Deciding the Physical Implementation of ETL Workflows

Vasiliki Tziovara
University of Ioannina
Ioannina, Hellas
vickit@cs.uoi.gr

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

Alkis Simitsis
IBM Almaden Research Center
San Jose, California, USA
asimits@us.ibm.com

ABSTRACT

In this paper, we deal with the problem of determining the best possible physical implementation of an ETL workflow, given its logical-level description and an appropriate cost model as inputs. We formulate the problem as a state-space problem and provide a suitable solution for this task. We further extend this technique by intentionally introducing sorter activities in the workflow in order to search for alternative physical implementations with lower cost. We experimentally assess our method based on a principled organization of test suites.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*data models, schema and subschema*

General Terms

Algorithms, Design

Keywords

Data Warehousing, ETL, Physical Design, Optimization

1. INTRODUCTION

Extract - Transform - Load (ETL) tools are special purpose software artifacts used to populate a data warehouse with up-to-date, clean source records. To perform this task, a set of operations should be applied on the source data. Nowadays, the majority of ETL tools organize such operations as a workflow. At the logical level, an ETL workflow can be considered as a directed acyclic graph (DAG) used to capture the flow of data from the sources to the data warehouse. The nodes of the graph are either recordsets used for storage purposes or transformation and cleansing activities that reject problematic records and reconcile data to the target warehouse schema. The edges of the graph represent input and output relationships between the nodes. Eventually, such an abstract logical design has to be implemented physically; i.e., to be mapped to a combination of executable programs and scripts that perform the ETL process. In this context, a logical-level activity can be physically implemented using various algorithmic methods, each with different cost in terms of time requirements or system resources (e.g., memory and space on disk).

An ETL process is extremely complex, error-prone, and time consuming [5]. Therefore, its optimization in terms of execution cost is a continuously increasing necessity. Surprisingly, despite its importance, no full-blown optimization technique for ETL processes has been provided either by the commercial ETL tools, or the related research efforts. To the best of our knowledge, the common practice indicates that the underlying DBMS should undertake the task of optimization. However, this policy does not allow the optimization of the whole ETL workflow, but only of isolated parts of it. To deal with the optimization of the ETL design, in a previous line of research, we have proposed a method for the logical optimization of ETL processes [4, 5]. Nevertheless, a limitation of that work is that the physical implementation of the ETL operations is not taken into consideration.

The objective of this work is to identify the best possible physical implementation for a given logical ETL workflow. The problem would not be novel if it concerned the optimization of the physical execution of queries in relational and post-relational environments. Still, *ETL workflows are NOT big queries*: their structure is not a left-deep or bushy tree, black box functions are employed, there is a considerable amount of savepoints to aid faster resumption in cases of failures, and different servers and environments are possibly involved. Moreover, frequently, the objective is to meet specific time constraints w.r.t. both regular operation and recovery (rather than the best possible throughput).

In our approach, different alternatives for the physical execution of each logical-level activity are employed. To facilitate the task of the designer, an automated mechanism for the mapping of a logical activity to a valid physical implementation is presented, based on a library of reusable templates for both logical and physical activities. The problem is modeled as a state-space search problem and different alternatives for state generation are considered for discovering the optimal physical implementation of a scenario. A generic cost model is considered as a discrimination criterion between physical representations, which is suitable for black-box activities with unknown semantics as well. Still, the determination of the best possible implementation for each activity in isolation does not guarantee the optimal implementation of the workflow; thus, a more elaborate search is required. To this end, the design is enriched by an additional set of special-purpose activities, called sorter activities, that apply on different positions of the design and sort the respective tuples according to the values of some, critical for the subsequent flow, attributes. Thus, more alternative physical implementations for a logical activity are provided; e.g, a join activity can use a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DOLAP'07, November 9, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-827-5/07/0011 ...$5.00.

Merge Join implementation once data arrive sorted at its inputs.

Finally, due to the lack of any standard, commonly agreed set of test suites for ETL workflows, we build upon a taxonomy for ETL workflows that classifies typical real-world ETL workflows in different template structures, to which we refer as *butterflies*, because of the shape of their graphical representation [8]. Our experimental evaluation is organized according to this classification and computes the best possible physical implementation for logical ETL workflows. The experiments demonstrate that the intentional introduction of sorters can make the difference in the determination of the final solution in several cases.

Roadmap. In Section 2, we formulate the problem of designing the physical implementation of ETL workflows as a state-space search problem. In Section 3, we incorporate extensions into the problem. In Section 4, we present our approach for obtaining the optimal physical configuration of an ETL scenario. In Section 5, we demonstrate our experimental evaluation. In Section 6, we discuss related work and in Section 7, we conclude with a prospect of the future.

2. PROBLEM FORMULATION

In this section, we present the formal definition of the problem under consideration. In what follows, we will employ the term *recordsets* to refer to any data store that obeys a schema (with *relational tables* and *record files* being the most popular kinds of recordsets in the ETL environment), and the term *activity* to refer to any software module that processes the incoming data, either by performing any schema transformation over the data or by applying data cleansing procedures. Activities and recordsets are *logical abstractions* of physical entities. At the logical level, we are interested in their schemata, semantics, and input-output relationships; however, we do not deal with the actual algorithm or program that implements the logical activity or with the storage properties of a recordset. When at a later stage, the logical-level workflow is refined at the physical level, the best possible implementation for an activity or physical layout for a recordset is considered, in order to optimize the execution cost of the workflow.

Each recordset has a *schema* that consists of a finite list of attributes. Each activity is characterized by the following properties: (a) a *name* (a unique identifier for the activity), (b) one *input schema* (*unary activity*) or *two input schemata* (*binary activity*), and (c) exactly one *output schema*. The following distinction holds for logical and physical activities:

Logical-level activity. A logical activity is considered as the declarative description of the relationship of its output schema with its input schema without delving into algorithmic or implementation issues. For example, an activity performing a logical-level join of tables R and S (denoted as $R \bowtie S$) does not provide further information on the implementation technique followed for the activity's execution.

