Incremental Consolidation of Data-Intensive Multi-flows

Petar Jovanovic, Oscar Romero, Alkis Simitsis, and Alberto Abelló

Abstract—Business intelligence (BI) systems depend on efficient integration of disparate and often heterogeneous data. The integration of data is governed by *data-intensive* flows and is driven by a set of information requirements. Designing such flows is in general a complex process, which due to the complexity of business environments is hard to be done manually. In this paper, we deal with the challenge of efficient design and maintenance of *data-intensive* flows and propose an incremental approach, namely *CoAl*, for semi-automatically consolidating data-intensive flows satisfying a given set of information requirements. *CoAl* works at the logical level and consolidates data flows from either high-level information requirements or platform-specific programs. As *CoAl* integrates a new data flow, it opts for maximal reuse of existing flows and applies a customizable cost model tuned for minimizing the overall cost of a unified solution. We demonstrate the efficiency and effectiveness of our approach through an experimental evaluation using our implemented prototype.

Index Terms—Business Intelligence, data-intensive flows, workflow management, data warehousing

1 Introduction

The complexity of business environments constantly grows, both with regard to the amount of data relevant for making strategic decisions and the complexity of included business processes. Today's dynamic and competitive markets often imply rapid (e.g., near real-time) and accurate decision making. Relevant data are stored across a variety of data repositories, possibly using different data models and formats, and potentially crossed with numerous external sources for various context-aware analysis. A data integration process combines data residing on different sources and provides a unified view of this data for a user [1]. For example, in a data warehousing (DW) context, data integration is implemented through extract-transform-load (ETL) processes. Generally, an ETL process represents a data-intensive flow (or simply, data flow) that extracts, cleans, and transforms data from multiple, often heterogeneous data sources and finally, delivers data for further analysis.

There are various challenges related to data flow design. Here we consider two: *design evolution* and *design complexity*.

A major challenge that BI decision-makers face relates to the *evolution* of business requirements. These changes are more frequent at the early stages of a DW design project [2] and in part, this is due to a growing use of agile methodologies in data flow design and BI systems in general [3]. But changes may happen during the entire DW lifecycle. Having an up-and-running DW system satisfying an initial set of requirements is still a subject to various changes as the business evolves. The data flows populating a DW, as other software artifacts, do not lend themselves nicely to evolution events and in general, due to their complexity, maintaining them manually is hard. The situation is even more critical in today's BI settings, where on-the-fly decision making requires faster and more efficient adapting to changes. Changes in business needs may result in new, changed or removed information requirements. Thus having an incremental and agile solution that can automatically absorb occurred changes and produce a flow satisfying the complete set of requirements would largely facilitate the design and maintenance of data-intensive flows.

In an enterprise environment data is usually shared among users with varying technical skills and needs, involved in different parts of a business process. Typical real-world data-intensive workloads have high temporal locality, having 80% of data reused in a range from minutes to hours [4]. However, the cost of accessing these data, especially in distributed scenarios, is often high [5]. At the same time, intertwined business processes may also imply overlapping of data processing. For instance, a sales department may analyze the *revenue* of the sales for the past year, while finance may be interested in the overall *net profit*. Computing the *net profit* can largely benefit from the total *revenue* already computed for the sales department and thus, it could benefit from the sales data flow too. The concept of reusing partial results is not new. Software and data reuse scenarios in data integration have been proposed in the past, showing that such reuse would result in substantial cost savings, especially for large, complex business environments [6]. Data flow reuse could result in a significant reduce in design complexity, but also in intermediate flow executions and thus, in total execution time too [5].

- *P. Jovanovic, O. Romero, and A. Abelló are with the Department of Service and Information System Engineering, Universitat Politècnica de Catalunya (BarcelonaTech), Barcelona, Spain. E-mail: {petar, oromero, aabello}@essi.upc.edu*
- *A. Simitsis is with HP Labs, Palo Alto, CA, USA. E-mail: alkis@hp.com*

In this paper, we address these challenges and present an approach to efficient, incremental consolidation of data-intensive flows. Following common practice, our method iterates over information requirements to create the final design. In doing that, we show how to efficiently accommodate a new information requirement to an existing design and also, how to update a design in lieu of an evolving information requirement. To this end, we describe a **Co**nsolidation **Al**gorithm (*CoAl*) for data-intensive flows. Without loss of generality, we assume that starting with a set of information requirements, we create a data flow per requirement. The final design satisfying all requirements comprises a *multi-flow*. As 'coal' is formed after the process and extreme compaction of layers of partially decomposed materials[1], *CoAl* processes individual data flows and incrementally consolidates them into a unified multi-flow.

CoAl deals with *design evolution* by providing designers with an agile solution for the design of data flows. *CoAl* assists the early stages of the design process when for only a few requirements we need to build a running data flow from scratch. But, it also helps during the entire flow lifecycle when the existing multi-flow must be efficiently accommodated to satisfy new, removed, or changed information requirements.

CoAl reduces *design complexity* with aggressive information and software reuse. Per requirement, it searches for the largest data and operation overlap in the existing data flow design. To boost the reuse of existing design elements when trying to satisfy a new information requirement (i.e., when integrating a new data flow), *CoAl* aligns the order of data flow operations by applying generic equivalence rules. Note that in the context of data integration, the reuse of both data and code (e.g., having a single computation shared by multiple flows as depicted in Figure 4) besides reducing flow complexity, might also lead to faster execution, better resource usage, and higher data quality and consistency [6], [7].

In addition, since data-intensive flows comprise critical processes in today's BI systems, *CoAl* accounts for the cost of produced data flows when searching for opportunities to integrate new data flows. *CoAl* uses a tunable cost model to perform multi-flow, logical optimization to create a unified flow design that satisfies all information requirements. Here, we focus on maximizing the reuse of data and operations, but the algorithm can be configured to work with different cost models, taking into account different quality factors of data flows (e.g., [8]).

As a final remark, *CoAl* works at the logical level and is therefore applicable to a variety of approaches that generate logical data flows from information requirements expressed either as high level business objects (e.g., [9], [10], [11]) or in engine specific languages (e.g., [12]).

In particular, our main contributions are as follows.

- We present a semi-automatic approach to the incremental design of data-intensive flows.
- We introduce a novel consolidation algorithm, called *CoAl*, that tackles the data flow integration problem from the context of data and code reuse, while at the same time taking into account the cost of the produced data flow design.
- We present generic methods for reordering and comparing data flow operations that are applied while searching for the consolidation solutions that will increase data and operation reuse.
- We experimentally evaluate our approach by using an implemented prototype. A set of empirical tests have been performed to assess the *CoAl*'s efficiency and improvements in overall execution time.

A short version of this paper was published in Jovanovic et al. [13].

Outline. Section 2 describes a running example and formalizes the problem at hand. Section 3 discusses the main challenges: operations reordering and comparison. Section 4 presents the *CoAl* algorithm. Section 5 reports on our experimental findings. Sections 6 and 7 discuss related work and conclude the paper, respectively.

2 Overview

2.1 Running Example

Figure 1 shows an abstraction of the TPC-H schema [14]. Figure 2 illustrates four example information requirements extracted from TPC-H queries. In a sense, our example here is adapted by reverse engineering the use case described by the TPC-H schema and queries.

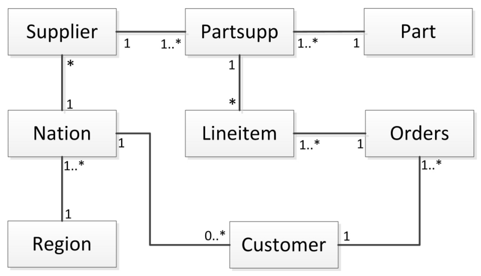

Figure 1. TPC-H Schema

We create a data flow per each requirement in Figure 2 (see Figures 3 and 6). In the literature, there are many methods dealing with such task, either manually (e.g., [9], [10]) or automatically (e.g., [11]). Independent of a method for creating data flows from single requirements, *CoAl* focuses on the problem of integrating these flows into a unified flow that satisfies all requirements.

Consider the DIF-1 and DIF-2 data flows depicted in Figure 3 that satisfy the requirements IR_1 and IR_2, respectively. We define the *referent* data flow as the integrated multi-flow satisfying a number of requirements already modeled (we start from IR_1) and the *new* data flow as the flow satisfying the new requirement (IR_2).

In terms of graphical notation, the squares represent source or target data stores, whereas the circles represent data flow operations. Operations of each data flow are uniquely named using the following notation: `OPNAME` =

1. src. Wikipedia

IR_1: Revenue of the sales for the parts ordered in the past year, per quarter.
IR_2: Net profit of the sales for the parts ordered in the last year, per quarter.
IR_3: Top 10 automobile industry customers based on the quantity of shipped parts, ordered in the last year.
IR_4: Sorted list of quantities of parts shipped to Spanish customers, ordered in the last year.

Figure 2. Information Requirements

`OPTYPE`+`OPID`+";"+{`FLOWIDs`}. Note that the flow IDs at the end define the set of data flows that share the given operation. They are optional and can be omitted for single (non integrated) data flows.

Observe that these two data flows have a number of common operations. *CoAl* exploits this and creates an alternative, equivalent data flow satisfying both requirements IR_1 and IR_2, such that the reuse of the data stores and operations of DIF-1 is maximal (see Figure 4). We then continue and integrate the remaining requirements to create a unified multi-flow that satisfies all four requirements (i.e., IR_1-IR_4), see Figure 7.

2.2 Preliminaries and Notation

We build upon past work on ETL workflow formalization [15] and model generic data flows as follows. A data-intensive flow (DIF) is formally defined as a directed acyclic graph consisting of a set of nodes ($\mathbf{V}$), which are either data stores ($\mathbf{DS}$) or operations ($\mathbf{O}$), while the graph edges ($\mathbf{E}$) represent a directed data flow among the nodes of the graph ($v_1 \prec v_2$). We write:

$DIF = (\mathbf{V}, \mathbf{E})$, such that: $\mathbf{V} = \mathbf{DS} \cup \mathbf{O}$,
$\forall e \in \mathbf{E} : \exists (v_1, v_2), v_1 \in \mathbf{V} \wedge v_2 \in \mathbf{V} \wedge v_1 \prec v_2$

In the rest of the paper we use the terms 'flow' and 'graph' interchangeably, while the same formalization of a data flow holds for an individual data flow, as well as for an integrated multi-flow.

