Optimizing ETL Workflows for Fault-Tolerance

Alkis Simitsis [*1], Kevin Wilkinson [*2], Umeshwar Dayal [*3], Malu Castellanos [*4]

[*] *HP Labs*
Palo Alto, CA, USA

[1] alkis@hp.com [2] Kevin.Wilkinson@hp.com [3] Umeshwar.Dayal@hp.com [4] Malu.Castellanos@hp.com

***Abstract*— Extract-Transform-Load (ETL) processes play an important role in data warehousing. Typically, design work on ETL has focused on performance as the sole metric to make sure that the ETL process finishes within an allocated time window. However, other quality metrics are also important and need to be considered during ETL design. In this paper, we address ETL design for performance plus fault-tolerance and freshness. There are many reasons why an ETL process can fail and a good design needs to guarantee that it can be recovered within the ETL time window. How to make ETL robust to failures is not trivial. There are different strategies that can be used and they each have different costs and benefits. In addition, other metrics can affect the choice of a strategy; e.g., higher freshness reduces the time window for recovery. The design space is too large for informal, ad-hoc approaches. In this paper, we describe our QoX optimizer that considers multiple design strategies and finds an ETL design that satisfies multiple objectives. In particular, we define the optimizer search space, cost functions, and search algorithms. Also, we illustrate its use through several experiments and we show that it produces designs that are very near optimal.**

I. Introduction

In current practice, the primary objective for an ETL project is correct functionality and adequate performance, i.e., the functional mappings from operational data sources to data warehouse must be correct and the ETL process must complete within a certain time window. Although, performance is indeed important, in reality ETL designers have to deal with a host of other quality objectives besides performance such as reliability, recoverability, maintainability, freshness, scalability, availability, flexibility, robustness, affordability, and auditability [1]. In previous work, we described how different quality objectives may influence the final ETL design [2]. In this paper, we use that approach to address a specific problem that is common in ETL engagements: how to design ETL flows for performance but also with objectives for fault-tolerance and freshness.

Our goal is to make ETL flows fault-tolerant yet still satisfy a freshness requirement to finish within a specified time window. One approach is simply to repeat a flow in the event of a failure. But, this may not be feasible if the dataset is large or the time window is short. So, a typical way to make ETL flows robust to failures is by adding recovery points (RP). A recovery point is a checkpoint of the ETL state at a fixed point in the flow (see Figure 1). This might be done, for example, by copying the data flow to disk. If a failure occurs, control goes back to this recovery point, the state is recovered, and the ETL flow resumes normally from that point. This is faster than restarting the entire flow since the ETL flow prior to the recovery point is not repeated. However, the cost for adding a recovery point involves the additional i/o overhead of the copy operation.

Clearly, it is not practical to place recovery points after every operation in a flow (as shown in Figure 1). So, the design question is where to place recovery points in a flow. Today, this issue is addressed informally based on the experience of the designer, e.g., a designer might place recovery points after every long-running operator. However, with complex flows and competing objectives there are too many choices. Hence, the resulting design is likely not optimal. If a secondary design objective was for high freshness, then recovery points may not even be feasible because they add latency such that the ETL would not complete within the time window.

A more systematic approach is to formulate the placement of recovery points as an optimization problem where the goal is to obtain the best performance when there is no failure and the fastest average recovery time in the event of a failure. Given an ETL flow with n operators, there are $n-1$ possible recovery points. Any subset of these is a candidate solution. Therefore, the search space is given by the total number of combinations of these $n-1$ recovery points:

$$totalRP = 2^{n-1} - 1$$

The cost of searching this space is exponential in the number of nodes $O(2^n)$. In fact, the search space is even larger because there are other strategies for fault-tolerance, e.g., repeating the flow as mentioned above or using redundant flows. Therefore, it is necessary to find heuristics to prune the space. In addition, the ETL design may have additional objectives and constraints that must be considered such as freshness, cost, storage space, and so on. Also, there are additional strategies to consider for improving performance such as parallelism. This expands the search space even more and requires additional heuristics appropriate for each objective and strategy.

In this paper, we address the problem of generating an optimal ETL design for performance plus fault-tolerance and freshness. Our approach is similar in spirit to the work of [3] and [4] in that we use heuristics to search the space of all possible ETL designs. However, past work only considered the performance objective. In our work, we incorporate the additional objectives of fault-tolerance and freshness. We describe different strategies for achieving these objectives, a cost model, and heuristics for searching the design space. We illustrate the value of our approach with several experiments, which demonstrate that our approach can produce near optimal

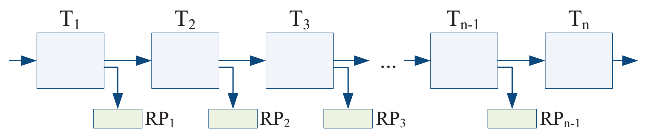


Fig. 1. Naive placement of recovery points after every operator

ETL designs even for complex scenarios.

Outline. The rest of the paper is structured as follows. In Section II, we formulate the problem as an optimization problem and describe the search space, the objective function, and the design alternatives. Section III describes the transitions within the search space. Section IV presents the exhaustive and heuristic optimization algorithms that search the space. Section V reports on our experimental findings for optimizing ETL flows for multiple objectives. Finally, Section VI discusses related efforts and Section VII concludes this paper.

II. Problem Formulation

Our approach to finding an optimal ETL design is to use heuristic search over the space of all possible ETL designs that meet the business objectives. In this section, we provide a formal statement of the problem. First, we state how to model an ETL flow as a graph. Then, we describe the quality objectives of interest, i.e., performance, fault-tolerance, and freshness and, for each objective, strategies for achieving them. Next, we present the cost model used to prune the search space by computing a cost function over each ETL design. Finally, the objective function is described. This is the function that the optimization algorithm seeks to minimize.

A. Preliminaries

1) Schema: Let $S^E = \{a_1, a_2, \ldots, a_m\}$ be the schema of an entity E containing a set of attributes a_i and A is the set of all possible attributes. An attribute a_i belonging to the schema of an entity E is denoted as a_i^E. In the ETL context, E is either an ETL transformation or a recordset residing on disk (i.e., a relation or a file). In the rest we will use S^E or $schema(E)$ interchangeably for representing the schema of an entity E.

2) Recordset: Let R be a recordset following a schema $S^R = \{a_1^R, \ldots, a_n^R\}$, which can be seen either as an input or output schema depending on the placement of the recordset in the workflow[1]. Recordsets represent data stores of any structure, like relational tables, flat files, xml files, and so on. Also, they represent any data store type that can be involved in an ETL process, like operational (source), warehouse (target), intermediate, landing, and staging data stores.

3) ETL Transformation: Let T be a transformation (also known as operator or operation) having a set of schemata $schemas(T)$. The $schemas(T)$ consists of the following schemata [4]:

- a set of input schemata s_i^T: each input schema is fed from an output schema of another transformation or the schema of a recordset.
- a set of output schemata s_o^T: each output schema is populated by the input schemata (or a subset of them) through a function characterizing the operational semantics of T. An output schema is to fed a recordset or an input schema of another transformation.
- a projected-out schema s_π^T = s_i^T - s_o^T: it contains the attributes that are projected out from the transformation (e.g., a simple function that concatenates $first\_name$ and $last\_name$ into $name$, it projects out the former two attributes).
- a generated schema s_g^T = s_o^T - s_i^T: it contains the attributes that are generated from the transformation (e.g., in the above example, the $name$ is a generated attribute).
- a parameter schema s_p^T = $params(T)$: it contains attributes belonging to the input schemata and a finite set of values. Essentially, these are the parameters that drive the transformation (e.g., the parameter schema of a filter $NN(a)$ that checks for null values on the attribute a is $s_p^{NN} = \{a\}$).

