State-Space Optimization of ETL Workflows

Alkis Simitsis, Panos Vassiliadis, and Timos Sellis

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. 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 an exhaustive and two heuristic algorithms toward the minimization of the execution cost of an ETL workflow. The heuristic algorithm with greedy characteristics significantly outperforms the other two algorithms for a large set of experimental cases.

Index Terms—Database management, database integration, data warehouse and repository, workflow management, heterogeneous databases.

1 Introduction

For quite a long time in the past, research has treated data warehouses as collections of materialized views, i.e., views whose data are locally stored and periodically refreshed [6], [23]. 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 data warehouse environments, where 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. This preparation of data before their actual loading in the warehouse for further querying is necessary due to quality problems, incompatible schemata, and unnecessary parts of source data not relevant for the purposes of the 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:

1. the identification of relevant information at the source side,
2. the extraction of this information,
3. the customization and integration of the information coming from multiple sources into a common format,
4. the cleaning of the resulting data set on the basis of database and business rules, and
5. the propagation of the data to the data warehouse and/or data marts.

- *A. Simitsis and T. Sellis are with the School of Electrical and Computer Engineering, Knowledge and Database Systems Laboratory, National Technical University of Athens, Iroon Politechnioy 9, Zographou 157 73, Athens, Hellas. E-mail: {asimi, timos}@dbnet.ece.ntua.gr.*
- *P. Vassiliadis is with the Department of Computer Science, University of Ioannina, 45110, Ioannina, Hellas. E-mail: pvassil@cs.uoi.gr.*

Manuscript received 22 Nov. 2004; revised 29 Mar. 2005; accepted 1 Apr. 2005; published online 18 Aug 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org and reference IEEECS Log Number TKDESI-0477-1104.

To give an idea of the complexity of the problem, we mention the characteristics of a vast data warehouse as cited in a recent experience report [2]. In this paper, the authors report that their data warehouse population system has to process, within a time window of 4 hours, 80 million records per hour for the entire process (compression, FTP of files, decompression, transformation, and loading) on a daily basis. The volume of data rises to about 2 TB with the main fact table containing about 3 billion records. The request for performance is so pressing that there are processes hard-coded in low-level Data Base Management Systems (DBMS) calls to avoid the extra step of storing data to a target file to be loaded to the data warehouse through the DBMS loader. The above clearly shows that intelligent techniques for data preparation can greatly improve the overall process of data warehouse population.

So far, research has only partially dealt with the problem of designing and managing ETL workflows. Typically, research approaches concern 1) the optimization of stand-alone problems (e.g., the problem of duplicate detection [21]) in an isolated setting and 2) problems mostly related to Web data (e.g., [10]). Recently, research on data streams [1], [4] 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], [4], [15]. 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 [12], [13], [17], [19] 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.

1041-4347/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

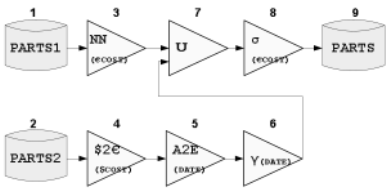

Fig. 1. A simple ETL workflow.

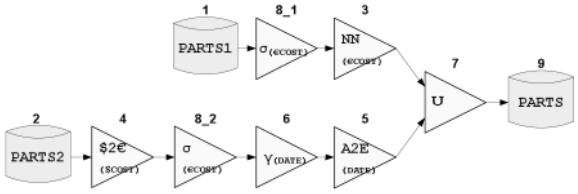

Fig. 2. A workflow equivalent to the one of Fig. 1.

One could argue that we can possibly express all ETL operations in terms of relational algebra and then optimize the resulting expression as usual. Later in this paper, we will 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 S_1 and S_2. In particular, it involves the propagation of data from the recordset *PARTS1(PKEY, SOURCE,DATE,COST)* of source S_1 that stores monthly information, as well as from the recordset *PARTS2(PKEY, SOURCE,DATE,DEPT,COST)* of source S_2 that stores daily information. In the *DW*, the recordset *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.

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*: a conversion from Dollars ($) to Euros (€) performed on attribute *COST*, *5*: dates *(DATE)* are converted from American to European format, and *6*: an aggregation for monthly supplies is performed and the unnecessary attribute *DEPT* (for department) is discarded from the flow. Then, *7*: the two flows are unified, and before being loaded to the warehouse, *8*: a final check is performed on the *COST* attribute, ensuring that only values above a certain threshold (e.g., $COST > 0$) 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 on negative values 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.
- 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*?

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 either before the transformation $2€ or before the aggregation. At the same time, there was a swapping between the aggregation and the *DATE* conversion function (*A2E*). In summary, the two problems that have risen are 1) to determine which operations over the workflow are legal and 2) which one is the best in terms of performance gains.

We take a novel approach to the problem by taking this peculiarity into consideration. 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 either by decreasing the total number of processes or by changing the execution order of the processes.

The paper deals with the specification of the design of an ETL workflow and its optimization. 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. The edges capture the flow of data among the nodes. Input and output schemata characterize activities. We provide a method for automating the computation of the input and output schemata of the activities depending on the setup of each state.

- Since the problem is modeled as a state space search problem, we define transitions from one state to another. We also provide details on how states are generated and the conditions under which transitions are allowed.
- Finally, we provide algorithms toward the optimization of ETL workflows. 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 and demonstrate the efficiency of the approach through a set of experimental results.

A short version of this paper has been accepted in [27]. 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 that concern the correct formulation of the states before and after transitions take place. In Section 4, we present three algorithms for the optimization of ETL processes, along with experimental results. In Section 5, we present related work. Finally, in Section 6, we conclude our results and discuss topics of future research.

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 1) both providers are activities (combined in a pipelined fashion) or 2) 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 consumer 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*. Wherever necessary, we will overload the term *output schema* with the role of the schema that provides the clean, consistent data to the subsequent consumers and the term *rejection schema* for the output schema that directs offending records to some log file or cleaning activity for further auditing. In all cases, the meaning of the terms will be clear from the context.

For each output schema, there is an algebraic expression characterizing the semantics of the data pushed to the respective schema. We consider an extended relational algebra involving the operators selection (σ), projection (π), Cartesian product ($\times$), join ($\bowtie$), aggregation (γ), ordering (Θ), union ($\cup$), difference ($-$), and function application (f). We will frequently employ the term projection-out (π_{out}) to refer to the complement of projection. The semantics of the above operators are obvious; for the case of function application, we assume the existence of a set of function types (e.g., arithmetic or string manipulation functions) which are instantiated each time by using attributes of the input schemata as input parameters and produce a new value as their return value.

Formally, an *activity* is a quadruple $\mathrm{A} = (\mathrm{Id}, \mathbf{I}, \mathbf{O}, \mathrm{S})$ such that:

1. Id is a unique identifier for the activity,
2. $\mathbf{I}$ is a finite set of one or more input schemata receiving data from the data providers of the activity,
3. $\mathbf{O}$ is a finite set of one or more output schemata that describe the placeholders for the rows that are processed by the activity, and
4. 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 $\mathbf{A}$, a finite set of recordsets $\mathbf{RS}$, and a finite list of provider relationships $\mathbf{Pr}$. Formally, an *ETL Workflow* is a directed acyclic graph (DAG), $\mathbf{G}(\mathbf{V}, \mathbf{E})$ such that $\mathbf{V} = \mathbf{A} \cup \mathbf{RS}$ and $\mathbf{E} = \mathbf{Pr}$. A subset of $\mathbf{RS}$, denoted by $\mathbf{RS_S}$, contains the sources of the graph (i.e., the source recordsets) and another subset of $\mathbf{RS}$, denoted by $\mathbf{RS_T}$, contains the sinks of the graph (representing the final target recordsets of the warehouse). $\mathbf{G}(\mathbf{V}, \mathbf{E})$ 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).

