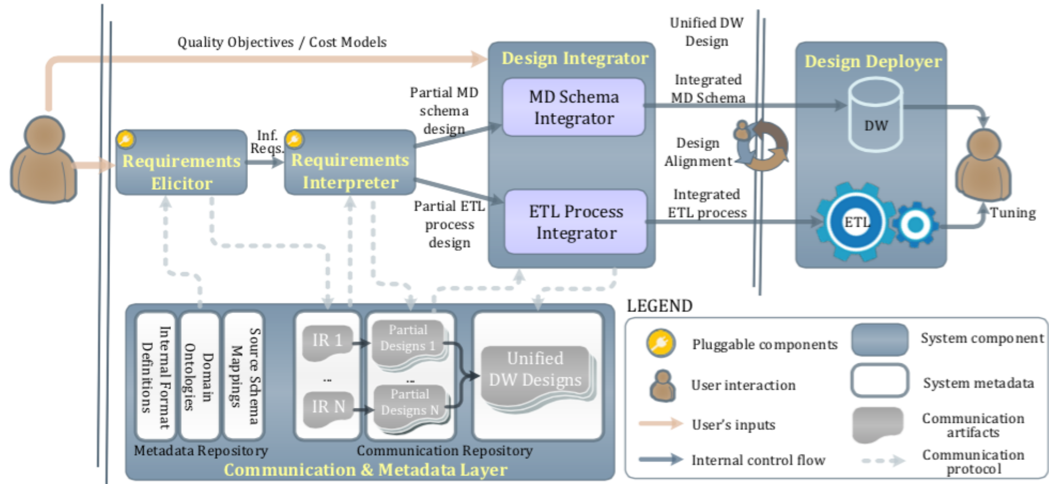


Fig. 6: High level overview of Quarry

Taking a look into the internals of the Quarry platform, depicted in figure 6, the first module we encounter is *Requirements Elicitor*, i.e., MineDigger. That is a high-level GUI that enables user interaction with the platform. User input is appropriately converted to information requirements and sent to *Requirement Interpreter*, i.e., GEM [41]. Its input is a set of data sources, whose semantics are captured by means of an OWL ontology, and a set of information requirements expressed in *xRQ*, a proprietary XML-based format. Each of the aforementioned requirements is processed individually in order to generate a MD schema and a data-intensive flow flow design that will satisfy the requirements at hand.

The outputs of GEM are further sent to the inputs of *Design Integrator* modules, i.e., ORE [24] and CoAl [21]. *MD Schema Integrator*, i.e., ORE, in an incremental and iterative way, generates a unified MD schema design from the set of individual MD schema designs resulted from the aforementioned (GEM) module, providing answers for each of the information requirements specified. ORE is configurable to use different non-functional requirements to drive the unified MD schema design, and as mentioned in [24], minimizing the structural complexity is the only one being used so far.

On the other hand, *ETL Process Integrator*, i.e., CoAl, is ORE's counterpart for data-intensive flow integration. Starting from individual flows, one for each information requirement, CoAl consolidates these, as before in an incremental and iterative way, into a unified

flow that answers the current set of information requirements. The consolidation process is driven by the non-functional requirement of minimizing the overall execution time of the unified flow. This is achieved by maximizing overlapping among input flows, by detecting potential duplicates in data sources and operations.