Also, an ETL transformation, depending on its operational semantics (or in other words, on how it treats the data), may belong to one of the following groups:

- pipeline operators, T^p: these are transformations that process each tuple separately, e.g., a filter transformation checking for null values or a function converting amounts in euros to amounts in dollars,
- blocking operators, T^b: these are transformations that require knowledge of the whole dataset, e.g., a grouping operator.

4) ETL Workflow: An ETL workflow comprises a set of ETL transformations, $\overline{T}$, and recordsets, $\overline{R}$, interconnected with each other forming a DAG. Let $G = (V, E)$ be a DAG representing an ETL workflow consisting of a set of vertices including the involved transformations, recordsets, along with the attributes, $\overline{A}$, contained in their schemata. Hence, $V = \overline{R} \cup \overline{T} \cup \overline{A}$. The edges E of the graph include provider edges Pr indicating the data flow and membership edges Po connecting the attributes with the respective schemata of T or R. Hence, $E = \overline{Pr} \cup \overline{Po}$. In the rest, we avoid overloading the notation of sets as e.g., $\overline{T}$; we write T, and whether this denotes a set or one of its elements is understood by the context.

An example ETL workflow is depicted in Figure 2. The output schema of transformation T populates the input schema of T' through a set of provider edges connecting the attributes of the former to the respective ones of the latter. The input and output schemas (i.e., the contained attributes) are connected with the transformations through membership edges. (The attributes are not shown here for clarity of presentation.)

B. Design Objectives

In previous work, we described a large set of quality metrics, termed QoX, that could be used to guide optimization of

[1]One could imagine two alike schemas implemented as two separate threads: a reader and a writer, that can be placed between the recordset and the workflow. In the rest, for presentation simplicity, we consider that recordsets have a single schema and we use the terms recordset schema and recordset interchangeably.

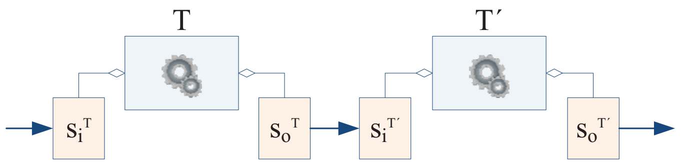


Fig. 2. Abstract part of an ETL workflow

ETL flows [1], [2]. In this section, we focus on the subset of those metrics that are needed for this paper. In particular, we describe four quality objectives, namely functional correctness, performance, reliability, and freshness, along with strategies for achieving them.

1) Functional Correctness Objective: Functional correctness is a requirement for a flow. In some sense, it is not a quality objective per se, but rather the starting point for optimization. A flow is functionally correct if it correctly implements the semantics of the schema mappings. Currently, there is no strategy or formal mechanism to generate a functionally correct flow from a specification. In the scope of this paper, we assume that the ETL flow provided is correct. We then aim at optimizing it according to the desired QoX metrics. The optimization process must preserve the functional correctness of the flow, i.e., the semantics of the optimized graph must match the semantics of the original, functionally correct sequential flow.

There are many possible designs for the initial, functionally correct flow. Note that an added advantage of ETL optimization is that it enables the initial design to be developed and expressed using techniques convenient to the designers. For example, if the designers feel that certain types of flows are easier to read and maintain (e.g., multiple short ETL flows rather than one long complicated ETL flow), they are free to do so without a performance penalty. Then, the optimizer will transform that initial design to meet the objectives.

Although the correctness of the initial flow is asserted and cannot be verified, it is possible to measure partial correctness because a designer may use test suites to validate certain aspects of a flow, e.g., consistency checks between sources and targets. For our purposes, we ignore these validation checks and treat them as part of the flow itself, i.e., a transformation that outputs success or failure. We need only to make sure that our optimization of the initial flow does not affect the semantics or correctness of these correctness measures.

2) Performance Objective: Performance is a metric of ETL process completion expressed in terms of time and resources used. In today's typical ETL engagements performance is the sole objective. Generally, there are two performance parameters, the frequency of performing the ETL process and the time window allowed for completing the ETL process. For example, ETL might be required to run on the first Saturday of each month and complete within eight hours. Or, ETL might be run daily from midnight for one hour.

The frequency of performing ETL is driven by the freshness requirement (see below). If the business only requires monthly reports, then running the ETL process once per month will suffice. The time window is driven by the availability of the various computing resources, specifically the source, target systems and the system used for the ETL process itself. It is also affected by the freshness requirement. Lower freshness means longer intervals between ETL processes which means larger input sizes and thus, longer time windows for the ETL process. We assume that the frequency and time window for the ETL process are part of the business requirements.

If the performance objective cannot be achieved by the initial, functionally correct ETL, there are several strategies for improving performance. These include alternative operator implementations, operator structural transformation, and parallelization. Just as a database management system may consider alternative implementations of join operators, an ETL optimizer may investigate different implementations of ETL operators to improve performance. In this paper, we are focused on optimizing the overall ETL flow so we ignore alternative operator implementations. We assume that the best implementation of an operator is provided or can be determined.

The placement of ETL operators may be transformed in several ways (see Section III and [3]). We may *swap* the order of two adjacent unary operators in an ETL flow. For example, given an ETL flow where a filter operator follows a sort operator, placing the filter first may reduce the amount of data to be sorted and so improve performance. Additionally, we may interchange the placement of a binary operator with two or more unary operators over parallel flows that converge to that binary operator. For example, a join of two data streams that is followed by a filter could be replaced by two separate filters over the two streams prior to the join.

An ETL design might consider two types of parallelism. Pipeline parallelism assigns two adjacent operators in an ETL flow to separate physical processors so they can be processed concurrently with a data stream from the producer operator to the consumer. Partition parallelism creates multiple, independent instances of an ETL flow on separate processors where each instance processes a different subset of the input. This requires a *splitter* operator to partition the input into multiple streams and a *merger* operator to combine the multiple result streams back into a single stream.

3) Reliability Objective: The reliability objective defines the ability of the ETL process to complete successfully despite failures. Any reason for not completing the process is considered a failure. Typical failures we face in the context of ETL are: network, power, human, resource or other miscellaneous failures. (In the following analysis we do not consider errors due to problematic data, which require different care.)

The reliability objective specifies the number of failures that the process can tolerate and still complete within its performance time window. We consider three strategies for achieving reliability: *repetition*, *use of recovery points*, and *redundancy*. The repetition strategy simply means repeating a flow in the event of a failure. This simple strategy may suffice when the ETL time window is long or for very simple flows.

A recovery point records the state of the ETL process at a specified point in the flow. In the event of a failure, the state is recovered and the ETL process continues from that point. Recovery points add latency to the flow both during normal processing (for staging data to it) and recovery (for reading data from it). In some cases, having recovery points is not feasible, e.g., due to large data volumes or a short processing time window.

Recovery points may be synchronous or asynchronous. These correspond roughly to consistent or fuzzy checkpoints, respectively, in a database management system. A synchronous recovery point is a blocking operation that makes a complete, consistent snapshot of the ETL process state at a given point in the flow. An asynchronous recovery point logs the state of data objects at a given point in the flow but does not block the flow. It has less latency but recovery is somewhat more complicated than for a synchronous recovery point. To simplify the presentation, we consider only synchronous recovery points. Asynchronous recovery points are optimized in the same manner by using a different cost function. However, this does not affect the algorithmic operation of our optimizer since it only gets the cost function as a parameter.

Redundancy refers to running multiple instances of a flow. It may be achieved using several techniques. *Replication* uses multiple, identical parallel instances of a flow and uses a *voting* technique to determine the correct result. *Fail-over* uses multiple, parallel instances of a flow, using one flow as primary and switching to a back-up flow in the event of a failure. A *diversity* strategy may be applied to either technique. Diversity uses alternative implementations for each flow. Note that these implementations could be generated by the ETL optimizer itself. In the rest, we consider only replication for redundancy.