2.2 A Reference Example ETL Workflow

Next, we introduce a reference example involving two source databases *S1* and *S2* as well as a central data warehouse *DW*. This simple workflow describes the population of a table of

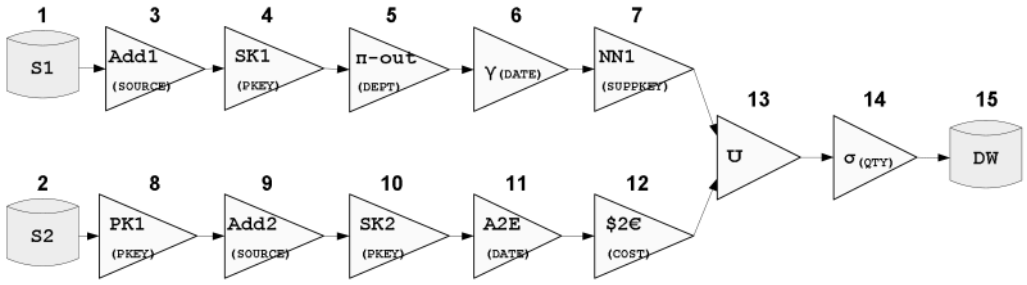

Fig. 3. A reference ETL workflow.

the data warehouse. In particular, it involves the propagation of data from the recordset *PARTSUPP(PKEY,SUPPKEY,EDATE,DEPT,QTY,COST)* of source *S1* that stores daily information, as well as from the recordset *PARTSUPP(PKEY,SUPPKEY,ADATE,QTY,COST)* of source *S2* that stores monthly information. In the data warehouse, *DW.PARTSUPP(PKEY,SUPPKEY,DATE,QTY,COST,SOURCE)* stores monthly information for the available quantity *(QTY)* and cost *(COST)* of parts *(PKEY)* per supplier *(SUPPKEY)*. We assume that 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.

In Fig. 3, we depict the full-fledged diagram of our reference example. Activities are numbered with their execution priority and tagged with the description of their functionality. Source recordsets have the highest priority (i.e., the smallest identifiers), while target recordsets have the lowest priority (i.e., the highest identifiers). The flow for source *S1* is *3*: data are enriched with an extra attribute *SOURCE* that describes their origin, *4*: their production keys are replaced with surrogate keys, *5*: the unnecessary attribute *DEPT* is discarded from the flow, *6*: the aggregation for monthly supplies is performed, and *7*: a check for *Not NULL* values is performed on attribute *SUPPKEY*. The flow for source *S2* is *8*: data are checked for duplicates, *9*: data are also enriched with an extra attribute *SOURCE* that describes their origin, *10*: their production keys are replaced with surrogate keys, *11*: their dates are converted from American to European format, and *12*: Dollars are converted to Euros. The two flows are then unified, and *13*: before being loaded to the warehouse, a final check is performed *14*: on whether the *QTY* attribute is not zero.

2.3 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 are based on the same input produce the same output. Practically, this is achieved in the following way:

1. 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,
2. by replacing common tasks in parallel flows with an equivalent task over a flow to which these parallel flows converge, and
3. by dividing tasks of a joint flow to clones applied to parallel flows that converge toward the joint flow.

Furthermore, we use the notation $S' = T(S)$ to denote the transition from a state S to a state S'. We say that we have *visited* a state if we use it to produce more states that can originate from it. Next, we introduce a set of logical transitions that we can apply to a state. These 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. 5a). Swap concerns only unary activities, e.g., selection, checking for nulls, primary key violation, projection, function application, and so on. 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 toward the involved binary activity. This is illustrated in Fig. 5b. 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 swaping 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

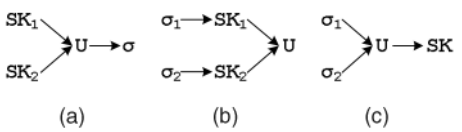

Fig. 4. Benefits from factorization and distribution.

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. We denote Merge and Split transitions as $MER(a_{1+2}, a_1, a_2)$ and $SPL(a_{1+2}, a_1, a_2)$, respectively.

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 toward 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 toward 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 eight rows in each flow and selectivities equal to 50 percent for process σ and 100 percent for the rest processes. Given $nlog_2n$ 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:

$$c_1 = 2nlog_2n + n = 56,$$
$$c_2 = 2(n + (n/2)log_2(n/2)) = 32, \text{ and}$$
$$c_3 = 2n + (n/2)log_2(n/2) = 24.$$

Thus, DIS (case b) and FAC (case c) can reduce the cost of a state.

Formally, the transitions are defined as follows:

- $Swap(a_1, a_2)$ over $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ produces a new graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$, where $\mathbf{V}' = \mathbf{V}$. Then, for each edge $e \in \mathbf{E}$ with $e = (v, a_1)$, introduce into $\mathbf{E}'$ the edge $e' = (v, a_2)$. Similarly, for each edge $e \in \mathbf{E}$ with $e = (a_1, \mathrm{v})$, introduce into $\mathbf{E}'$ the edge $e' = (a_2, \mathrm{v})$. Replace the edge (a_1, a_2) with the edge (a_2, a_1). The rest of the edges of the graph remain the same.
- $Factorize(a_b, a_1, a_2)$ over $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ produces a new graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$. In this new graph, remove nodes a_1, a_2 and introduce a new node a to $\mathbf{V}' = \mathbf{V} - \{a_1, a_2\} \cup \{\mathrm{a}\}$. For each edge $e \in \mathbf{E}$ with $e = (v, a_1)$ or $e = (v, a_2)$, introduce into $\mathbf{E}'$ the edge $e' = (v, a_b)$. For each edge $e \in \mathbf{E}$ with $e = (a_b, v)$, introduce into $\mathbf{E}'$ the edge $e' = (a, v)$. Add the edge (a_b, a). The rest of the edges of the graph remain the same.
- $Distribute(a_b, a)$ over $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ produces a new graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$. In this new graph, remove node a and introduce two new nodes a_1 and a_2 to $\mathbf{V}' = \mathbf{V} \cup \{a_1, a_2\} - \{a\}$. Remove the edge (a_b, a). For each edge $e \in \mathbf{E}$ with $e = (v, a_b)$, introduce into $\mathbf{E}'$ the edges $e' = (v, a_1)$ and $e'' = (v, a_2)$. For each edge $e \in \mathbf{E}$ with $e = (a, v)$, introduce into $\mathbf{E}'$ the edge $e' = (a_b, v)$. Introduce into $\mathbf{E}'$ the edges $e_1 = (a_1, a_b)$ and $e_2 = (a_2, a_b)$. The rest of the edges of the graph remain the same.
- $Merge(a_{1+2}, a_1, a_2)$ over $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ produces a new graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$. In this new graph, introduce a new node a_{1+2} to $\mathbf{V}' = \mathbf{V} - \{a_1, a_2\} \cup \{a_{1+2}\}$. Naturally, the edge $e = (a_1, a_2)$ is removed. For each edge $e \in \mathbf{E}$ with $e = (v, a_1)$, introduce into $\mathbf{E}'$ the edge $e' = (v, a_{1+2})$. For each edge $e \in \mathbf{E}$ with $e = (a_2, v)$, introduce into $\mathbf{E}'$ the edge $e' = (a_{1+2}, v)$. The rest of the edges of the graph remain the same.
- $Split(a_{1+2}, a_1, a_2)$ over $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ produces a new graph $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$. In this new graph, introduce two new nodes a_1 and a_2 to $\mathbf{V}' = \mathbf{V} - \{a_{1+2}\} \cup \{a_1, a_2\}$. For each edge $e \in \mathbf{E}$ with $e = (v, a_{1+2})$, introduce into $\mathbf{E}'$ the edge $e' = (v, a_1)$. For each edge $e \in \mathbf{E}$ with $e = (a_{1+2}, v)$, introduce into $\mathbf{E}'$ the edge $e' = (a_2, v)$. Add the edge $e = (a_1, a_2)$. The rest of the edges of the graph remain the same.

