Integrating ETL Processes from Information Requirements

Petar Jovanovic[1], Oscar Romero[1], Alkis Simitsis[2], and Alberto Abelló[1]

[1] Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain ({petar|oromero|aabello}@essi.upc.edu)
[2] HP Labs, Palo Alto, CA, USA (alkis@hp.com)

Abstract. Data warehouse (DW) design is based on a set of requirements expressed as service level agreements (SLAs) and business level objects (BLOs). Populating a DW system from a set of information sources is realized with extract-transform-load (ETL) processes based on SLAs and BLOs. The entire task is complex, time consuming, and hard to be performed manually. This paper presents our approach to the requirement-driven creation of ETL designs. Each requirement is considered separately and a respective ETL design is produced. We propose an incremental method for consolidating these individual designs and creating an ETL design that satisfies all given requirements. Finally, the design produced is sent to an ETL engine for execution. We illustrate our approach through an example based on TPC-H and report on our experimental findings that show the effectiveness and quality of our approach.

1 Introduction

Organizations share their common Data Warehouse (DW) constructs among users of different skills and needs, involved in different parts of the business process. Information requirements coming from such users may consider different analytical perspectives; e.g., Sales is interested in analyzing suppliers data, while Finance analyzes different data like cost or net profit. Complex business models, often make these data intertwined and mutually dependent. Taking into account dynamic enterprise environments with constantly posed information requirements, we need a means for dealing with the complexity of building a complete target schema and supporting extract-transform-load (ETL) process from the early design phases. In addition, due to typical maintenance tasks of such constructs, a great challenge is to provide the designer with the means for dynamic and incremental building of such designs considering real business needs.

In this paper, we focus on ETL design and present our approach to the incremental consolidation of ETL processes, each created to satisfy a single business requirement. For new projects, we create the ETL design from scratch based on a given set of requirements. If an ETL process already exists, we build upon it and extend it according to new or changed requirements.

For these tasks, we propose the *CoAl* algorithm. As 'coal' is formed after the process and extreme compaction of layers of partially decomposed materials (src.

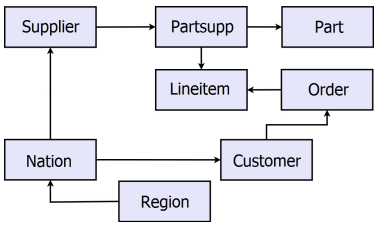

Fig. 1. TPC-H Schema

IR1: For East European nations, the total, revenue of supplied parts.
IR2: For Spanish suppliers, the total revenue of supplied parts.
IR3: For North European nations excluding Norway, the shiped quantity of ordered parts.
IR4: For parts, the shiped quantity ordered by customer "Cust. 1".
IR5: For North European nations excluding Norway, the total revenue of ordered parts.

Fig. 2. Information Requirements

Wikipedia), *CoAl* processes partial ETL designs, each satisfying a single business requirement, and consolidates them into a unified design satisfying the entire set of requirements. The algorithm is flexible and applies various equivalence rules to align the order of ETL operations for finding the appropriate matching part among different input ETL designs. At the same time, it accounts for the cost of ETL designs, searching for near-optimal solutions. At the end, the solution suggested by *CoAl* is sent to an ETL engine for execution. Hence, we provide a novel, end-to-end, requirement-driven solution to the ETL design problem. Our experiments show the effectiveness and usefulness of the proposed method.

Contributions. In particular, our main contributions are as follows.

- We present our approach to the incremental integration of new information requirements into new or existing ETL designs.
- We introduce a novel consolidation algorithm, called *CoAl*, that deals with both structural and content comparison of ETL designs, identifies the maximal matching area among them, and finally, taking into account the cost, produces an ETL design satisfying all requirements.
- We show a set of experiments showing the effectiveness and quality of *CoAl*.

Outline. Section **??** introduces a running example used throughout the paper. Section **??** presents the ETL design consolidation problem, describes equivalence rules, and formalizes operation comparisons. Section **??** presents the *CoAl* algorithm and Section **??** reports on our experimental findings. Finally, Sections **??** and **??** discuss related work and conclude the paper, respectively.

2 Running example

To illustrate our approach, we use a scenario based on the TPC-H schema [**?**]. Figure **??** shows an abstraction of the TPC-H schema. Assuming a set of five requirements (IR1, ..., IR5) over the TPC-H schema, as shown in Figure **??**, we describe how we automatically produce a design that fulfills all five requirements.

First, we create an ETL design for each of these requirements. In the literature there are methods for dealing with such a task (e.g., [**?**]). Having a design per requirement at hand, in this paper, we focus on integrating the individual ETL designs into a design that satisfies all requirements. Considering Figure **??**, we define the *referent* ETL design as the integrated ETL design for a number of requirements already modeled (we start from IR1) and the *new* ETL design as the design for a requirement not integrated yet (IR2). In terms of graphical