4) Freshness Objective: Freshness concerns the latency between the occurrence of a business event at a source system –e.g., receipt of a purchase order– and the reflection of that event in the target system –e.g., the data warehouse. The freshness objective determines the frequency of the ETL process, i.e., the ETL process should be executed as frequently or more frequently than the business specification determines. For example, a freshness requirement that data must be loaded within two hours could be achieved by running the ETL process hourly or at least once every two hours. Strategies for higher freshness include increasing the frequency of the ETL process and decreasing the execution time of the ETL process. Generally, the execution time window for ETL is primarily determined by resource availability (e.g., run during off-peak hours). But, as the freshness requirement approaches real-time, the execution time window may also be affected by the freshness requirement.

Alternatively, freshness can be seen as the data volume processed per ETL execution. When high freshness is required smaller batches are processed, whereas when off-line ETL execution is preferred, then data are being processed in larger batches. However, in the latter case, conceptually there is one single batch, since all source data are available at the same time.

C. Cost Model

Assuming that transformation T gets n incoming tuples, it outputs $n' = g(n)$ tuples. T has a processing cost c_T, a recovery cost c_R, and a probability of failure p_f.

1) Processing cost: The processing cost c_T is estimated as a function over the incoming data volume; thus, by a generic formula $c_T = f(n)$. The f function depends on two aspects:

- the operational semantics of T as a function of the input size and captured as $h_p(n)$; e.g., for a transformation having a sorting algorithm as its core functionality, the $h_p(n)$ function is of the form $n \times log(n)$, and
- the fixed, auxiliary cost for the transformation captured as $h_a(n)$.

The operational semantics models the per-tuple processing cost for a transformation. The auxiliary cost models the per-instance overhead for an instance of a transformation. This includes time to initiate and terminate the operation (process and pipeline creation), time to acquire resources (memory allocation, file creation), and so on.

Therefore, the processing cost of T is a composite function $c_T = f(n) = f(h_p(n), h_a(n))$. For a large input size, the processing cost is likely dominated by the per-tuple cost, $h_p(n)$. However, as the input size shrinks, the auxiliary cost, $h_a(n)$ may come to dominate. Other QoX can affect c_T through $h_a(n)$. For example, a requirement for high freshness will, in general, result in smaller input sizes. However, at some point, freshness cannot be increased because the reduced processing time for smaller input sizes is outweighed by the auxiliary costs for the transformation. As another example, low data availability increases the processing cost, in the sense that the need and criticality for timely processing the involved data increases as well.

In general, the processing cost of an ETL workflow F involving l transformations would be:

$$c_{T(F)} = \sum_{i=1}^{l} c_{T_i} \tag{1}$$

2) Partitioning cost: The processing cost of an ETL flow is given by the equation (1). In general, the parallelization of flow execution, where permissable, reduces that cost. However, parallelizing an ETL flow is not a panacea for improving ETL performance [2]. For achieving parallelization, first we need to partition the data. Depending on the partitioning technique additional cost should be considered. Conceptually, two additional transformations are needed: first, a *splitter* for partitioning the flow and then, a *merger* for combining the parallel branches. Figure 3 shows generic examples of splitter T_S and merger T_M transformations.

Assuming a degree of parallelism d_N then the cost c_P for partitioning the flow into d_N parallel branches is:

$$c_{P(F)} = \max_j \left(\frac{\sum_{i=1}^{l} c_{T_i}}{d_{N_j}}\right) + c_{T_S} + c_{T_M} \tag{2}$$

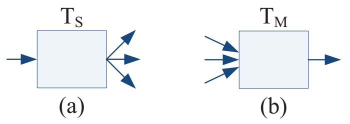


Fig. 3. (a) Splitter and (b) Merger transformations

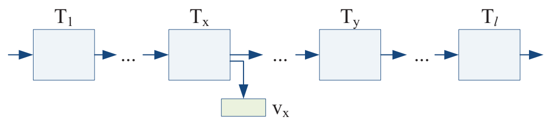


Fig. 4. Example ETL flow with a recovery point

Although the number of tuples processed does not change (otherwise, the ETL semantics are not maintained), the execution time is divided by the degree of parallelism d_N and the amount $\sum c_T/d_{N_j}$ is the cost of each branch j. As the execution time of each branch may vary due to several reasons (e.g., different loads), the slowest branch determines the execution time of the partitioned part of the flow.

3) Recovery cost: The recovery cost $c_{R(F)}$ represents the cost of a flow F for maintaining recovery points (or savepoints). Assuming a recovery point v_T that stores the output of T is a relation or file stored on the disk, then the i/o cost for communicating with the disk is usually significantly greater than c_T when T is executed in memory. Figure 4 depicts an example ETL flow containing a recovery point at an intermediate stage.

Assume a flow F containing l transformations $T_1, \ldots, T_l$. Let v_x be a recovery point attached to the transformation T_x. The recovery cost of the flow F without considering the recovery point would be equal to its processing cost:

$$c_{R(F)} = c_{T(F)} = \sum_{i=1}^{x} c_{T_i} + \sum_{i=x+1}^{l} c_{T_i} = c_{T(F')} + \sum_{i=x+1}^{l} c_{T_i} \tag{3}$$

Should we consider the cost of maintaining a recovery point v_x at T_x then the cost of the flow becomes:

$$c_{R(F)} = c_{T(F')} + \frac{c_{v_x}}{2} + \sum_{i=x+1}^{l} c_{T_i} \tag{4}$$

So, the additional cost we need to pay is $\frac{c_{v_x}}{2}$ (this is the i/o cost only for writing to the disk) for each recovery point we add to the flow. Essentially, for a data volume of n tuples this cost can be estimated as $c_{v_x} = \frac{c_{i/o}}{z_p} \times n$, where $c_{i/o}$ is the cost of writing and reading[2] one page of size z_p tuples to the disk.

Assume that a failure occurs at the transformation T_y, which is placed after T_x in the flow. The recovery cost of the ETL process differs depending on the configuration used: with (w/) or without (w/o) recovery points. When no recovery points are present, the ETL process should start from scratch for a cost:

$$c_{R(F)_{w/o}} = 2 \times \{c_{T(F')} + \sum_{i=x+1}^{y} c_{T_i}\} + \sum_{i=y+1}^{l} c_{T_i} \tag{5}$$

For the scenario with the recovery point the process resumes from the previous recovery point, i.e., v_x, and thus, the cost becomes:

[2]For HDDs, we can assume equal i/o costs for writes and reads; for today's SSDs, random writes are about 10x slower than random reads [5].

$$c_{R(F)_{w/}} = c_{T(F')} + c_{v_x} + 2 \times \{\sum_{i=x+1}^{y} c_{T_i}\} + \sum_{i=y+1}^{l} c_{T_i} \tag{6}$$

From the equations (5) and (6) it turns out that when a single failure occurs we may limit the recovery cost if the following condition holds:

$$c_{v_x} < c_{T(F')} \Leftrightarrow c_{v_x} < \sum_{i=1}^{x} c_{T_i} \tag{7}$$

This observation generates optimization opportunities in the ETL design and more concretely, an interesting challenge is to identify in which places of the flow the addition of recovery points will reduce the recovery cost.

4) Replication Cost: The replication cost $c_{L(F)}$ represents the cost for replicating a flow F into m replicas. The cost of creating the replicas is the cost of a *replicator* operator T_R that sends a copy of the same data to its output threads. After the replication, the cost of merging the flow is determined by the cost of a *voter* operator. We assume a voter that decides based on majority and there are two solutions. The fast voter T_{V_F} decides based on a simple $count(*)$ of the tuples coming from each replica; thus, it gets the semantics of a scan operator. The accurate voter T_{V_A} decides based on a check of each individual tuple; hence, it gets the semantics of a join operator. We assume the accurate voter here.