Figs. 5a, 5b, and 5c illustrate abstract examples of the usage of these transition, while in Figs. 5d, 5e, and 5f, we concretely instantiate these examples over our reference example.

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 that consists of n activities 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 $C(S_{\mathrm{MIN}})$ is minimal.

In the literature [8], [11], [14], [20], 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. Having described the problem in detail, we move next to discussing problems that arise in generating states and applying transformations on them.

3 State Generation and Transition Applicability

In this section, we will deal with several nontrivial issues in the context of our modeling for the optimization of ETL

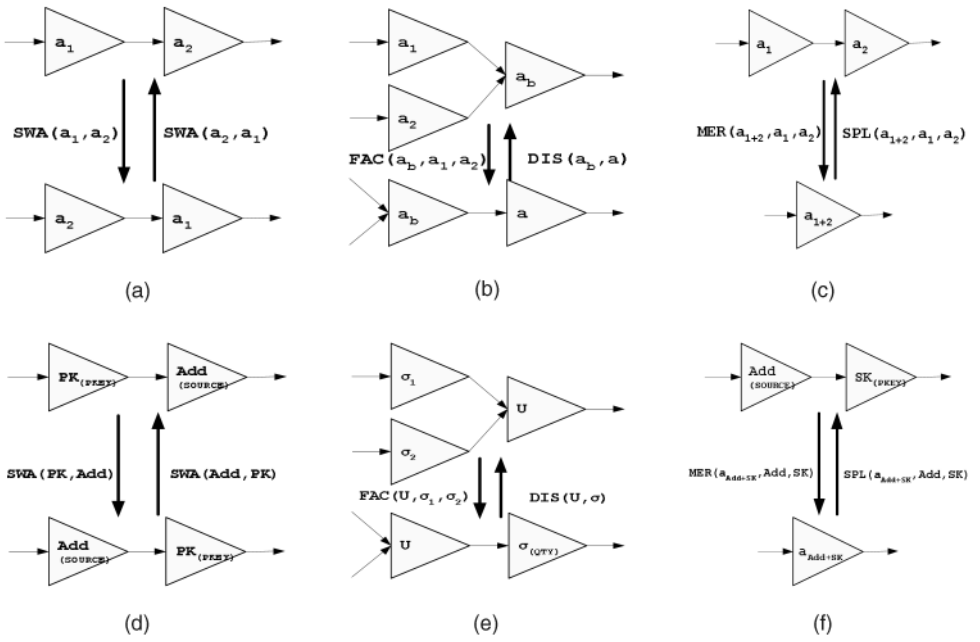

Fig. 5. Abstract (a), (b), (c) and concrete (d), (e), (f) examples of transitions. (a) Swap. (b) Factorize and Distribute. (c) Merge and Split. (d) Swap *PK(PKEY), Add(SOURCE)*. (e) Factorize and Distribute $\sigma(QTY)$, U. (f) Merge and Split *Add(SOURCE), SK(PKEY)*.

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, a problem for the optimization of ETL workflows is that different attributes names do not always correspond to different entities of the real world.

To handle this problem, we resort to a simple naming principle: 1) all synonyms refer to the same entity of the real world and 2) all different attribute names, at the same time, refer to different things in the real world. Naturally, it might be the case that the employed recordsets violate this principle. For example, in Fig. 3, the attributes *COST* of the two sources (and the resulting data flows) correspond to different meaning in the real world (Dollars and Euros, respectively). At the same time, it is quite possible that we do not care to trace the distinction between the attributes *EDATE* and *ADATE* of the different flows (see Section 3.3 for more on this). To deal with the possibility of such conflicts, we employ a mapping from the original attribute names of the involved recordsets to a set of reference attribute names that do not suffer from this problem. This is a procedure that cannot be fully automated, therefore, we request from the designer to resolve the synonymity problems (although, there already exist results for the semiautomatic mapping of attributes). Formally, we introduce:

1. 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 and
2. a simple naming principle: All synonymous attributes are semantically related to the same attribute name in Ω_{n} and no other mapping to the same attribute is allowed.

For the reference example of Fig. 3, we can perform the following mappings (original names are in capital and reference attributes are in lowercase letters):

S1.PARTSUPP		Ω_{n}	S2.PARTSUPP		Ω_{n}
PKEY	→	pkey	PKEY	→	pkey
SUPPKEY	→	suppkey	SUPPKEY	→	suppkey
EDATE	→	date	ADATE	→	date
DEPT	→	dept	QTY	→	qty
QTY	→	qty	COST	→	dollar_cost
COST	→	euro_cost			

3.2 State Generation

In [26], 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. In this section, we explain how this generation is performed.

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 1) which are the parameters for the activity (functionality schema) and 2) which are the new or the nonnecessary attributes of the template. Then, these attributes are properly instantiated at the construction of the ETL workflow.

Once the auxiliary schemata of the activities have been determined, we can automate the construction of the input/output schemata (called Schema Generation) of the activities as follows: 1) we topologically sort the graph of the workflow, 2) following the topological order, we assign the input schema of an activity to be the same with the (output) schema of each provider, and 3) the output schema of an activity is equal to (the union of) its input schema(ta), augmented by the generated schema, minus the projected-out attributes. After each transition, we determine the schemata of the activities based on the new graph of the state. The automatic construction is coded in the algorithm *Schema Generation* (SGen) that is depicted in Fig. 6.

In principle, we would be obliged to recompute the activity schemata for the whole graph due to the fact that there might be output schemata acting as providers for more than one consumer (i.e., the node which models the activity has more than one outgoing edges toward different activities or recordsets). An obvious pruning involves the exploitation of the topological order of the workflow: In this approach, whenever we perform a transition, we need to examine only the activities following (in terms of topological order) the modified part of the workflow. There is a case, still, where we can do better. In the case where all activities have exactly one output and one consumer for each output schema, the impact of a change due to a transition affects only the involved activities, while the rest of the workflow remains intact. In this case, instead of running the algorithm Schema Generation for the whole graph, we simply recompute the new schemata for the subgraph that includes the affected activities.

3.3 Local Groups and Homologous Activities

We introduce the notion of local groups to capture activities forming a linear path of execution and homologous activities to capture the cases of activities that do the same job while being in different local groups.

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. 3, the local groups of the state are *{3,4,5,6,7}*, *{8,9,10,11,12}*, and *{14}*.

Homologous Activities. Two activities are homologous when 1) they are found in converging local groups, 2) they have the same semantics (as an algebraic expression), and 3) they have the same functionality generated and projected-out schemata.