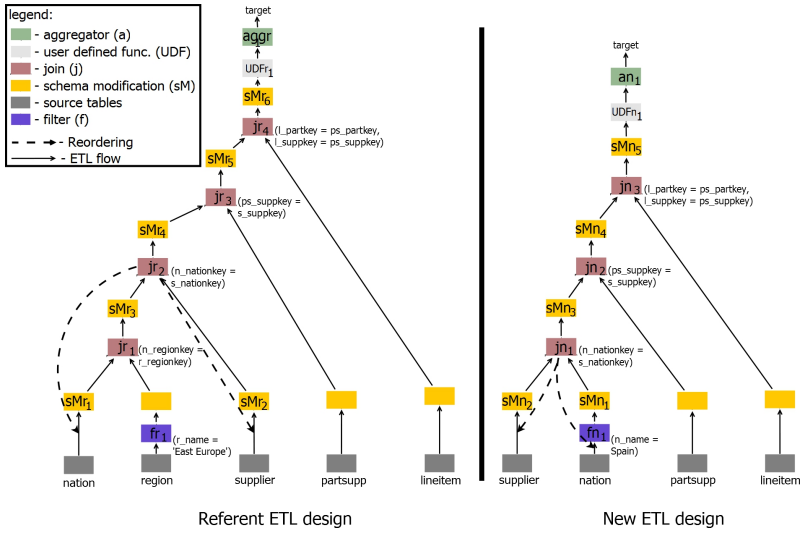

Fig. 3. ETL designs satisfying IR1 and IR2

notation, the gray bottom rectangles represent data sources, whereas the other boxes represent operations. The design for IR1, say G1, contains four join operations, jr_k, k=1...4. The design for IR2, G2, has three joins jn_l, l=1...3. Both designs contain other operations like filters, and so on.

Observe that the two designs have a number of common operations, like for example those on the paths involving the source *nation* (shaded paths in Figure **??**). For both performance and maintainability purposes, we need to create an alternative, equivalent design having the minimum number of overlapping operations. Figure **??** (left) shows an alternative ETL design that satisfies both IR1 and IR2 requirements and the common computation is realized only once.

Once the designs satisfying IR1 and IR2 have been integrated, we iteratively proceed with the remaining requirements, IR3, IR4, and IR5, until we consolidate all five ETL designs into one (assuming that all five designs share operations; otherwise, a design is not merged with the others). For this example, the ETL design that satisfies all five requirements is shown in the right part of Figure **??**.

For the sake of presentation we discuss here functional requirements, but our approach works seamlessly for non-functional requirements too. For example, a requirement regarding availability might lead to a fault-tolerant design that uses replication. Such a design would involve a splitter and a voter operations –to create and merge back the replicas, respectively– whereas the flow fragment between these two operations would have been replicated by a factor –for example– of 3 (triple-modular redundancy). In such cases, the design integration proceeds using the same techniques we describe next by means of the running example.

3 The Design Consolidation Problem

In this section, we describe the problem of consolidating ETL designs satisfying single requirements. We first discuss the challenges that need to be solved

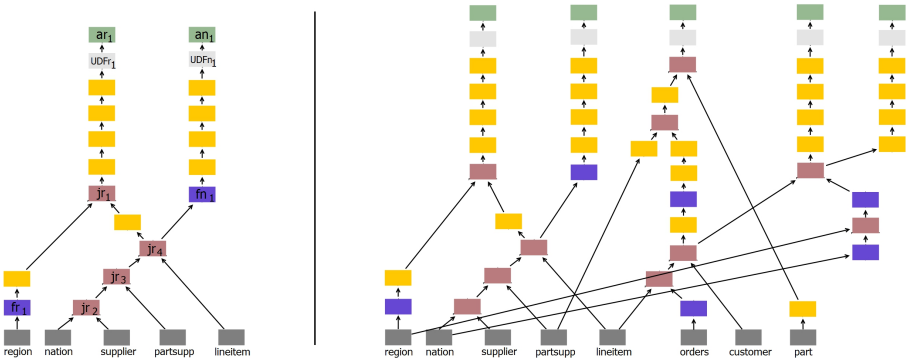

Fig. 4. ETL designs satisfying IR1 and IR2 (left) and all five requirements (right)

and then, we formally present the theoretical underpinnings regarding design equivalence and operation comparisons.

3.1 Goals and Challenges

Typically, an ETL design is modeled as a directed acyclic graph. The nodes of the graph are data stores and operations, and the graph edges represent the data flow among the nodes.

Intuitively, for consolidating two ETL designs, a referent G_1 and a new G_2 designs, we need to identify the maximal overlapping area in G_1 and G_2. Therefore, we proceed as follows:

1. First, we identify the common source nodes between G_1 and G_2. In terms of our running example, let us assume that G_1 satisfies IR1 and G_2 satisfies IR2. The common sources for the two designs are *nation*, *supplier*, *partsupp*, and *lineitem* (see Figure **??**).
2. For each source node, we consider all paths up to a target node and search for common operations in both designs. Starting with the *nation* source node, we identify the paths up to the target in both designs (shaded paths in Figure **??**). In these paths, we search for common operations that could be consolidated into a single operation in the new design.

Deciding which operations can be consolidated and how is not an easy task. If two operations, each placed in a different design, can be matched, then we have a *full match*. For example, the join operation jn_3 in the new design G_2 fully matches the join operation jr_4 in the referent design G_1 (see also Figure **??**). If two operations, one in the referent design and the other in the new design, partially overlap, then we have a *partial match*. For example, if jn_3 involved only predicate $ps\_partkey = l\_partkey$, then the referent design would partially overlap, since in that case jr_4 would be more specific and thus would provide only a subset of the necessary results set.