Note that under replication, each transformation is doing m times as much work but using the same number of resources as in the unreplicated flow. Consequently, the cost of each transformation is weighted to account for the resource sharing and additional work. Formally, the replication cost c_L of a flow F containing l transformations $T_1, \ldots, T_l$, is the following:

$$c_{L(F)} = c_{T_R} + \sum_{i=1}^{l} (w_i \times c_{T_i}) + c_{T_V} \tag{8}$$

5) Freshness cost: Freshness is generally improved by increasing the frequency of extract operations and thus, decreasing the number of tuples processed through a flow. Formally, let f denote the freshness requirement in time units (e.g., seconds). Let c denote the cycle time (or frequency) of data extraction from the source, i.e., perform an extract every c time units. Let $b(c)$ denote the number of data items extracted from the source during the cycle time c. Let $x(n)$ denote the ETL execution time for n data items. Then, the freshness requirement is related to ETL cost as:

$$c + x(b(c)) < f \tag{9}$$

In other words, the cycle time plus the time to perform the ETL for the extracted data must be less than the freshness requirement. To minimize interference with the data source, we need to choose the maximum cycle time, c, that satisfies the inequality 9.

D. Objective Function

Our problem scenario involves three objectives: performance, fault-tolerance, and freshness. The basic performance requirement is the time window for running the ETL process. We assume the time window is given as part of the technical specifications for the ETL design. However, as mentioned earlier, high freshness (e.g., near real-time) may affect the time window. The fault-tolerance objective specifies the number of failures that the ETL process must tolerate and still complete within the time window. The technical specifications also include expectations about the size of the input recordset; e.g., an arrival rate. Obviously, this is also affected by the freshness requirement, i.e., the input recordset size decreases when the ETL frequency increases.

Let n be the input recordset size, w be the execution time window for the ETL process represented as a flow F, and k be the number of faults that must be tolerated. Then the ETL flow has the constraint that:

$$time(F(n,k)) < w \tag{10}$$

In other words, the time to process n tuples with up to k failures during the process must be less than the time window w. Consequently, the objective function for optimization is to find the ETL flow F with the minimal cost and that satisfies the above constraint.

$$\begin{aligned}\text{OF}(F,n,k,w): \; & minimize \; c_{T(F)}, \\ & where \; time(F(n,k)) \; < \; w\end{aligned} \tag{11}$$

III. States and Transitions

In this section, we model the problem of ETL optimization for fault-tolerance as a state space search problem. We define the states and transitions between states, which define the state space searched by the optimizer.

A. States

A state S is an ETL workflow graph, i.e., a directed acyclic graph $G = (V, E)$. As discussed in Section II, each node in V is either a recordset, an ETL transformation or an attribute participating in a recordset or ETL transformation schema. Each edge in E is either a provider or a membership edge. However, the attributes are not taken into account by the transitions that produce new states (see below). Attributes are used only to ensure the correctness of the state production; thus, the formulae representing the s^T_π, s^T_g, and s^T_p schemata of Section II-A should hold. For example, we cannot swap two subsequent transformations when the intersection of their parameter schemata is not the empty set [4].

B. State Transitions

Each transition transforms an ETL flow graph G into an equivalent ETL flow graph G'. In other words, when a transition f is applied onto a state S it produces a new state S', and we write $S' = f(S)$. Two ETL flow graphs –i.e., states– are equivalent if they produce the same output, given the same input.

In general, there are a large number of possible transitions. Earlier work on ETL optimization considered performance alone, and introduced three types of transitions [4], which we also use in this paper.

- $swap(v_1, v_2)$: this transition applies to a pair of unary operations, v_1 and v_2, which occur in adjacent position in an ETL flow graph G, and produces a new ETL flow graph G' in which the positions of v_1 and v_2 have been interchanged;
- $factorize(v_b, v_1, v_2, v)$ and $distribute(v_b, v_1, v_2, v)$: this pair of transitions interchange the positions of a binary node v_b and a pair of unary nodes that are functionally equivalent; *factorize* replaces two unary nodes v_1 and v_2 with a single node v that performs the same function, and places v immediately after v_b in the flow; *distribute* is the “inverse” of factorize, and replaces v with two functionally equivalent nodes v_1 and v_2, and moves them immediately before v_b in the flow;
- $compose(v_{12}, v_1, v_2)$ and $decompose(v_{12}, v_1, v_2)$[3]: this pair of transitions are used to combine and split the operations of two nodes; *compose* takes adjacent unary nodes v_1 and v_2, and replaces them with a single unary node v_{12} that performs the composition of the functions of v_1 and v_2; *decompose* has the inverse effect of taking a unary node v_{12} and replacing it with two adjacent unary nodes v_1 and v_2 whose composition produces the same output as v_{12}.

To optimize for additional objectives, we extend the set of transitions with three new types of transitions, which we describe below.

1) partition(v1,v2,n,P): This transition is used to inject partitioned parallelism into the flow graph (see Figure 5-bottom). It splits the section of the flow graph between transformations v_1 and v_2 into n streams based on partitioning policy P by inserting a *splitter* node T_S after v_1 and a *merger* node T_M before v_2. Note that we are agnostic as to what partitioning policy P is used (range, hash, round-robin, or some other) as long as it guarantees that the union of the set of tuples directed to the n streams is equal to the output of v_1. For this transition to be applicable to an ETL flow graph, the resulting flow graph must be equivalent to the original. This requires that the operations occurring in the flow between v_1 and v_2 be distributive over union. Filter, join, and surrogate-key transformations and some aggregation operations (sum, count, max, min) have this property. Other operations (grouping,

[3]In [4], these transitions were called *merge* and *split*. We change their names to avoid confusion with the merger and splitter operations that we describe for the partition transition below.

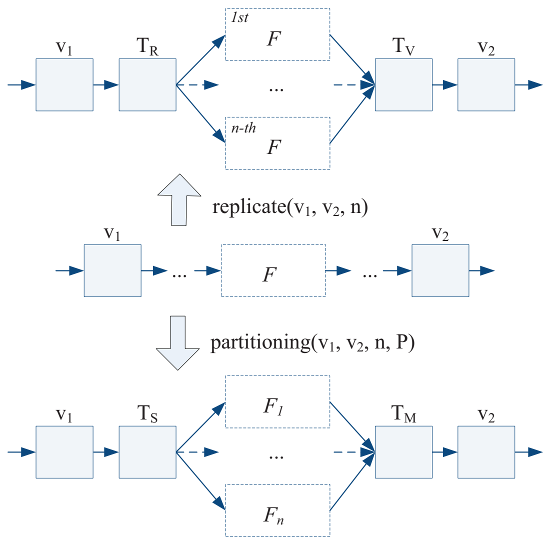


Fig. 5. Replicate and Partition transitions

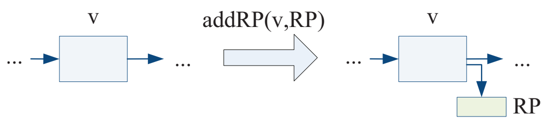


Fig. 6. Add recovery point transition

sorting, non-distributive aggregates, user-defined functions in general) may require modification of the operations in the flow followed by post-processing in the *merger* operation. This is a well-studied problem and we will not elaborate further. The benefit of partitioned parallelism, of course, is that it can potentially improve the freshness of the data by reducing the execution time of the ETL flow.

2) add_recovery_point(v,RP): This transition creates a new recordset node RP, and a new edge from node v to node RP (see Figure 6). The semantics are that RP is a recovery point, i.e., a point in the data flow where the state of the data flow is made durable, so that it can be used to recover from failures. Note that adding a recovery point does not change the semantics of the ETL flow, and hence the new flow produced by this transition is equivalent to the original flow.