For example, consider the ETL workflow depicted in Fig. 7 that concerns the population of a target recordset *RS3(SKEY,SOURCE,QTY,COST)* from two source recordsets *RS1(PKEY,SOURCE,QTY,COST)*, and *RS2(PKEY, SOURCE,QTY,COST)*. In this state, the two surrogate key activities (3 and 4) are homologous since they both are found in converging linear paths, have the same algebraic expression, and the same functionality *{PKEY,SOURCE}*, generated *{SKEY}*, and projected-out *{PKEY}* schemata.

Algorithm Schema Generation (SGen)

1. **Input**: A state S, i.e., a graph G=(V, E) involving n activities
2. **Output**: The state S annotated with fully defined input and output activity schemata
3. **Begin**
4. topologically sort S;
5. assign a priority p(i) to each activity and let a_i denote the activity with priority p(i);
6. **for** (i=1;i<n;i++) { //i.e., for each activity
7. **for** each input schema $s_{i,j}$ of a_i {
8. $s_{i,j}$ =output_schema(provider($s_{i,j}$));
9. }
10. output_schema(a_i) = $\cup_{(j)} s_{i,j}$ ∪ generated_schema(a_i) - projected_out(a_i);
11. }
12. **End**.

Fig. 6. Schema generation for the activities of a state after a transition.

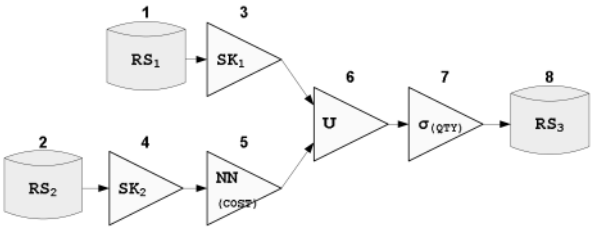

Fig. 7. Homologous activities and local groups.

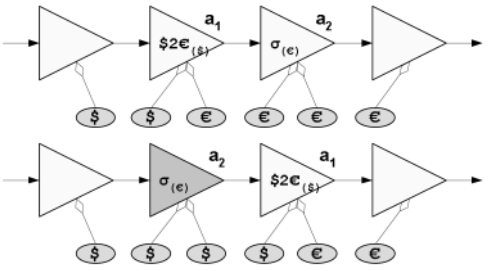

Fig. 8. Necessity for swap-condition 3.

3.4 Transition Applicability

So far, we have not answered the issue of transition applicability. In other words, we need to define the rules which allow or prohibit the application of a transformation to a certain state. We have deliberately delayed this discussion until schema generation was introduced since the correctness of transitions depends on the correct generation of schemata. Now, we are ready to present a formal framework for the applicability of these transitions in the rest of this section.

Swap. Swapping raises quite a few issues in our setting. 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 downward," 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.

Let us motivate the discussion with both a counterexample and an example for the opportunity for operator "pushing." First, the counterexample (Fig. 8): Assume the case where activity a_1 performs a conversion of attribute cost from Dollars to Euros and its consumer, activity a_2, selects only the tuples with an amount in Euro higher than *100€*. Clearly, we are not allowed to blindly swap these activities. On the other hand, if we disallow any swapping in the presence of functions, then, practically, we forbid many useful operations since ETL workflows are quite heavy on functions (mostly arithmetic and string processing ones). For example, consider the case of Fig. 9, where activity $a_1(A2E)$ performs a conversion of attribute date from American date format to European date format and activity $a_2(GDAY)$ isolates the name of a day (e.g., Thursday) from a date (e.g., *Thursday, November 13, 2003*) and produces a new attribute named *day*. In this case, the swapping is permissible.

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), and
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. 8, where activity a_1 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 a_2, 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 a_2 is put in advance of activity a_1, the swapping will be rejected. Schema Generation assigns attribute names from the sources towards the targets: Then, the attributes of activity a_2 will be named as *dollar_cost* and, therefore, the functionality attribute *euro_cost* will not be in the input schema of the activity.

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 a_1 and a_2. 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 Section 3.1.

Coming back to the example of Fig. 9, the swapping of activities a_1 and a_2 is allowed in our setting, provided that both activities have the same name for the attribute date in their input and output schemata. In Fig. 9, we present 1) the

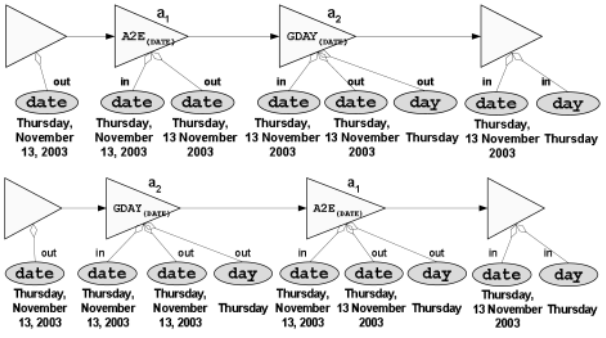

Fig. 9. Applicability of swap for functions.

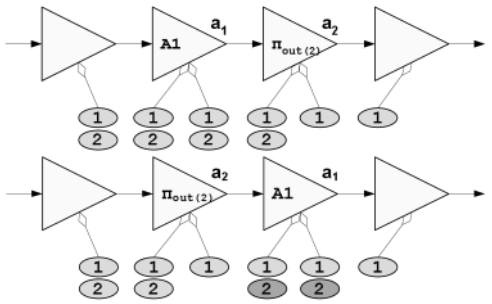

Fig. 10. Necessity for swap-condition 4.

originating state and 2) the state after the swap transition and the application of the Schema Generation algorithm which gives the new input and output schemata.

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. 10). 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. Clearly, we tackle this problem with the automatic schema generation that is described in Section 3.2.

In terms of the example of Fig. 3, we can observe another case where the swapping is also allowed, involving activities 11 and 12. The four conditions governing swapping are met; thus, Fig. 11 presents all the steps of this transition (concerning attribute *COST*): the initial state (Fig. 11a), the activities after swapping (Fig. 11b), and the final state after the application of the Schema Generation (Fig. 11c).

Factorize/Distribute. We factorize two 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, and
2. a_1 and a_2 have a common consumer, say a_b, which is a binary operation (e.g., union, difference).

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}, and
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.

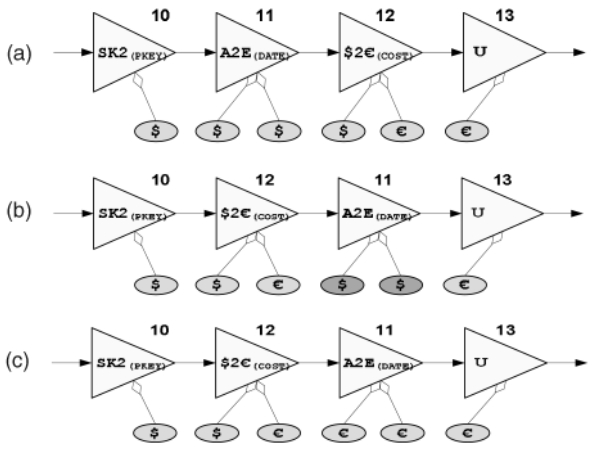

Fig. 11. The whole process of swapping transition.

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 a and $b + c$.

3.5 Correctness of the Introduced Transitions

An obvious question that arises concerns the correctness of the transitions that we introduce. In other words, can we guarantee 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? In this section, we will prove the correctness of the transitions we introduce.