Physical-level activity. A physical activity includes detailed information on the implementation techniques that must be employed for the activity's execution. For example, assume an activity a_1 which performs the join of tables R and S. If we define that this join is implemented using the Nested-Loops algorithm, the execution engine is informed about the inner procedures followed to produce the result of the join and can be aware of the cost of this implementation w.r.t. time or system resources. A physical implementation of an activity is characterized by the following elements:

- The algorithm, and executable code, employed for the physical execution of the activity.
- The implementation cost in terms of time or system resources such as memory or disk space allocation.
- A set of input preconditions, i.e., a set of conditions that the input data must hold, in order for the physical activity to be executed. Certain physical implementations can be employed only if specific conditions are met by the source data. For example, a Merge Join requires both inputs to be sorted on the join attribute.
- A set of extra constraints possibly defined by the designer, to guarantee certain properties of the solution; e.g., that the the resources consumed, the resumption cost of the scenario or the execution cost of a part of it do not exceed certain thresholds.

ETL workflows. Formally, we model an ETL workflow as a directed acyclic graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$. Each node $v \in \mathbf{V}$ is either an activity $a \in \mathbf{A}$ or a recordset $r \in \mathbf{R}$. An edge $(a, b) \in \mathbf{E}$ denotes that b receives data from node a for further processing. In this *provider relationship*, node a plays the role of the data provider, while node b is the data consumer. The following well-formedness constraints determine the interconnection of nodes in ETL workflows:

- The data consumer of a recordset cannot be another recordset. Still, more than one consumer is allowed for recordsets.
- Each activity must have at least one provider, either another activity or a recordset. When an activity has more than one data providers, these providers can be other activities or activities combined with recordsets.
- Each activity must have exactly one consumer, either another activity or a recordset.
- Feedback of data is not allowed; i.e., the data consumer of an activity cannot be the same activity.

Templates. To facilitate the mapping of a logical-level activity to its alternative physical-level implementations, we consider a template library for activities that are customized per scenario. A template activity is a customizable design artifact with commonly agreed semantics concerning its ability of performing a specific task over its -yet unspecified- input data. Thus, each template has a set of properties, which involve some predefined semantics and a set of parameters that, once set, determine the functionality of the template.

In our setting, we employ two categories of templates: (a) *logical* templates that materialize logical-level activities, and (b) *physical* templates that materialize physical-level activities. Similar to logical and physical activities, there is a 1:N mapping between logical and physical templates, depicted in Figure 1.

Using logical and physical templates, a certain physical-level activity is materialized as follows. First, for a given logical activity a logical template from the template library is picked. Then, for the creation of the activity's logical instance, the necessary schemata and concrete values for the logical template parameters should be specified. Next, the logical-to-physical mapping of templates produces one or more physical templates, and finally, the physical template chosen, according to certain constraints and preconditions, is instantiated to a physical instance.

For example, consider the case of a logical activity performing a *Not Null* check on the source data stemming from a recordset $R(A, B, C)$, over attribute B. The appropriate logical template for this case is the $NotNull_T^L$. This template requires the customization of (a) its input schema, which is set to (A, B, C), (b) its output schema, which in this case, is identical to the input schema,

$$\begin{tikzcd}
\textit{Logical activity} & a^L \arrow[r, "T"] & t^L \arrow[d, "\mathcal{M}(1:N)"] & \textit{Logical template} \\
\textit{Physical activity} & a_P & t^P \arrow[l, "C"'] & \textit{Physical template}
\end{tikzcd}$$

Figure 1: The mechanism for the mapping of a logical activity to a physical implementation

and (c) a parameter NN_attr, which is the name of the attribute over which the Not Null check will be performed; e.g., attribute B. Therefore, the generic semantics of the template activity's output that are $\{x \mid x \in Input(\emptyset), x.NN\_attr \neq NULL\}$, are customized as $\{x \mid x \in Input(A, B, C), x.B \neq NULL\}$.

In general, different implementations can be chosen for the same logical activity, depending on the physical characteristics of the data. For instance, the logical template $NotNull_T^L$ can be associated with two physical-level templates: $Filter_T^P$ and $Sorted\_filter_T^P$. The first checks all the input extensions for records that fulfill the filtering condition, whereas the second requires a precondition to be met: a total order of the input over the checked value, and stops its execution whenever the last record with the particular value is met. (If $NULL$ is the last value in the total order of the domain of attribute B, once the first record with a $NULL$ value is met, the activity completes its execution.)

Not all the physical implementations constitute legal mappings. For instance, the usage of a $Sorted\_filter_T^P$, in the above example is only allowed in the presence of sorted data over a specific attribute. Therefore, each physical implementation has a set of *preconditions* that determine whether its usage in a certain scenario is valid or not. To confront with this constraint, in Section 3, we discuss the possibility of artificially introducing sorters in the workflow to allow a broader range of available implementations in order to improve the execution cost of the ETL process.

Assume a template library containing a set of logical template activities $L^L = \{t_1^L, t_2^L, \ldots, t_n^L\}$, a set of physical template activities $L^P = \{t_1^P, t_2^P, \ldots, t_k^P\}$ with $n \leq k$, and a $1{:}N$ mapping $\mathcal{M}$ among activities from both sets that maps each logical template to a set of valid physical template implementations.

Let $\mathbf{A}$ be an infinitely countable set of activities and $\mathbf{\Omega}$ an infinitely countable set of templates. We define a mapping T, such that: $T : \mathbf{A} \to \mathbf{\Omega}$. Thus, for an activity $a \in \mathbf{A}$ that is a materialization of a template $t \in \mathbf{\Omega}$, the following holds: $T(a) = t$. We define also the mapping C as the inverse of T. Then, $C : \mathbf{\Omega} \to \mathbf{A}$. Thus, for a certain ETL scenario, the customization of a template $t \in \mathbf{\Omega}$ for an activity $a \in \mathbf{A}$ is given by the mapping: $C(t) = a$. The mappings T and C apply both to logical and physical level activities.

For a physical activity a^P, $constr(a^P)$ is a predicate in a simple conjunctive form denoting the set of constraints $c_i = 1, \ldots, q$ that exist for the physical implementation of activity a^P; i.e., $constr(a^P) = \{c_1 \wedge c_2 \wedge \cdots \wedge c_q\}$.