3) replicate(v1,v2,n): This transition is used to make the flow more resilient to failure. It introduces a *replicator* node T_R after v_1 and a *voter* node T_V before v_2 which produces n copies of the flow between v_1 and v_2 (see Figure 5-top). The *voter* node compares the outputs of the copies and produces an output equal to that produced by the majority of the copies. Clearly, replication does not change the semantics of the operations, and hence the new flow produced by this transition is equivalent to the original flow (under the assumption that at least one copy completes). Note that this transition may also contain a voter policy V in its definition. We have already discussed example voting policies in II-C.4 like the fast and accurate voters. For clarity of presentation, we do not show V when we refer to the replicator transition.

```
Algorithm EXHAUSTIVE SEARCH
   Input: An initial state $S_G$ representing an ETL graph
          $G = (V, E)$
   Output: A state $S_{MIN}$ having the minimum cost
 1 begin
 2   $S_{MIN} = S_G$;
 3   $open \leftarrow S_G$;
 4   $close = \oslash$;
 5   while $open <> \oslash$ do
 6     $S \leftarrow open$;
 7     for all $S' = f(S)$ do
 8       if $S' \notin open$ and $S' \notin close$ then
 9         if $C(S') < C(S_{MIN})$ then $S_{MIN} = S'$;
10         $open \leftarrow S'$;
11     $close \leftarrow S$;
12   return $S_{MIN}$
13 end
```

IV. Search Space QoX Optimization

In this section, we present two algorithms for searching the state space with the goal of finding the state having the minimum cost as defined in Section II-C.

A. Exhaustive Search

First, working as in [4], we explore the state space exhaustively using the EXHAUSTIVE SEARCH algorithm (ES). ES generates all possible states by applying all the applicable transitions to every state.

ES uses two lists *open* and *close* for keeping track of unvisited and visited states, respectively. It also uses a state variable S_{MIN} for storing through its iterations, the state having the minimum cost. The algorithm starts from the initial ETL graph, i.e., state S_G, which is given as an input parameter. At the beginning *open* contains S_G and *close* is empty. For every state S in *open*, we apply any permissable transition f to it. For every newly generated –but not already visited– state S', if its cost is less that the minimum cost discovered so far, then S' becomes the new S_{MIN}. In any case, S' is marked as visited and we proceed until there is no other state to create. Then, ES returns the optimal state S_{MIN}.

Clearly, the state space is finite and the algorithm terminates after having generated all possible states. Also, it is obvious that the returned state is the optimal one. However, the state space is exponentially large and in realistic environments, it is practically infeasible to explore it in all its extent. In an experimental environment, though, it can be extremely useful, since it provides the optimal solution, and any other proposed method can compare its effectiveness against it.

B. Pruning the Search Space

The vastness of the state space requires more efficient exploration methods than the EXHAUSTIVE SEARCH. For improving the search performance, an obvious solution is to prune the state space. In the following, we propose techniques for achieving that.

1) Heuristics: First, we present heuristics that can drive us close to the optimal solution quickly.

Heuristics for Performance.

- (H1): *operator push-down*. Move more restrictive operators to the start of the flow to reduce the data volume; e.g., rather than $extract \rightarrow surrogate\ key\ gen \rightarrow filter$ do $extract \rightarrow filter \rightarrow surrogate\ key\ gen$.
- (H2): *operator grouping*. Place pipelining operators belonging to T^p together and separately from blocking operators belonging to T^b; e.g., rather than $filter \rightarrow sort \rightarrow function \rightarrow group$ do $filter \rightarrow function \rightarrow sort \rightarrow group$ (if such swaps are allowed; see [4]).
- (H3): *pipeline parallelism*. Place adjacent operators on separate physical processors and execute them in parallel with tuple flow from the producer operator to the consumer; e.g., in the flow $filter_1 \rightarrow filter_2$ assign the $filter_1$ and $filter_2$ operators to run on separate processors so they execute concurrently. The subflows to execute in parallel should be chosen such that their latency is approximately equal, i.e., if there are n processors, break the flow into n subflows $x_1, \ldots, x_n$ of equal (or near-equal) execution time, such that $max(time(x_i))$ is minimized.
- (H4): *partition parallelism*. Split the data stream into two or more sub-streams and process the streams on separate physical processors concurrently. For example, given a flow $lookup \rightarrow filter \rightarrow sort$, produce a new flow $splitter \rightarrow lookup \rightarrow filter \rightarrow sort \rightarrow merge$ where $splitter$ splits the data into n streams, $merge$ combines the streams, and the sub-streams are processed on separate processors in parallel. Partition parallelism is effective when the overhead of split and merge is low (see equation 2). However, note that partitioning in general is useful when the incoming volumes are large enough so that processing them all at once it might get too expensive. One interesting case is when the tuple order is not important; then, merging back the data can be done in less expensive manner. However, when the order does matter, the partition schema should be $s_{part} = \bar{a}_{part}$, where $\bar{a}_{part} \in s_P^T : c_T \geq c_{T'}$, $T' \in \{T_1, \ldots, T_l\} \setminus \{T\}$ (s_P^T is the parameter schema of T); i.e., $\bar{a}_{part}$ is based on the most expensive T of the partitioned flow.
- (H5): *split-point placement*. Given a flow to be partitioned, a candidate split point is before a blocking operator. The intuition is that the blocking operator will have less data to process when run in parallel and so it will be faster. (A requirement is that there is a parallelizable implementation of the blocking operator.)
- (H6): *merge-point placement*. Given a flow to be partitioned, a candidate merge point is after a large data reduction operator (e.g., highly selective filter or aggregate), since merging is faster when there is less data to merge.

Heuristics for Recoverability.

- (H7): *blocking recovery point*. Add recovery points after blocking or, in general, time-consuming operators.
- (H8): *phase recovery point*. Add recovery points at the end of an ETL phase, e.g., after extraction or transformation phases.
- (H9): *recovery feasibility*. Adding a recovery point at some position should take under consideration the inequality 7.

Heuristic for Reliability.

- (H10): *reliable freshness*. If a flow requires very fresh data (i.e., the execution time window w is small), and also high reliability (i.e., the number k of failures to be tolerated is high), recovery points may not be usable due to their additional latency. In this situation, we should consider using redundant flows.

2) QoX Heuristic Search (QHS): Based on the abovementioned heuristics, we present algorithm QHS that prunes the state space extensively in order to improve the objective function OF (equation 11).

QHS is a heuristic algorithm having as inputs an initial state (i.e., the initial ETL workflow) and a list of requirements. More specifically, the desired time window w in which that ETL process should terminate and the maximum number of failures k, which must be tolerated; i.e., the number of failures that the ETL should sustain and despite them will terminate correctly in w time units. Our algorithm is general in that it gets as input a generic objective function OF; in other words, it can get any other OF and not necessarily only the one provided by the equation 11. At the end, QHS returns the state that had the minimum cost among all others it has visited.

The first action that QHS takes is to optimize the workflow for performance (ln:3 – i.e., line 3 in QHS). For that, we use the algebraic optimization techniques described in [4] for getting the best we can according to the heuristic (H1). In a nutshell, these include the creation of large chains of unary operators before or after binary ones. [4] guarantees that the most selective operators have been pushed at the front of the design; at least, whenever that is possible. We have modified appropriately the algebraic optimization method of [4] and we bias the creation of chains containing pipeline operators, in order to exploit the heuristic (H2). Thus, if in a chain (or a local group to use the terminology of [4]) there exist both pipeline and blocking operators, then we group them separately.