For being able to guarantee full or partial matching, we should also look at the operations performed before the ones considered for the matching; i.e.,

we need to check the input paths of each operation considered for matching. For example, we cannot consolidate operations jr_2 and jn_1 until we ensure that their predecessors have been already fully matched too. In order to enable better matching, we also consider design restructuring by moving operations before or after those considered for matching. This task is performed by our *CoAl* algorithm described in detail in Section **??**.

Before presenting *CoAl*, we first describe two theoretical aspects that set the foundations of our method. In **??**, we show how to reorder operations within the same design, in order to facilitate the search for full or partial matchings. In **??**, we show how partial and full matches may be identified between operators.

3.2 Equivalence Rules

Reordering operations within an ETL design may be desirable for several reasons; e.g., for improving performance by pushing selective operations early in the flow. Here, we focus on design restructuring with the goal of favoring operation matching between two different designs.

In order to change the structure of a design, we need to ensure that the change, called transition, is valid and leads to a semantically equivalent design. For this reason, all possible transitions obey to a set of equivalence rules, which guarantees the equivalence of designs after a transition has taken place. We consider some of the transitions previously proposed in the context of ETL optimization [**?**]. These transitions include: *swap*, *distribute*, and *factorize*. Swap (swp) interchanges the position of two adjacent unary operations. Factorize (fct) and distribute (dst) represent the factorization and distribution of unary operations over an adjacent n-ary one. It is proven that these transitions are sound and produce equivalent designs, as long as some conditions based on the schemata of operations hold. For example, two unary operations o_1 and o_2 cannot be swapped if o_2 has as a parameter an attribute generated by o_1. For further details and a complete list of these conditions, we refer the interested reader to [**?**]. In addition, we use another two transitions: *association* (asc), which refers to the associativity rule of n-ary operations of the same kind (e.g., joins), and *n-ary distribution* (distr), which refers to the distributive rule between n-ary operators (e.g., join and union), all with their well-known properties.

Table **??** shows the applicability of equivalence rules for an example set of operations (we explain them in **??**). Although, for the sake of presentation, we list here a limited set of operations (as those needed for the running example), the transitions work for a much broader set of operation as discussed in the literature (e.g., [**?**,**?**]). The table reads as follows. For each cell we present how the operation of the column can be rearranged and pushed down the adjacent operation of the row. A tick ($\surd$) means that the unconditional equivalence rule(s) exists between these operations. The additional label(s) besides this symbol refer to which transitions are allowed. If there is a conflict and no equivalence rule can be applied over operations, the cell is crossed ($\times$). Furthermore, in the cases of partial conflicts the cell is marked with ($\sim$) and has an appropriate label. This happens when certain equivalence rules can be applied only if certain conditions

Table 1. Equivalence rules for the running example

oper.	**f**	**sM**	**j**	∪	**a**	**UDF**	**SK**
f	$\surd_{swp}$	$\sim_{swp}$	$\surd_{dst/fct}$	$\surd_{dst/fct}$	$\sim_{swp}$	$\surd_{swp}$	$\surd_{swp}$
sM	$\surd_{swp}$	×	$\surd_{dst/fct}$	$\surd_{dst/fct}$	$\sim_{swp}$	$\surd_{swp}$	$\surd_{swp}$
j	$\surd_{dst/fct}$	$\sim_{dst/fct}$	$\surd_{asc}$	×	$\sim_{dst/fct}$	$\sim_{dst/fct}$	$\sim_{dst/fct}$
∪	$\surd_{dst/fct}$	$\surd_{dst/fct}$	$\surd_{distr}$	$\surd_{asc}$	×	$\surd_{dst/fct}$	$\sim_{dst/fct}$
a	$\sim_{swp}$	$\sim_{swp}$	$\sim_{dst/fct}$	×	×	$\sim_{swp}$	$\sim_{swp}$
UDF	$\sim_{swp}$	$\sim_{swp}$	$\sim_{dst/fct}$	$\surd_{dst/fct}$	$\sim_{swp}$	$\sim_{swp}$	$\sim_{swp}$
SK	$\sim_{swp}$	$\sim_{swp}$	$\sim_{dst/fct}$	$\surd_{dst/fct}$	$\sim_{swp}$	$\sim_{swp}$	×

hold. In all cases, *CoAl* considers only valid transitions based on the equivalence rules.

For example, note that reordering sMr_4 in the referent design of the running example is not allowed, since it projects out *s_suppkey* included in the predicate of jr_3. As another example, a filter and an aggregator can be swapped, assuming that the input schema of filter does not have an attribute contained in the grouping attributes of the aggregate. In addition, it is possible to dst/fct aggregate over join if afterwards the specified set of functional dependencies that ensures the equivalence of such transition holds [**?**]. Other operations behave similarly.

3.3 Operation Comparisons

Next, we describe how we determine whether between two operations, say o_{ref} (placed in the referent design) and o_{new} (placed in the new design) there exists either full or partial or no match.

As we discussed before (see **??**), two operations can be consolidated if they match and if their input data flows also coincide. Even if they do not coincide at first, after finding the matching, either full or partial, we try design restructuring based on the equivalence rules until we meet this condition (if it is possible). We discuss this in the next section. Here, we describe comparison of two operations o_{ref} and o_{new}, without considering their input flows.

Figure **??** illustrates the four possible outcomes of operation comparison.

(1) The compared operations are equal: $o_{ref} = o_{new}$. Then, we consolidate the two operations as a single one in the integrated design.