Cost. Also, $cost(a^P)$ is a function that returns the cost of a physical activity a^P. This cost depends on (a) the cost model considered; (b) the position of the activity in the workflow (specifically, its cardinality m_i); and (c) the physical implementation that is selected for its execution. Activities that implement well-known relational operators can be accompanied with detailed cost models. Still, there are also activities that depart from the typical relational operators. For such a black-box activity i with known or unknown semantics, the following generic formula estimates its computational cost, $cost_C$:

$$cost_C(i) = m_i \times cpt(i) \tag{1}$$

where $cpt(i)$ is the cost of activity i to process a single record. In this case, micro-benchmarks can be used for the estimation of $cpt(i)$. The total cost of a physical ETL scenario comprising n activities is obtained by summarizing the costs of all its activities and is given by the following formula:

$$Cost(G) = \sum_{i=1}^{n} cost(a_i^P) \tag{2}$$

The State-Space Nature of the Problem. We model the problem of finding the physical implementation of an ETL process as a state-space search problem. Given a logical-level ETL workflow, $\mathbf{G^L}$, a set of physical-level scenarios is generated. These scenarios have equivalent semantics with the original logical scenario, but may have different costs. Our objective is to determine the physical-level scenario $\mathbf{G^P}$ such that (i) all logical recordsets, activities, and (ii) provider edges are mapped to their physical counterparts, (iii) all constraints are respected and, (iv) $\mathbf{G^P}$ has the minimal cost among all other alternative solutions explored.

States. A state is a graph $\mathbf{G^P}$ that represent a physical-level ETL workflow. The initial state $\mathbf{G_0^P}$ is produced after the random assignment of physical implementations to logical activities w.r.t. preconditions and constraints.

Transitions. Given a state $\mathbf{G^P}$, a new state $\mathbf{G^P}'$ is generated (a) by replacing the implementation of a physical activity a^P of $\mathbf{G^P}$ with another valid implementation for the same activity and (b) by the introduction of a physical-level sorter activity as a new node in the graph.

Formal definition of the problem. Given a logical-level ETL workflow $\mathbf{G^L}(\mathbf{V^L}, \mathbf{E^L})$, where $\mathbf{V^L} = \mathbf{A^L} \cup \mathbf{R}$, $\mathbf{A^L}$ is the set of logical activities and $\mathbf{R}$ is the set of recordsets, determine the physical-level graph $\mathbf{G^P}(\mathbf{V^P}, \mathbf{E^P})$, where $\mathbf{V^P} = \mathbf{A^P} \cup \mathbf{R}$, $\mathbf{A^P}$ is the set of physical activities, such that:

- $a_i^P = C(\mathcal{M}(T(a_i^L)))$, $a_i^P \in \mathbf{A^P}$, $a_i^L \in \mathbf{A^L}$
- $\forall e^L = (x^L, y^L)$, $e^L \in \mathbf{E^L}$, $x^L, y^L \in \mathbf{V^L}$, introduce e^P to $\mathbf{E^P}$ where $e^P = (x^P, y^P)$, $x^P, y^P \in \mathbf{V^P}$
- $\wedge_i constr(a_i^P) = true$, $a_i^P \in \mathbf{A^P}$
- $\sum_i cost(a_i^P) = minimal$, $a_i^P \in \mathbf{A^P}$

3. ADDITION OF SORTERS

Traditional query optimization is based on the notion of interesting orders to find the optimal plans for relational database queries [3]. In our context, we can exploit orderings of data in several ways, mainly by applying fast algorithms in the presence of orderings (e.g., Merge-Join and Merge-based aggregation are the fastest algorithms for their respective families, but they depend upon an appropriate pre-existing ordering of the data to be applicable). Still, algorithms in traditional query processing exploit existing orderings to determine the best execution plan. In our case, we aim at deriving an algorithm that exploits the possibility of *intentionally introducing orderings, towards obtaining physical plans of lower cost*. To this end, we introduce special-purpose, physical-level transformations, called Sorter activities or Sorters. Sorter activities apply on stored recordsets and rearrange the order of their input tuples according to the values of some specified attributes.

Impact of Adding Sorters. Adding sorter activities to a physical-level graph does not impose changes to the functionality of the

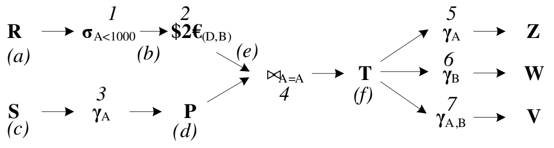


Figure 2: Candidate places for sorters

workflow: each graph node produces the same output tuples as it used to in the original scenario. The addition of each sorter activity to the graph causes an increase to the total cost of the workflow, of the order of $O(n * log\ n)$, where n is the number of tuples the sorter has to order. This raise of the workflow's operational cost is significant, especially for large values of n. On the other hand, if the ordering imposed by the sorter can be exploited by activities that follow the sorter due to the adoption of a cheaper implementation technique that exploits this ordering, the cost of these activities declines considerably. Altogether, as we will discuss in Section 5, there exist several cases where the total cost of the workflow can be reduced significantly and the gain in the performance of the scenario can be crucial.

Issues Raised by the Introduction of Sorters. The following issues need to be addressed regarding the introduction of sorters: (a) where the orderings of data should be introduced; (b) over which attributes the orderings should be applied; and (c) what type of ordering (ascending or descending) should be preferred.

Candidate Positions for the Introduction of Sorter Activities. Given a logical graph $\mathbf{G^L(V^L, E^L)}$, our first consideration is to discover all the candidate places to insert sorter activities, and specifically: (a) source tables; (b) data staging area (DSA) tables; and (c) edges that connect two activities, i.e., edges whose data provider and data consumer are both activities.

Figure 2 shows an ETL scenario involving two source databases R and S, a DSA table P, a fact table T, and three materialized views Z, W and V. There are also 7 activities, performing filtering, conversion, join, and aggregation. Figure 2 illustrates the candidate positions to place sorter activities. Candidate positions for sorters are marked with letters enclosed in parentheses (places (a) to (f), i.e., on the data of tables R, S, P, and T and on the edges among activities, i.e., edges $(1, 2)$ and $(2, 4)$.

Interesting Orders for Sorters. A second issue concerns the choice of the attributes of each recordset's schema, or activity's output schema, over which an ordering of data will be performed. We adopt the traditional technique of [3], in the setting of which, an interesting order is a set of attributes present in the join, grouping, and ordering conditions of a query. In a broader sense, an interesting order is a set of attributes such that, an ordering of the data over them can lead to a cheaper evaluation plan for a query. Therefore, our objective is to identify the set of interesting orders for each activity of the workflow. The interesting orders for an activity are defined at the physical-level templates. Then, they are customized per scenario.

Consider the example for the $NotNull^L_T$ template, discussed in Section 2, that corresponds to two physical implementations $Filter^P_T$ and $Sorted\_filter^P_T$. The template $NotNull^L_T$ comprises a parameter NN_attr, to be specified by the designer. The interesting order for the template is NN_attr and it is customized to the value of this attribute. If the data are sorted by the attribute of the interesting order, i.e., by NN_attr, then both physical implementations apply. Otherwise, only the implementation $Filter^P_T$ is applicable.