In doing so, we identify all possible local groups lp of pipeline operators and we place them in a list Q_{lp}. Actually, since the algorithms proposed in [4] already provide us with chains of unary operators, we just work with these chains, without having to check again for them in the graph. Then, based on the formulae presented in Section II-C, we estimate the cost $c_{T(lp_i)}$ of each lp_i and we keep Q_{lp} sorted in increasing order of that estimated cost (ln:4-6).

In all cases, whenever we have a chain of pipeline operators, we exploit the heuristic (H3) and assign their execution in different processors using a simple round robin scheduling policy (not shown in the formal description of QHS).

Then (ln:7-9), we identify all candidate positions for placing a recovery point using the heuristics (H7-H8) and by taking

```
Algorithm QoX HEURISTIC SEARCH
   Input: An initial state $S_G$ representing an ETL graph $G = (V, E)$, an objective function OF, and requirements for execution time
          window $w$ and the maximum number of failures to tolerate $k$
   Output: A state $S_{MIN}$ having the minimum cost
1  begin
2    $S_{MIN} = S_{CURRENT} = S_G$;
3    optimize for performance based on [4];
4    create local groups $lp_i$ of pipeline operators;
5    estimate cost $c_{T(lp_i)}$ for all $lp_i$;
6    $Q_{lp} \leftarrow lp_i$, $\forall lp_i \in S_G$ ordered by increasing order of $c_{T(lp_i)}$;
7    find candidate positions $pos$ for recovery points $RP$ based on the inequality (7);
8    estimate cost $c_{P_i}$ for adding $RP_i$ at $pos_i$; //essentially for connecting $RP_i$ with node $v_i$ at position $pos_i$
9    $Q_{RP} \leftarrow < RP_i, pos_i >$, ordered by increasing order of $c_{P_i}$;
10   $candidates \leftarrow S_{current}$ ;
11   while $\underline{candidates <> \oslash}$ do
12     release = $true$ ;
13     while $\underline{Q_{RP} <> \oslash}$ do
14       $< rp, p > \leftarrow Q_{RP}$ ;
15       $S'$ = add_RP($p$,$rp$,$S_{CURRENT}$) ;
16       release = $false$;
17       while $\underline{\text{release} = false}$ do
18         if $\underline{\text{OF}(S',n,k,w) \text{ holds}}$ then
19           if $\underline{C(S_{MIN}) > C(S')}$ then $S_{MIN} = S'$ ;
20           release = $true$;
21           $candidates \leftarrow S'$ ;
22         else
23           pick a $lp$ from $Q_{lp}$, s.t. $lp$ is the highest ranked in $Q_{lp}$ in terms of expected gain and is placed before $p$ ;
24           $open \leftarrow S_{current}$ ;
25           $S'_{MIN} = S_{current}$;
26           while $\underline{open <> \oslash}$ do
27             $S \leftarrow open$ ;
28             for $\underline{\text{all } \{d_N, P\}}$ do
29               $S'$ = partition($lp.first$,$lp.last$,$d_N$,$P$,$S$) if $\underline{C(S'_{MIN}) > C(S')}$ then $S'_{MIN} = S'$ ;
30               $open \leftarrow S'$ ;
31           if $\underline{\text{OF}(S'_{MIN},n,k,w) \text{ holds}}$ then
32             if $\underline{C(S_{MIN}) > C(S'_{MIN})}$ then $S_{MIN} = S'_{MIN}$ ;
33             release = $true$; $candidates \leftarrow S'_{MIN}$ ;
34           $open \leftarrow S_{current}$ ;
35           $S'_{MIN} = S_{current}$;
36           while $\underline{open <> \oslash}$ do
37             $S \leftarrow open$ ;
38             for $\underline{\text{all } \{r_N\}}$ do
39               $S'$ = replicate($lp.first$,$lp.last$,$r_N$,$S$) if $\underline{C(S'_{MIN}) > C(S')}$ then $S'_{MIN} = S'$ ;
40               $open \leftarrow S'$ ;
41           if $\underline{\text{OF}(S'_{MIN},n,k,w) \text{ holds}}$ then
42             if $\underline{C(S_{MIN}) > C(S'_{MIN})}$ then $S_{MIN} = S'_{MIN}$ ;
43             release = $true$; $candidates \leftarrow S'_{MIN}$ ;
44   return $\underline{S_{MIN}}$
45 end
```

under consideration the recovery feasibility (H9) at each place using the inequality (7). In each case, we estimate the expected cost c_{P_i} for adding the recovery point RP_i at the position pos_i (i.e., to connect it with the node that is placed in pos_i). We store the recovery points found satisfying the abovementioned heuristics in list Q_{RP}, which we maintain ordered by increasing order of the expected cost c_{P_i}.

Every state visited by QHS is stored in list $candidates$, where first we put the initial state. Then, we start exploring the state space, by visiting only states that we know based on heuristics (H1-H10) that are good candidates for giving us a better cost. We start with examining the possibility of adding in our workflow the recovery points stored in Q_{RP}. For each, we examine if we can afford it, and if the answer is positive then we add it to the graph (ln: 18-21). Note that since Q_{RP} is sorted in increasing order of the expected cost (or equally, in decreasing order of the potential gain), we pick first in a greedy fashion the most promising recovery points.

However, if we cannot afford a specific recovery point, we examine the possibility to make some room in our budget by boosting the performance using parallelization and in particular, partitioning (ln:23-30) (pipelining is already used wherever possible). Thus, we consider all local groups lp_i stored in Q_{lp} that are placed before the position of interest for the considered recovery point. For each lp_i, we examine possible partitioning schemes using a partitioning policy P and a degree of parallelism d_N.

Regarding the partitioning policy P, the algorithm is agnostic to it. In our implementation though, we use round robin as the default partitioning policy, when the sort order does not matter, and sort-based or range partitioning, when the sort order matters, in order to reduce the cost of merging (i.e., merge-sort in this case) at the end of lp_i.

An appropriate d_N should be chosen based on the inequality 7, so that the partitioning will make room in terms of cost from the addition of the recovery point v_p at position p. Thus, d_N should be at least: $d_N = \frac{\sum_{lp_i=lp.first}^{lp.last} c_{T_{(lp_i)}}}{c_{v_p} - c_{T_S} - c_{T_M}}$. An upper bound for d_N is determined by the cost of merging at the end of lp_i (if d_N is too large, then paying the cost of merging is not worth it). Hence, we examine integer values of d_N (if any) in the above range. If we cannot find a satisfactory d_N (ln:31-33), then we continue with another lp_i. If no lp_i can allow the addition of v_p, then either we try a different v_p at a different position p or we use replication.

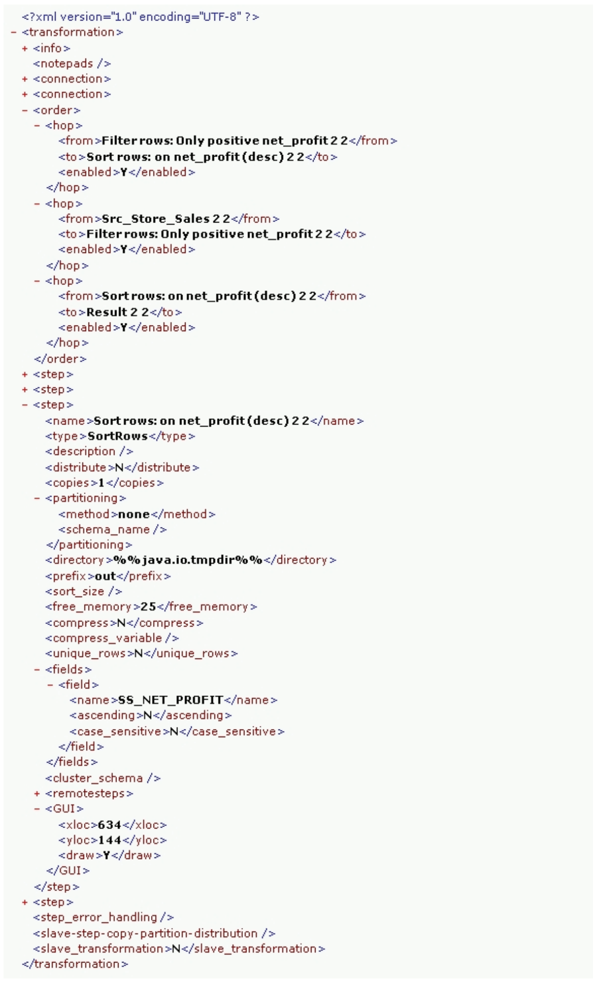
For replication, we check different options of creating the r_N replicas for the chain lp_i (ln:36-40). The r_N values tested belong to a range similar to the one described before for d_N and are estimated using equation 8. The only difference is that here we are interested in odd integer values of d_N since the voter chooses based on the majority of votes. Depending on how accurate results we want (this is an external requirement not shown in the formal description of the algorithm) we choose either the fast or the accurate voter (see Section II-C for their operation).

