Optimizing ETL Processes in Data Warehouses

Alkis Simitsis
Nat. Tech. Univ. of Athens
asimi@dbnet.ece.ntua.gr

Panos Vassiliadis
University of Ioannina
pvassil@cs.uoi.gr

Timos Sellis
Nat. Tech. Univ. of Athens
timos@dbnet.ece.ntua.gr

Abstract

Extraction-Transformation-Loading (ETL) tools are pieces of software responsible for the extraction of data from several sources, their cleansing, customization and insertion into a data warehouse. Usually, these processes must be completed in a certain time window; thus, it is necessary to optimize their execution time. In this paper, we delve into the logical optimization of ETL processes, modeling it as a state-space search problem. We consider each ETL workflow as a state and fabricate the state space through a set of correct state transitions. Moreover, we provide algorithms towards the minimization of the execution cost of an ETL workflow.

1. Introduction

For quite a long time in the past, research has treated data warehouses as collections of materialized views. Although this abstraction is elegant and possibly sufficient for the purpose of examining alternative strategies for view maintenance, it is not enough with respect to mechanisms that are employed in real-world settings. Indeed, in real-world data warehouse environments, instead of automated mechanisms for the refreshment of materialized views, the execution of operational processes is employed in order to export data from operational data sources, transform them into the format of the target tables and finally, load them to the data warehouse. The category of tools that are responsible for this task is generally called *Extraction-Transformation-Loading* (ETL) tools. The functionality of these tools can be coarsely summarized in the following prominent tasks, which include: (a) the identification of relevant information at the source side; (b) the extraction of this information; (c) the customization and integration of the information coming from multiple sources into a common format; (d) the cleaning of the resulting data set, on the basis of database and business rules, and (e) the propagation of the data to the data warehouse and/or data marts.

So far, research has only partially dealt with the problem of designing and managing ETL workflows. Typically, research approaches concern (a) the optimization of stand-alone problems (e.g., the problem of duplicate detection [16]) in an isolated setting and (b) problems mostly related to web data (e.g., [7]). Recently, research on data streams [1], [2] has brought up the possibility of giving an alternative look to the problem of ETL. Nevertheless, for the moment research in data streaming has focused on different topics, such as on-the-fly computation of queries [1], [2]. To our knowledge, there is no systematic treatment of the problem, as far as the problem of the design of an optimal ETL workflow is concerned.

On the other hand, leading commercial tools [9], [10], [13], [14] allow the design of ETL workflows, but do not use any optimization technique. The designed workflows are propagated to the DBMS for execution; thus, the DBMS undertakes the task of optimization. Clearly, we can do better than this, because, an ETL process cannot be considered as a "big" query. Instead, it is more realistic to treat an ETL process as a complex transaction. In addition, in an ETL workflow, there are processes that run in separate environments, usually not simultaneously and under time constraints.

One could argue that we can possibly express all ETL operations in terms of relational algebra and then optimize the resulting expression as usual. In this paper we demonstrate that the traditional logic-based algebraic query optimization can be blocked, basically due to the existence of data manipulation functions. Consider the example of Fig. 1 that describes the population of a table of a data warehouse `DW` from two source databases `S1` and `S2`. In particular, it involves the propagation of data from the recordset `PARTS1(PKEY,SOURCE,DATE,COST)` of source `S1` that stores monthly information, as well as from the recordset `PARTS2(PKEY,SOURCE,DATE,DEPT, COST)` of source `S2` that stores daily information. In the `DW`, `PARTS(PKEY,SOURCE,DATE,COST)` stores monthly information for the cost in Euros `(COST)` of parts `(PKEY)` per source `(SOURCE)`. We assume that both the first supplier and the data warehouse are European and the second is American; thus, the data

coming from the second source need to be converted to European values and formats.

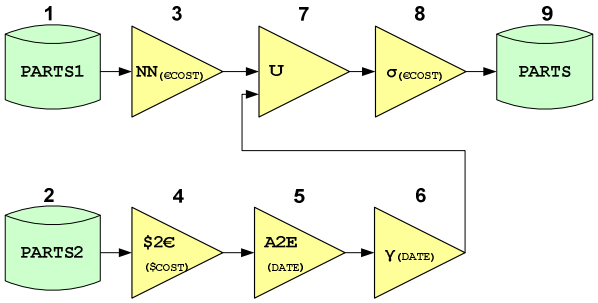


Figure 1. A simple ETL workflow

In Fig. 1, activities are numbered with their execution priority and tagged with the description of their functionality. The flow for source `S1` is: (3) a check for `Not Null` values is performed on attribute `COST`. The flow for source `S2` is: (4) Dollar costs (`$COST`) are converted to Euros (`€COST`); (5) dates (`DATE`) are converted from American to European format; (6) an aggregation for monthly supplies is performed and the unnecessary attribute `DEPT` (for department) is discarded from the flow. The two flows are then unified (7) and before being loaded to the warehouse, a final check is performed on the `€COST` attribute (8), ensuring that only values above a certain threshold are propagated to the warehouse.

There are several interesting problems and optimization opportunities in the example of Fig. 1:

- Traditional query optimization techniques should be directly applicable. For example, it is desirable to push selections all the way to the sources, in order to avoid processing unnecessary rows.
- Is it possible to push the selection for values above a certain threshold early enough in the workflow? As far as the flow for source `PARTS1` is concerned, this is straightforward (exactly as in the relational sense). On the other hand, as far as the second flow is concerned, the selection should be performed after the conversion of dollars to Euros. In other words, the activity performing the selection *cannot* be pushed before the activity applying the conversion function.
- Is it possible to perform the aggregation, before the transformation of American values to Europeans? In principle, this should be allowed to happen, since the dates are kept in the resulting data and can be transformed later. In this case, the aggregation operations *can be* pushed, before the function (as opposed to the previous case).
- How can we deal with naming problems? `PARTS1.COST` and `PARTS2.COST` are homonyms, but they do not correspond to the same entity (the first is in Euros and the second in Dollars). Assuming that the transformation `$2€` produces the attribute `€COST`, how can we guarantee that corresponds to the same real-world entity with `PARTS1.COST`?