Data store nodes ($\mathbf{DS}$) can represent either a *source* data store ($\mathbf{DS}_S$, e.g., input DB table, file, etc.) or a *result* data store which in general is not necessarily materialized ($\mathbf{DS}_R$, e.g., output DB table, file, report, virtual cube, etc.), i.e., $\mathbf{DS} = \mathbf{DS}_S \cup \mathbf{DS}_R$. Data store nodes are defined by a schema (i.e., finite list of attributes) and a connection to a source or a target storage for respectively extracting or loading the data. Furthermore, we formally define a data flow *operation* as a quintuple:

$$o = (\mathbb{I}, \mathbb{O}, \mathbb{S}, \mathbf{Pre}, \mathbf{Post})$$

- $\mathbb{I}$ represents a set of input schemata, where each schema ($\mathbf{I}_i$) characterizes an input from a single predecessor operation and is defined with a finite set of attributes coming from that operation (i.e., $\mathbf{I} = \{a_1, .., a_{n_I}\}$). This definition is generic in that it allows the arbitrary input arity of flow operations.
Example. Notice in Figure 3 that some operations like `UDF3` are *unary* and thus have only one input schema, while operations like `Join2` are binary and expect two input schemata. □

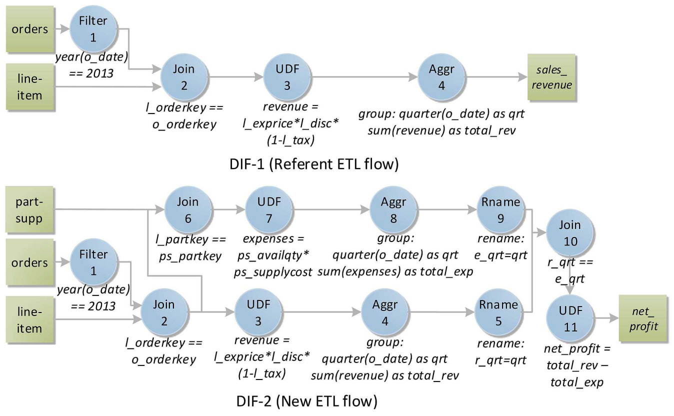

Figure 3. Example data-intensive flows for IR_1 and IR_2

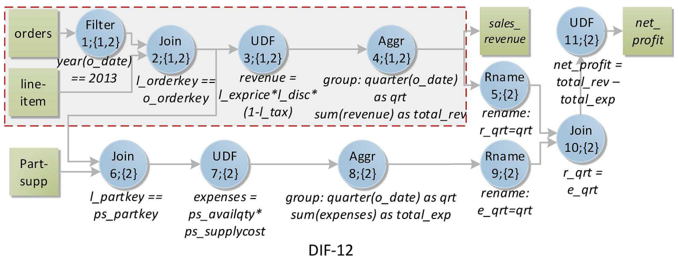

Figure 4. Integrated data-intensive multi-flow (IR_1-IR_2)

- $\mathbb{O}$ represents a set of output schemata, where each schema ($\mathbf{O}_i$) characterizes an output to a single succeeding operation and is defined with a finite set of attributes (i.e., $\mathbf{O} = \{a_1, .., a_{nO}\}$).
Example. The operations in DIF-12 of Figure 4 can have either a single output schema (e.g., `UDF3`, `Filter1` and `Aggr4`), or as it is the case of `Join2` two equivalent output schemata sending the same data to two different subflows. □
- $\mathbb{S}$ represents the formalization of operator's semantics, i.e., a finite set of expressions that interprets the processing semantics of an operator. Due to their inherent complexity and diversity, in order to automate the processing of data flow operations (e.g., comparison, discovery of different operation properties), we express the semantics of a generic data flow operation as a finite set of normalized expression trees. That is, a binary search tree, alphanumerically ordered on the expression elements (i.e., operators, function calls, variables, and constants), whilst respecting the valid order of operators when evaluating the expression. *CoAl* assumes that the semantics' formalization is generic, and similar formalization techniques (e.g., [16]) can be used seamlessly in our approach.
Example. To express the semantics of data flow operations in DIF-1 (Figure 3) we build the normalized expression trees showed in Figure 5. □

To further determine how the operations' semantics affect their interdependence in a data flow, we use a set of data flow operation properties. Relying solely on a set of algebraic properties of data flow operations (e.g.,

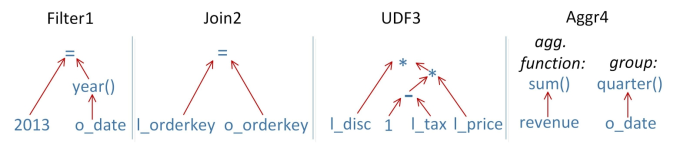

Figure 5. Normalized expression trees (DIF-1 operations)

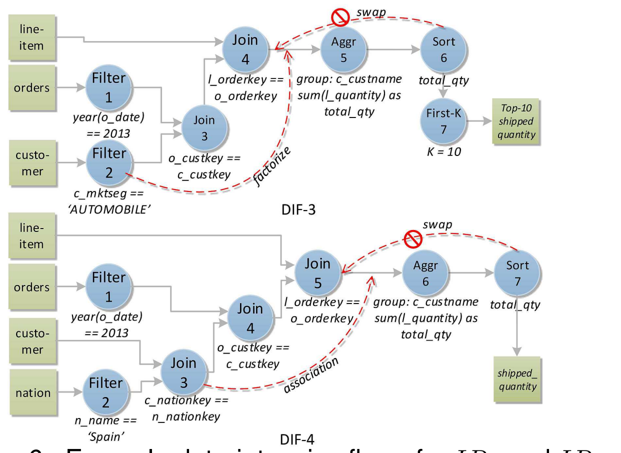

Figure 6. Example data-intensive flows for IR_3 and IR_4

[17], [15]) is not enough to take full advantage of the potential for the data flow analysis. Different 'physio-logical' properties that can be extracted from a data flow, additionally boost the automation of the flow analysis and equivalent operation reordering. For example, such idea has been introduced for the context of optimizing generalized MapReduce data flows by extracting properties like attribute values [18]. Additionally motivated by this work, we extend and generalize the idea of having a customizable set of properties characterizing data flow operations. In this paper, we considered the following set of operation properties:

- *Schema (S).* Attributes being used, emitted or removed by an operation.
- *Values (V).* Attributes whose values are used or produced by an operation.
- *Order (O).* Indicator if the order of the tuples (i.e., rows or records) in the processed dataset affects or is being produced by an operation.

We have analyzed the applicability of these properties over the types of operations in the example ETL tools; both a commercial, i.e., Oracle Warehouse Builder (OWB 11.2), and an open source data integration (ETL) tool, i.e., Pentaho Data Integration (PDI 5.0). This analysis has showed us that these three properties cover the frequently used operations, while our approach is generic and allows the extension to other categories of data processing operations as needed.

Furthermore, in terms of these three properties, for each instance of a data flow operation, we define pre- (**Pre**) and post-conditions (**Post**) of a data flow operation.

Depending on the properties that an *operation "consumes"* (i.e., the results of an operation are affected by the specific value of that input property), we define the pre-conditions of an operation as:

Pre = $(S_{pre}, V_{pre}, O_{pre})$, such that:

- S_{pre} (*consumed schema*) is a subset of attributes of input schemata ($\mathbb{I}$) that are used by an operation.
- V_{pre} (*consumed values*) is a subset of attributes of the consumed schema ($V_{pre} \subseteq S_{pre}$) whose values are used by an operation.
- O_{pre} (*consumed order*) is a boolean indicator that specifies whether the results of the operation processing are affected by the order of input dataset.

Example. The `UDF3` operation of DIF-1 in Figure 3 uses the attributes `l_exprice`, `l_disc`, and `l_tax` from the input (i.e., $S_{pre} = \{l\_exprice, l\_disc, l\_tax\}$), and moreover it uses their values for computing the value of the output attribute `revenue` (i.e., $V_{pre} = \{l\_exprice, l\_disc, l\_tax\}$). On the other hand, notice that the `Rename` operations in DIF-2, require the attribute `qrt` at the input (i.e., $S_{pre} = \{qrt\}$), while the concrete value of that attribute is not used by these operations (i.e., $V_{pre} = \emptyset$). Likewise, the value of the `revenue` attribute resulting from the `UDF` operation, is not affected by the order of the input dataset (i.e., $O_{pre} = false$). □

In a similar way, but now depending on the properties an *operation "produces"* (i.e., generates the specific value of that property at the output), we define the post-conditions of a data flow operation as:

Post = $(S_{post\_gen}, S_{post\_rem}, V_{post}, O_{post})$, such that:

- S_{post_gen} (*generated schema*) is a finite set of new attributes that the operation generates at the output.
- S_{post_rem} (*removed schema*) is a subset of input attributes that the operation removes from the output.
- V_{post} (*produced values*) is a finite set of attributes whose values are either produced or modified by an operation.
- O_{post} (*produced order*) is a boolean indicator that specifies whether the operation processing generates the specific order of the output dataset.

Note that we need to distinguish two sets (i.e., S_{post_gen} and S_{post_rem}) to specify the *schema* property of a post-condition in order to determine the dependency of operations in a data flow. We clarify this when discussing the generic equivalence rules in Section 3.1.

Example. The operation `Aggr4` of DIF-1 in Figure 3 modifies the schema provided at the input and produces the new schema at the output, removing all the input attributes (i.e., $S_{post\_rem} = \{\#all\}$) and generating the grouping attribute `qrt` and the aggregated attribute `total_revenue` (i.e., $S_{post\_gen} = \{qrt, total\_revenue\}$). `Aggr4` also produces the new value for the aggregated `revenue` attribute and the value of the grouping attribute `qrt` (i.e., $V_{post} = \{qrt, total\_revenue\}$), and affects the order of the output dataset (i.e., $O_{post} = true$). □

For extending the set of considered "physio-logical" properties, an instantiation of each property must be defined both at the input (**Pre**) and the output (**Post**) of each data flow operation. If an operation type does not *consume/produce* a property, the corresponding instantiation is empty (or `false`, see the *order* property).