If replication fails (ln:41-43) then we cannot satisfy the recovery point under consideration, and we proceed with the next available one from Q_{RP}. Each time a valid solution is found, we put it in the list of candidates, in order to check later on if we can enrich it with additional recovery points that would fit in the given requirements.

V. Experimental Evaluation

A. System Architecture

The QoX optimizer presented in this paper is agnostic to the specific ETL engine. The optimizer gets as an input a graph representing an ETL workflow and produces another graph with same semantics. Modern ETL tools, both commercial, like Informatica's PowerCenter[4] and open-source, like Pentaho's Kettle[5], support import and export of ETL designs in XML files. An example ETL workflow in Kettle's XML format is depicted in Figure 7. Thus, an appropriate parser transforms the XML file into our supported graph format, and vice versa, it transforms an ETL workflow from our supported graph format to an XML file. Therefore, the optimization techniques described here can be used on top of any ETL engine that supports this functionality.



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <transformation>
  + <info>
    <notepads />
  + <connection>
  + <connection>
  - <order>
    - <hop>
        <from>Filter rows: Only positive net_profit 2 2</from>
        <to>Sort rows: on net_profit (desc) 2 2</to>
        <enabled>Y</enabled>
      </hop>
    - <hop>
        <from>Src_Store_Sales 2 2</from>
        <to>Filter rows: Only positive net_profit 2 2</to>
        <enabled>Y</enabled>
      </hop>
    - <hop>
        <from>Sort rows: on net_profit (desc) 2 2</from>
        <to>Result 2 2</to>
        <enabled>Y</enabled>
      </hop>
    </order>
  + <step>
  + <step>
  - <step>
      <name>Sort rows: on net_profit (desc) 2 2</name>
      <type>SortRows</type>
      <description />
      <distribute>N</distribute>
      <copies>1</copies>
    - <partitioning>
        <method>none</method>
        <schema_name />
      </partitioning>
      <directory>%%java.io.tmpdir%%</directory>
      <prefix>out</prefix>
      <sort_size />
      <free_memory>25</free_memory>
      <compress>N</compress>
      <compress_variable />
      <unique_rows>N</unique_rows>
    - <fields>
      - <field>
          <name>SS_NET_PROFIT</name>
          <ascending>N</ascending>
          <case_sensitive>N</case_sensitive>
        </field>
      </fields>
      <cluster_schema />
    + <remotesteps>
    - <GUI>
        <xloc>634</xloc>
        <yloc>144</yloc>
        <draw>Y</draw>
      </GUI>
    </step>
  + <step>
    <step_error_handling />
    <slave-step-copy-partition-distribution />
    <slave_transformation>N</slave_transformation>
  </transformation>
```

Fig. 7. Example ETL workflow

B. Experimental Setup

For the experimental assessment of our methods, we used a set of 30 ETL workflows of varying size and structure. According to their sizes, the workflows used can be categorized as follows: (a) small flows, containing from 20 to 30 operators, (b) medium flows, containing from 31 to 54 operators, and (c) large flows, containing from 55 to 80 operators.

Example operators considered in our evaluation are filters, functions (type conversions, string manipulation, etc.), schema changing transformations (like pivot etc.), surrogate key assignment, lookup operations, diff operation, union, join, aggregating operations, and so on. For each transformation, we used a cost function determined by its core operations (e.g., an operator having sort as its core function has a cost similar to $n \times log(n)$). For more complex operators (e.g., user-defined functions), we used the open source ETL tool called Kettle

[4]http://www.informatica.com/products_services/powercenter/
[5]http://kettle.pentaho.org/

and went through the implementation code of the operations in order to get an estimate of their complexity and cost.

All experiments have been conducted in a Dual Core 2 PC at 2.13 GHz with 1 GB main memory and a SATA disk of 230 GB. The optimizer is implemented in C++ and makes use of the Boost library for graphs[6].

C. Evaluation

A significant aspect is to understand the size of the optimization problem and to validate that our solution works. Exhaustive search ES produces the optimal solution, but realistic ETL flows can be too large for exhaustive search. However, our findings show that heuristic search QHS is feasible in space (Figure 8) and time (Figure 9) and provides solution of quality close to the optimal (Figure 10).

In Figure 8, the number of graph states (possible ETL flows) evaluated by the optimizer is plotted against the size of the original, unoptimized flow. As expected, as the size of the flow increases, the number of states grows. The sharp knee for exhaustive search at 40 nodes is because, beyond 40 nodes, exhaustive search was intentionally terminated at 35 hours of search time and the best solution as of that time was returned. The reason the number of states visited declines beyond 40 nodes is because the processing time to generate new search states increases with the size of the flow. Thus, for a fixed amount of search time, exhaustive search will consider fewer states for a large flow than for a small flow.

Heuristic search behaves well even for large flows. As the flow size increases, the number of states visited for heuristic search increases at a much lower rate than for ES.

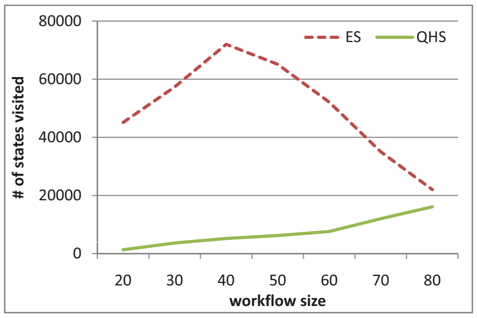


Fig. 8. Number of states visited versus worklow size

In Figure 9, the optimizer execution time is plotted against the size of the original, unoptimized flow. Note the time is log-scale. The heuristic algorithm finishes orders of magnitude faster than exhaustive search. Recall that exhaustive search was terminated at 35 hours beyond 40 nodes which is why that exhaustive line is nearly flat. We see that even for the largest flows, heuristic search completes within a reasonable amount of time (a couple of hours). Flow optimization will be a relatively infrequent event so spending a few hours to optimize flows is acceptable.

Figure 10 compares the quality (in terms of cost) of solutions found by heuristic search against the solution found

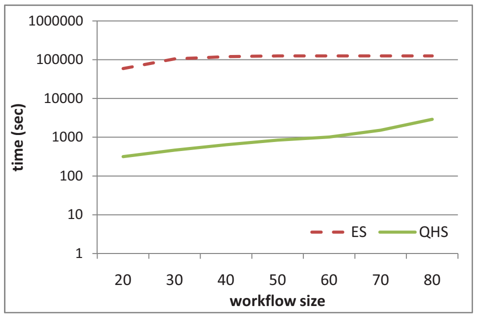


Fig. 9. Optimization time versus workflow size

by exhaustive search. Up to 40 nodes, the exhaustive solution is optimal and we see that the heuristic solution is within a few percent of optimal. Beyond 40 nodes, the quality of the heuristic solution declines but it is compared against a presumably sub-optimal solution returned by exhaustive search that was terminated early. Consequently, the results in this range are harder to interpret. However, the overall trend line is nearly linear across the entire range so we can speculate that the sub-optimal solution returned by exhaustive search was not too far off.

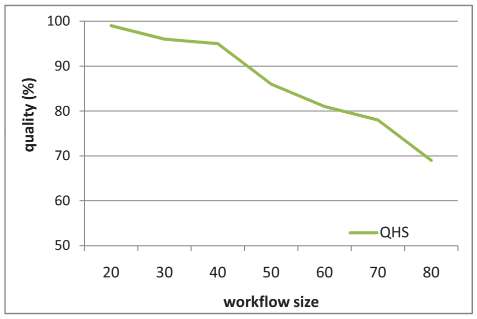


Fig. 10. Solution quality of QHS versus workflow size