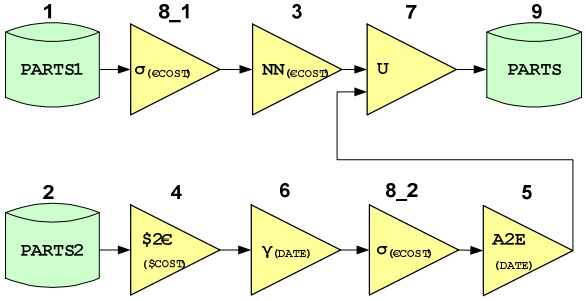


Figure 2. An equivalent ETL workflow

In Fig. 2, we can see how the workflow of Fig. 1 can be transformed in an equivalent workflow performing the same task. The selection on Euros has been propagated to both branches of the workflow so that low values are pruned early. Still, we cannot push the selection, neither before the transformation `$2€`, nor before the aggregation. At the same time, there was a swapping between the aggregation and the `DATE` conversion function (`A2E`). In summary, the two main problems that can be highlighted in this setting are (a) to determine which operations over the workflow are legal and (b) to determine the best workflow configuration in terms of performance gains.

We take a novel approach to the problem by taking into consideration the aforementioned peculiarities. Moreover, we employ a workflow paradigm for the modeling of ETL processes, i.e., we do not strictly require that an activity outputs data to some persistent data store, but rather, activities are allowed to output data to one another. In such a context, I/O minimization is not the primary problem. In this paper, we focus on the optimization of the process in terms of logical transformations of the workflow. To this end, we devise a method based on the specifics of an ETL workflow that can reduce its execution cost by changing either the total number or the execution order of the processes. Our contributions can be listed as follows:

- We set up the theoretical framework for the problem, by modeling the problem as a state space search problem, with each state representing a particular design of the workflow as a graph. The nodes of the graph represent activities and data

stores and the edges capture the flow of data among the nodes.
- Since the problem is modeled as a state space search problem, we define transitions from one state to another that extend the traditional query optimization techniques. We prove the correctness of the introduced transitions. We also provide details on how states are generated and the conditions under which transitions are allowed.
- Finally, we provide algorithms towards the optimization of ETL processes. First, we use an exhaustive algorithm to explore the search space in its entirety and to find the optimal ETL workflow. Then we introduce greedy and heuristic search algorithms to reduce the search space that we explore, and demonstrate the efficiency of the approach through a set of experimental results.

The rest of this paper is organized as follows. Section 2 presents a formal statement for our problem as a state space search problem. In Section 3 we discuss design issues and correctness of our setting. In Section 4, we present algorithms for the optimization the ETL processes, along with experimental results. In Section 5 we present related work. Finally, in Section 6 we conclude with our results and discuss topics of future research. A long version of the paper, with all the proofs is found at [19].

2. Formal Statement of the problem

In this section, we show how the ETL optimization problem can be modeled as a state space search problem. First, we give a formal definition of the constituents of an ETL workflow and we describe the states. Then, we define a set of transitions that can be applied to the states in order to produce the search space. Finally, we formulate the problem of the optimization of an ETL workflow.

2.1. Formal definition of an ETL workflow

An ETL workflow is modeled as a directed acyclic graph. The nodes of the graph comprise *activities* and *recordsets*. A recordset is any data store that can provide a flat record schema (possibly through a gateway/wrapper interface); in the rest of this paper, we will mainly deal with the two most popular types of recordsets, namely relational tables and record files. The edges of the graph denote *data provider* (or *input/output*) *relationships*: an edge going out of a node n_1 and into a node n_2 denotes that n_2 receives data from n_1 for further processing. In this setting, we will refer to n_1 as the *data provider* and n_2 as the *data consumer*. The graph uniformly models situations where (a) both providers are activities (combined in a pipelined fashion) or (b) activities interact with recordsets, either as data providers or data consumers.

Each node is characterized by one or more *schemata*, i.e., finite lists of *attributes*. Whenever a schema is acting as a data provider for another schema, we assume a one-to-many mapping between the attributes of the two schemata (i.e., one provider attribute can possibly populate more than one consumers while a consumer attribute can only have one provider). Recordsets have only one schema, whereas activities have at least two (input and output). Intuitively, an activity comprises a set of *input schemata*, responsible for bringing the records to the activity for processing and one or more *output schemata* responsible for pushing the data to the next data consumer (activity or recordset). An activity with one input schema is called *unary*, and an activity with two input schemata is called *binary*.

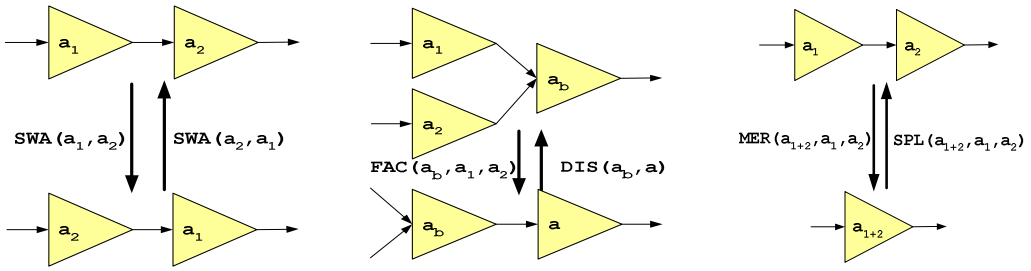
Formally, an *activity* is a quadruple `A=(Id,I,O,S)`, such that: (a) *Id* is a unique identifier for the activity; (b) **`I`** is a finite set of one or more input schemata, receiving data from the data providers of the activity; (c) **`O`** is a finite set of one or more output schemata that describe the placeholders for the rows that are processed by the activity; and (d) `S` is one or more expressions in relational algebra (extended with functions) characterizing the semantics of the data flow for each of the output schemata. This can be one expression per output schema or a more complex expression involving intermediate results too.