Finally, notice that the **Pre** and **Post** conditions of a data flow operation provide the needed knowledge to determine the dependencies among data flow operations when performing equivalence transformations for reordering operations in a generic data flow. We discuss this in more detail in Section 3.1.

2.3 Problem Statement

We formalize the problem of the incremental data flow consolidation, by introducing the three main design operations to *integrate*, *remove*, and *change* a data flow.

Integrate a data flow ($\cdot_{int}$): When a new information requirement comes, we need to integrate the data flow that satisfies it, into the existing data flow. Considering that a data flow at the logical level is modeled as a directed acyclic graph (*DAG*), in the context of integrating new data flow, at each step we assume two graphs:

- *Referent graph.* An existing multi-flow satisfying the n current information requirements.
  $DIF_{ref} = (\mathbf{V}_{ref}, \mathbf{E}_{ref}) : DIF_{ref} \models \{IR_1, .., IR_n\}$
- *New graph.* A data flow satisfying the upcoming requirement.
  $DIF_{new} = (\mathbf{V}_{new}, \mathbf{E}_{new}) : DIF_{new} \models IR_{new}$

In addition, for each information requirement (IR_i) and a data flow (DIF_{ref}), we define a *requirement subgraph* function (i.e., $DIF_i = (V_i, E_i) = G(DIF_{ref}, IR_i)$), such that G returns a subgraph DIF_i of DIF_{ref}, if the requirement IR_i can be satisfied by DIF_{ref} using DIF_i. Otherwise, it returns $NULL$.

Example. Notice that the shaded subgraph within the multi-flow DIF-12 in Figure 4, ending in the *sales_revenue* data store, is a *requirement subgraph* satisfying IR_1. □

Intuitively, to integrate two data flow graphs ($DIF_{ref} \cdot_{int} DIF_{new}$), we look for the maximal overlapping of their nodes (i.e., data sources and operations). As a result, the integrated multi-flow (i.e., DIF_{int}) must logically subsume the *requirement subgraphs* of all the requirements satisfied by DIF_{ref} and DIF_{new} and consequently satisfy the entailed information requirements.

We define a consolidated multi-flow as:

$DIF_{int} = DIF_{ref} \cdot_{int} DIF_{new} = (\mathbf{V}_{int}, \mathbf{E}_{int})$, s.t.:
$\forall IR_i, i = 1, .., n : G(DIF_{int}, IR_i) <> NULL,$
$G(DIF_{int}, IR_{new}) <> NULL.$

Thus, we say that: $DIF_{int} \models \{IR_1, .., IR_n, IR_{new}\}$

Example. The multi-flow integrated from IR_1-IR_4 is shown in Figure 7, where the overlapping operations are shown inside the shaded areas. □

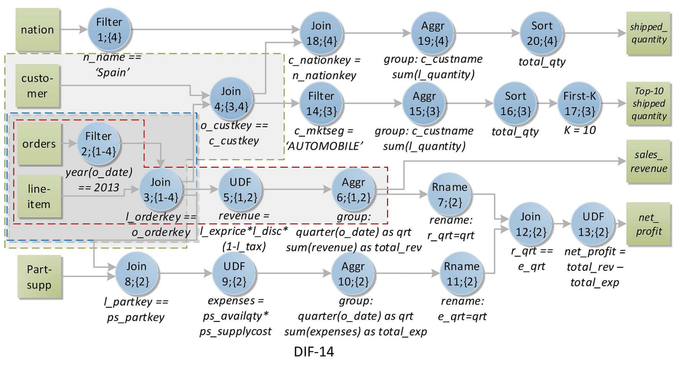

Figure 7. Integrated data-intensive multi-flow (IR_1-IR_4)

Remove a data flow ($\cdot_{rem}$): In the case a user wants to remove an information requirement (IR_{rem}) from her analysis (i.e., $DIF_{ref} \models \{IR_1, .., IR_n\}$), we need to remove its *requirement subgraph* (i.e., $DIF_{rem} = (V_{rem}, E_{rem}) = G(DIF_{ref}, IR_{rem})$), without affecting the satisfiability of other requirements. Formally:

$DIF_{int} = DIF_{ref} \cdot_{rem} DIF_{rem}$, s.t.:
$\forall IR_i, i = \{1, .., n\} \backslash \{rem\} : G(DIF_{int}, IR_i) <> NULL,$
$((V_{rem} \backslash \bigcup_{\forall i \in \{1,..,n\} \backslash \{rem\}} V_i) \cap V_{int}) = \varnothing,$
$((E_{rem} \backslash \bigcup_{\forall i \in \{1,..,n\} \backslash \{rem\}} E_i) \cap E_{int}) = \varnothing.$

To this end, while integrating data flows, *CoAl* maintains metadata, consisting of two maps that for each node (datastore or operation) and edge of the integrated multi-flow, keeps a *share counter* for the total number of input data flows that use that node (i.e., $\forall v \in V_{int} : \exists c > 0$) or edge (i.e., $\forall e \in E_{int} : \exists c > 0$). For target data store nodes that satisfy requirements at hand, *CoAl* also keeps the requirement identifier.

Thus, when a user decides to remove an information requirement, the system will search through its *requirement subgraph*, starting from a target node, and decrements the *share counter* of the visited nodes and edges. If the counter drops to zero, the system removes the node or the edge from the graph, as it is not used by any of the remaining input data flows anymore.

Change a data flow ($\cdot_{chg}$): Similarly, changing an information requirement ($IR_{chg} \rightsquigarrow IR_{chg'}$), results in modifying the subgraph of a referent data flow that satisfies this requirement ($DIF_{chg} \rightsquigarrow DIF_{chg'}$), while preserving the satisfiability of other requirements in the analysis. Intuitively, the operation for changing a data flow can be reduced to the sequence of the previous two operations, i.e., remove and integrate. Formally:

$DIF_{int} = DIF_{ref} \cdot_{chg} (DIF_{chg} \rightsquigarrow DIF_{chg'}) =$
$\quad = (DIF_{ref} \cdot_{rem} DIF_{chg}) \cdot_{int} DIF_{chg'}.$

Next, we discuss the challenges in data flow consolidation and present the *CoAl* algorithm.

3 Data Flow Consolidation Challenges

To search for the overlapping between DIF-12 and DIF-3 (i.e., Figures 4 and 6, respectively), we first find `orders` and `lineitem` as the shared source data stores. Then, starting from these nodes we proceed with comparing operations going further in the encompassing subgraphs towards the result data source nodes, respectively, `Top-10 shipped quantity` and `sales_revenue`. Taking into account the previous example, we notice several challenges that arise when searching the overlapping operations in the referent and the new data flows.

1) Going from the `orders` nodes in DIF-12 and DIF-3, notice that after the common *Filter* operations we identify *Join* operations in both data flows, `Join2` and `Join3`, respectively. However, even though one input of these *Join* operations coincides in both flows, the second input differs, and thus we do not

proceed with comparing these operations.
Incremental advancement. To guarantee the semantical overlapping of two data flows, we must proceed incrementally starting from the common source nodes. That is, for comparing any two operations of two data flows, we must ensure that the predecessors of both operations coincide.

2) Although we do not compare the two *Join* operations, we continue our search. Note that in DIF-3 there is another *Join* operation (i.e., `Join4`) and that it is possible to exchange the order of this operation with the previously discussed `Join3` without affecting the semantics of the DIF-3 flow. As a result, we find a larger set of overlapping operations between DIF-3 and DIF-12.
Operation reordering. Operation reordering under a set of equivalence rules is a widely used optimization technique, e.g., for pushing selective operators early in a flow. When comparing data flows we can benefit from such technique to boost finding the maximal overlapping of operations whilst keeping the equivalent semantics of input data flows.

3) In each step, like after reordering the *Join* operations in DIF-3, when we find that the predecessors of two operations coincide, we proceed with comparing these two operations. Thus the comparison of data flow operations arises a third challenge in consolidating data flows considering the inherent complexity and variety of the data flow operations.
Operation comparison. To integrate the operations of two data flows we need to compare them to ensure that they provide the equivalent dataset at the output. To automate their comparison we need to formalize the semantics of data flow operations.

Before presenting *CoAl* that solves the first challenge, we discuss the theoretical aspects of the last two challenges.

3.1 Operation reordering

Operation reordering has been widely studied in the context of data flow optimization, both for traditional relational algebra operators (e.g., [17]) and generic data flows (e.g., [15], [18]). Different reordering scenarios have proven to improve the performance of data flows (e.g., pushing selective operations early in the flow). Conversely, we observe that reordering techniques can also be used for consolidating data flows by finding the maximal overlapping of flow operations. Thus, here we introduce a set of data flow transformations and generic equivalence rules which ensure that these transformations lead to a semantically equivalent data flow.

Following the previously proposed set of flow transformations in the context of ETL processes [15], in *CoAl* we extend this set considering also the associative property of n-ary operations (e.g., Join) and thus rely on the following four flow transformations used for reordering the operations.

- **Swap.** Applied to a pair of adjacent unary operations, it interchanges the order of these operations.
- **Distribute/Factorize.** Applied on a unary operation over an adjacent n-ary operation, it respectively distributes the unary operation over the adjacent n-ary operation or factorize several unary operations over the adjacent n-ary operation.
- **Merge/Split.** Applied on a set of adjacent unary operations, it respectively merges several operations into a single unary operation or splits a unary operation into several unary operation.
- **(Re-)associate.** Applied on a pair of mutually associative n-ary operations, it interchanges the order in which these operations are executed.