(2) The results of o_{new} can be obtained from the results of o_{ref}. Then, both operations can be partially collapsed as depicted in Figure **??**. Hence, the output of o_{new} can be computed from the output of o_{ref} (i.e., $o_{ref} \prec o_{new}$) and thus, it partially benefits from the transformations already performed by o_{ref}. Also, the consolidation of the partially matched operation o_{new} may involve a transformation of this operation for obtaining the original output data. For example, if jr_4 in Figure **??** involved only the predicate $ps\_partkey = l\_partkey$ we could then only identify partial matching between jn_3 and jr_4 and the consolidation of these operations would require an extra operation to filter data according to the remaining predicate ($ps\_suppkey = l\_suppkey$).

(3) The results of o_{ref} can be obtained from the results of o_{new} ($o_{new} \prec o_{ref}$).

(4) Finally, it may happen that neither o_{new} can benefit from o_{ref} nor the opposite. Then, the two operations cannot be consolidated. In such cases, we use

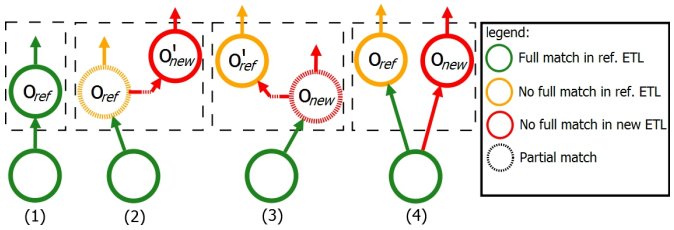

Fig. 5. Integration of the operations

a *fork* in the already matched ETL subset, as shown in Figure **??**(4). (Note that the fork is implemented as a copy-partitioning operation in the physical design.)

In general, each operation is characterized by its *input* (I) and *output* (O) schemata (see also [**?**]). We also consider the *semantics* (S) involved in the computation performed by this operation. To express the wide complexity of ETL flows we can define the semantics of their operations as *programs* with corresponding *precondition* (Pre) and *postcondition* (Post) predicates. Accordingly, we can formally represent an ETL operation o as $\mathbf{o}(I, O, S, Pre, Post)$.

- $\mathbf{o}_1(I_1, O_1, S_1, Pre_1, Post_1) = \mathbf{o}_2(I_2, O_2, S_2, Pre_2, Post_2)$ iff $I_1 = I_2 \wedge O_1 = O_2 \wedge Pre_1 \equiv Pre_2 \wedge Post_1 \equiv Post_2$;
- $\mathbf{o}_1(I_1, O_1, S_1, Pre_1, Post_1) \prec \mathbf{o}_2(I_2, O_2, S_2, Pre_2, Post_2)$ iff $\exists \mathbf{o}_3(I_3, O_3, S_3, Pre_3, Post_3) : I_1 = I_2 \wedge O_1 = I_3 \wedge O_3 = O_2 \wedge Pre_1 \equiv Pre_2 \wedge Post_3 \equiv Post_2 \wedge Post_1 \Rightarrow Pre_3$;

The definition of a specific operation semantics (S) along with the corresponding postconditions and preconditions predicates and their implications ($\Rightarrow$) are provided by template definitions for operations. Hence, any given ETL design uses instances of these operations that inherit such properties from their generic template definitions. This process is straightforward and a detailed discussion on this topic falls out of the scope of this paper.

Next, we show formal definitions for operation comparison for the operations shown in Table **??**. Due to space considerations, we do not elaborate here on other operations, but the process is similar.

- **Filter** - $\mathbf{f}_{\psi}(R)$
  For comparing filter operations, besides the input schema, we also check the comparison of included predicates ψ. The comparison of the equivalence or logical implication of these predicates ($\psi_1 \Leftarrow \psi_2$) can be facilitated by generic reasoners. We compare filter operations as follows:
  - $\mathbf{f}_{\psi_1}(R) = \mathbf{f}_{\psi_2}(S)$ iff R=S $\wedge$ $\psi_1 \equiv \psi_2$;
  - $\mathbf{f}_{\psi_1}(R) \prec \mathbf{f}_{\psi_2}(S)$ iff R=S $\wedge$ $\psi_1 \Leftarrow \psi_2$;
- **Schema Modification** - $\mathbf{sM}_{a1,a2,..,an}(R)$
  For comparing schema modifications, besides the input relations, we also compare the attributes that are modified. Therefore, we compare schema modification operations as follows:
  - $\mathbf{sM}_{a1,..,an}(R) = \mathbf{sM}_{b1,..,bm}(S)$ iff R=S $\wedge$ {a1,..,an} = {b1,..,bm};
  - $\mathbf{sM}_{a1,..,an}(R) \prec \mathbf{sM}_{b1,..,bm}(S)$ iff R=S $\wedge$ {a1,..,an} $\supset$ {b1,..,bm};