There is more than one ways to establish the correctness of the introduced transitions. One alternative would be to treat activities as white-box modules with semantics specified in a certain language. In this case, one must employ a reasoning mechanism that checks the equivalence of the semantics of the two workflows in terms of the employed language. Candidate language for this task would include formal specification languages, like Z or VDM, or database languages like SQL or Datalog. Especially for the latter category, there are already results for the modeling of ETL activities in LDL++ [26]. Although feasible, both such approaches suffer from the main drawbacks of formal specification (too much programming for little effort, lack of maintainability, etc.) and the cost of performing the equivalence check for each transition.

Under these thoughts, we decided to pursue a black-box approach. 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). 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 $NN(Age)$, we implicitly know that the outgoing data fulfill a constraint that the involved variable (attribute Age) is not null.

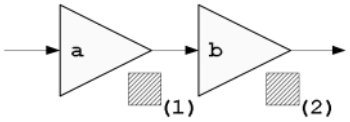

Fig. 12. Data "packets" as processed by a sequence of activities.

Fig. 13. Example of postcondition for a *Not Null* activity.

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: 1) they must produce data under the same schema and 2) they must produce exactly the same records (i.e., the same predicates are true) at the end of their execution.

The semantics of this predicate postcondition is that the predicate is set to true once the activity has successfully completed. Otherwise, the predicate is false. In other words, as data flow from the sources toward the warehouse and pass through the activities, they set the postcondition to true. Observe Fig. 12. Assume that the packets in positions 1 and 2 are the two last packets that are processed by activities a and b. In position 1, the postcondition $cond_a$ says that data fulfill the properties described by $cond_a$. As data are forwarded more, once activity b is completed, both $cond_a$ and $cond_b$ hold for the propagated data.

An obvious consideration involves the interpretation of the predicate in terms of the semantics it carries. Assume the case of Fig. 13, where the depicted activity is characterized by the postcondition $NN(AGE)$. One would obviously wonder, why is it clear that we all agree to interpret the semantics of NN as the check for not null values over the parameter variable (here, AGE)? To tackle this problem, we build upon the work of [26], 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, $NN(\#vrbl_1)$ is the definition of the postcondition for *Not Null* attributes at the template level. In Fig. 13, this is instantiated as $NN(AGE)$, where AGE 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.

Activity Predicate. Each activity or recordset is characterized by a logical postcondition, which we call *activity predicate* or *activity postcondition*, having as variables: 1) the attributes of the functionality schema in the case of activities or 2) the attributes of the recordset schema, in the case of recordsets.

For each node $n \in \mathbf{V}$ of a workflow $\mathrm{S} = \mathrm{G}(\mathbf{V}, \mathbf{E})$, there is a predicate p that acts as postcondition $cond_n$ 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 three usual cases for node n:

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: $\{\#vrbl_1, \ldots, \#vrbl_k\} = n.in_1.fun$ and $\{\#vrbl_{k+1}, \ldots, \#vrbl_N\} = n.in_2.fun$.
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 postconditions 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 postcondition. Each workflow is also characterized by a *workflow postcondition*, $Cond_{WF}$, which is a Boolean expression formulated as a conjunction of the postconditions of the workflow activities arranged in the order of their execution (as provided by a topological sort).

For example, Fig. 14 depicts an example workflow along with its postcondition $Cond_{WF}$.

The functionality schema of Diff activity comprises the primary key attributes of the two provider activities of *Diff*. In this example, the primary key of SK is $SK.SKEY$ and NN is $NN.SKEY$, so the functionality schema of *Diff* contains the attributes of the input schemata of *Diff* that are mapped to $SK.SKEY$ and $NN.SKEY$, i.e., $Diff.IN_1.SKEY$ and $Diff.IN_2.SKEY$, respectively.

Now, we are ready to define when two workflows (states) are equivalent. Intuitively, this happens when 1) the schema of the data propagated to each target recordset is identical and 2) the postconditions of the two workflows are equivalent.

Equivalent workflows. Two workflows $W1$ and $W2$ are equivalent when:

1. the schema of the data propagated to each target recordset is identical and
2. $Cond_{W1} \equiv Cond_{W2}$.

In the following theorems, we assume a set of affected activities $\mathbf{G_A}$; for each transition type, we will define the set $\mathbf{G_A}$ precisely.

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

Proof. We distinguish the different cases of transitions.

Swap transition $SWA(a, b)$. Let a sequence of unary activities $k, a, b, l \in A$. In this case, $G_A = \{a, b\}$. If the applicability conditions of swap are met for the activities a and b, then the following holds.

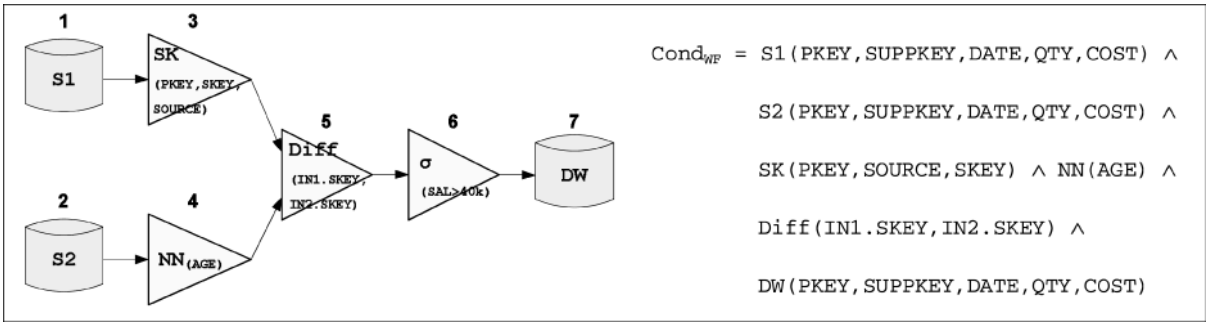

Fig. 14. Exemplary workflow and workflow postconditions.

Before $SWA(a,b)$: The input schema of the activity l is equal to the output schema of the activity b:

$$\begin{aligned} l.in_{(before)} &= b.out = (b.in \cup b.gen - b.pro) \\ &= (a.out \cup b.gen - b.pro) \\ &= (a.in \cup a.gen - a.pro \cup b.gen - b.pro) \\ &= (k.out \cup a.gen - a.pro \cup b.gen - b.pro). \end{aligned}$$

After $SWA(a,b)$:

$$\begin{aligned} l.in_{(after)} &= a.out = (a.in \cup a.gen - a.pro) \\ &= (b.out \cup a.gen - a.pro) \\ &= (b.in \cup b.gen - b.pro \cup a.gen - a.pro) \\ &= (k.out \cup b.gen - b.pro \cup a.gen - a.pro). \end{aligned}$$

Obviously, $l.in_{(before)} - l.in_{(after)} = \emptyset$. Recall that:

$$a_i.in = a_{i-1}.out.$$

Factorize transition $FAC(d, c_1, c_2)$. Assume the example of Fig. 15. In this case, $G_A = \{d, c_1, c_2, c_{1;2}\}$. Before the transition, we have that:

$$\begin{aligned} d.in_1 &= c_1.out = (a.out \cup c_1.gen - c_1.pro), \\ d.in_2 &= c_2.out = (b.out \cup c_2.gen - c_2.pro), \\ f.in_{(before)} &= d.out = (a.out \cup c_1.gen - c_1.pro) \cup \\ &\quad (b.out \cup c_2.gen - c_2.pro) \cup d.gen - d.pro. \end{aligned}$$

After the transition, we have that:

$$\begin{aligned} &d.in1 = a.out, \\ &d.in2 = b.out, \\ &d.out = (a.out \cup b.out) \cup d.gen - d.pro \end{aligned}$$

and

$$\begin{aligned} &f.in_{(after)} = c_{1;2}.out = \\ &(a.out \cup b.out \cup d.gen - d.pro \cup c_{1;2}.gen - c_{1;2}.pro). \end{aligned}$$

Then,

$$\begin{aligned} &f.in_{(before)} - f.in_{(after)} = \\ &(c_1.gen - c_1.pro \cup c_2.gen - c_2.pro) - (c_{1;2}.gen - c_{1;2}.pro). \end{aligned}$$

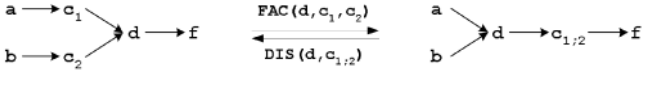

Fig. 15. Example of factorization and distribution.