Example. Examples of these transformations applied to integrate DIF-3 and DIF-4 into the referent data flow (Figure 4) are showed in Figure 6. □

Furthermore, we define here the equivalence rules applicable to a generic set of data flow operations, which must hold in order to guarantee a semantically equivalent data flow after performing the previous reordering transformations. Our generic equivalence rules are expressed in terms of the operation properties defined in Section 2.2 (i.e., *schema*, *values*, *order*). Notice that the equivalence rules defined in terms of these properties can be conservative in some cases and prevent some valid reorderings, but whenever applied, they do guarantee the semantic equivalence of a reordered data flow.

Let's consider two adjacent operations o_A and o_B in a data flow, such that o_A precedes o_B (i.e., $o_A \prec o_B$). Thus, to ensure the equivalence transformation, *CoAl* checks if there is no conflict of the properties among these two operations, i.e., if the following constraints hold.

- **Schema conflict.** We must guarantee that all attributes of the schemata used by operations o_A and o_B are available also after the operation reordering.
$(S_{post\_rem_B} \cap S_{pre_A} = \varnothing) \wedge (S_{post\_gen_A} \cap S_{pre_B} = \varnothing)$
Example. In DIF-2 (Figure 3), notice that `Aggr4` generates the attribute `qrt` that is accessed by the `Rname5` operation, and thus there is a conflict of the *schema* property, which prevents the *swap* between these two operations □
- **Values conflict.** We must guarantee that none of the attributes' values used by one operation are modified by another operation.
$(V_{post_B} \cap V_{pre_A} = \varnothing) \wedge (V_{post_A} \cap V_{pre_B} = \varnothing)$
Example. Similarly, in DIF-3 (Figure 6), `Aggr5` generates the value of attribute `total_qty` that is consumed by the `Sort6` operation, and thus there is a conflict of the *value* property, which prevents the *swap* between these two operations. □
- **Order conflict.** We must guarantee that if the results of one operation are affected by a specific order of an input dataset, another operation does not modify the order of the dataset at the output.
$(O_{post_B} \Rightarrow \neg O_{pre_A}) \wedge (O_{post_A} \Rightarrow \neg O_{pre_B})$

Example. For the operations `Sort6` and `First-K7`[2] of DIF-3, *CoAl* finds an order conflict, since `Sort6` affects the order of the tuples in the output dataset, while the results of `First-K7` are obviously affected by the order of the input dataset. □

Besides these, if both o_A and o_B are n-ary operations, and hence *CoAl* tries to apply the *(re-)associate* transformation, we must also ensure that o_A and o_B are mutually associative operations.

Example. Notice that the *equi-join* operations (e.g., `Join2` and `Join6` operations in DIF-12 in Figure 3) are associative, as well as the typical set *union* and *intersection* operations, whilst for example *outer joins* (left, right, and full) and set *difference* in general do not satisfy associative property and cannot be reordered using the *(re-)associate* transformation. □

Finally, only if for all the above conditions, *CoAl* detects no conflict, it can consider reordering o_A and o_B.

Proof. For the proof that the first three flow transformations (i.e., *swap*, *distribute/factorize*, and *merge/split*) lead to a semantically equivalent data flow, we refer the reader to the work of Simitsis et. al, [15]. Furthermore, the proof that the *association* transformation leads to a semantically equivalent data flow, when applied over the operations that satisfy the associative property, is based on the assumption of the equivalence of the output *schemata* and the output *datasets* before and after the *association* is applied.

1) *Schema equivalence.* Regardless of the order of the two adjacent n-ary operations that satisfy the associative property, the equivalent output schemata are provided at the output (e.g., concatenated schemata in the case of equi-join, or equivalent schemata in the case of set union and set intersection).
2) *Dataset equivalence.* The associative property, if satisfied by the two adjacent n-ary operations, guarantees the equivalence of the output datasets before and after the *(re-)associate* transformation is applied.

3.2 Operations comparison

Another challenge for consolidating two data flows is finding the *matching* operations between these flows. In general, operations o_A and o_B, only *match* if they imply the equivalent semantics and the subgraphs having o_A and o_B as their sinks provide equal datasets.

We consider four possible outcomes of comparing two operations, o_A and o_B, from a referent and a new data flow, respectively (see Figure 8).

(1) *Full match:* the compared operations are equivalent (i.e., $o_A \equiv o_B$). In that case, we can consolidate the two operations as a single one in the integrated multi-flow.

Example. In DIF-1 and DIF-2 of Figure 3 we find some *fully matching* operations, e.g., `Filter1` or `UDF3`. □

2. The *First-K* operation keeps only the top K tuples of the input dataset (e.g., `LIMIT` in SQL syntax).

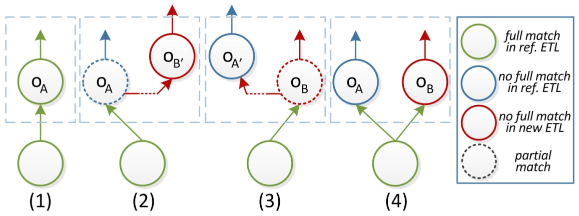

Figure 8. Integration of data flow operations

(2) *Partial (referent) match:* the results of o_B are subsumed by the results of o_A, thus o_B can partially benefit from the transformations already performed by o_A (i.e., $o_B \sqsubseteq o_A$). Then, both operations can be partially collapsed as depicted in Figure 8(2). Furthermore, the consolidation of the partially matched operation o_B may involve an additional transformation (i.e., $o_{B'}$) for obtaining the original output data.

(3) *Partial (new) match:* the results of o_A are subsumed by the results of o_B (i.e., $o_A \sqsubseteq o_B$). Similarly, o_A can benefit from the transformations performed by o_B.

Example. Consider an alternative scenario where the requirement IR_2 only takes into account urgent orders, i.e., `Filter1` uses `year(o_date)=2013 AND o_orderprior = '1-URGENT'`. Then, we would find a *partial match* of the *Filter* operations in DIF-1 and DIF-2, since `Filter1` in DIF-2 could partially benefit from `Filter1` in DIF-1 and would need an additional filter using `o_orderprior = '1-URGENT'`. □

(4) *No match:* Finally, it may happen that neither o_B nor o_A can benefit from one another, (i.e., $o_A \not\sqsubseteq\not\sqsupseteq o_B$). Then, the two operations cannot be consolidated. Thus, we introduce a *fork* in the already matched subflow, as shown in Figure 8(4). The fork in such case requires the copy-partition functionality (i.e., parallel flows).

Following the notation in Section 2.2, here we formalize the comparison of two operations o_A and o_B as follows:

- $\mathbf{o}_A(\mathbb{I}_A, \mathbb{O}_A, \mathbf{S}_A, \mathbf{Pre}_A, \mathbf{Post}_A) \equiv \mathbf{o}_B(\mathbb{I}_B, \mathbb{O}_B, \mathbf{S}_B, \mathbf{Pre}_B, \mathbf{Post}_B)$ *iff* $\mathbb{I}_A = \mathbb{I}_B \wedge \mathbb{O}_A = \mathbb{O}_B \wedge \mathbf{S}_A \equiv \mathbf{S}_B$;
- $\mathbf{o}_A(\mathbb{I}_A, \mathbb{O}_A, \mathbf{S}_A, \mathbf{Pre}_A, \mathbf{Post}_A) \sqsupseteq \mathbf{o}_B(\mathbb{I}_B, \mathbb{O}_B, \mathbf{S}_B, \mathbf{Pre}_B, \mathbf{Post}_B)$ *iff* $\exists \mathbf{o}_{B'}(\mathbb{I}_{B'}, \mathbb{O}_{B'}, \mathbf{S}_{B'}, \mathbf{Pre}_{B'}, \mathbf{Post}_{B'}) : \mathbb{I}_A = \mathbb{I}_B \wedge \mathbb{O}_A = \mathbb{I}_{B'} \wedge \mathbb{O}_{B'} = \mathbb{O}_B \wedge \mathbf{S}_B \equiv \mathbf{S}_A \circ \mathbf{S}_{B'}$;

Intuitively, we find the equivalence (i.e., *full match*) of two data flow operations, if their input and output schemata coincide, while at the same time the operations define the equivalent semantics (i.e., $\mathbf{S}_A \equiv \mathbf{S}_B$). To find the *partial match* between two operation o_A and o_B (i.e., cases (2) and (3) in Figure 8), we need to check if the result of one operation (o_B) can be partially obtained from the results of another operation (o_A), i.e., if the results of o_B are subsumed by the results of o_A. To this end, we look for a new operation (o'_B) so that the semantics of operations o_A and o'_B can be *functionally composed* to imply the equivalent semantics as the operation o_B and hence provide the same result dataset. Note that in general, finding the subsumption among the expressions is known to be a challenging problem. Thus for arbitrary operation expressions, we rely on the current state of the art for reasoning over the expressions and assume that the expressions are in conjunctive normal form.

4 Consolidation Algorithm

We now present the *CoAl* algorithm for data flow consolidation.

Intuitively, the *incremental advancement* property defined in the previous section, requires that for solving the problem of integrating any two data flow graphs, by maximizing the reuse of their data and operations, we first need to recursively solve the subproblems of integrating their subgraphs, from the data sources, and following a topological order of nodes in the graphs.

Given the clear ordering and dependencies between these subproblems, we formulate the problem of integrating data-intensive flows as a dynamic programming problem. We devise a bottom-up, iterative variant of the algorithm that efficiently solves the problem in our case.

In particular, *CoAl* starts with two data flow graphs, the referent (DIF_{ref}) and the new (DIF_{new}), and proceeds following a topological order of the data flow operations in a graph, starting from the matched leaf (source) nodes. *CoAl* iteratively searches larger matching opportunities between their operations by applying the generic equivalence rules (Section 3.1), hence considering reordering without modifying the semantics of the involved data flows. At each iteration, *CoAl* compares two operations (one from each data flow), and continues only if a *full match* is found (Section 3.2). This guarantees the following two invariants:

(**I**$_1$): At each step, only one pair of operations of referent and new data flows can be partially or fully matched.
(**I**$_2$): A new match is added to the set of matched operations if and only if the operations themselves match and their input flows have been previously fully matched.