- **Join** - $R \ \mathbf{j}_{\psi} \ S$
  To compare joins, we take into account the commutative property that applies over the inputs of a join. As with filter, we compare the corresponding join predicates. Thus, we compare joins as follows:
  - $P \ \mathbf{j}_{\psi_1} \ Q = R \ \mathbf{j}_{\psi_2} \ S \ iff$ ((P=R $\wedge$ Q=S) $\vee$ (P=S $\wedge$ Q=R)) $\wedge \ \psi_1 \equiv \psi_2$;
  - $P \ \mathbf{j}_{\psi_1} \ Q \prec R \ \mathbf{j}_{\psi_2} \ S \ iff$ ((P=R $\wedge$ Q=S) $\vee$ (P=S $\wedge$ Q=R)) $\wedge \ \psi_1 \Leftarrow \psi_2$;
- **Union** - R $\cup$ S
  To compare unions, we only compare their input relations, as they do not have any additional parameters defined. Here, we also consider the commutative property. Thus, we compare unions as follows:
  - $P \cup Q = R \cup S \ iff$ ((P=R $\wedge$ Q=S) $\vee$ (P=S $\wedge$ Q=R));
- **Aggregator** - ${}_{g1,..,gm}\mathbf{a}f1(A1'),..,fk(Ak') \ (R)$
  To compare aggregators, besides the input relations, we also compare the grouping attributes regarding equality or functional dependency between them ($g_i \rightarrow t_i$). Currently, we consider the set of aggregation functions to be equal. However, this can be extended considering the class of expandable aggregation functions, discussed in [**?**]. Thus, we compare them as follows:
  - ${}_{g1,..,gn}\mathbf{a}... \ (R) = {}_{t1,..,tm}\mathbf{a}... \ (S) \ iff$ R=S $\wedge$ m=n $\wedge \ \forall i = 1..m, g_i{=}t_i$;
  - ${}_{g1,..,gn}\mathbf{a}... \ (R) \prec {}_{t1,..,tm}\mathbf{a}... \ (S) \ iff$ R=S $\wedge$ m$\leq$n $\wedge \ \forall i = 1..m, g_i = t_i \vee g_i{\rightarrow}t_i$;
- **User Defined Function (UDF) - UDF**(R)
  A udf is expressed as ${}_{f:I\rightarrow O}\mathbf{o}(R)$. For comparing udfs we consider their behavior over the input records (r), as follows:
  - ${}_{f_1:I_1\rightarrow O_1} \ \mathbf{o}_1$(R) $=$ ${}_{f_2:I_2\rightarrow O_2} \ \mathbf{o}_2$(S) $\ iff \ R = S \wedge \forall r \in R : f_1(r) = f_2(r)$;
  - ${}_{f_1:I_1\rightarrow O_1} \ \mathbf{o}_1$(R) $\prec$ ${}_{f_2:I_2\rightarrow O_2} \ \mathbf{o}_2$(S) $\ iff \ \exists_{f_3:I_3\rightarrow O_3} \ \mathbf{o}_3$(P):$R = S \wedge O_1 = I_3 \wedge O_3 = O_2 \wedge \forall r \in R : f_3(f_1(r)) = f_2(r)$;
- **Surrogate Key Assignment (SK) - SK**(R, S)
  Surrogate key assignment (SK) is a typical ETL operation that joins the incoming data (R) with a lookup dimension table (S) and replaces the pair "source of data, primary key"$(value)$ with a unique identifier for DW called "surrogate key"(sk). If a surrogate key does not exist for the pair "source, primary key", then a new surrogate key is generated, typically by a function producing values like $max(SK) + 1$. The comparison is as follows:
  - $\mathbf{SK}(R_1, S_1) = \mathbf{SK}(R_2, S_2) \ iff \ R_1 = R_2 \wedge (\forall t \in R_1 : (\exists t' \in S_1, t[value] = t'[value] \wedge \exists t'' \in S_2, t[value] = t''[value]) \rightarrow t'[sk] = t''[sk])$;

  Due to specific semantics of the SK transformations, the above comparison does not actually test equality of two SK transformations, but their ability to be consolidated. Therefore, we define that two SK transformations can be consolidated iff there is no conflict between their lookup tables, i.e., iff the SK values can be found either in one or in none of the tables.

4 Consolidation Algorithm

The *CoAl* algorithm looks for all matching opportunities between operators from the referent and new designs and at the end, it produces a consolidated design.

For each source node, we explore its paths up to a target following a *topological order* of the nodes in the design. At each iteration of the algorithm, we only match two operations (one from each design), and we only add this match to the final result if and only if all previous nodes in the path have been fully matched. To ensure this, we proceed using the equivalence rules between the operations at hand and taking into account the performance cost of the design.

Returning to the running example that shows how we consolidate the designs for IR1 and IR2 (see figure **??**), we identify a full match between jr_2 and jn_1 operations. However, their input paths have not been fully matched yet. One solution to handle this is to check if we can push jr_2 and jn_1 down to their respective sources (*supplier* and *nation*). This is possible because on the one side jr_2 may move over sMr_2, sMr_3, jr_1, and sMr_1, while on the other side operation jn_1 may move over sMn_1, sMn_2, and fn_1, and we may still produce equivalent designs that fulfill our constraint.

As we discussed, only when a full match is not possible (either directly or after reordering of operations), we search for a partial match. Partial matches finish our exploration in the considered branch, as we do not fulfill our constraint: fully match of the input paths is required in order to keep exploring the branch. Hence, if at the end a complete match is not found –i.e., the new ETL cannot be completely subsumed by the referent ETL– we explore the partial matchings identified and estimate their costs. The cheapest solution according to the cost model considered (discussed later in this section) is chosen for integration.

Formally, *CoAl* starts with two ETL designs, the referent and the new, and iterates following a topological order of the ETL operations, which guarantees the following two invariants:

(I_1): At each iteration, only one pair of operations can be partially or fully matched.
(I_2): A new match is added to the set of already matched operations iff the input flows of the operations involved in the new match have been fully matched in previous iterations.