In general, the interesting order of a sorter S_X depends on its position on the workflow and on the activities in the output of the sorter. We discern two usual cases for sorters. The first case occurs when *a sorter activity is placed between two subsequent activities* a and b (e.g., activities 1 and 2 in Figure 2), the candidate attributes X for orderings depend exclusively on activity b. In other words, the interesting orders of activity b determine the ordering X imposed by the sorter S_X. The second case occurs when *a sorter activity directly affects a source or a DSA relation*, the output of the respective relation should be examined. We already mentioned that recordsets can forward their data to more than one destination. Thus, we discover the interesting orders of the activities that receive data from the relation. Then, the interesting orders identified are combined into a single set, under the condition that there is no overlap of interesting orders, i.e., each interesting order is considered once in the set. Regarding this consideration, observe the example of Figure 2. The fact relation T populates three aggregators. The first one aggregates data by attribute A, so the interesting order is $\{A\}$. The second aggregator has as interesting order the set $\{B\}$, while the third one has an interesting order $\{A, B\}$. Therefore, the set of interesting orders for relation T is $\{\{A\}, \{B\}, \{A, B\}\}$.

Ascending vs. Descending Ordering of Data. For lack of space, we simply mention that the appropriate ascending or descending order is considered depending on the semantics of the activity that receives the sorted output of a sorter and refer the interested reader to [7] for more details. For example, a filter of the form $\sigma_{A<1000}(R)$ (Figure 2) requires a sorter in ascending order for performance gains. The order would be descending if the condition was $\sigma_{A>1000}(R)$.

4. OBTAINING THE OPTIMAL PHYSICAL SCENARIO

In this section, we first describe signatures, which are used as string representations of physical-level scenarios and then we present a method towards the identification of the best possible physical implementation of a logical scenario.

Signatures. To efficiently implement any algorithm that generates states representing physical implementations of ETL workflows, we need a compact way to represent these flows. In this paper, we extend the idea of representing an ETL scenario with a string, called *signature*, originally introduced in [4], with (a) DSA tables that may have more than one output, (b) more than one target recordsets, (c)the physical implementation of activities, and, (d) any sorters artificially introduced in the workflow.

Definition. A signature is a string that represents a graph $G = (V, E)$ and it is formed using the following rules: (a) a physical implementation p of a logical level activity a is denoted as $a@p$, (b) the names of the activities that form a linear path are separated with dots ("."), (c) concurrent paths are delimited by a double slash ("//"), while, each path is enclosed in parentheses, (d) whenever a sorter that orders data according to attributes A,B is placed on an edge (a, b) among activities a and b, the sorter is named a_b and the signature contains the notation $a\_b(A, B)$ to convey the sorter, and (e) a sorter introduced on table V that orders tuples according to A,B is denoted as $V!(A, B)$.

For example, one possible signature for the scenario depicted in Figure 2 is:

$$((R.1.2)//(S.S!(A).3@HB.P)).4@NL.T.$$
$$\hookrightarrow \quad (((5@SO.Z)//(6@SO.W))//(7@SO.V))$$

The signature shows that (a) data on S is sorted on attribute A, (b) the aggregation of activity 3 is based on hashing (HB), (c) the join activity 4 is based on nested loops (NL) and (d) the three last aggregation activities are based on sorting (SO). The following signature:

```
Algorithm: Exhaustive Ordering (EO)
Input: An logical graph $\mathbf{G^L} = (\mathbf{V^L}, \mathbf{E^L})$ with $n$ nodes
Output: A signature $S_{MIN}$ having minimal cost
1  Begin
2      $S_0 \leftarrow Compute\_Signature$
3      $Cost(S_0) \leftarrow Compute\_Cost(S_0)$, $S_{MIN} = S_0$
4      Let $D$ be a dictionary that contains signatures and
       respective costs, add $S_0$ and $Cost(S_0)$ to $D$
5      Let $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ be the set of all possible
       combinations of candidate positions for sorters
6      Given a $\gamma \in \Gamma$, let $P_\gamma = \{p_{\gamma_1}, p_{\gamma_2}, \ldots, p_{\gamma_k}\}$ be the set of
       possible positions of combination $\gamma$
7      Given a position $p_\gamma$, let $O_{cs} = \{o_1, o_2, \ldots, o_n\}$ be the set
       of candidate sorters over $p_\gamma$ (including no sorter)
8      Foreach $\underline{\gamma \in \Gamma}$ {
9          Foreach $\underline{p_\gamma \in P_\gamma}$ {
10             Foreach $\underline{o \in O_{cs}}$ {
11                 generate a new signature $S'$
12                 If $\underline{(S' \notin D)}$ {
13                     $Cost(S') \leftarrow Compute\_Cost(S')$
14                     Store $S'$ and $Cost(S')$ to $D$
15                     If $\underline{(Cost(S') < Cost(S_{MIN}))}$ {
16                         $S_{MIN} = S'$
17                 }}
18     }}}
19     Return $\underline{S_{MIN}}$
20 End
```

Figure 3: Algorithm Exhaustive Ordering

$$((R.1.2.2\_4(A))//(S.S!(A).3@HB.P)).4@MJ.T.$$
$$\hookrightarrow \quad (((5@SO.Z)//(6@SO.W))//(7@SO.V))$$

contains a sorter between activities 2 and 4, so that data are sorted on attribute A for both sources and a Merge Join (MJ) takes place for activity 4.

Obtaining the best physical implementation. The algorithm *Exhaustive Ordering* (EO), depicted in Figure 3, takes as input an initial logical-level graph $\mathbf{G^L} = (\mathbf{V^L}, \mathbf{E^L})$ with n nodes and generates all possible states that can be generated by placing all possible sorters over the candidate positions of the graph and by using all possible combinations of the different physical implementation for each activity. The algorithm proceeds in finding the state having minimal cost and returns it as output.