In addition, when searching for next operations to match, the following invariant must also hold:

(**I**$_3$): An operation can be reordered to be matched next, if and only if such reordering does not change the semantics (i.e., output) of a data flow.

The consequence of these invariants is that a pair of matched operations is eventually consolidated in the output, integrated multi-flow, if the flows they belong to can be reordered so that their children are fully matched.

Thus, the correctness of the *CoAl* algorithm, in the sense that it integrates input data flows without affecting their outputs, is guaranteed by the three characteristics of the algorithm, –i.e., *operation comparison*, *incremental advancement*, and *equivalent operation reordering*,– discussed in Section 3, and demonstrated respectively with the invariants **I**$_1$, **I**$_2$, and **I**$_3$.

Example. We illustrate different steps of *CoAl*, using the scenario of integrating data flow DIF-3 (Figure 6), into the referent DIF-12 multi-flow (Figure 4). □

In general, *CoAl* comprises four phases (see Algorithm 1): i) *search for the next operations to match*; ii) *compare the next operations*; iii) *reorder input data flows* if a match has been found; and iv) *integrate the solution* with the lowest estimated cost. The first three steps are executed in each iteration of *CoAl*, while the last one is executed once, when no matching possibility is left.

Algorithm 1 CoAl

```
inputs: $DIF_{ref}$, $DIF_{new}$; output: $DIF_{int}$
 1: $altList := \{[[DIF_{ref}, DIF_{new}, \emptyset, \emptyset], cost_{noInt}]\}$;                ▷ no int. alternative
 2: $matchQ$ := matchLeafs($DIF_{ref}$, $DIF_{new}$);
 3: while $matchQ \neq \emptyset$ do
 4:     $matchDIFPair[DIF'_{ref}, DIF'_{new}, allMatches, lastMatches]$ := dequeue($matchQ$);
 5:     $[matchOpsPair[o_{ref}, o_{new}], edge_{ref}]$ := dequeue($lastMatches$);
 6:     $nextOps_{ref}$ := findNextForMatch($DIF'_{ref}, o_{ref}, edge_{ref}$);          ▷ Algorithm 2
 7:     $nextOps_{new}$ := findNextForMatch($DIF'_{new}, o_{new}, edge_{o'_{new}}$);       ▷ Algorithm 2
 8:     for all $pair(o'_{ref} \in nextOps_{ref}, o'_{new} \in nextOps_{new})$ do
 9:         if $o'_{ref} \equiv o'_{new} \vee o'_{ref} \sqsubseteq o'_{new} \vee o'_{new} \sqsubseteq o'_{ref}$ then
10:             $DIF''_{ref}$ := reorder($DIF'_{ref}, o'_{ref}, o_{ref}$);
11:             $DIF''_{new}$ := reorder($DIF'_{new}, o'_{new}, o_{new}$);
12:             insert($allMatches$, $[[o'_{ref}, o'_{new}], intInfo]$);
13:             if $o'_{ref} \equiv o'_{new}$ then                                    ▷ full match
14:                 for i:=1 to deg($o'_{ref}$) enqueue($lastMatches$, $[[o'_{ref}, o'_{new}], edge_{ref_i}]$);
15:                 enqueue($matchQ$, $[DIF''_{ref}, DIF''_{new}, allMatches, lastMatches]$);
16:             else if $o'_{ref} \sqsubseteq o'_{new} \vee o'_{new} \sqsubseteq o'_{ref}$ then          ▷ partial match
17:                 if $lastMatches = \emptyset$ then                ▷ no further matchings avail.
18:                     insert($altList$, $[[DIF''_{ref}, DIF''_{new}, allMatches, lastMatches], cost'']$);
19:                 else
20:                     enqueue($matchQ$, $[DIF''_{ref}, DIF''_{new}, allMatches, lastMatches]$);
21:                 end if
22:             end if
23:         end if
24:     end for
25:     if no matching found then
26:         if $lastMatches = \emptyset$ then                        ▷ no further matchings avail.
27:             insert($altList$, $[[DIF'_{ref}, DIF'_{new}, allMatches, lastMatches], cost']$);
28:         else
29:             enqueue($matchQ$, $[DIF'_{ref}, DIF'_{new}, allMatches, lastMatches]$);
30:         end if
31:     end if
32: end while
33: $bestAlt$ := findMin($altList$);
34: $DIF_{int}$ := integrate($bestAlt$);
```

Before detailing the four phases, we present the main structures maintained by *CoAl* while looking for the final consolidation solution.

1) *matchQ:* A priority queue that contains pairs of referent and new data flows with currently overlapping areas which can be potentially extended with new matching operations.
   matchQ ::= matchDIFPair, matchQ|matchDIFPair;
2) *altList:* A list of all alternative solutions ending up in a partial or full overlapping of two data flows (referent and new), together with the estimated costs of such consolidation solution.
   *altList ::=[matchDIFPair,**cost**],altList| [matchDIFPair,**cost**];*

Each element of *matchQ* and *altList* contains information of integrated data flows, i.e.,
matchDIFPair ::= $[DIF_{ref}, DIF_{new},$ *allMatches,lastMatches*];

- A pair of data flows (DIF_{ref} and DIF_{new}), potentially reordered for such integration.
- Pairs of pointers to all matched operations (*allMatches*), with information about the matching type and integration (*intInfo*).
  *allMatches ::= [matchOpsPair,**intInfo**],allMatches|*
  *[matchOpsPair,**intInfo**];*
- A **queue** with pairs of pointers to the last matched operations (*lastMatches*), and an out-edge in a referent graph (*edge$_{ref}$*).
  lastMatches $::=$ [*matchOpsPair*, *edge$_{ref}$*], *lastMatches*|
  [*matchOpsPair*, *edge$_{ref}$*];

Algorithm 2 FindNextForMatch

```
inputs: DIF, $o_{cur}$, out-edge; output: $nextOps$
1: nextOps := ∅;
2: for all $path \in$ findPathsToForks(DIF, $o_{cur}$, out-edge) do
3:     for i:=1 to length(path) do
4:         if canReorder($path, i$) ∧ fulfillsI2($path[i]$) then
5:             insert($nextOps$,path[i]);
6:         end if
7:     end for
8: end for
9: return $nextOps$;
```

matchOpsPair ::=[o_{ref}, o_{new}];

CoAl first initializes the list of alternative solutions by adding the alternative with no integration of DIF_{new} and DIF_{ref}, together with the cost of having these two data flows separately, i.e., $cost_{noInt}$ (Step 1).

CoAl then starts by searching for the matching leaf (source) nodes of new and referent data flows (i.e., **matchLeafs**, Step 2). The source data stores are compared based on their main characteristics, i.e., *source type*, *source name* or *location*, and *extracted attributes*. *CoAl* initializes *matchQ* with the pair of the referent and new data flows, together with the found matching pairs of source data stores (i.e., initially both *allMatches* and *lastMatches*).

Example. When integrating DIF-3 (Figure 6), into the referent flow, i.e., DIF-12 (Figure 4), we first identify common source nodes of the two data flow graphs (i.e., `orders` and `lineitem`). □

The four phases of *CoAl* are as follows:

i) *Search for the next operations to match.* At each iteration, we consider extending a single pair of currently overlapping data flows from the priority queue (*matchQ*) with a new pair of matching operations. For a dequeued pair of data flows (i.e., **dequeue**, Step 4), we identify the operations in these flows to be compared next, starting from a pair of previously identified full matches (i.e., o_{ref} and o_{new} dequeued from *lastMatches*; **dequeue**, Step 5). Finding operations to be compared next in both data flows is performed by means of two calls to the function **FindNextForMatch** (i.e., Algorithm 2) in steps 6 and 7). In **FindNextForMatch** we apply the generic equivalence rules explained in Section 3.1, and find the operations that can be pushed down towards the last fully matched operations, thus fulfilling I_2 (i.e., **canReorder** and **fulfillI2**, Step 4 in Algorithm 2). Notice that we search until we reach the operation that has multiple outputs (i.e., **findPathsToForks**, Step 2), since swapping operations down a fork would affect the semantics of other branches in a data flow.
Example. For the fully matching source nodes `orders`, of DIF-12 and DIF-3, we find the following sets of operations to be compared next: $orders_{DIF-12}$ = {`Filter1`, `Join2`}; $orders_{DIF-3}$ = {`Filter1`, `Join3`, `Join4`}. □

ii) *Compare the next operations. CoAl* then compares each pair of operations from the previously produced sets (i.e., $nextOps_{ref}$ and $nextOps_{new}$), using the comparison rules discussed in Section 3.2. Depending on the result, it identifies: (a) a *full match*, $o\prime_{ref} \equiv o\prime_{new}$ (Step 13); (b) a *partial match*, $o\prime_{new} \sqsubseteq o\prime_{ref} \vee o\prime_{ref} \sqsubseteq o\prime_{new}$ (Step 16) or (c) *no match*, $o\prime_{ref} \not\sqsubseteq\not\sqsupseteq o\prime_{new}$.
Example. From the two sets of operations that can be compared next, we find two full matches between `Filter1`(DIF-12) and `Filter1`(DIF-3), and `Join2`(DIF-12) and `Join4`(DIF-3). □
It may also happen that no matching is found for any pair of operations (Step 25).

iii) *Reorder the input data flows.* If *CoAl* finds a (*full* or *partial*) *match*, it proceeds (if needed) with operation reordering to align the input data flows and enable integration of the previously matched operations, i.e., to satisfy I_2, (i.e., **reorder**, steps 10 and 11).
Example. Following the above example, for the *full match* of the *Filter* operations in DIF-12 and DIF-3, no additional operation reordering is necessary and *CoAl* directly adds `Filter1` to the current maximum overlapping area (i.e., I_2 is satisfied). But, for the *full match* between `Join2`(DIF-12) and `Join4`(DIF-3), *CoAl* must perform operation reordering (i.e., *(re-)associate* `Join4` down `Join3` in DIF-3), so that the `Join4` operation could be matched next (I_2). □
CoAl then extends the overlapping of input data flows with matching pair of operations to *allMatches*, together with their integration information (i.e., **insert**, Step 12). Next, based on the type of the previously found match, *CoAl* proceeds as follows:

- For a *full match*, it enqueues back to priority queue the two data flows (possibly reordered) to further extend the matching in the next iterations (i.e., **enqueue**, Step 15), starting from the last matched pair of operations added (i.e., *lastMatches*). Notice that *CoAl* needs to enable the search in all possible output branches of the referent data flow, thus we enqueue back the currently matched operations once for each of the next *out-edges* to be followed from the previously matched operations (see Step 14).
- For a *partial match*, if there are no other previously matched operations from which it can extend matching (Step 17), *CoAl* estimates the cost of the current solution and inserts it, along with its cost, to the list of alternatives (Step 18). Otherwise, it enqueues back to *matchQ* the two data flows to further extend the matching in other branches. (i.e., **enqueue**, Step 20).