These invariants have some interesting consequences. Two matched operations are eventually consolidated in the output, integrated design if the designs they belong to can be reordered so that their children are fully matched. Since we are looking for the maximal overlapping area between the two designs, we can guarantee that any two operations that fully match can be immediately added to the output. The proof based on contradiction is straightforward.

Suppose that o_{ref1} and o_{ref2} are two operations from the referent design and o_{new1} and o_{new2} are two operations from the new design. Let us assume that o_{ref1} fully matches with o_{new1}, from now $pair_1$, and o_{ref2} fully matches with o_{new2}, from now $pair_2$. Both belong to the maximal overlapping area between both designs and there is no other full match left to identify. If the order to add them to the output matters, it means that one of these pairs, say $pair_1$, should be added to the result before $pair_2$. But this can only happen if no equivalence rules can be applied between the corresponding operators in each design (i.e., between o_{ref1} and o_{ref2} in the referent design and between o_{new1}

inputs: G_1, G_2, **output:** G_{int}

1. matchingsQueue := matchLeafs(G_1, G_2);
2. alternativeList := $\emptyset$;
3. **while** (*matchingsQueue* is not empty) **do**
   (a) currentMatchings($G_1\prime$,$G_2\prime$) := dequeue(matchingsQueue);
   (b) newOperationsForMatching(LOps_ref, LOps_new) := explore($G_1\prime$,$G_2\prime$);
   (c) **if** (LOps_new is empty) **then**
      i. insert($G_1\prime$,$G_2\prime$, $\emptyset$, 0) into alternativeList;
   (d) **foreach** pair(O_{new} from LOps_new, O_{ref} from LOps_ref)
      i. **if** (O_{new} fully matches O_{ref}) **then**
         A. $G_1\prime\prime$ =: reorder($G_1\prime$); $G_2\prime\prime$ =: reorder($G_2\prime$);
         B. enqueue(matchingsQueue, {$G_1\prime\prime$,$G_2\prime\prime$}, {O_{new},O_{ref}});
      ii. **else if** (O_{new} partially matches O_{ref}) **then**
         A. $G_1\prime\prime$ =: reorder($G_1\prime$); $G_2\prime\prime$ =: reorder($G_2\prime$);
         B. insert($G_1\prime\prime$, $G_2\prime\prime$, {O_{new}, O_{ref}}, Cost($G_1\prime\prime$, $G_2\prime\prime$)) into alternativeList ;
   (e) **if** (no matching found) **then**
      i. insert($G_1\prime$, $G_2\prime$, $\emptyset$, Cost($G_1\prime$, $G_2\prime$)) into alternativeList;
4. **if** (alternativeList is not empty)
   (a) G_{int} := integrate(cheapestAlternativeMatching);
5. **Return** G_{int};

Fig. 6. Pseudocode for *CoAl*

and o_{new2} in the new design). In such a case, and knowing that they belong to the maximal overlapping area, $pair_1$ can be moved according to the equivalence rules down in both designs, so that all their input data flows are fully matched and consequently, they will be added to the output in the next iteration. After $pair_1$ has been integrated, we use another finite set of equivalence rules for pushing $pair_2$ down and fulfill (I_2). Thus, in the next iteration it will be also added to the output. Relevantly, this is also the proof that our algorithm will eventually finish. Due to this property we define rule R_1:

(R_1): When looking for matchings, the first two operations to be compared from the referent and the new designs are those that fulfill (I_2).

Although it favors the cheapest solutions (i.e., that do not require any reordering), R_1 nevertheless does not eliminate better solutions that may appear.

CoAl comprises four steps (see Figure **??**): i) *search for the next operations to match*; ii) *compare the next operations*; iii) *reorder input designs* if a match has been found; and iv) *integrate the alternative matching* with the lowest estimated cost. The three first are executed in each iteration of the algorithm, whereas the last one is executed only once, when no match is pending.

Through the algorithm we maintain two structures. First, a priority queue that contains fully matched areas that may be further extended with new matching operations. Each queue element contains the list of matching operations together with the input ETL structures specifically reordered for such matchings. Second, a list for keeping all alternative matching combinations ending up in a partial matching found through the algorithm, along with the estimated costs of such matchings. The algorithm starts by matching the source nodes of the referent and the new designs (step **??**). The comparison of the source nodes is based on the parameters that characterize them: source type, source name, and extracted fields. The steps of *CoAl* are as follows.

Search for the next operations to match. We identify the operations to be compared next (step **??**). We start with comparing operations according to (R_1). If there is no full match, the algorithm identifies all operations that can be

reordered, by applying equivalence rules, and pushes them down according to (I_2) and hence it identifies different possibilities for comparing. As a result, two sets of operations to be compared (LOps_new and LOps_ref) are produced. In terms of the running example, for the paths starting from the matching source *nation* (see Figure **??**), in the referent design we identify the set: (jr_1, jr_2, and sMr_1) and in the new design: (jn_1, sMn_1, and fn_1).

Compare the next operations. We then produce the cartesian product of these two sets (step **??**). For each pair, we proceed as explained in subsection **??** depending on the result of the comparison: (a) we can identify a *full match* (equality) (step **??**); (b) a *partial match* (step **??**) or (c) *no match* (step **??**).