In our approach, we will model an ETL workflow as a graph. Assume a finite list of activities **`A`**, a finite set of recordsets **`RS`**, and a finite list of provider relationships **`Pr`**.

Formally, an *ETL Workflow* is a directed acyclic graph (DAG), **`G(V,E)`** such that **`V`**=**`A`**∪**`RS`** and **`E`**=**`Pr`**.

A subset of **`RS`**, denoted by $\mathbf{RS_S}$, contains the sources of the graph (i.e., the source recordsets) and another subset of **`RS`**, denoted by $\mathbf{RS_T}$, contains the sinks of the graph (representing the final target recordsets of the warehouse). **`G(V,E)`** can be topologically ordered, therefore a unique *execution priority* can be assigned to each activity as its unique identifier.

Finally, all activities of the workflow should have a provider and a consumer (either another activity or a recordset). Each input schema has exactly one provider (many providers for the same consumer are captured by `UNION` activities).



(a) Swap (b) Factorize and Distribute (c) Merge and Split

Figure 3. Abstract examples of transitions

2.2. The problem of ETL workflow optimization

We model the problem of ETL optimization as a state space search problem.

States. Each state S is a graph as described in Section 2.1, i.e. states are ETL workflows; therefore, we will use the terms 'state' and 'ETL workflow' interchangeably.

Transitions. Transitions T are used to generate new, equivalent states. In our context, equivalent states are assumed to be states that based on the same input produce the same output. Practically, this is achieved in the following way:

(a) by transforming the execution sequence of the activities of the state, i.e., by interchanging two activities of the workflow in terms of their execution sequence;

(b) by replacing common tasks in parallel flows with an equivalent task over a flow to which these parallel flows converge;

(c) by dividing tasks of a joint flow to clones applied to parallel flows that converge towards the joint flow.

Next, we introduce a set of logical transitions that we can apply to a state. We use the notation $S' = T(S)$ to denote the transition T from a state S to a state S'. The introduced transitions include:

- *Swap*. This transition can be applied to a pair of unary activities a_1 and a_2 and interchange their sequence, i.e., we swap the position of the two activities in the graph (see Fig. 3a). Swap concerns only unary activities, e.g., selection, checking for nulls, primary key violation, projection, function application, etc. We denote this transition as $SWA(a_1,a_2)$.
- *Factorize and Distribute*. These operations involve the interchange of a binary activity, e.g., union, join, difference, etc., and at least two unary activities that have the same functionality, but are applied over different data flows that converge towards the involved binary activity. This is illustrated in Fig. 3b. In the upper part, the two activities a_1 and a_2 have the same functionality, but they are applied to different data flows that converge towards the binary activity a_b. The Factorize transition replaces the two activities a_1 and a_2 with a new one, a, which is placed right after a_b. Factorize and Distribute are reciprocal transitions. If we have two activities that perform the same operation to different data flows, which are eventually merged, we can apply Factorize in order to perform the operation only to the merged data flow. Similarly, if we have an activity that operates over a single data flow, we can distribute it to different data flows. One can notice that Factorize and Distribute essentially model swapping between unary and binary activities. We denote Factorize and Distribute transitions as $FAC(a_b,a_1,a_2)$ and $DIS(a_b,a)$ respectively.
- *Merge* and *Split*. We use these two transitions to "package" and "unpackage" a pair of activities without changing their semantics. Merge indicates that some activities have to be grouped according to the constraints of the ETL workflow; thus, for example, a third activity may not be placed between the two, or these two activities cannot be commuted. Split indicates that a pair of grouped activities can be ungrouped; e.g., when the application of the transitions has finished, we can ungroup any grouped activities. The benefit is that the search space is proactively reduced without sacrificing any of the design requirements. Merge transition is denoted as $MER(a_{1+2},a_1,a_2)$ and split transition is denoted as $SPL(a_{1+2},a_1,a_2)$.

The reasoning behind the introduction of the transitions is quite straightforward.

- Merge and split are designated by the needs of ETL design as already described.

- Swapping allows highly selective activities to be pushed towards the beginning of the workflow, in a meaning similar to the case of traditional query optimization.
- Factorization allows the exploitation of the fact that a certain operation is performed only once (in the merged workflow) instead of twice (in the converging workflows). For example, if an activity can cache data (like in the case of surrogate key assignment, where the lookup table can be cached), such a transformation can be beneficial. On the other hand, distributing an activity in two parallel branches can be beneficial in the case where the activity is highly selective and is pushed towards the beginning of the workflow. Observe Fig. 4. Consider a simple cost model that takes into account only the number of processed rows in each process. Also, consider an input of 8 rows in each flow, and selectivities equal to 50% for process σ and 100% for the rest processes. Given $n\log_2 n$ and n as the cost formulae for SK and σ respectively (for simplicity, we ignore the cost of U), the total costs for the three cases are: $c_1=2n\log_2 n+n=56$, $c_2=2(n+(n/2)\log_2(n/2))=32$, $c_3=2n+(n/2)\log_2(n/2)=24$. Thus, DIS (case 2) and FAC (case 3) can reduce the cost of a state.

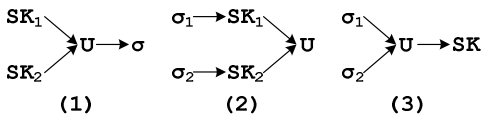


Figure 4. Factorization benefits

Due to lack of space we omit the details regarding the formal definitions of the above transitions. The interested reader is referred to the long version of the paper [19].

So far, we have demonstrated how to model each ETL workflow as a state and how to generate the state space through a set of appropriate transformations (transitions). Naturally, in order to choose the optimal state, the problem requires a convenient discrimination criterion.

Such a criterion is a cost model. Given an activity a, let $c(a)$ denote its cost (possibly depending not only on the cost model, but also on its position in the workflow graph). Then, the total cost of a state is obtained by summarizing the costs of all its activities. The total cost $C(S)$ of a state S is given by the next formula:

$$C(S) = \sum_{i=1}^{n} c(a_i)$$

The *problem of the optimization of an ETL workflow* involves the discovery of a state S_{MIN}, such that $C(S_{MIN})$ is minimal.