Similarly, if no match is found (Step 25), this solution along with its estimated cost is also added to the list of potential alternatives only if it is not possible to further extend the matching.
Example. In the given example (i.e., DIF-3 and DIF-12), this occurs after we match the join operations (i.e., `Join2` from DIF-12 and `Join4` from DIF-3). Going further in data flows DIF-12 and DIF-3, we

cannot find any matching operation that can enlarge the common areas of these two data flows. Thus, we add the currently matched data flows as an alternative solution, resulting in an integrated multi-flow branching after the matched join operation (i.e., `Join3;{1-4}` in Figure 7) $\square$
Otherwise, *CoAl* continues matching in other branches (i.e., **enqueue**, Step 29).
The algorithm finishes the matching process when all operations of input data flows are explored and compared (i.e., no more elements in the *matchQ*).

iv) *Integrate an alternative solution.* After all iterations finish, *CoAl* analyzes the list of the found alternatives, looks for the one with the lowest estimated cost (i.e., **findMin**, Step 33), and integrates it using the integration information (i.e., **integrate**, Step 34).

Finally, *CoAl* returns the integrated multi-flow (i.e., DIF_{int}).

4.1 Computational complexity

To integrate a referent ($DIF_{ref} = (V_{ref}, E_{ref})$) and a new ($DIF_{new} = (V_{new}, E_{new})$) data flow, the *CoAl* algorithm at first glance indicates at worst quadratic complexity (in terms of $|V_{ref}| \cdot |V_{new}|$), due to the Cartesian product of operations that can be compared next (see Step 8). However, there are several characteristics that either directly from the invariants of the *CoAl* algorithm or from empirical experiences show that this is not a realistic upper bound of the algorithm.

- Under the assumption that input data flows are compact in terms that they do not have *redundant* operations (i.e., operations of a single flow that can be *fully matched*; see Section 3.2), it is impossible that multiple alternative paths branching from a single operation in a multi-flow completely match with a path of another data flow.
- The search is led by the size of paths in the new data flow (i.e., $|p_{new}|^{avg}$), which is typically shorter than the paths in the referent data flow.
- When searching next operations to compare (i.e., Algorithm 2), due to conflicting dependencies among operations (see Section 3.1), it is also unrealistic that all the operations in the paths of encompassing *requirement subgraphs* could be reordered to be compared next, which drastically reduces the actual number of comparisons inside the loop.

We further analyze the complexity of the *CoAl* algorithm based on the search space of the algorithm while looking for the next operations to match. That is, we take into account the number of the main loop iterations (see Step 3), and for each loop, the number of operations searched to be compared next in each loop (see Algorithm 2). While the cost of Algorithm 2 for a new data flow in bounded by the maximal size of new data flow graphs, the cost for a referent one grows iteratively as the size of the data flow graph grows, hence we take the latter one into account when estimating the complexity of the *CoAl* algorithm. Additionally, notice that the number of the main loop iterations (Step 3) depends on the number of elements previously enqueued to the *matchQ* and *lastMatched*, which occurs only when we find a *full match* between two data stores or two operations (see steps 2 and 13 in Algorithm 1). In the worst case for the complexity, we can find a *full match* for all operations in a path of a new data flow, i.e., the path is completely subsumed by the referent flow.

We start by estimating the number of iterations of the main loop. The complexity for a single path of new data flow (i.e., p_{new}) can be obtained as follows:

$$c(p_{ref} \cdot_{int} p_{new}) = \sum_{i=1}^{|p_{new}|} deg(o_{ref_i}) \qquad \square$$

If we consider an average *outdegree* of a referent data flow graph (i.e., $deg^{avg}(DIF_{ref})$):

$$c(p_{ref} \cdot_{int} p_{new}) = \sum_{i=1}^{|p_{new}|} deg(o_{ref_i}) = \\ = |p_{new}|^{avg} \cdot deg^{avg}(DIF_{ref}) \qquad \square$$

Furthermore, *CoAl* performs such search for all paths starting from the previously matched source data stores, i.e., maximally for $|\mathbf{DS}_{new_S}| \cdot |\mathbf{DS}_{ref_S}|$ paths.

$$c_1(DIF_{ref} \cdot_{int} DIF_{new}) = c_1 = \\ = |\mathbf{DS}_{new_S}| \cdot |\mathbf{DS}_{ref_S}| + |\mathbf{DS}_{new_S}| \cdot |p_{new}|^{avg} \cdot deg^{avg}(DIF_{ref}) \square$$

Using graph theory, we can further represent the average *outdegree* of a directed graph (i.e., $DIF_{ref} = (V_{ref}, E_{ref})$) as $\frac{|E_{ref}|}{|V_{ref}|}$. At the same time, the average length of a *source-to-target* path $|p_{new}|^{avg}$ can be obtained as the average depth of a new graph. Assuming that a graph resulting from a single information requirement is a tree, and the tree is balanced, we can obtain its depth as $\log_{\frac{|E_{new}|}{|V_{new}|-|\mathbf{DS}_{new_S}|}} |DS_{new_S}|$. That is:

$$c_1 = |\mathbf{DS}_{new_S}| \cdot |\mathbf{DS}_{ref_S}| + |\mathbf{DS}_{new_S}| \cdot \\ \cdot (\log_{\frac{|E_{new}|}{|V_{new}|-|\mathbf{DS}_{new_S}|}} |\mathbf{DS}_{new_S}|) \cdot \frac{|E_{ref}|}{|V_{ref}|} \qquad \square$$

Next, for each loop, we estimate the complexity of searching for the next operations to compare in the referent data flow graph. Starting from a pair of last matched operations, we search in all paths, and identify the next operations that are candidates to be reordered and compared next (see Algorithm 2). Such search in general resembles the tree traversal with the last matched operation as a root and target data stores as leaf nodes. We estimate the size of such tree and thus the complexity of its traversal with the average depth of a tree (i.e., the average length of a *source-to-target* path), multiplied by the average outdegree of the graph. Again, based on the graph theory, we can express such estimations in terms of the size of a referent data flow graph. That is:

$$c_2 = depth^{avg}(DIF_{ref}) \cdot deg^{avg}(DIF_{ref}) = \\ = (\log_{\frac{|E_{ref}|}{|V_{ref}|-|\mathbf{DS}_{ref_R}|}} |\mathbf{DS}_{ref_S}|) \cdot \frac{|E_{ref}|}{|V_{ref}|} \qquad \square$$

Thus, we estimate the complexity of the algorithm as:

$$c_{int} = c_1 \cdot c_2 = \\ = (|\mathbf{DS}_{new_S}| \cdot |\mathbf{DS}_{ref_S}| + |\mathbf{DS}_{new_S}| \cdot (\log_{\frac{|E_{new}|}{|V_{new}|-|\mathbf{DS}_{new_S}|}} |\mathbf{DS}_{new_S}|) \cdot \\ \frac{|E_{ref}|}{|V_{ref}|}) \cdot ((\log_{\frac{|E_{ref}|}{|V_{ref}|-|\mathbf{DS}_{ref_R}|}} |\mathbf{DS}_{ref_S}|) \cdot \frac{|E_{ref}|}{|V_{ref}|}) \qquad \square$$

Assuming that the average size (i.e.,$|E_{new}|$, $|V_{new}|$) and the number of source data stores (i.e., $|\mathbf{DS}_{new_S}|$) in a

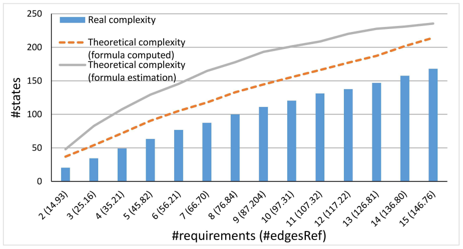

Figure 9. Search space exploration

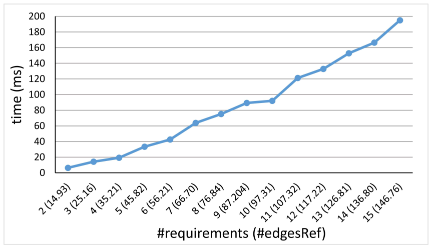

Figure 10. *CoAl*'s execution time

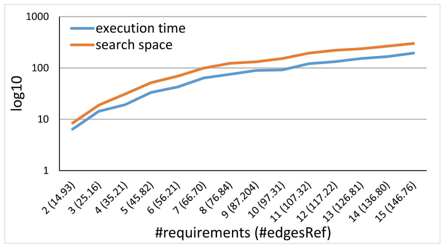

Figure 11. Space/time correlation

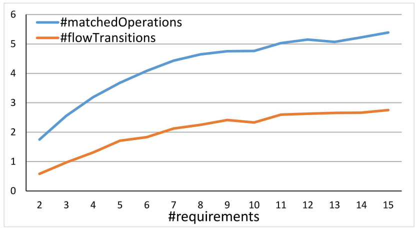

Figure 12. *CoAl* characteristics

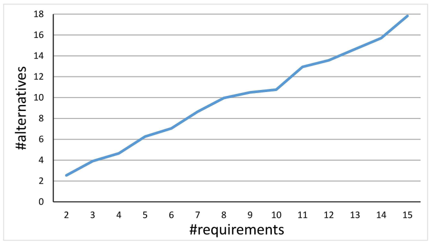

Figure 13. Alternative solutions

new data flow is constant, the theoretical complexity for the problem of integrating data flows can be given as a function of the size of the referent data flow (i.e., $|E_{ref}|$), the number of its data sources (i.e., $|\mathbf{DS}_{ref_S}|$), and its average outdegree (i.e., $\frac{|E_{ref}|}{|V_{ref}|}$).