This practically means that we can guarantee the locality of the impact if $c_{1+2}.pro = (c_1.pro \cup c_2.gen \cup c_2.pro)$ or $c_{1+2}.pro = (c_1.pro \cup c_1.gen \cup c_2.pro)$.

Distribute transition $DIS(d, c_{1;2})$. Similarly, provided again that we can guarantee that

$$\begin{aligned} &f.in_{(before)} - f.in_{(after)} = \\ &(c_1.gen - c_1.pro \cup c_2.gen - c_2.pro) - (c_{1;2}.gen - c_{1;2}.pro) = \emptyset. \end{aligned}$$

Merge and Split transitions $MERGE$, and $SPLIT$. Obvious. □

Theorem 2. *All transitions produce equivalent workflows.*

Proof. For each transition, we must show the validity of the conditions 1 and 2 of the previous definition. Theorem 1 guarantees that the application of all the transitions does not affect the schema of the workflow activities. Thus, condition 1 is fulfilled for all transitions. Next, we prove that condition 2 is fulfilled too, i.e., that if $S_2 = T(S_1)$ holds then $Cond_{S1} = Cond_{S2}$ holds too.

Swap transition. Without loss of generality, assume a part of a workflow involving two adjacent activities a and b. Before the application of $SWA(a,b)$ the data fulfill the properties described by the expression: $cond_a \wedge cond_b$. After swapping the two activities, the postcondition is: $cond_b \wedge cond_a$. Obviously, the postconditions before and after the application of the SWA transitions are equal.

Factorize transition. The general case for the application of the factorization operator is depicted in Fig. 16. In this case, before the application of $FAC(c, d\_1, d\_2)$ transition, the postcondition at the end of the group of involved activities is: $cond_a \wedge cond_b \wedge cond_c \wedge cond_d$. After the application of $FAC(c, d\_1, d\_2)$, the postcondition at the same point is: $cond_a \wedge cond_{d\_1} \wedge cond_b \wedge cond_{d\_2} \wedge cond_c$. It suffices to show that $cond_{d\_1} \wedge cond_{d\_2} \equiv cond_d$, which is obvious since:

1. d_1, d_2 are homologous, i.e., identical; therefore, $cond_{d\_1} \equiv cond_{d\_2} \Rightarrow cond_{d\_1} \wedge cond_{d\_2} \equiv cond_{d\_1}$.
2. $cond_{d\_1} \equiv cond_d$ by definition.

Distribute transition. The proof for distribute transition is similar to the one for factorize transition. With respect to the example of Fig. 16, before the application of $DIS(c, d)$, the postcondition at the end of the group of involved activities is: $cond_a \wedge cond_{d\_1} \wedge cond_b \wedge cond_{d\_2} \wedge cond_c$. After the application of $DIS(c, d)$, the postcondition at the same point is: $cond_a \wedge cond_b \wedge cond_c \wedge cond_d$. Again, it

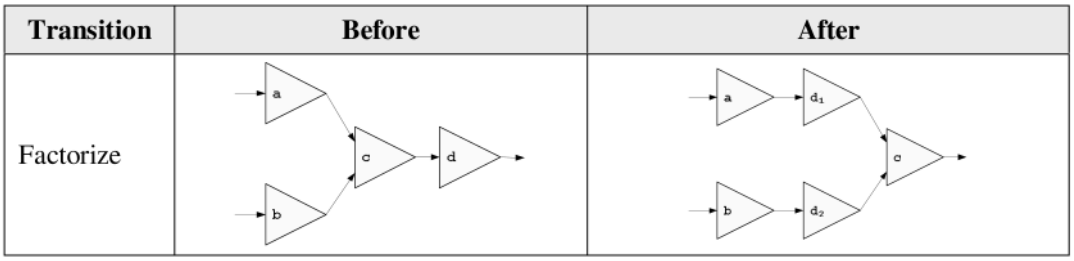

Fig. 16. Factorize transition.

suffices to show that $cond_{d\_1} \wedge cond_{d\_2} \equiv cond_d$ and this is true as we show before.

Merge/Split transitions. The proof is obvious. Since merge and split are mechanisms to "package" and "unpackage" activities, they do not affect the postconditions, i.e., the semantics, of the activities. For example, the application of merge transition to two subsequent activities a and b results to the production of a "new" merged activity $a+b$. Then, the postcondition before $MER(a+b, a, b)$ after activity b is: $cond_a \wedge cond_b$. After the merge, the postcondition at the same point is: $(cond_a \wedge cond_b)$. Note that the postcondition of a merged activity $a+b$ is not $cond_{a+b}$ because merge does not logically affect the semantics of the merged activities.

In the same sense, split does not affect the postconditions of the involved activities too. Thus, condition 2 is fulfilled for all five transitions. □

4 State Space Search-Based Algorithms

In this section, we present three algorithms toward the optimization of ETL processes. First, we use, wherever it is possible, an exhaustive approach to construct the search space in its entirety and to find the optimal ETL workflow. Next, we introduce a heuristic algorithm that reduces the search space, as such a greedy version of it. Finally, we present our experimental results.

4.1 State Identification

During the application of the transitions, we need to be able to discern states from one another so that we avoid generating (and computing the cost of) the same state more than once. As already mentioned, a state is a directed graph. Moreover, it is characterized from extra constraints as 1) it has some source nodes (source recordsets), 2) some sinks (the ultimate warehouse tables or views), and 3) it can be topologically ordered on the basis of the provider-consumer relationship so that a unique execution priority can be assigned to each activity. 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.

Based on the above observations, it is easy to assign an identification scheme to each activity such that: 1) each linear path is denoted as a string where the activities of the path are delimited by points, and 2) concurrent paths are delimited by a double slash. We call the string that characterizes each state as the *signature* of the state. Between concurrent paths, the signature of the state starts with the path including the identifier with the lowest value.

Observe Fig. 3, where we have tagged each element of the workflow with a unique identifier. The signature of that state is *((1.3.4.5.6.7)//(2.8.9.10.11.12)).13.14.15*. Each linear path is traced as dot delimited list of identifiers included in parentheses. If we swap activities 3 and 4, then the signature becomes *((1.4.3.5.6.7)//(2.8.9.10.11.12)).13.14.15*.

In Fig. 17, we present the algorithm *Get Signature* (GSign) that we use to get the signature of a state. As an input, GSign gets a state and a target node of the state. Starting from the target node, GSign recursively adds the Ids of all activities to the signature. If an activity is unary, then we just put "." and its Id to the signature, else, if the activity is binary, GSign takes into account that it has to follow two separate paths.