In the literature [8], [11], [15] there exists a variety of cost models for query optimization. Our approach is general in that it is not in particular dependent on the cost model chosen.

3. State generation, transition applicability and corrrectness

In this section, we will deal with several non-trivial issues in the context of our modeling for the optimization of ETL processes as a state space search problem. We consider equivalent states as workflows that, based on the same input, produce the same output. To deal with this condition, we will first discuss in detail how states are generated and then we will deal with the problem of transition applicability.

3.1. Naming principle

As we have already seen in the introduction, an obvious problem for the optimization of ETL workflows is that different attributes names do not always correspond to different entities of the real world and vice versa.

To handle this problem, we resort to a simple naming principle: (a) all synonyms refer to the same entity of the real world, and (b) all different attribute names, at the same time, refer to different things in the real world. Since it is possible that the employed recordsets violate this principle, we map the original attribute names of the involved recordsets to a set of reference attribute names that do not suffer from this problem. Formally, we introduce:

(a) a *set of reference attribute names at the conceptual level*, i.e., a finite set of unique attribute names Ω_n, and a mapping of each attribute of the workflow to this set of attribute names;

(b) a *simple naming principle*: all synonymous attributes are semantically related to the same attribute name in Ω_n; no other mapping to the same attribute is allowed.

In the example of Fig. 1, we can employ the same reference attribute name for both American and European dates, since we will treat them equivalently as groupers. At the same time, COST attributes, expressed in Dollars and in Euros should be mapped in different reference attributes (due to the presence of the selection predicate on Euros). In the sequel, we will

employ only reference attribute names in our discussions.

3.2. Issues around activity schemata

In [18] we have presented a set of template activities for the design of ETL workflows. Each template in this library has predefined semantics and a set of parameters that tune its functionality: for example, when the designer of a workflow materializes a `Not Null` template he/she must specify the attribute over which the check is performed. In order to construct a certain ETL workflow, the designer must specify the input and output schemata of each activity and the respective set of parameters. Although this is a manual procedure, in the context of this paper, the different states are automatically constructed; therefore, the generation of the input and output schemata of the different activities must be automated, too. For lack of space, the automation of this procedure is presented in the long version of the paper [19].

For the purpose of state transitions, (e.g., swapping activities), apart from the input and output schemata, each activity is characterized by the following schemata:

1. *Functionality (or necessary) schema*. This schema is a list of attributes, being a subset of (the union of) the input schema(ta), denoting the attributes which take part in the computation performed by the activity. For example, an activity having as input schema s_i=[A,B,C,D] and performing a `Not Null(B)` operation, has a functionality schema s_f =[B].
2. *Generated schema*. This schema involves all the output attributes being generated due to the processing of the activities. For example, a function-based activity `$2€` converting an attribute `dollar_cost` to Euros, i.e., `euro_cost = $2€(dollar_cost)`, has a generated schema s_g= `[euro_cost]`. Filters have an empty generated schema.
3. *Projected-out* schema. A list of attributes, belonging to the input schema(ta), not to be propagated further from the activity. For example, once a surrogate key transformation is applied, we propagate data with their new, generated surrogate key (belonging to the generated schema) and we project out their original production key (belonging to the projected-out schema).

These auxiliary schemata are provided at the template level. In other words, the designer of the template library can dictate in advance, (a) which are the parameters for the activity (functionality schema) and (b) which are the new or the non-necessary attributes of the template. Then, these attributes are properly instantiated at the construction of the ETL workflow.

Local Groups. A *local group* is a subset of the graph (state), the elements of which form a linear path of unary activities. In the example of Fig. 1, the local groups of the state are `{3}`, `{4,5,6}` and `{8}`.

Homologous Activities. Also, we introduce the notion of *homologous* activities to capture the cases of activities. Two activities are homologous if: (a) they are found in converging local groups; (b) they have the same semantics (as an algebraic expression); (c) they have the same functionality, generated and projected-out schemata.

3.3. Transition applicability

In this subsection, we define the rules which allow or prohibit the application of a transformation to a certain state.

Swap. One would normally anticipate that swapping is already covered by traditional query optimization techniques. Still, this is not true: on the contrary, we have observed that the swapping of activities deviates from the equivalent problem of "pushing operators downwards", as we normally do in the execution plan of a relational query. The major reason for this deviation is the presence of functions, which potentially change the semantics of attributes. Relational algebra does not provide any support for functions; still, the "pushing" of activities should be allowed in some cases, whereas, in some others, it should be prevented.

Remember the two cases from the introductory example of Fig. 1 and 2. It is not allowed to push selection on Euros before their transformation and aggregation. On the contrary, it should be permitted to push the aggregation on `DATE` before a function transforming the `DATE` from American to European format.

Formally, we allow the swapping of two activities a_1 and a_2 if the following conditions hold:

1. a_1 and a_2 are adjacent in the graph (without loss of generality assume that a_1 is a provider for a_2)
2. both a_1 and a_2 have a single input and output schemata and their output schema has exactly one consumer
3. the functionality schema of a_1 and a_2 is a subset of their input schema (both before and after the swapping)

4. the input schemata of a_1 and a_2 are subsets of their providers, again both before and after the swapping

Conditions (1) and (2) are simply measures to eliminate the complexity of the search space and the name generation. The other two conditions though, cover two possible problems. The first problem is covered by condition (3). Observe Fig. 5, where activity $2€ transforms Dollars to Euros and has an input attribute named dollar_cost, a functionality schema that contains dollar_cost and an output attribute named euro_cost. Activity σ(€), at the same time, is specifically containing attribute euro_cost in its functionality schema (e.g., it selects all costs above 100€). When a swapping has to be performed and activity σ(€) is put in advance of activity $2€, the swapping will be rejected.

The guard of condition (3) can be easily compromised if the designer uses the same name for the attributes of the functionality schemata of activities $2€ and σ(€). For example, if instead of dollar_cost and euro_cost, the designer used the name cost, then condition (3) would not fire. To handle this problem, we exploit the usage of the naming principle described in subsection 3.1.