5 Evaluation

5.1 Prototype

CoAl works at the logical level and integrates data flows coming from either high level business requirements (e.g., [11]) or platform-specific programs (e.g., queries, scripts, ETL tool metadata; [19]). We built a prototype that implements the *CoAl* algorithm. The prototype is integrated into a larger ecosystem for the design and deployment of data flows from information requirements [20]. Communication with different *external modules* of the ecosystem for import/export of data flows is enabled using a platform-independent representation of a data flow, namely *xLM* (i.e., XML encoding of data flow metadata [21]). Data flows in other languages could be translated to/from *xLM* using external tools (e.g., [12]).

5.2 Experimental setup

We selected a set of 15 data flows, translated from the referent TPC-H benchmark queries[3]. Notice that even though the TPC-H benchmark provides a relatively small set of input queries, such set covers different sizes and complexities of input data flows and suffices to demonstrate the functionality of the *CoAl* algorithm. Thus the obtained results are generalizable to other inputs. *CoAl*'s flexibility to deal with different complexities of data flow operations is previously showed in sections 2 and 3. Considered data flows (similar to those presented in Section 2.1), span from only 4 to the maximum of 20 operations, performing various data transformations, i.e., *filters*, *joins*, *projections*, *aggregations*, *user defined functions*. More information about the selected queries can be found in the TPC-H specification [14]. We translated the selected SQL queries, into the platform-independent representation that *CoAl* understands (i.e., *xLM*).

To cover a variety of input scenarios (i.e., different orders in which input data flows are provided), we have considered different permutations of incoming data flows. Since the total number of different permutations for the chosen 15 queries is not tractable (i.e., 15!, $>$ 1307 billions), we have randomly sampled, a uniformly distributed set of 1000 permutations and obtained the average values of the observed numbers. For each permutation, we have simulated the incremental arrival of input data flows, starting by integrating the first 2 data flows, and then incrementally adding the other 13.

5.3 Scrutinizing *CoAl*

We first analyzed the distribution of the values obtained in the considered sample of permutations. For all of them we confirmed a positive (right) skew, which indicated the stability of our data flow integration algorithm. Thus, in the reminder of this section, we report the mean values obtained from the considered permutations.

Search Space. As shown in Figure 9 (*Real complexity*), the search space (i.e., *#states* refers to the complexity in Section 4.1) grows linearly with the number of input data flows. For input flows of an average size of 15 operations, the number of states considered starts from only several when integrating 2 data flows and go up to the maximum of 170 states when integrating the 15^{th} data flow (Figure 9 reports the average values).

We additionally estimated the theoretical computational complexity of *CoAl* for given inputs, following the formula in Section 4.1 and compared it to the obtained real values (see *Theoretical complexity (formula estimated)* in Figure 9). We observed that the overestimation and a

3. Selected TPC-H queries: q1, q2, q3, q4, q5, q6, q9, q10, q11, q13, q16, q17, q18, q19, q21. Other queries are discarded due to limitations of the available external SQL translation module.

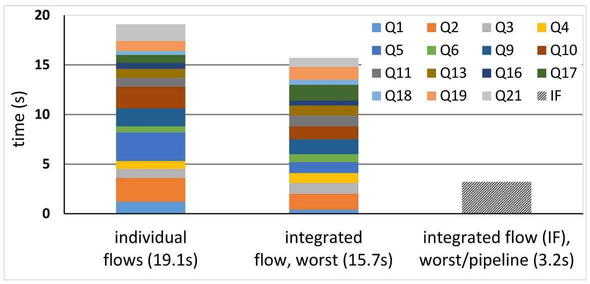

Figure 14. Performance gains (worst integration case)

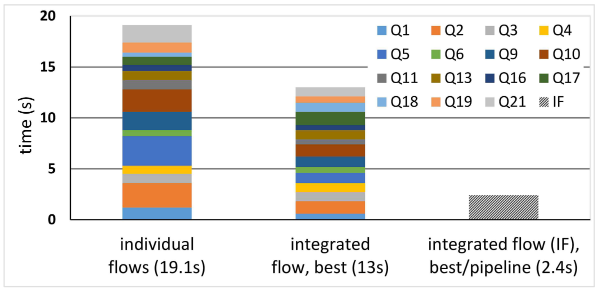

Figure 15. Performance gains (best integration case)

slight deviation of the tendency of the formula estimated complexity comes from the generalizations adopted for estimating the average depth and the degree of a data flow graph. Fitting the formula with the average values directly computed in the execution (see *Theoretical complexity (formula computed)* in Figure 9), showed the coinciding tendency with the real complexity and smaller overestimation resulted from the averaged values.

On the other side, following the complexity discussion in Section 4.1, we analyzed the time needed to complete the search in terms of the size of the referent data flow (i.e., number of edges). This analysis showed that the time also grows linearly following the size of a referent data flow (see Figure 10), starting from only 6.4 ms with the initial size of a referent data flow (i.e., 15 edges), and going up to the maximum of 195ms when integrating the fifteenth data flow over the referent data flow with 147 edges. These values showed a very low overhead of *CoAl*, making it suitable for today's right-time BI applications. Moreover, such results further show *CoAl*'s scalability for larger input complexities. We additionally analyzed the correlation of the time and the search space (see Figure 11), and showed that the growth of execution time follows the same (linear) trend as the complexity growth, which validated our complexity estimations discussed in Section 4.1.

Algorithm characteristics. We also studied the behavior of *CoAl* internal characteristics. Figure 12 shows how the number of matched operations (#matchedOperations) and the number of flow transitions (#flowTransitions), related to the I_2 invariant, are affected by the size of the problem. The average number of matches increases from 1 to 4 (excluding the matched data sources), until the sixth integrated data flow, and then this trend slows only up to 5 matched for the following data flows. This happens because an integrated flow may impose branching (multiple outputs) for the subflows shared among the input data flows (see Section 4). Such behavior restricts the operation reorderings from one branch down the fork, as it would change the semantics of the shared subflow, and thus of all the dependent branches.

This trend is also confirmed by the number of different flow transitions (i.e., number of different operation reorderings; see Figure 12). This further showed that in the basic integration scenario different orders of incoming data flows might produce different integration solution (although all of them will be semantically equivalent). Notice, however, that tracing the metadata of original data flows and all integration alternatives (whose number grows linearly with the size of the problem; see Figure 13), would facilitate the maintenance of integrated multi-flows and the revision of some integration choices.

Improvement in the overall execution time. Additionally, by reviewing the integrated multi-flows for the considered sample of order permutations, we have identified a certain variation of the result characteristics (i.e., a relative standard deviation of the output size is around 20%), and thus we isolated two permutations whose outputs we further analyzed, i.e., (1) *the best case* - among the considered permutations, the permutation that produces the largest overlapping (i.e., the most matched operations) between the input data flows, and (2) *the worst case* - among the considered permutations, the permutation that produces the smallest overlapping (i.e., the least number of matched operations) between the input data flows. For these two cases, we analyzed the execution time of the multi-flow after integrating all 15 data flows from the input, and compared it with the total execution time for the 15 individually executed data flows. Notice that we do not present here optimal solutions in terms of performance, but rather analyze how different degrees of data and transformation reuse affect the overall execution time of a data-intensive multi-flow. For these experiments we ran data flows in Pentaho Data Integration tool using a dataset of 10k to 20k tuples generated from the TPC-H data generator.

The results are illustrated in Figures 14 and 15 for the *worst* and the *best* overlapping case, respectively. We first individually executed the data flows from the input, and observed that it took 19.1s in total to execute 15 data flows with the maximum of 2.4s for executing the data flow of Q2 from TPC-H. We further executed the integrated solutions of the *best* and the *worst* case, as explained above. Notice in Figures 14 and 15 that some data flows are penalized by the integration (e.g., Q17), i.e., their execution time increased due to unfavorable reordering of more selective operations (e.g., filters over joins) to achieve larger overlapping with the referent data flow. Conversely, some larger data flows (e.g., Q2 and Q5) largely benefit from the integration by reusing already performed data processing of other data flows. Note that in both cases the overall execution time of the integrated multi-flow decreased. The *best case* solution (see Figure 15) took 13s for the overall execution, whilst the *worst case* solution (see Figure 14) took 15.7s. We thus observed approx. 31.9% of improvement of the overall execution time for the *best* and 17.8% for the *worst case*, which finally confirms our initial assumptions.

We additionally confirmed that the improvement of the overall execution time is correlated with the amount of overlapping (i.e., number of shared data and operations). For instance, the improvement in the overlapping size from the *worst* to the *best* integration case discussed above (i.e., 51 in the *worst* to 79 in the *best* case) showed to be approx. proportional to the improvement of the overall execution time for these two cases.

Furthermore, integration of multiple data flows, enabled extra optimization inside the considered execution tool, by allowing the pipelined execution of the unified data flow. This finally resulted with 2.4s of the overall execution pipeline for the *best case*, and 3.2s for the *worst case*. Notice that we report these results for showing one of the benefits of integrating different data flows (see shaded bars in Figures 14 and 15), while for the fair comparison we used the results not taking into account the advantage of the enabled pipeline execution.

6 Related Work

From the early years, the reuse in data integration setting has been encouraged and envisioned as beneficial [6], since organizations typically perform data integration efforts that involve overlapping sets of data and code. However, such problem has also been characterized as challenging due to inherently complex enterprise environments. Different guidelines and approaches have been proposed to tackle this issue in various scenarios.