The optimizer selects different fault-tolerance strategies depending on the QoX objectives. To illustrate this, Figure 11 compares the number of recovery points (RP) and the number of replicas (RPL) used in solutions for various workflow sizes and various input sizes. The workflow sizes ranged from 20 to 80 nodes. Three input sizes were considered: 1000 tuples, 100,000 tuples and 1 million tuples.

For the smallest flow of 20 nodes, we see only a single strategy is used. The larger input sizes used only recovery points. However, the smaller input size used replication, presumably because the additional latency due to recovery points would not allow the workflow to complete within its time window. As the flow size increases up to 80 nodes, solutions tend to use a mixture of strategies with recovery points interspersed with flows that are replicated. However, in general, we see that the smaller input sizes favor a replication strategy over a recovery point strategy.

Essentially, Figure 11 shows the trade-off between freshness (which affects input size) and fault-tolerance, while the workflow size is examined as a third dimension. For achieving

[6]http://www.boost.org/doc/libs/1_39_0/libs/graph/doc/index.html

higher freshness, a typical strategy is to process smaller batches of data and this should be done fast. Hence, for smaller input sizes (red bar) replication is favored, whereas for larger batches the optimizer tends to use more recovery points. Such tradeoffs are captured in the theoretical analysis presented in Section II-C and II-D and constitute the key contribution of our paper. Note that equation 11 gives the optimization objective, combining the freshness and fault-tolerance requirements into a constraint on the optimal solution. Figure 11 shows why the optimization problem is non-trivial: the optimal solution must include the "best" mix of recovery points, replicas, and so on.

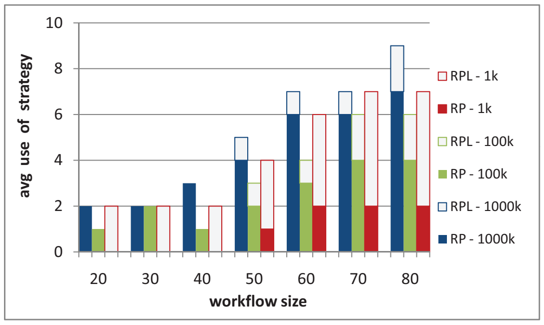


Fig. 11. Recovery points and replicas used in heuristic solutions

VI. Related Work

Commercial ETL tools (e.g., [6], [7], [8], [9]) provide little support for automatic optimization. They only provide hooks for the ETL designer to specify for example which flows may run in parallel or where to partition flows for pipeline parallelism. Some ETL engines such as PowerCenter [7] support PushDown optimization, which pushes operators that can be expressed in SQL from the ETL flow down to the source or target database engine. The remaining transformations are executed in the data integration server. The challenge of optimizing the entire flow remains. To the best of our knowledge, today's ETL tools do not support automatic optimization for the qualities considered in this paper.

Despite the significance of optimization, so far the problem has not been extensively considered in the research literature on ETL. The existing studies mainly focus on a black-box optimization approach at the logical level, and concern the order with which the activities are placed in the flow [3], [4]. More general rewriting that takes into account the semantics of the flow operations has not yet been addressed. A major inhibitor to progress here is the lack of a formal language at the logical level (analogous to the relational algebra).

Other research efforts have dealt with optimizing specific parts of ETL (esp. in real-time ETL), such as the loading phase (e.g., [10]), individual operators (e.g., [11]), scheduling policies (e.g., [12]), and so on. However, none of these efforts has dealt with the problem of optimizing the entire ETL workflow.

Furthermore, following the tradition of query optimization, the prior work on ETL optimization considered performance as the only objective. This paper is the first effort to consider other QoX objectives as well. In our earlier papers, we introduced the QoX framework [1] and showed examples of tradeoffs that lead to different ETL designs in order to meet different QoX objectives [2]. Here, we study the problem of optimizing for performance under freshness and fault-tolerance constraints.

Object-relational optimization has also provided results for queries with methods. For example, earlier work has dealt with left-deep or bushy relational query plans [13]. However, ETL workflows have more complex structure and functionality, and thus, do not necessarily meet the assumptions made in [13].

VII. Conclusions

ETL projects today are designed for correct functionality and adequate performance, i.e., to complete within a time window. However, the task of optimizing ETL designs is left to the experience and intuition of the ETL designers. In addition, ETL designs face additional objectives beyond performance.

In this paper, we have presented an approach to developing ETL designs that satisfy multiple objectives beyond performance. We illustrated this approach for designs that support fault-tolerance and freshness. Our approach considers multiple strategies for these objectives and uses heuristics to search a large design space of possible solutions. Our experiments demonstrate the feasibility of our technique by comparing the heuristic solutions to those found by exhaustive search. The heuristic algorithm is orders of magnitude faster than the exhaustive search and yet finds solutions that are within a few percent of the optimal solution.

As future work, we intend to consider optimizing ETL flows for additional QoX objectives.

References

[1] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson, "Data integration flows for business intelligence," in EDBT, 2009, pp. 1–11.

[2] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos, "QoX-Driven ETL Design: Reducing the Cost of the ETL Consulting Engagements," in SIGMOD, 2009.

[3] A. Simitsis, P. Vassiliadis, and T. K. Sellis, "Optimizing ETL Processes in Data Warehouses," in ICDE, 2005, pp. 564–575.

[4] ——, "State-Space Optimization of ETL Workflows," IEEE Trans. Knowl. Data Eng., vol. 17, no. 10, pp. 1404–1419, 2005.

[5] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query Processing Techniques for Solid State Drives," in SIGMOD Conference, 2009, pp. 59–72.

[6] IBM, "IBM Data Warehouse Manager," in the Web, available at: http://www-01.ibm.com/software/data/integration/, 2009.

[7] Informatica, "PowerCenter," in the Web, available at: http://www.informatica.com/products/powercenter/, 2009.

[8] Microsoft, "SQL Server Integration Services (SSIS)," in the Web, available at: http://www.microsoft.com/sqlserver/2008/en/us/integration.aspx, 2009.

[9] Oracle, "Oracle Warehouse Builder 10g," in the Web, available at: http://www.oracle.com/technology/products/warehouse/, 2009.

[10] C. Thomsen, T. B. Pedersen, and W. Lehner, "RiTE: Providing On-Demand Data for Right-Time Data Warehousing," in ICDE, 2008, pp. 456–465.

[11] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell, "Supporting Streaming Updates in an Active Data Warehouse," in ICDE, 2007, pp. 476–485.

[12] L. Golab, T. Johnson, and V. Shkapenyuk, "Scheduling Updates in a Real-Time Stream Warehouse," in ICDE, 2009, pp. 1207–1210.

[13] J. M. Hellerstein, "Optimization Techniques for Queries with Expensive Methods," ACM TODS, vol. 23, no. 2, pp. 113–157, 1998.