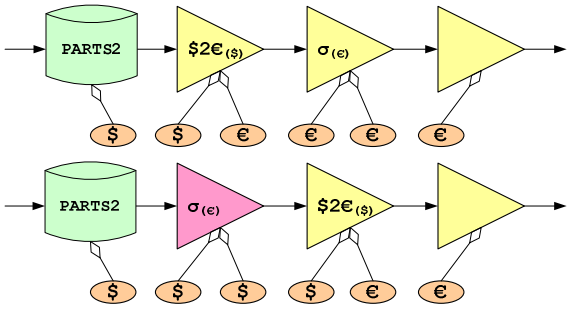


Figure 5. Necessity for swap-condition (3)

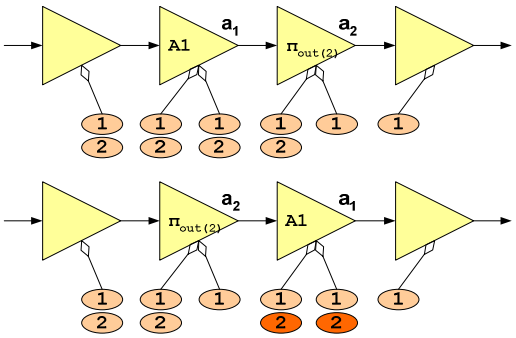


Fig. 6. Necessity for swap-condition (4)

The second problem, confronted by condition (4) is simpler. Assume that activity a_2 is a π_{out} (projected-out) activity, rejecting an attribute at its output schema (Fig. 6). Then, swapping will produce an error, since after the swapping takes place, the rejected attribute in the input schema of activity a_1 (now a consumer of a_2) will not have a provider attribute in the output schema of a_1.

Factorize/Distribute. We factorize two homologous activities a_1 and a_2, if we replace them by a new activity a that does the same job to their combined flow. Formally, the conditions governing factorization are as follows:

1. a_1 and a_2 have the same operation in terms of algebraic expression; the only thing that differs is their input (and output) schemata
2. a_1 and a_2 have a common consumer, say a_b, which is a binary operation (e.g., union, difference, etc.)

Obviously, a_1 and a_2 are removed from the graph and replaced by a new activity a, following a_b. In other words, each edge (x,a_1) and (x,a_2) becomes (x,a_b) for any node x, edges (a_1,a_b) and (a_2,a_b) are removed, the nodes a_1 and a_2 are removed, a node a is added, the edge (a_b,a) is added and any edge (a_b,y) is replaced by (a,y) for any node y.

The distribution is governed by similar laws; an activity a can be cloned in two paths if:

1. a binary activity a_b is the provider of a and two clones, activities a_1 and a_2 are generated for each path leading to a_b
2. a_1 and a_2 have the same operation in terms of algebraic expression with a

Naturally, a is removed from the graph. The node and edge manipulation are the inverse from the ones of factorize.

Merge/Split. Merge does not impose any significant problems: the output schema of the new activity is the output of the second activity and the input schema(ta) is the union of the input schemata of the involved activities, minus the input schema of the second activity linked to the output of the first activity. Split requires that the originating activity is a merged one, like, for example, a+b+c. In this case, the activity is split in two activities as (i) a and (ii) b+c.

3.4. Correctness of the introduced transitions

In this section, we prove the correctness of the transitions we introduce. In other words, we prove that whenever we apply a transition to a certain state of the problem, the derived state will produce exactly the same data with the originating one, at the end of its execution.

There are more than one ways to establish the correctness of the introduced transitions. We have decided to pursue a black-box approach and in our setting, we annotate each activity with a predicate, set to true whenever the activity successfully completes its execution (i.e., it has processed all incoming data and passed to the following activity or recordset). Otherwise, the predicate is set to false. The predicate consists of a predicate name and a set of variables. We assume fixed semantics for each such predicate name. In other words, given a predicate $2€(COST) we implicitly know that the outgoing data fulfill a constraint that the involved variable (attribute COST) is transformed to Euros.

Once a workflow has executed correctly, all the activities' predicates are set to true. Within this framework, it is easy to check whether two workflows are equivalent: (a) they must produce data under the same schema and (b) they must produce exactly the same records (i.e., the same predicates are true) at the end of their execution.

An obvious consideration involves the interpretation of the predicate in terms of the semantics it carries. Assume the function $2€ of Fig. 1 that is characterized by the post-condition $2€(COST). One would obviously wonder, why is it clear that we all agree to interpret the semantics of $2€ as the conversion of Dollars costs to Euros over the parameter variable (here, COST)? To tackle this problem, we build upon the work of [18], where *template definitions* are introduced for all the common categories of ETL transformations. In this case, every template has a "signature" (i.e., a parameter schema) and a set of well-defined semantics in LDL. For example, $2€(#vrbl$_1$) is the definition of the post-condition for $2€ function at the template level. In Fig. 1 this is instantiated as $2€(COST), where COST materializes the #vrbl$_1$. The scheme is extensible since, for any other, new activity, that the designer wishes to introduce, explicit LDL semantics can be also given. For our case, it is sufficient to employ the signature of the activity in a black box approach, both for template-based or individual activities.

A second consideration would involve the commonly agreed upon semantics of variables. We tackle this problem by introducing the common scenario terminology Ω_n, and the naming principle of Section 3.1.

Next, we give the formal definitions of the activity and workflow post-conditions.

Activity Predicate. Each activity or recordset is characterized by a logical post-condition, which we call *activity predicate* or *activity post-condition*, having as variables: (a) the attributes of the functionality schema in the case of activities or (b) the attributes of the recordset schema, in the case of recordsets.