In other words, algorithm *EO* generates the initial state, signature S_0, by invoking the function $Compute\_Signature$ (line 2). For the logical-to-physical mapping for the initial state, the physical implementation for each activity is randomly chosen among the valid alternative implementations. Then, the cost of the initial scenario is evaluated by the function $Compute\_Cost()$ (line 3). Both S_0 and its relevant cost are stored to dictionary D (line 4). The algorithm needs to compute all possible combinations of positions where sorters can be artificially introduced as described in the Section 3 (line 8). Assuming the example of Figure 2, all combinations of positions $(a), (b), \ldots, (f)$ constitute the set Γ (line 5). For each such combination, e.g., $\{a, b, d\}$, the set P_γ involves the positions that are the members of the set (Line 6). For each such position, (line 9), the algorithm determines all candidate sorters that can be inserted among graph nodes (all candidate sorters compose the set O_{sc} defined in line 7) and uses them to generate all possible signatures that can be produced if one or more sorters are added to the initial graph (line 10). Each new signature S' produced corresponds to a new scenario that differs from the original scenario in the sense that (a) it contains one or more additional sorter activities or (b) it contains alternative implementation methods for the activities (line 11). Then, if the new signature S' does not already exists in dictionary D (line 12), its cost is computed and added along with the signature S' to the dictionary (lines 13-14). Finally, the signature having the minimal cost (S_{MIN}) is returned as the solution to the problem.

For more details and explanations on implementation issues for functions $Compute\_Cost()$ and $Compute\_Signature$, we refer the interested reader in [7].

5. EXPERIMENTS

In this section, we present a series of experiments for various types of workflows. However, as far as we are aware of, in the literature and practice there is a lack of standard benchmark or experimental setup for ETL workflows. In [8], we have introduced a principled method of constructing ETL workflows, which we adopt for the purposes of this paper. The main constructs of the experimental method are a broad category of workflows, called *Butterflies* due to their shape. The following section briefly summarizes the main ideas of [8]. Then, we present our experimental results.

5.1 Butterflies as an Experimental Method

A butterfly is an ETL workflow that consists of three distinct components: (a) the left wing, (b) the body and (c) the right wing of the butterfly. The left and right wings are two non-overlapping groups of nodes which are attached to the body of the butterfly. Specifically:

- The left wing of the butterfly includes one or more sources, activities and auxiliary data stores used to store intermediate results. This part of the butterfly performs the ETL processing of the workflow and forwards the processed data to the body of the butterfly.
- The body of the butterfly is a detailed warehouse (a.k.a. *fact*) table that is populated with the cleansed and transformed data produced by the left wing. Still, other structures like *dimension* tables can act as bodies of a butterfly, too.
- The right wing gets the data stored at the body and utilizes them to support reporting and analysis activity. The right wing consists of materialized views, reports, spreadsheets, as well as the activities that populate them. In our setting, we abstract all the aforementioned static artifacts as materialized views.

Regarding the example of Figure 5, the body of the butterfly is the fact table V. The left wing of the butterfly, includes the source relations R and S, the DSA relation P, and the activities 1, 2, and 3. The right wing of the butterfly includes the materialized aggregate views Z and W, as well as the aggregation activities 4 and 5.

Balanced Butterflies. A butterfly that includes medium-sized left and right wings is called a *Balanced butterfly* and stands for a typical ETL scenario where incoming source data are merged to populate a warehouse table along with several views or reports defined over it. Figures 2 and 5 are examples of this class of butterflies. Still, we have constructed several butterfly variants, in order to cover other real-world cases, too, where there is no symmetry between the left and the right part. We classify these variants according to their graph structure (see Figure 4(b)). In terms of graphical notation, the bodies of the butterflies are underlined. The two fundamental wing components can be either Lines or Combinations, which we present right ahead.

Lines. Lines are sequences of activities and recordsets such that: (a) no recordsets are directly linked; and (b) all activities have ex-

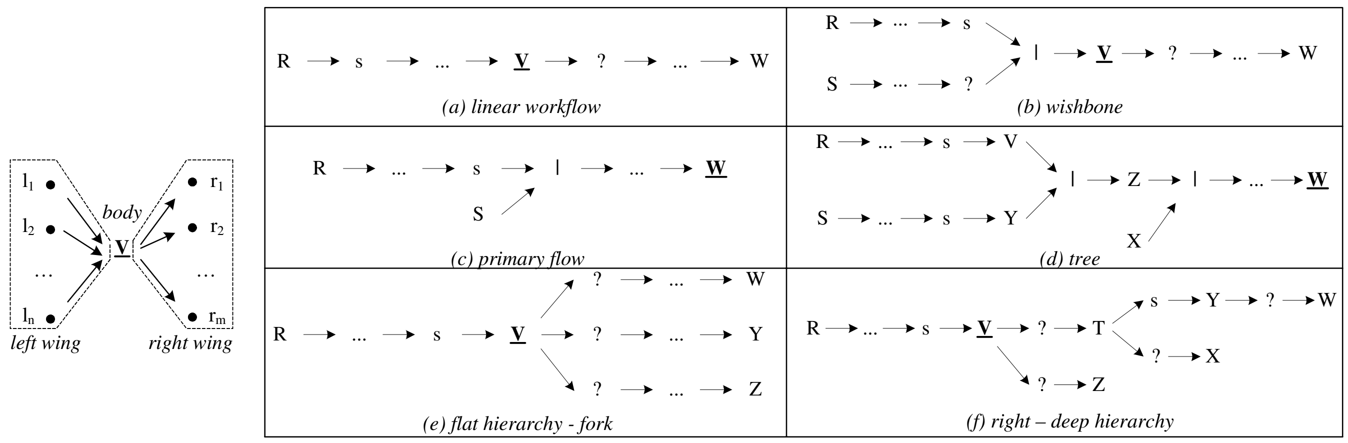


Figure 4: Butterfly components and classes

actly one input and one output, i.e., unary activities. In these workflows, nodes form a single data flow.

Combinations. A combinator activity is a binary activity that merges parallel data flows through some join (e.g., a relational join, or diff operation) or union variant. A combination is built around a combinator with lines or other combinations as its inputs. We differentiate combinations as left-wing and right-wing combinations.

Left-wing combinations are constructed by lines and combinations forming the left wing of the butterfly. The left wing contains at least one combination. The inputs of the combination can be:

- Two lines. Two parallel data flows are unified into a single flow using a combination. These workflows are shaped like the letter Y and we call them *Wishbones*.
- A line and a recordset. This refers to the practical case where data are processed through a line of operations, some of which require a lookup to persistent relations. In this setting, the *Primary Flow* of data is the line part of the workflow.
- Two or more combinations. The recursive usage of combinations leads to many parallel data flows. These workflows are called *Trees*.