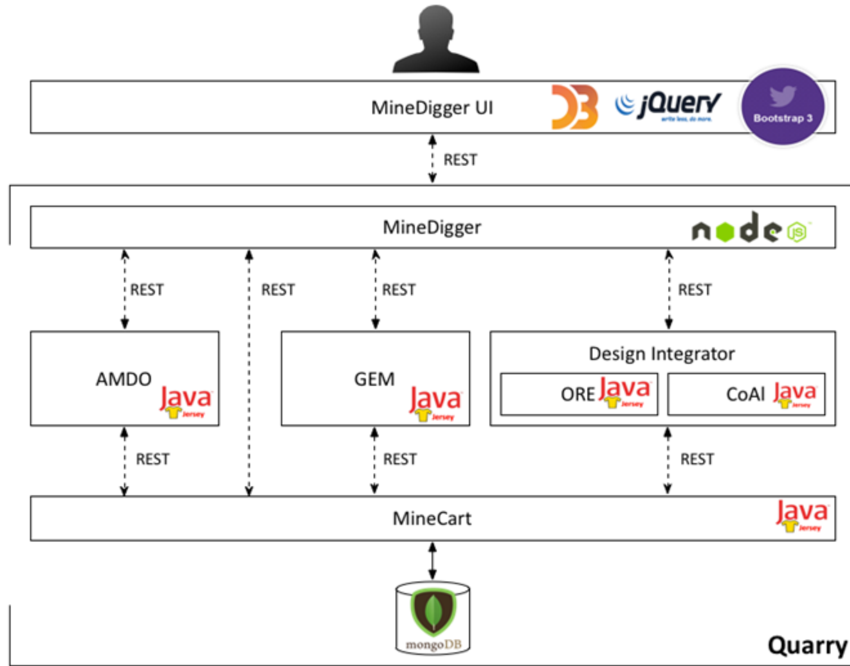


Fig. 7: Service-oriented architecture of Quarry with technological details

B.2 Forge: Implementation details

Forge is a mid-size project with a total of 52 Java classes. Figure 8 depicts a simplified version of the UML class diagram where the most important classes are shown, note they are clustered by type.

However, in order to simplify coding we reuse on some existing libraries throughout the project, some of the most used and relevant are:

- JGraphT[5], a graph library that provides mathematical graph-theory objects and algorithms
- ETLFlowGraph, developed as part of the work in [25], it enhances the graph structure to convert it to an ETL graph with all components presented in section 2.2.
- AIMA[6], which provides implementation for all algorithms described in the book *Artificial Intelligence - A Modern Approach* [43].
- Google Guava[7], several of Google's core libraries. Mostly used for complex data structures and set operations.

Besides selecting a DW configuration C, within Quarry the task of Forge is to deploy a DW solution and to divide the input data-intensive flow into 2 parts, loading and query

[5] http://jgrapht.org/
[6] http://aima.cs.berkeley.edu/
[7] https://github.com/google/guava