For each node $n \in \mathbf{V}$ of a workflow $S = \mathbf{G(V,E)}$ there is a predicate p that acts as post-condition $cond_n$ for node n.

$$p \equiv cond_n(\#vrbl_1, \ldots, \#vrbl_k, \#vrbl_{k+1}, \ldots, \#vrbl_N)$$

Since $n \in \mathbf{V}=\mathbf{A} \cup \mathbf{RS}$, we discern the following cases:

1. n is a unary activity: the attributes of the functionality schema of the activity acting as the variables of the predicate.
   $\{\#vrbl_1, \ldots, \#vrbl_N\} = n.fun$
2. n is a binary activity: the attributes of the functionality schemata of both activities acting as the variables of the predicate.
   $\{\#vrbl_1, \ldots, \#vrbl_k\} = n.in_1.fun$
   $\{\#vrbl_{k+1}, \ldots, \#vrbl_N\} = n.in_2.fun$
3. n is a recordset: the attributes of the recordset acting as the variables of the predicate.

Once *all* activities of a workflow are computed, there is a set of post-conditions which are set to true. Therefore, we can obtain an expression describing what properties are held by the data processed by the workflows, once the workflow is completed.

Workflow post-condition. Each workflow is also characterized by a *workflow post-condition*, $Cond_G$, which is a Boolean expression formulated as a conjunction of the post-conditions of the workflow activities, arranged in the order of their execution (as provided by a topological sort). For example, in the workflow of Fig. 1, the workflow post-condition $Cond_G$ is given by the following formula:

$$Cond_G = PARTS1(PKEY,SOURCE,DATE,COST) \wedge PARTS2(PKEY,SOURCE,DATE,DEPT,COST) \wedge NN(COST) \wedge \$2€(COST) \wedge A2E(DATE) \wedge \gamma_{SUM}(PKEY,SOURCE,DATE,€COST) \wedge U() \wedge \sigma(€COST) \wedge DW(PKEY,SOURCE,DATE,€COST)$$

Now, we are ready to define when two workflows (states) are equivalent. Intuitively, this happens when (a) the schema of the data propagated to each target recordset is identical and (b) the post-conditions of the two workflows are equivalent.

Equivalent workflows. Two workflows (i.e., states) G1 and G2 are equivalent when:

a. the schema of the data propagated to each target recordset is identical
b. $Cond_{G1} \equiv Cond_{G2}$

Finally, we can express the following theorems which guarantee that the state transitions that we have defined are correct, in the sense that they produce equivalent workflows (i.e., states). All proofs are found in [19].

Theorem 1: Let a state S be a graph $G=(V,E)$, where all activities have exactly one output and one consumer for each output schema. Let also a transition T derive a new state S′, i.e., a new graph $G'=(V',E')$, affecting a set of activities $G_A \subset V \cup V'$. Then, the schemata for the activities of $V-G_A$ are the same with the respective schemata of $V'-G_A$. □

Theorem 2. All transitions produce equivalent workflows. □

Having presented the theoretical setup of the problem of ETL optimization, we can now present the search algorithms that we propose for this problem.

4. State space search based algorithms

In this section, we present three algorithms towards the optimization of ETL processes: (a) an exhaustive algorithm, (b) a heuristic algorithm that reduces the search space and (c) a greedy version of the heuristic algorithm. Finally, we present our experimental results.

4.1. Preliminaries

In order to speed up the execution of algorithms, we need to be able to uniquely identify a state.

State Identification. During the application of the transitions, we need to be able to discern states from one another, so that we avoid to generate (and compute the cost of) the same state more than once. In order to automatically derive *activity identifiers* for the full lifespan of the activities, we choose to assign each activity with its priority, as it stems from the topological ordering of the workflow graph, as given in its initial form. By making use of these unique identifiers, we create a string that characterizes each state and we name it the *signature* of the state. For example, the signature of the state depicted in Fig. 1 is `((1.3)//(2.4.5.6)).7.8.9`.

Finally, note that the computation of the cost of each state in all algorithms is realized in a semi-incremental way. That is, the variation of the cost from the state S to the state S′ can be determined by computing only the cost of the path from the affected activities towards the target in the new state and taking the difference between this cost and the respective cost in the previous state.

4.2. Exhaustive and heuristic algorithms

Exhaustive Search. In the exhaustive approach we generate all the possible states that can be generated by applying all the applicable transitions to every state. In general, we formalize the state space as a graph, where the nodes are states and the edges possible transitions from one state to another.

The *Exhaustive Search* algorithm (ES) employs a set of *unvisited* nodes, which remain to be explored and a set of *visited* nodes that have already been explored. While there are still nodes to be explored, the algorithm picks an `unvisited` state and produces its children to be checked in the sequel. The search space is obviously finite and it is straightforward that the algorithm generates all possible states and then terminates. Afterwards, we search the `visited` states and we choose the one with the minimal cost as the solution of our problem. The formal definition of the Exhaustive algorithm can be found in [19].

Heuristic Search. In order to avoid exploring the full state space, we employ a set of heuristics, based on simple observations, common logic and on the definition of transitions.

Heuristic 1. The definition of FAC indicates that it is not necessary to try factorizing all the activities of a state. Instead, a new state should be generated from an old one through a factorize transition (FAC) that involves only homologous activities and the respective binary one.

Heuristic 2. The definition of DIS indicates that a new state should be generated from an old one through a distribute transition (DIS) that involves only activities that could be distributed and the respective binary one. Such activities are those that could be transferred in front of a binary activity.

Heuristic 3. According to the reasons of the introduction of merge transition, it should be used where it is applicable, before the application of any other transition. This heuristic reduces the search space.

Heuristic 4. Finally, we use a simple "divide and conquer" technique to simplify the problem. A state is divided in local groups, thus, each time optimization techniques are applied in a part of, instead on the whole, graph.

The input of the algorithm *Heuristic Search* (HS) (Fig. 7) is the initial workflow (state S_0) and a list of merge constraints that are used at the pre-processing stage. Next, we present the various phases of this algorithm.

Pre-processing (Ln: 4-8). First, all the activities that the workflow constraints indicate, are merged. Such constraints might be (a) the semantics of the individual activities, e.g., before the application of a surrogate key assignment, we must enrich the data with information about the respective source, and (b) user-defined constraints, which capture the fact that a user may indicate that some activities should be merged in order to reduce the search space. Also, HS finds all the homologous (**H**) and distributable (**D**) activities of the initial state and then, it divides the initial state S_0 in local groups (**L**) (having as borders the binary activities, e.g., join, difference, intersection, etc. and/or the recordsets of the state).