Observe that in the cases of trees and primary flows, the target warehouse acts as the body of the butterfly (i.e., there is no right wing). This situation covers (a) fact tables without materialized views and (b) the case of dimension tables that also need to be populated through an ETL workflow.

Right-wing combinations are constructed by lines and combinations on the right wing of the butterfly. These lines and combinations form either a flat or a deep hierarchy.

- *Flat Hierarchies*. These configurations have small depth (usually 2) and large fan-out. An example of such a workflow is a *Fork*, where data are propagated from the fact table to the materialized views in two or more parallel data flows.
- *Right - Deep Hierarchies*. To push the experiments to their limits, we employ configurations with *right-deep hierarchies*. These configurations have significant depth and medium fan-out. Figure 4 (f) shows a right-deep hierarchy with depth 6.

General issues. In the above analysis as well as in the examples introduced so far, in order to simplify the presentation, we have considered activities such as filters and aggregations. In practice, more complex and composite activities are also used; e.g., pivot, difference, slowly changing dimensions (SCD), and so on. In addition, nowadays, most commercial ETL tools support activities with more than one output; e.g., splitter and switch. Our principled

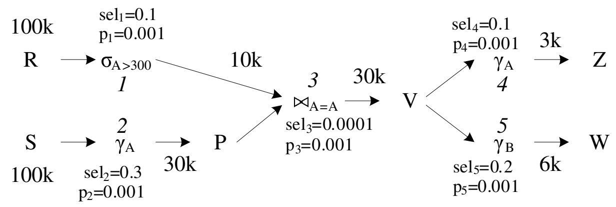


Figure 5: Balanced butterfly

method for constructing ETL workflows supports any generic type of activity [8]. However, for consideration of space, in the rest of the paper we will not elaborate more on this issue.

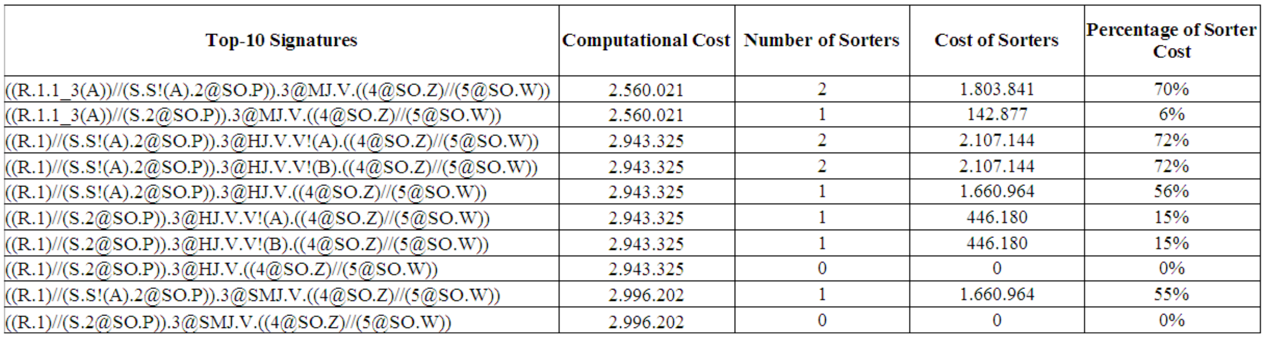
5.2 Experimental results

In this section, we experimentally evaluate the proposed approach using different classes of ETL workflows. All the experiments were conducted on an Intel(R) Pentium(R) M running at 1,86 GHz with 1 GB RAM and the machine has been otherwise unloaded during experiments. For lack of space, in this subsection we present detailed results for balanced and right-deep butterflies only. In Section 5.3, we briefly describe our findings for the rest butterfly classes. For more details on our experimental analysis, we refer the interested reader in [7].

Balanced Butterflies. First, we start with "equi-weight" balanced butterflies. The workflow characteristics for the scenario of Figure 5 are: (a) number of nodes = 11, (b) time to discover optimal scenario = 28 sec, and (c) total number of generated signatures = 181.

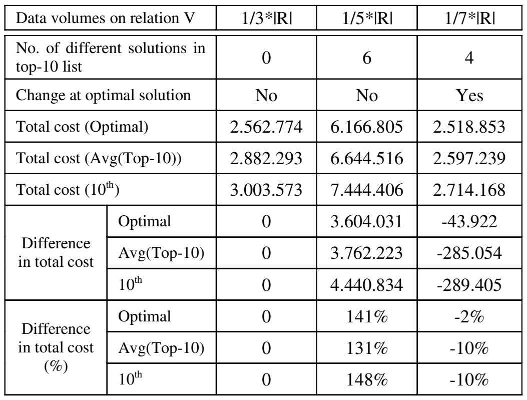
Figure 6 shows the 10 signatures having the lowest total cost for the scenario. Observe that the optimal physical representation for this butterfly contains a sorter on edge 1_3 and a sorter on table S. Observe also that this scenario has a highly selective left wing. Thus, the introduction of a sorter is beneficial even for the source data of S. Furthermore, the difference between the 10th solution and the optimal signature is small (in particular 17%).

Overhead of Sorters. Figure 6 presents the number of sorters contained in each of the aforementioned top-10 signatures, the cost of the sorters, and the percentage of sorters' cost. Observe that the cost of the sorters is significant for the best solution (70% as percentage). This means that the addition of the two sorters minimizes the cost of the rest of the activities so much, that the total cost of the entire scenario is minimized, so that this solution is definitely a winner compared to the best possible solution without any sorters. The fluctuation of the sorter's overhead, nevertheless, is impres-



Top-10 Signatures	Computational Cost	Number of Sorters	Cost of Sorters	Percentage of Sorter Cost
((R.1.1_3(A))//(S.S!(A).2@SO.P)).3@MJ.V.((4@SO.Z)//(5@SO.W))	2.560.021	2	1.803.841	70%
((R.1.1_3(A))//(S.2@SO.P)).3@MJ.V.((4@SO.Z)//(5@SO.W))	2.560.021	1	142.877	6%
((R.1)//(S.S!(A).2@SO.P)).3@HJ.V.V!(A).((4@SO.Z)//(5@SO.W))	2.943.325	2	2.107.144	72%
((R.1)//(S.S!(A).2@SO.P)).3@HJ.V.V!(B).((4@SO.Z)//(5@SO.W))	2.943.325	2	2.107.144	72%
((R.1)//(S.S!(A).2@SO.P)).3@HJ.V.((4@SO.Z)//(5@SO.W))	2.943.325	1	1.660.964	56%
((R.1)//(S.2@SO.P)).3@HJ.V.V!(A).((4@SO.Z)//(5@SO.W))	2.943.325	1	446.180	15%
((R.1)//(S.2@SO.P)).3@HJ.V.V!(B).((4@SO.Z)//(5@SO.W))	2.943.325	1	446.180	15%
((R.1)//(S.2@SO.P)).3@HJ.V.((4@SO.Z)//(5@SO.W))	2.943.325	0	0	0%
((R.1)//(S.S!(A).2@SO.P)).3@SMJ.V.((4@SO.Z)//(5@SO.W))	2.996.202	1	1.660.964	55%
((R.1)//(S.2@SO.P)).3@SMJ.V.((4@SO.Z)//(5@SO.W))	2.996.202	0	0	0%