Schema mapping management. The *data exchange* and *data integration* problems set the theoretical underpinnings of the complexity of what we today call an ETL process [22]. Schema mappings, as a set of logical assertions relating the elements of source and target schemata, play a fundamental role in both data exchange [23] and data integration [1] settings. Intuitively, schema mappings are predecessors to more complex ETL transformations [22]. Various approaches and tools dealt with automating schema mapping creation (e.g., [23], [24]), while others further proposed high-level operations over the set of mappings, i.e., *composition*, *inversion*, and *merge*, (e.g., [25]). We find the problems of *composing* and *merging* schema mappings especially interesting for the context of data flow consolidation in terms of reducing *information redundancy* and *minimality*. The remarks of these works motivated our research, but moving towards more complex scenarios of today's BI introduced new challenges both regarding the complexity of schema mappings (e.g., grouping, aggregation or "black-box" operations) and the diversity of data sources, that these approaches could not support (i.e., only relational or XML data formats have been previously considered). Conversely, we propose the generic solutions for both operation reordering and operation comparison problems that solve the problem for an arbitrary set of data flow operations.

Workflow optimization. Equivalence rules used in data flow optimization can be conveniently applied in the context of consolidating data flows to maximize the data and operation sharing (see Section 3.1). Both traditional query optimization [17] and multi-query optimization approaches [7] focus on performance and consider a different subset of operations than those typically encountered in complex data flows (e.g., operations with "black-box" semantics). Recently, more attention has been given to solving the data flow optimization problem for a generic set of complex operations (e.g., [15], [18]). In the former work, the problem of ETL optimization has been modeled as a state-space search problem [15], with a set of generic equivalence transitions used to generate new (eventually optimal) states. Such equivalence transitions inspired those presented in Section 3.1 (i.e., *swap*, *factorize/distribute*, *merge/split*), but state generation is based solely on the information about the schemata used and produced by ETL operations. We propose a less conservative approach where we distinguish three different properties of data flow operations (i.e., *schema*, *value*, and *order*) and thus we are able to detect more promising reordering (optimization) opportunities. On the other side, the later approach [18] does consider the attributes' *value* as an important operation property, but overlooks the dataset *order*. The main reason is that the approach focuses solely on the set of second order operations written in an imperative language for a big data analytics system (e.g., *Map*, *Reduce*, *Cross*, etc.).

Recent optimization approaches (e.g., [26]) discuss the problem of finding the optimal global query plan for a set of input queries by means of data and operation sharing. Another approach considers two non-orthogonal challenges when looking for the optimal data flow design: operation *sharing* and *reordering* [26]. However, unlike *CoAl*, this approach focuses mainly on the tradeoffs of using these two approaches in the context of data flow optimization and does not study how operation *reordering* can enhance and maximize data and operation sharing among different data flows. Moreover, these approaches are limited to the typical relational algebra operators, while *CoAl* provides a generic framework for the comparison and reordering of arbitrary data flow operations (see Section 2).

Data flow design. The modeling and design of data-intensive flows is a thoroughly studied area, both in the academia [10], [16], [27] and industry, where many tools available in the market often provide overlapping functionalities for the design and execution of these flows. However, neither the research nor the available tools provide the means for automatically adapting data flow designs to changing information requirements.

To the best of our knowledge, the only work tackling the integration of ETL processes is in Albrecht and Naumann [28]. The authors propose a set of high level operators for managing the repository of ETL processes. However, the work lacks the formal definition and automatic means for such operators. Additionally, the authors do not consider the incremental consolidation of data flows led by information requirements.

7 Conclusions and Future Work

We have presented *CoAl*, our approach to facilitate the incremental consolidation of data-intensive flows. *CoAl* starts from data flows that satisfy single information requirements. Iteratively, *CoAl* identifies different possibilities for integrating new data flows into the existing multi-flow, focusing on the maximal data flow reuse. Finally, *CoAl* suggests a unified data flow design evaluating it with the user-specified cost model.

We have developed a prototype that implements the complete functionality of *CoAl*. We used it to evaluate the efficiency, scalability, and the quality of the output solutions of our approach, reporting the improvement of the overall execution time as well as other benefits of integrated multi-flows.

The final goal of our overall work is to provide an end-to-end platform for self-managing the complete lifecycle of BI solutions, from information requirements to deployment and execution of data-intensive flows [20].

References

[1] M. Lenzerini, "Data integration: A theoretical perspective," in *PODS*, 2002, pp. 233–246.

[2] D. Caruso, "Bringing Agility to Business Intelligence," February 2011, Information Management, http://www.information-management.com/infodirect/2009_191/business_intelligence_metadata_analytics_ETL_data_management-10019747-1.html.

[3] R. Hughes, *Agile Data Warehousing: Delivering world-class business intelligence systems using Scrum and XP*. IUniverse, 2008.

[4] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[5] M. Böhm, D. Habich, and W. Lehner, "Multi-flow optimization via horizontal message queue partitioning," in *ICEIS*, 2010, pp. 31–47.

[6] A. Rosenthal and L. J. Seligman, "Data integration in the large: The challenge of reuse," in *VLDB*, 1994, pp. 669–675.

[7] P. Kalnis and D. Papadias, "Multi-query optimization for on-line analytical processing," *Inf. Syst.*, vol. 28, no. 5, pp. 457–473, 2003.

[8] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal, "QoX-driven ETL design: reducing the cost of ETL consulting engagements," in *SIGMOD Conference*, 2009, pp. 953–960.

[9] L. Bellatreche, S. Khouri, and N. Berkani, "Semantic Data Warehouse Design: From ETL to Deployment à la Carte," in *DASFAA (2)*, 2013, pp. 64–83.

[10] Z. E. Akkaoui, E. Zimányi, J.-N. Mazón, and J. Trujillo, "A BPMN-Based Design and Maintenance Framework for ETL Processes," *IJDWM*, vol. 9, no. 3, pp. 46–72, 2013.

[11] O. Romero, A. Simitsis, and A. Abelló, "GEM: Requirement-driven generation of ETL and multidimensional conceptual designs," in *DaWaK*, 2011, pp. 80–95.

[12] P. Jovanovic, A. Simitsis, and K. Wilkinson, "Babbleflow: a translator for analytic data flow programs," in *SIGMOD*, 2014, pp. 713–716.

[13] P. Jovanovic, O. Romero, A. Simitsis, and A. Abelló, "Integrating ETL processes from information requirements," in *DaWaK*, 2012, pp. 65–80.

[14] "TPC-H," last accessed March, 2015, http://www.tpc.org/tpch/.

[15] A. Simitsis, P. Vassiliadis, and T. K. Sellis, "State-Space Optimization of ETL Workflows," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 10, pp. 1404–1419, 2005.

[16] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos, "A generic and customizable framework for the design of ETL scenarios," *Inf. Syst.*, vol. 30, no. 7, pp. 492–525, 2005.

[17] T. Neumann, "Query optimization (in relational databases)," in *Encyclopedia of Database Systems*. Springer US, 2009, pp. 2273–2278.

[18] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *PVLDB*, vol. 5, no. 11, pp. 1256–1267, 2012.

[19] P. Jovanovic, A. Simitsis, and K. Wilkinson, "Engine independence for logical analytic flows," in *ICDE*, 2014, pp. 1060–1071.

[20] P. Jovanovic, O. Romero, A. Simitsis, A. Abelló, H. Candón, and S. Nadal, "Quarry: Digging up the gems of your data treasury," in *EDBT*, 2015, pp. 549–552.

[21] A. Simitsis and K. Wilkinson, "The specification for xLM: an encoding for analytic flows," 2014.

[22] P. Vassiliadis, "A survey of extract-transform-load technology," *IJDWM*, vol. 5, no. 3, pp. 1–27, 2009.

[23] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, "Clio: Schema mapping creation and data exchange," in *Conceptual Modeling: Foundations and Applications*, 2009, pp. 198–236.

[24] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou, "Orchid: Integrating Schema Mapping and ETL," in *ICDE*, 2008, pp. 1307–1316.

[25] M. Arenas, J. Pérez, J. L. Reutter, and C. Riveros, "Foundations of schema mapping management," in *PODS*, 2010, pp. 227–238.

[26] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann, "Shared workload optimization," *PVLDB*, vol. 7, no. 6, pp. 429–440, 2014.

[27] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, "Conceptual modeling for ETL processes," in *DOLAP*, 2002, pp. 14–21.

[28] A. Albrecht and F. Naumann, "METL: Managing and Integrating ETL Processes," in *VLDB PhD Workshop*, 2009.

Petar Jovanovic is a PhD candidate at the MPI research group at Universitat Politècnica de Catalunya inside the IT4BI-DC Erasmus Mundus Joint Doctorate. His research interests mainly fall into the business intelligence field, namely management and optimization of data-intensive flows, data warehouses, and ETL. He has authored publications in renowned international journals (Information Systems), conferences (DaWaK, ICDE, EDBT, SIGMOD) and workshops (DOLAP) on these subjects.

Oscar Romero is a tenure-track lecturer at UPC. He obtained his PhD in Computing in 2010. His research interests concerns data modeling and storage for next generation data warehousing systems, focusing on distributed data management, management and optimization of data intensive flows, semantic-aware information integration and the development of user-centric functionalities for service-oriented BI.

Alkis Simitsis is a senior research scientist in the Analytics Lab at HP Labs. His research career spans multi-objective optimization of hybrid analytics flows, real-time business intelligence, massively parallel processing, query processing/optimization, data warehouses, ETL, web services, and user-friendly query interfaces focusing on keyword search and NLP techniques. He has published more than 90 papers in refereed international journals and conferences in the above areas and has served in various roles in many program committees including SIGMOD, VLDB, ICDE, and EDBT. He is a member of ACM, IEEE, and the Technical Chamber of Greece.

Alberto Abelló is Tenure Track professor. PhD in Informatics, UPC. Local coordinator of the Erasmus Mundus PhD program IT4BI-DC. Active researcher with more than 50 peer-reviewed publications and H-factor 18, his interests include Data Warehousing and OLAP, Ontologies, NOSQL databases and BigData management. He has served as Program Chair of DOLAP and MEDI, being member also of the PC of other database conferences like DaWaK, CIKM, VLDB, etc.