For the cases where new activities are derived from existing ones, we use the following rules:

1. If two existing activities a and b are merged to create a new one, then the new activity is denoted as $a+b$,
2. if two existing activities a and b are factorized to create a new one, then the new activity is denoted as $a;b$,
3. if an activity is cloned in more that one paths, then each of its clones adopts the identifier of the original activity followed by an underscore and a unique integer as a partial identifier (e.g., a_1, a_2), and
4. if a "composite" activity is split, we reuse the identifiers of its components (e.g., $a+b$ or $a;b$ can be split to a and b, respectively).

Again, the lowest partial identifier is assigned to the path including the activity identifier with the lowest value, etc. For example, if we distribute activity *14* in Fig. 3, then the concurrent linear paths end to activities *14_1* and *14_2* respectively; the signature of the state is then *((1.3.4.5.6.7.14_1)//(2.8.9.10.11.12.14_2)).13.15*. Observe how we assign the partial identifiers *_1* and *_2* to the individual paths. If we merge activities *11* and *12*, then a merged activity *11 + 12* is interleaved between activities *10* and *13* and the signature of the state becomes *((1.3.4.5.6.7)// (2.8.9.10.11+12)).13.14.15*.

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

```
Algorithm Get Signature (GSign)
1.  Input: A state S, i.e., a graph G=(V,E), a state S'
    with edges in the reverse direction than the ones of S,
    and a node v that is a target node for the state S
2.  Output: The signature sign of the state S
3.  Begin
4.    Id ← find the Id of v;
5.    sign = "." + ID + sign;
6.    if (outdeg(v)==2) {
7.     v1←next of v with the lowest (Id);
8.     GSign(S,v1,s1);
9.     v2 ←next of v with the highest (Id);
10.    GSign(S,v2,s2);
11.    sign = "(("+s1+")//("+s2+"))" + sign;
12.   }
13.   else if (outdeg(v)==1) {
14.    v ← next of v;
15.    GSign(S,v,sign);
16.   }
17.   sign = sign.replace_all("(.","(");
18. End.
```

Fig. 17. Algorithm for state identification.

```
Algorithm Exhaustive Search (ES)
1.  Input: An initial state S_O, i.e., a graph
    G=(V,E)
2.  Output: A state S_MIN having the minimal cost
3.  Begin
4.   S_MIN = S_O ;
5.   unvisited ← S_o;
6.   visited = ∅;
7.   for each S in unvisited {
8.    for each S'=T(S) {
9.     if ((S'∉unvisited)&&(S'∉visited))
       {
10.       S_MIN = Min(Cost(S'),Cost(S_MIN));
11.       S' = SGen(S');
12.       unvisited ← S';
13.    }
14.   }
15.   visited ← S;
16. }
17. return S_MIN;
18. End.
```

Fig. 18. Algorithm Exhaustive Search (ES).

from the affected activities toward 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. The *Exhaustive Search* algorithm (ES) (Fig. 18) 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. Afterward, we search the visited states and we choose the one with the minimal cost as the solution of our problem.

Heuristic Search. In order to avoid exploring the full state space, we employ a set of heuristics, based on simple observations, common practice (heuristic 4), and on the definition of transitions (heuristics 1, 2, and 3).

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 algorithm *Heuristic Search* (HS) is illustrated in Fig. 19 and explained in detail in [27].

HS-Greedy. One could argue that the first part of HS (lines 11-15) seems to be expensive, considering its repetition in the end of the algorithm. 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 these parts 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*.

4.3 Experimental Results

In order to validate our method, we implemented the proposed algorithms in C++ and experimented on the variation of measures like time (we present it in Fig. 20 as the volume of visited states), volume of processed rows, improvement, and 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 [20] 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 of various kinds. 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. This is the reason for the peak and then the decrement of the