Figure 6: Results for balanced butterfly



Data volumes on relation V		1/3*|R|	1/5*|R|	1/7*|R|
No. of different solutions in top-10 list		0	6	4
Change at optimal solution		No	No	Yes
Total cost (Optimal)		2.562.774	6.166.805	2.518.853
Total cost (Avg(Top-10))		2.882.293	6.644.516	2.597.239
Total cost (10th)		3.003.573	7.444.406	2.714.168
Difference in total cost	Optimal	0	3.604.031	-43.922
	Avg(Top-10)	0	3.762.223	-285.054
	10th	0	4.440.834	-289.405
Difference in total cost (%)	Optimal	0	141%	-2%
	Avg(Top-10)	0	131%	-10%
	10th	0	148%	-10%

Figure 7: Impact of selectivity

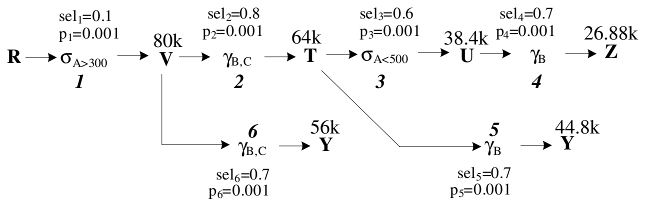


Figure 8: Right-deep butterfly

sive: the overhead ranges from 6% (for the second best solution) to approximately 70% for the best, third and fourth solution. This is due to the balanced nature of the butterfly, which has many candidate positions for the introduction of sorters: therefore, the results of a combination of sorters can produce significant variances.

Effect of Input Size. In the reference scenario, each of the sources R and S contains 100,000 rows. We have experimented by varying the size of data extracted from each source to 200,000 rows. This experiment returns the same 8 solutions out of the top-10 list produced by the reference scenario. A general observation is that the total cost of the optimal solution in all cases is practically linearly dependent upon the input size. This observation holds also for the average cost of the top-10 signatures and the 10th solution.

Effect of the Overall Selectivity of the Workflow. In this set of experiments, the selectivity values of the workflow are modified in such a way that small, medium or large data volumes pass through the butterfly's body. Specifically, we appropriately tune the selectivities, so that the ratio of the data of table V over the input data is: 0.1, 0.3, 0.5 or 0.7.

Observe that the latter case of $1/7 * |R|$ data reaching table V, is the representation of a butterfly configuration with a highly selective left wing. The differences in total cost of the optimal solution, average and 10th solution are negative. This means that this is cheaper than $1/3 * |R|$. Since the volume of data reaching V is small, a sorter can be placed on this table without causing a severe increase in total cost. In fact, the optimal solution for the $1/7 * |R|$ case contains a sorter on the data of table V. Thus, we can conclude that highly selective left wings of butterflies highly favor the butterfly's body as a good candidate position for a sorter.

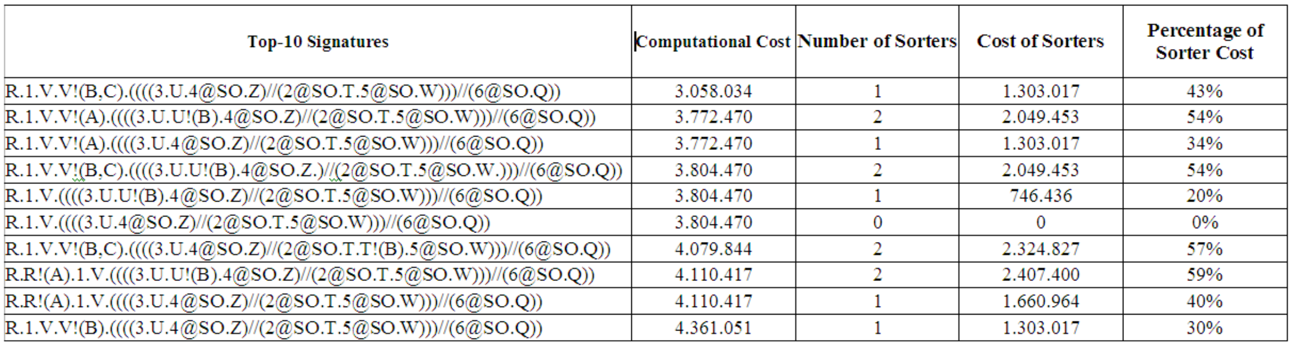
Right-Deep Butterflies. Another type of workflow under consideration is a Right - Deep Hierarchy. We illustrate a reference Right-Deep scenario in Figure 8 with the following characteristics: (a) number of nodes = 13, (b) time to discover optimal scenario = 14 sec, and (c) total number of generated signatures = 49.

Figure 9 shows the 10 signatures having the lowest total cost. The optimal physical representation for this scenario contains a sorter on table V that orders data according to attributes B, C.

In Figure 9 we also depict the sorter costs for the top-10 solutions. Interestingly, although the number of activities is small (only 6 activities in a graph of 13 nodes) the best solutions typically contain sorters. This is an interesting result since it highlights the possibility of exploiting sorters in recordset-heavy configurations. The sorter costs remain significant in the range 20% - 60% for the top-10 list.

Effect of Complexity (Depth and Fan-out) of the Right Wing. In this set of experiments (Fig. 10), we vary the following metrics: the size, the depth, and the average fan-out (w.r.t. the recordsets) of the butterfly's right wing. Then, we measure the candidate positions for sorters and the completion time of the exhaustive algorithm. Observe that the third case of Figure 10 contains 14 nodes on the right wing, whereas the fourth case 15 nodes, still the latter has 6 times larger completion time than the former. This is due to the fact that the third case contains a small number of candidate positions for sorters compared to the fourth case. The only factor that increases the completion time of the third case is the number of possible physical implementations of each activity. The outcome of these experiments is that the depth and the average fan-out do not play a determinant role to the completion time, whereas the number of candidate positions for sorters plays a certain role and the critical factor is ultimately, the size of the right wing.