Reorder the input designs. If *CoAl* finds a (full or partial) match between two operations, then it tries reordering of the input designs to guarantee (I_2) (steps **??** and **??**). Considering the running example, when we find a full match between joins jr_2 and jn_1, the algorithm pushes them down to the sources *nation* and *supplier*. *CoAl* then adds the match found to the integrated design and depending on the type of match found it proceeds as follows.

- For a full match, it enqueues back to priority queue the two designs (possibly reordered) to further extend the matching in next iterations (step **??**).
- For a partial match, it estimates the cost of such a solution and then adds it, along with its cost, to the list of integration alternatives (step **??**).
- Finally, if there is no match, this alternative, along with its estimated cost, is also added to the list of potential integration alternatives (step **??**).

Regarding a cost estimation model, *CoAl* is not tied to a specific cost model; in fact, it is extensible to any given cost model. Example cost models for ETL designs can be found in the literature (e.g., [**?**]).

The matching process ends when the algorithm finishes with all possible paths and the comparison among their operations (i.e., when there are no more elements in the priority queue). Alternatively, the algorithm terminates when a complete matching of the new design is identified (step **??**). This extreme case happens only when the new design is completely subsumed by the referent one and thus the cost of such an alternative (i.e., the referent design itself) is 0.

Integrate an alternative match. After the iterations finish, *CoAl* checks if there is any alternative matching. It checks the list of all possible alternatives and chooses the one with the lowest estimated cost. Then, it continues the design consolidation with that alternative (step **??**).

Finally, *CoAl* returns the consolidated design.

5 Evaluation

This section describes our prototype and reports on our experimental findings.

CoAl* in *GEM. Our work revolves around *GEM*, which is a prototype for the creation of multidimensional (MD) schemata and the respective ETL design based on a given set of business requirements. In a nutshell, starting from a set of requirements expressed in a proprietary XML-like form, we semi-automatically

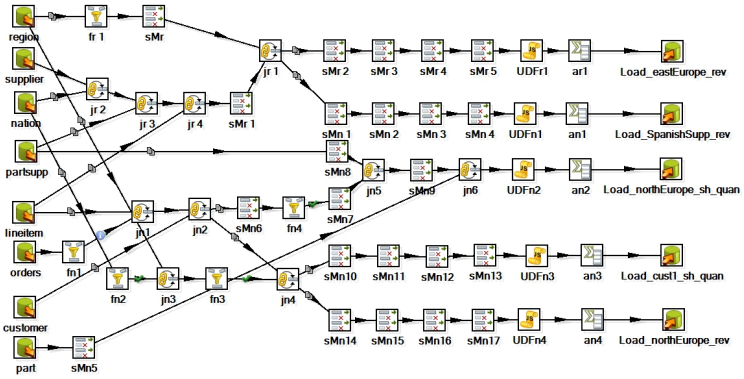

Fig. 7. Physical ETL design satisfying (IR1 - IR5)

construct the resulting MD schemata and ETL designs using Semantic Web technology for inferring the necessary mappings [**?**]. The outcome of this process is a conceptual MD and an ETL designs. The conceptual ETL design is encoded in an XML-like format, namely xLM, previously proposed in [**?**]. Our method produces one ETL design per business requirement and then, we use *CoAl* for consolidating the results into a unified ETL process.

As a next step, our prototype translates the conceptual ETL design into a physiological ETL model, expressed again in xLM, which then may be executed in an ETL engine. Figure **??** shows a physical rendition for the running example. For now, *GEM* is connected to an open source ETL engine (PDI, a.k.a. Kettle [**?**]). For the connection, we translate xLM to the engine-specific XML form for storing ETL metadata, and thus, we are able to import a design into the engine and execute it. Our design choice of use an XML-like encoding was made for achieving a greater extensibility, since many modern ETL engines use XML encoding to import/export ETL metadata. Thus, *GEM* may connect to any of them, assuming that the correct XSLT to tool-specific XML parser is provided.

Experimental Methodology. We constructed designs based on the TPC-H [**?**] schema and queries (information requirements). We first used *GEM* to build designs corresponding to individual requirements and then, we launched *CoAl* to consolidate these designs. We considered all order permutations of the provided designs. Here, due to space considerations, we present our representative results for six TPC-H queries: Q3, Q5, Q7, Q8, Q9, and Q10. For each permutation, we first started by consolidating two requirements, and then we incrementally added the other four.

Scrutinizing *CoAl*. Next we report on our experimental findings.

Search Space. As shown in Figure **??**(right), for a naive search, the search space grows with the ♯requirements. For input designs of an average size of 28 operations, the number of states considered starts from 1.2k for 2 requirements and go up to 9.7k for 6 requirements. At the same time, the time needed to complete the search grows exponentially with the ♯requirements –see Figure

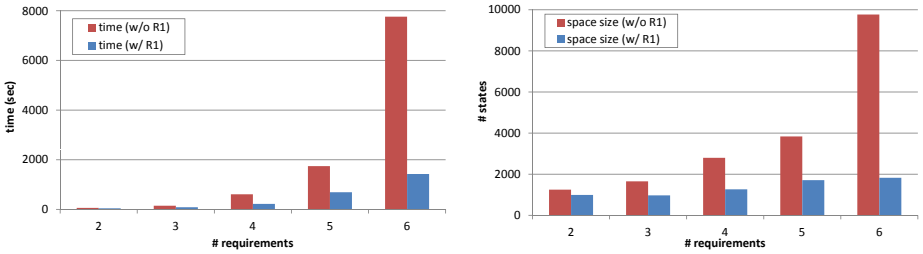

Fig. 8. Search space exploration

??(left)– starting from 60sec for 2 requirements and go up to 7.7ksec for 6 requirements. Hence, it is obvious that we need to prune the state space.