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 MERs according to merg_cons;
5.  unvisited = ∅;
6.  visited = ∅;
7.  SMIN = S0;
8.  H ← Find_Homologous_Activities(S0);
9.  D ← Find_Distributable_Activities(S0);
10. L ← Find_Local_Groups(S0);
11. for each gi in L {
12.   for each pair (ai,aj) in gi {
13.      SNEW ← SWA(ai,aj);
14.      if (c(SNEW)<c(SMIN)) SMIN = SNEW;
15. }}
16. visited ← SMIN;
17. for each pair (ai,aj) in H {
18.   if((ShftFw(ai,ab))and(ShftFw(ai,ab))){
19.      SNEW ← FAC(ab,ai,aj);
20.   if (c(SNEW)<c(SMIN)) SMIN = SNEW;
21.      visited ← SNEW;
22. }}
23. unvisited = visited;
24. for each Si in unvisited {
25.   for each au in D {
26.     if (ShftBw(au,ab)) {
27.        SNEW ← DIS(ab,au);
28.        if (c(SNEW)<c(SMIN)) SMIN = SNEW;
29.        visited ← SNEW;
30. }}}
31. for each Si in visited {
32.   L ← Find_Local_Groups(Si);
33.   for each gi in L {
34.     for each pair (ai,aj) in gi {
35.        SNEW ← SWA(ai,aj);
36.        if (c(SNEW)<c(SMIN)) SMIN = SNEW;
37. }}}
38. apply all SPLs according to spl_cons;
39. return SMIN;
40.End.
```

Fig. 19. Algorithm Heuristic Search (HS).

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.

Fig. 20. Number of visited states and improvement with respect to initial state.

search space of ES algorithm depicted in Fig. 22a because, as the number of activities increases, the search space explored in a constant portion of time (40h) decreases.

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 (Fig. 21 and Fig. 22b). Fig. 20 depicts the number of visited states for each 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 by at least 86 percent (average value was 92 percent). For medium ETL workflows, HS finds a better solution than HS-Greedy (in a range of 13-38 percent). 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 percent of the cost of the initial state, while HS-Greedy returns "unstable" results in a low average value of 47 percent.

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 (Fig. 23). 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 percent. Finally, we mention that the variation of the volume of rows did not change the

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

Fig. 21. Quality of solution. The values with * are compared to the best of ES when it stopped.

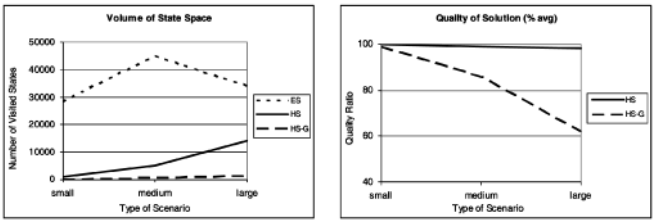

Fig. 22. (a) Volume of the state space produced by each one of the three algorithms. (b) Quality of solution.

aforementioned results. This was expected since our approach is general in that it does not in particular depend on the cost model chosen.

5 Related Work

In this section, we discuss the state of art and practice in the field of ETL along with any related research efforts. There exist a variety of ETL tools in the market; we mention a recent review [9] and several commercial tools [12], [13], [17], [19]. 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 [5], [7], [16]. Also, we mention three research prototypes: 1) AJAX [10], 2) Potter's Wheel [22], and 3) ARKTOS II [24], [26]. 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], [4], [15]. 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 we discuss in the context of ETL. Nevertheless, stream management techniques are not directly applicable to typical ETL problems 1) due to the fact that real time replication (necessary for the application of stream management techniques) is not always applicable to legacy systems and 2) pure relational querying, as studied in the field of stream management is not sufficient for ETL purposes. Also, to our knowledge, research in stream management does not provide an optimal solution to the optimization of the processes, basically due to the requirement for on-the-fly computation of results. For example, the authors of [1] verify that they cannot produce a globally optimal plan, although they do apply local optimization in an ad-hoc manner. Another technical part of our work concerns the derivation of the schemata of the involved activities. Quite a long line of research has dealt with the problem in its general context [3], [18], [25]; nevertheless, we need a fully automated solution for our particular centralized user-controlled setting, therefore, we devised our own solution to the problem.

6 Conclusions and Future Work

In this paper, we have concentrated on the problem of optimizing ETL workflows. 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. Activities are characterized by input and output schemata. 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 and the conditions under which transitions can be applied to states. Finally, we have presented search algorithms. First, we have described an exhaustive approach to construct the search space in its entirety in order to find the optimal ETL workflow. Then, we have introduced a heuristic algorithm and its greedy variant to reduce the explored search space. Experimental results suggest that the benefits of our method are significant. Several research issues are left open as a continuation of this work, including the physical optimization of ETL workflows, the smooth adaptation of the ETL workflow to changes in the schema of the underlying data stores, the exploitation of common tasks in different workflows and the generalization of our results to non-ETL workflows. The theoretical challenge of providing a complete set of transitions is also open.

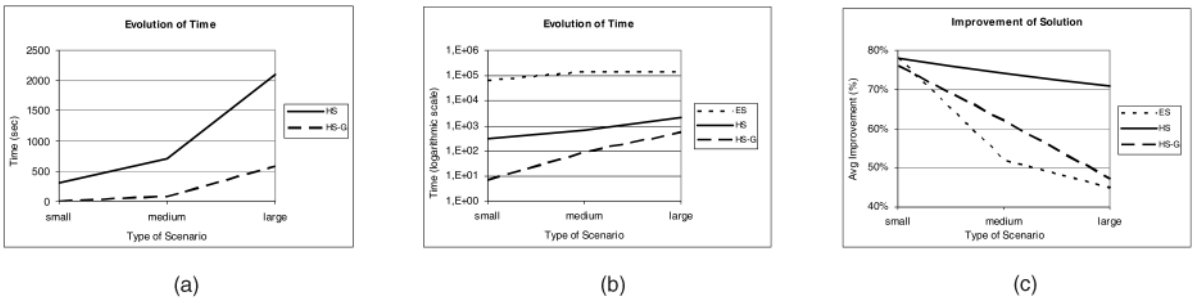

Fig. 23. Time characteristics for all three algorithms: (a) Evolution of time for HS and HS-G algorithms, (b) evolution of time for all three algorithms (in logarithmic scale), and (c) improvement of solution for all three algorithms.

REFERENCES

[1] D.J. Abadi et al. "Aurora: A New Model and Architecture for Data Stream Management," The Very Large Data Bases J., vol. 12, no. 2, pp. 120-139, 2003.

[2] J. Adzic and V. Fiore, "Data Warehouse Population Platform," *Proc. Fifth Int'l Workshop Design and Management of Data Warehouses,* 2003.

[3] S. Alagic and P.A. Bernstein, "A Model Theory for Generic Schema Management," *Proc. Eighth Int'l Workshop Database Programming Languages,* pp. 228-246, 2001.

[4] S. Babu and J. Widom, "Continuous Queries over Data Streams," *SIGMOD Record,* vol. 30, no. 3, pp. 109-120, 2001.

[5] V. Borkar, K. Deshmuk, and S. Sarawagi, "Automatically Extracting Structure from Free Text Addresses," *Bull. Technical Committee on Data Eng.,* vol. 23, no. 4, pp. 27-32, 2000.

[6] J. Chen, S. Chen, and E.A. Rundensteiner, "A Transactional Model for Data Warehouse Maintenance," *Proc. 21st Int'l Conf. Concept Modelling,* pp. 247-262, 2002.

[7] Y. Cui and J. Widom, "Lineage Tracing for General Data Warehouse Transformations," *The Very Large Data Bases J.,* vol. 12, pp. 41-58, 2003.

[8] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems.* Addison-Wesley Pubs, 2000.

[9] Gartner, "ETL Magic Quadrant Update: Market Pressure Increases," http://www.gartner.com/reprints/informatica/112769.html, 2002.

[10] H. Galhardas, D. Florescu, D. Shasha, and E. Simon, "Ajax: An Extensible Data Cleaning Tool," *Proc. ACM Int'l Conf. Management of Data,* p. 590, 2000.

[11] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 5, no. 2, pp. 73-170, 1993.

[12] IBM, "IBM Data Warehouse Manager," http://www-3.ibm.com/software/data/db2/datawarehouse, 2004.

[13] Informatica, "PowerCenter," http://www.informatica.com/products/data+integration/power-center/default.htm, 2004.

[14] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys,* vol. 16, no. 2, pp. 111-152, 1984.

[15] D. Lomet and J. Gehrke, "Special Issue on Data Stream Processing," *Bull. Technical Committee on Data Eng.,* vol. 26, no. 1, 2003.

[16] W. Labio, J.L. Wiener, H. Garcia-Molina, and V. Gorelik, "Efficient Resumption of Interrupted Warehouse Loads," *Proc. ACM Int'l Conf. Management of Data,* pp. 46-57, 2000.

[17] Microsoft, "Data Transformation Services," www.microsoft.com, 2004.

[18] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan, "Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice," *Information Systems,* vol. 19, no. 1, pp. 3-31, 1994.

[19] Oracle Corp., "Oracle9i Warehouse Builder User's Guide, Release 9.0.2," http://otn.oracle.com/products/warehouse/content.html, Nov. 2001.

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

[21] E. Rahm and H. Do, "Data Cleaning: Problems and Current Approaches," *Bull. Technical Committee on Data Eng.,* vol. 23, no. 4, pp. 3-13, 2000.

[22] V. Raman and J. Hellerstein, "Potter's Wheel: An Interactive Data Cleaning System," *Proc. 27th Intl. Conf. Very Large Data Bases,* pp. 381-390, 2001.

[23] D. Theodoratos and T.K. Sellis, "Data Warehouse Configuration," *Proc. 23rd Int'l Conf. Very Large Data Bases,* pp. 126-135, 1997.

[24] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos, "Modeling ETL Activities as Graphs," *Proc. Fourth Int'l Workshop Design and Management of Data Warehouses,* pp. 52-61, 2002.

[25] Y. Velegrakis, R.J. Miller, and L. Popa, "Mapping Adaptation under Evolving Schemas," *Proc. 29th Int'l Conf. Very Large Data Bases,* pp. 584-595, 2003.

[26] P. Vassiliadis, A. Simitsis, P. Georgantas, and M. Terrovitis, "A Framework for the Design of ETL Scenarios," *Proc. 15th Conf. Advanced Information System Eng.,* pp. 520-535, 2003.

[27] A. Simitsis, P. Vassiliadis, and T. Sellis, "Optimizing ETL Processes in Data Warehouses," *Proc. 21st IEEE Int'l Conf. Data Eng.,* pp. 564-575, 2005.

Alkis Simitsis received the PhD degree from the School of Electrical and Computer Engineering of the National Technical University of Athens (NTUA) in 2004. His research interests include extraction-transformation-loading (ETL) processes in data warehouses and query processing/optimization.

Panos Vassiliadis received the PhD degree from the National Technical University of Athens in 2000. He is a lecturer at the University of Ioannina. His research interests include data warehousing, Web services, and database design and modeling. He has published more than 25 papers in refereed journals and international conferences in the above areas.

Timos Sellis received the PhD degree from the University of California at Berkeley in 1986. He is a full professor at the National Technical University of Athens. His research interests include extended relational database systems, data warehouses, and spatial, image, and multimedia database systems. He has published more than 100 articles in refereed journals and international conferences in the above areas.