Completion time and early termination. Clearly, the completion time of the exhaustive algorithm is exponential to the size of the butterfly. Practically though, during our experiments we have observed that the number of involved sorters in the good solutions does not exceed the number of intermediate DSA relations in the graph. In all categories of workflows, the optimal signature is found relatively early in the execution of the exhaustive algorithm; i.e., the signature with minimal cost is usually found in the first 50-60 signatures produced by the algorithm. The explanation is that after adding relatively few sorters, if more sorters are added to the workflow, any benefits gained from cheaper physical implementations due to the first sorters are outweighed by the extra sorting cost.



Top-10 Signatures	Computational Cost	Number of Sorters	Cost of Sorters	Percentage of Sorter Cost
R.1.V.V!(B,C).((((3.U.4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	3.058.034	1	1.303.017	43%
R.1.V.V!(A).((((3.U.U!(B).4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	3.772.470	2	2.049.453	54%
R.1.V.V!(A).((((3.U.4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	3.772.470	1	1.303.017	34%
R.1.V.V!(B,C).((((3.U.U!(B).4@SO.Z.)//(2@SO.T.5@SO.W.)))//(6@SO.Q))	3.804.470	2	2.049.453	54%
R.1.V.((((3.U.U!(B).4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	3.804.470	1	746.436	20%
R.1.V.((((3.U.4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	3.804.470	0	0	0%
R.1.V.V!(B,C).((((3.U.4@SO.Z)//(2@SO.T.T!(B).5@SO.W)))//(6@SO.Q))	4.079.844	2	2.324.827	57%
R.R!(A).1.V.((((3.U.U!(B).4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	4.110.417	2	2.407.400	59%
R.R!(A).1.V.((((3.U.4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	4.110.417	1	1.660.964	40%
R.1.V.V!(B).((((3.U.4@SO.Z)//(2@SO.T.5@SO.W)))//(6@SO.Q))	4.361.051	1	1.303.017	30%

Figure 9: Results for right-deep butterfly

Case	1	2	3	4
Right wing size (nodes)	8	10	14	15
Depth	4	6	4	6
Avg(Fan-out)	2	0.8	2.3	2
Candidate positions for sorters	2	3	3	6
Completion time (sec)	5	14	59	311

Figure 10: Effect of Depth and Fan-out

5.3 Findings for specific categories of butterflies

Balanced butterflies. The general case of butterflies is characterized by many candidate positions for sorters. Overall, the introduction of sorters appears to benefit the overall cost. The body of the butterfly is a good candidate to place a sorter, especially when the left wing is highly selective.

Butterflies with a right-deep hierarchy. These butterflies behave similarly to the general case of balanced butterflies. The size of the right wing is the major determinant of the overall completion cost of our algorithms due to the large number of candidate positions for sorters.

Lines. The generated space of alternative physical representations of a linear scenario is linear to the size of the workflow (without addition of sorters). In our experiments we have observed that due to the selectivities involved, the left wing might eventually determine the overall cost (and therefore, placing filters as early as possible is beneficial, as one would typically expect).

Butterflies with no right wing. In principle, the butterflies that comprise just a left wing are not particularly improved when sorters are involved. In particular, the introduction of sorters in Wishbones and Trees does not lead to the reduction of the total cost of the workflow. However, there are certain cases, in trees, where sorters might help - provided that the data pushed through the involved branch has a small size or a large number of activities share the same interesting order.

Forks. Sorters are highly beneficial for forks. This is clearly anticipated since a fork involves a high reusability of the butterfly's body. Therefore, the body of the butterfly is typically a good candidate for a sorter.

6. RELATED WORK

Previous efforts on the *optimization of ETL workflows* propose a set of transitions that generate equivalent logical workflows, possibly with lower cost, by changing the execution order of logical activities [4, 5]. Physical properties are not considered in this line of work, though.

Object-relational optimization has also provided results for queries with methods. Hellerstein deals with left-deep or bushy relational query plans [1]; however, ETL workflows have more complex structure and functionality, and therefore, do not necessarily meet the assumptions made by Hellerstein. The *exploitation of orderings* starts with the paper of P. Selinger et al. [3] that exploits existing orderings in the data (in the form of interesting orders) in the context of query optimization. Still, the intentional introduction of new orderings has not been considered per se. Furthermore, while many of the studies consider the generation of plans without unnecessary, overlapping order or group operators, they rely on functional dependencies and predicates applied over data, without handling orders more abstractly [2, 6, 9]. Thus, this line of work is orthogonal to our problem, since these results can be plugged-in to our algorithm in a straightforward fashion.

7. CONCLUSIONS

In this paper, we have dealt with the problem of determining the best possible physical implementation of an ETL workflow, given its logical-level description and an appropriate cost model as inputs. We have experimented with artificially introducing sorters in the physical representation of the workflow that allow the usage of a richer set of order-dependent implementations for each logical activity to improve the overall workflow performance. The long version of the paper [7] includes further results when failures are considered for the execution of the workflows.

Future work can follow in terms of linking the results of this work with the appropriate scheduling policy in the execution engine. Also, the usage of hash-groups can be considered as an alternative to the usage of orderings, mainly for real-time ETL.

8. REFERENCES

[1] J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. ACM Trans. Database Syst., 23(2):113–157, 1998.

[2] T. Neumann and G. Moerkotte. An Efficient Framework for Order Optimization. In ICDE, pages 461–472, 2004.

[3] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In SIGMOD, pages 23–34, 1979.

[4] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In ICDE, pages 564–575, 2005.

[5] A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-Space Optimization of ETL Workflows. IEEE Trans. Knowl. Data Eng., 17(10):1404–1419, 2005.

[6] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In SIGMOD, pages 57–67, 1996.

[7] V. Tziovara. Order-Aware ETL Workflows. Master's thesis, University of Ioannina. Available as a TR at http://www.cs.uoi.gr, 2006.

[8] P. Vassiliadis, A. Karagiannis, V. Tziovara, and A. Simitsis. Towards a Benchmark for ETL workflows. In QDB'07 (in conj. with VLDB'07), 2007.

[9] X. Wang and M. Cherniack. Avoiding Ordering and Grouping In Query Processing. In VLDB, pages 826–837, 2003.