For that, we used the (R_1) rule (see Section **??**). For evaluating the effectiveness of (R_1), we performed the same set of experiments with and without the rule. This is shown in Figure **??**: the red bars represent the naive search and the blue bar shows the results of applying the (R_1) rule into our search. The improvement is obvious in terms of both space and time.

As another experiment, we studied the behavior of the internal characteristics of *CoAl*. Figure **??** shows how ♯matches, ♯maxTransitions (this relates to the (I_2) invariant), ♯firstMatches (R_1-effect), and ♯solutions are affected by the size of the problem. While the number of matches increases with the number of requirements, both the numbers of solutions and reorganizations drop as we encounter additional requirements. At first, *CoAl* aggressively matches different designs, but as the incrementally integrated design matures and the design space is covered, there are lesser novel, valid moves. This is also verified by the ♯visited states (not shown in the graph) that increases with the ♯requirements.

Quality of our solutions. Figure **??** presents our findings regarding the quality of solutions provided by *CoAl* with respect to optimal designs, which were manually constructed. Figure **??**(left) shows a comparison based on a combination of design metrics that measure the coverage of the optimal cases by the respective designs. Interestingly, the quality of *CoAl* increases with the ♯requirements. Figure **??**(right) reports on an individual metric, namely ♯operations. In all cases, the ♯operations in designs produced by *CoAl*, follows the same pattern as in the respective optimal cases. Moreover, *CoAl* matches all data stores (not shown in the figure). It is worth noting that the time needed for finding the optimal case was 3-4x larger than the respective time needed for getting the designs automatically. In addition, *CoAl* produces equivalent designs; i.e., designs that when executed produce the exact same results.

6 Related Work

ETL. Previous work on ETL has studied modeling and optimization issues. Regarding modeling, there are two directions: the use of ad-hoc formalisms (e.g., [**?**]) and standard modeling languages (e.g., [**?**,**?**,**?**]). These approaches do not describe how the ETL design adapts to change of requirements. Past work has also

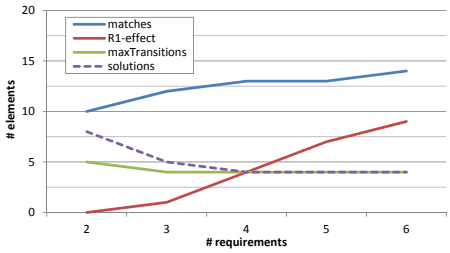

Fig. 9. *CoAl* characteristics

tackled the problem of optimizing ETL designs for a variety of objectives (e.g., performance, fault-tolerance, etc.) without showing how to deal with business requirements [**?**,**?**].

Query optimization. Both traditional query optimization [**?**] and multi-query optimization approaches [**?**] focus on performance and consider a different subset of operations than those typically encountered in ETL. Also, database optimizers do not work well for operations with 'black-box' semantics [**?**]. Our equivalence rules, however, are based on transitions that have been proved to work for a wider range of operations [**?**](e.g., arbitrary user functions, data mining transformations, cleansing operations, etc.).

Data mappings and data exchange. Data mapping specifications aim at bridging the heterogeneities between source and target schemas by *mapping* the relationships between schemas [**?**]. The data exchange problem aims at restructuring data structured under one source schema in terms of a given target schema [**?**]. However, current algorithms and tools generating automatic data mappings (e.g., [**?**,**?**,**?**]) either cannot tackle grouping and aggregation or overlook complex transformations like those with black-box semantics.

7 Conclusions

We have presented *CoAl*, our approach to facilitate the incremental consolidation of ETL designs based on business requirements. *CoAl* identifies different possibilities for consolidation and suggests near-optimal designs taking into account their processing cost too. Our method can be used either at the early stages of an ETL project for creating the ETL design or at later stages, to facilitate the burdensome process of adapting an ETL design to evolving requirements. *CoAl* is integrated in our prototype tool called *GEM*, which connects to an ETL engine for the actual execution of the produced designs. Our experiments show that *CoAl* successfully automates the design process, a task that is largely infeasible to be performed manually in a timely fashion.

Our future plans include the optimization of our method by exploiting heuristics based on the observation of past execution results for a variety of designs.

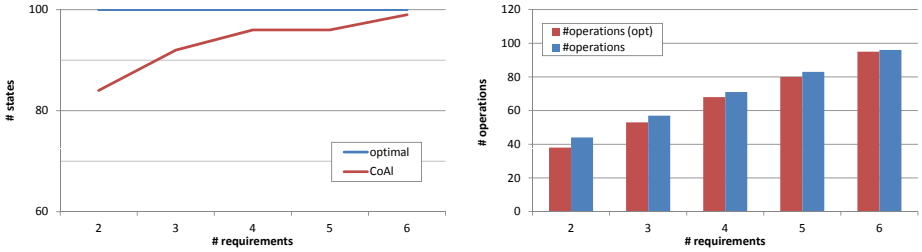

Fig. 10. Quality of *CoAl*