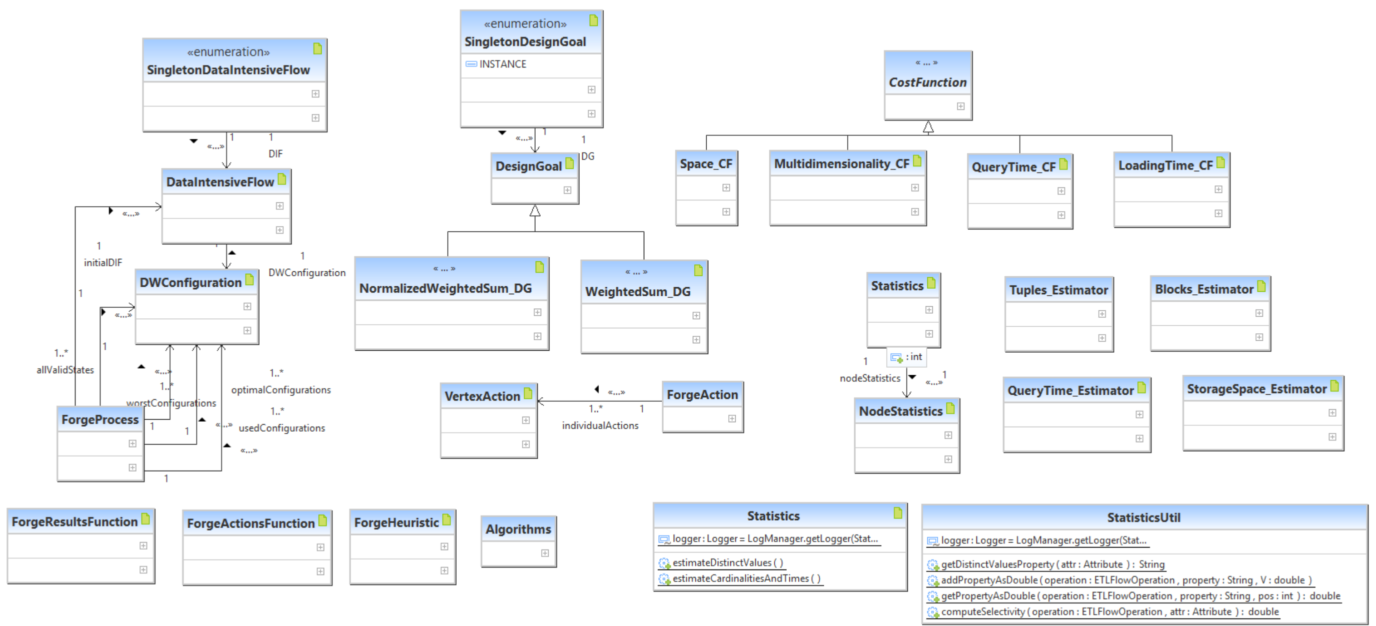


Fig. 8: Simplified UML class diagram of Forge

parts. Deploying a data warehouse is achieved by creating tables in a database with the output schema of all selected nodes in the DW configuration.

Once the schema of the data warehouse is created, nodes to insert the output of the selected nodes to the tables are created. On the other hand, a separate DIF is created, which reads the inputs from the populated tables and continues the logics of the original DIF.

Appendix C Extended Related Work

In section 6 we presented related work on materialized view selection, however this work additionally deals with two major research areas, namely: measurement non-functional requirements in data warehouses and estimation of ETL operators cost. In this appendix we present related work concerning those two areas.

C.1 Measuring Non-functional Requirements in Data Warehouses

Sometimes referred as *quality factors*, non-functional requirements have been extensively studied in software engineering [9]. Probably, one of the most applied techniques to measure such quality factors is the goal-question-metric method (GQM) [6]. This method is based on the fact that quality *goals* are usually hard to assess, however it is possible to validate those by answering a set of *questions*. In a similar manner, such quality questions are hard to be directly assessed, but they can be measured by a set of *metrics* which can quantify their degree of compliance.

The work in [20] defines a broad set of quality dimensions for data warehouses relying on the GQM method. Additionally, a meta model and architecture is proposed where queries representing quality measurements can be issued. In [37], the NFR framework [10] is adopted and extended with metrics for data warehouses. This framework specifies non-functional requirements as a graph of softgoals that can be conflicting or harmonious.

On the other hand, as previously said, ETL processes are the most error-prone and time-consuming tasks, which some estimations quantify in 70%. Therefore, this fact has led extensive research in quality-oriented design of ETL processes. [48] proposes a set of quality metrics, named QoX, which can be tracked throughout the design process. Such metrics are used in [44] in order to define a generic benchmark for ETL processes. In [55], a clear set of quantifiable metrics are defined for such quality factors. Further works focus on optimizing some particular requirements such as [49], where they target performance, fault-tolerancy and data freshness; or [38] for maintenance and evolution.

C.2 Estimating ETL Operators Cost

Optimization in ETL processes has been usually pushed down to the underlying DBMS in order to take advantage of the extensive optimization techniques that exist for such systems. It is clear though that the elements contained in an ETL process are not restricted to those offered in DBMS, i.e. those inherited from the relational algebra [11]. Therefore, this highlights the need for specific optimization techniques tailored to such processes. Many works [45,46,49] rely on the reordering of the ETL workflow, for instance to move restrictive operations such as selections or aggregations in the early stages of the process. However, most of those works overlook the problem of estimating the cost of each operator and tend to follow a black-box approach.

In [46], the problem of optimizing ETL workflows is tackled with a simple cost model where only the number of processed rows is considered. Extending the work, [49] proposes a set of cost functions for the complete ETL design, where each represents an ETL quality metric. The cost of each operation is calculated by means of a generic function paremetrized with the number of input and output tuples. Recent works are moving towards a cost-based approach for optimizing ETL workflows [27–29], albeit it is not clear the used cost function per operator, or they rely on its previous execution to infer the that cost.

In the past, different approaches dealing with ETL optimization [12,56] have considered UDFs as black boxes, as they support arbitrary code written in any programming or database

query language. That clearly yields difficulties when evaluating its complexity. However, recent systems [4] are considering UDFs are first class citizens in order to broaden their processing capabilities. Some works [19] are trying to tackle the issue estimating the cost of UDFs by means of static code analysis techniques. In [18] a method for calculating the cost of a UDF function is proposed by means of a recursive formula, it evaluates and aggregates different CPU usage parameters for the tree representation of the UDF.