Phase I (Ln: 9-13). In this phase, HS proceeds with all the possible swap transitions in each local group. Every time that HS meets a state with better cost than the minimum cost that already exists, it assigns it to S_{MIN}. Note, that all local groups are part of the same state: S_0; thus, the output of this phase is a state with a global minimum cost (concerning only the swap transitions).

Phase II (Ln: 14-20). This phase checks for possible commutations between two adjacent local groups. For each pair of homologous activities of S_0, it tests if both activities can be pushed to be adjacent to their next binary operator (function `ShiftFrw()`). Then, it applies factorization over the binary operator and the pairs of that satisfy the aforementioned condition. HS adds every new state to the `visited` list and keeps record of a state with a new minimum cost.

Phase III (Ln: 21-28). HS searches every new state of Phase II for activities that could be distributed with binary operators (function `ShiftBkw()`). Obviously, it is not eligible to distribute again the activities that we factorize in Phase II; hence, HS uses the distributable activities of the initial state (**D**). Again, HS adds every new state to the `visited` list and keeps record of a state with a new minimum cost.

Phase IV (Ln: 29-35). Intuitively, in the previous two phases, HS produces every possible local group, because in the lifetime of each local group there exist its original activities minus any activities that might be distributed to other local groups union any activities that might be factorized from other groups. In the forth phase, HS repeats Phase I in order to get all the possible states that could be produced with the application of the swap (`SWA`) transition in all the nodes that a local group can have. Of course, if it is necessary, HS updates S_{MIN}. After the completion of the above phases, HS applies split (`SPL`) transitions in order to split all the activities that were merged during the pre-processing stage. The constraints for the `SPL` transition are reciprocal to the `MER` transition constraints. Finally, HS returns the state with the minimum cost.

Algorithm Heuristic Search (HS)

```
1.  Input: An initial state S0, i.e., a graph G =(V,E)
    and a list of merge constraints merg_cons
2.  Output: A state SMIN having the minimal cost
3.  Begin
4.   apply all MER according to merg_cons;
5.   unvisited=∅; visited=∅; SMIN=S0;
6.   H ← FindHomologousActivities(S0);
7.   D ← FindDistributableActivities(S0);
8.   L ← FindLocalGroups(S0);
9.   for each gi in L {
10.    for each pair (ai,aj) in gi {
11.      SNEW ← SWA(ai,aj);
12.      if (c(SNEW)<c(SMIN)) SMIN = SNEW;
13.  }}
14.  visited ← SMIN;
15.  for each pair (ai,aj) in H {
16.    if    (    (ShiftFrw(ai,ab))    and
     (ShiftFrw(ai,ab)) ) {
17.      SNEW ← FAC(ab,ai,aj);
18.      if (c(SNEW)<c(SMIN)) SMIN = SNEW;
19.      visited ← SNEW;
20.  }}
21.  unvisited = visited;
22.  for each Si in unvisited {
23.    for each au in D {
24.      if (ShiftBkw(au,ab)) {
25.        SNEW ← DIS(ab,au);
26.        if (c(SNEW)<c(SMIN)) SMIN = SNEW;
27.        visited ← SNEW;
28.  }}}
29.  for each Si in visited {
30.    L ← FindLocalGroups(Si);
31.    for each gi in L {
32.      for each pair (ai,aj) in gi {
33.        SNEW ← SWA(ai,aj);
34.        if (c(SNEW)<c(SMIN)) SMIN = SNEW;
35.  }}}
36.  apply all SPLs according to spl_cons;
37.  return SMIN;
38. End.
```

Fig. 7. Algorithm Heuristic Search (HS)

After each transition has taken place, the input and output schemata of each activity are automatically re-generated (see [19]). We assign a unique signature to each state; thus, the two lists `visited` and `unvisited` do not allow the existence of duplicate

states. Obviously, due to the finiteness of the state space and the identification of states, the algorithm terminates.

HS-Greedy. One could argue that Phase I seems to overcharge HS, considering its repetition in Phase IV. Experiments have shown that the existence of the first phase leads to a much better solution without consuming too many resources. Also, a slight change in Phase I (and respectively in Phase IV) of HS improves its performance. In particular, if, instead of swapping all pairs of activities for each local group, HS swaps only those that lead to a state with less cost that the existing minimum, then HS becomes a greedy algorithm: *HS-Greedy*.

Experimental results. In order to validate our method, we implemented the proposed algorithms in C++ and experimented on the variation of measures like time, volume of visited states, and improvement of the solution and the quality of the proposed workflow. We have used a simple cost model taking into consideration only the number of processed rows based on simple formulae [15] and assigned selectivities for the involved activities. As test cases, we have used 40 different ETL workflows categorized as small, medium, and large, involving a range of 15 to 70 activities. All experiments were run on an AthlonXP machine running at 1.4GHz with 768Mb RAM. As expected, in all cases, the ES algorithm was slower compared to the other two, and in most cases it could not terminate due to the exponential size of the search space. As a threshold, in most cases, we let ES run up to 40 hours. Thus, we did not get the optimal solution for all the test cases, and consequently, for medium and large cases we compare (quality of solution) the best solution of HS and HS-Greedy to the best solution that ES has produced when it stopped (Table 1). Table 2 depicts the number of visited states for each algorithm, the execution time of the algorithm and the percentage of improvement for each algorithm compared with the cost of the initial state.

We note that for small workflows, HS provides the optimal solution according to ES. Also, although both HS and HS-Greedy provide solutions of approximately the same quality, HS-Greedy was faster at least 86% (average value was 92%). For medium ETL workflows, HS finds better solution than HS-Greedy (in a range of 13-38%). On the other hand, HS-Greedy is a lot faster than HS, while the solution that it provides could be acceptable. In large test cases, HS proves that it has an advantage because it returns workflows with improved cost over 70% of the cost of the initial state; while HS-Greedy returns "unstable" results in a low average value of 47%.

Table 1. Quality of solution

workflow category	ES quality of solution % (avg)	HS quality of solution % (avg)	HS-Greedy quality of solution % (avg)
small	100	100	99
medium	-	99*	86*
large	-	98*	62*

* The values are compared to the best of ES when it stopped.

The time needed for the execution of the algorithms is satisfactory compared to the time we will earn from the execution of the optimized workflow, given that usual ETL workflows run into a whole night time window. For example, the average worst case of the execution of HS for large scenarios is approximately 35 minutes, while the gain from the execution of the proposed workflow outreaches a percentage of 70%.

5. Related work

There exists a variety of ETL tools in the market; we mention a recent review [6] and several commercial tools [9], [10], [13], [14]. Although these tools offer GUI's to the developer, along with other facilities, the designer is not supported in his task with any optimization tools. Therefore, the design process deals with this issue in an ad-hoc manner. Research efforts also exist in the ETL area, including [4], [3], [5], [12]. Also, we mention three research prototypes: (a) AJAX [7], (b) Potter's Wheel [17], and ARKTOS II [18]. The first two prototypes are based on algebras, which we find mostly tailored for the case of homogenizing web data; the latter concerns the modeling of ETL processes in a customizable and extensible manner. To our knowledge, no work in the area of ETL has dealt with optimization issues so far.

In a similar setting, research has provided results for the problem of stream management [1], [2]. Techniques used in the area of stream management, which construct and optimize plans on-the-fly, come the closest that we know of to the optimization style that we discuss in the context of ETL. Nevertheless, stream management techniques are not directly applicable to typical ETL problems (a) due to the fact that real time replication is not always applicable to legacy systems and (b) pure relational querying, as studied in the field of stream management is not sufficient for ETL purposes.

Table 2. Execution time, number of visited states and improvement wrt initial state

type of workflow		ES			HS			HS-Greedy		
category	**volume of activities (avg)**	**visited states (avg)**	**improvement % (avg)**	**time sec (avg)**	**visited states (avg)**	**improvement % (avg)**	**time sec (avg)**	**visited states (avg)**	**improvement % (avg)**	**time sec (avg)**
small	20	28410	78	67812	978	78	297	72	76	7
medium	40	45110*	52*	144000*	4929	74	703	538	62	87
large	70	34205*	45*	144000*	14100	71	2105	1214	47	584

* The algorithm did not terminate. The depicted values refer to the status of the ES when it stopped.

6. Conclusions and future work

In this paper, we have concentrated on the problem of optimizing ETL workflows. We have set up the theoretical framework for the problem, by modeling the problem as a state space search problem, with each state representing a particular design of the workflow as a graph. Since the problem is modeled as a state space search problem, we have defined transitions from one state to another. We have also made a thorough discussion on the issues of state generation, correctness and transition applicability. Finally, we have presented search algorithms. Experimental results on these algorithms suggest that the benefits of our method can be significant.

Several research issues are left open, such as the physical optimization of ETL workflows, (i.e., taking physical operators and access methods into consideration) or the impact analysis of changes and failures in the workflow environment that we describe.

Acknowledgment. This work is supported by the Greek Ministry of Education and the European Union through the EPEAEK and the Pythagoras Programs.

7. References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, et al. Aurora: a new model and architecture for data stream management. The VLDB Journal, 12(2):120-139, 2003.

[2] S. Babu, J. Widom. Continuous Queries over Data Streams. SIGMOD Record 30(3): 109-120.

[3] V. Borkar, K. Deshmuk, S. Sarawagi. Automatically Extracting Structure from Free Text Addresses. Bulletin of the Technical Committee on Data Eng., 23(4), 2000.

[4] J. Chen, S. Chen, E.A. Rundensteiner. A Transactional Model for Data Warehouse Maintenance. ER'02, LNCS 2503, pp. 247–262, 2002.

[5] Y. Cui, J. Widom. Lineage Tracing for General Data Warehouse Transformations. The VLDB Journal, 12:41-58, 2003.

[6] Gartner. ETL Magic Quadrant Update: Market Pressure Increases. Available at: www.gartner.com/reprints/informatica/112769.html

[7] H. Galhardas, D. Florescu, D. Shasha and E. Simon. Ajax: An Extensible Data Cleaning Tool. SIGMOD'00, pp.590, Texas, 2000.

[8] G. Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 25(2), 1993.

[9] IBM. IBM Data Warehouse Manager. Available at www-3.ibm.com/software/data/db2/datawarehouse

[10] Informatica. PowerCenter. Available at: www.informatica.com/products/data+integration/power-center/default.htm

[11] M. Jarke, J. Koch. Query Optimization in Database Systems. ACM Computing Surveys 16(2), 1984.

[12] W. Labio, J.L. Wiener, H. Garcia-Molina, V. Gorelik. Efficient Resumption of Interrupted Warehouse Loads. SIGMOD'00, pp. 46-57, Texas, USA, 2000.

[13] Microsoft. Data Transformation Services. Available at www.microsoft.com

[14] Oracle Corp. Oracle9i™ Warehouse Builder User's Guide, Release 9.0.2. November 2001. Available at: http://otn.oracle.com/products/warehouse/content.html

[15] M. Tamer Ozsu, P. Valduriez. Principles of Distributed Database Systems. Prentice Hall, 1991.

[16] E. Rahm, H. Do. Data Cleaning: Problems and Current Approaches. Bulletin of the Technical Committee on Data Engineering, 23(4), 2000.

[17] V. Raman, J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. VLDB'01, pp. 381-390, Roma, Italy, 2001.

[18] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis. A Framework for the Design of ETL Scenarios. CAiSE'03, Klagenfurt, Austria, 2003.

[19] A. Simitsis, P. Vassiliadis, T. Sellis. Optimizing ETL Processes in Data Warehouse Environments (long version). Available at http://www.dbnet.ece.ntua.gr/~asimi/publications/SiVS04.pdf