QoX-Driven ETL Design: Reducing the Cost of ETL Consulting Engagements

Alkis Simitsis
HP Labs
Palo Alto, Ca, USA
alkis@hp.com

Kevin Wilkinson
HP Labs
Palo Alto, Ca, USA
kevin.wilkinson@hp.com

Malu Castellanos
HP Labs
Palo Alto, Ca, USA
malu.castellanos@hp.com

Umeshwar Dayal
HP Labs
Palo Alto, Ca, USA
umeshwar.dayal@hp.com

ABSTRACT

As business intelligence becomes increasingly essential for organizations and as it evolves from strategic to operational, the complexity of Extract-Transform-Load (ETL) processes grows. In consequence, ETL engagements have become very time consuming, labor intensive, and costly. At the same time, additional requirements besides functionality and performance need to be considered in the design of ETL processes. In particular, the design quality needs to be determined by an intricate combination of different metrics like reliability, maintenance, scalability, and others. Unfortunately, there are no methodologies, modeling languages or tools to support ETL design in a systematic, formal way for achieving these quality requirements. The current practice handles them with ad-hoc approaches only based on designers' experience. This results in either poor designs that do not meet the quality objectives or costly engagements that require several iterations to meet them. A fundamental shift that uses automation in the ETL design task is the only way to reduce the cost of these engagements while obtaining optimal designs. Towards this goal, we present a novel approach to ETL design that incorporates a suite of quality metrics, termed QoX, at all stages of the design process. We discuss the challenges and tradeoffs among QoX metrics and illustrate their impact on alternative designs.

Categories and Subject Descriptors

H.2.7 [**Database Administration**]: Data warehouse and repository.

General Terms

Management, Performance, Design, Reliability, Experimentation.

Keywords

ETL, Data Warehouses, Metrics, QoX, Quality, Modeling.

1. INTRODUCTION

The backstage of the Data Warehouse architecture consists of Extract-Transform-Load (ETL) processes. ETL processes are responsible for extracting data from distributed and often heterogeneous sources, cleaning and transforming that data according to business requirements, and finally, loading it to the data warehouse. ETL design and implementation constitutes 70% (by some estimates) of the effort in data warehousing projects, and today is offered as a time-consuming, labor-intensive consulting service. Apart from the fact that these projects are expensive, ETL is the critical path from business events (at the data sources) to business analysis and action (at the warehouse). Thus, delays in ETL engagements directly affect business operational effectiveness. There are strong motivations for making ETL engagements less expensive and faster.

The lifecycle of a typical ETL engagement begins with the gathering of business and technology requirements. The business requirements specify information needs and service level objectives like overall cost, latency between operational event and warehouse load, provenance needs, and so on. The technology requirements specify the details of data sources and targets, transformations, infrastructure, dependencies and constraints on extracts and loads, and system availability. These requirements are synthesized into specifications that are combined to form a high-level conceptual design. This is followed by the construction of logical and physical models to capture the data flows from operational systems to a data warehouse. The final step is to express the physical model in an implementation language such as SQL, a scripting language or some ETL engine (e.g., Ab Initio, DataStage or Informatica).

In current commercial and research solutions, the focus for ETL is on correct functionality and adequate performance, i.e., the functional mappings from data sources to warehouse must be correct and the ETL runs must complete within a certain time window. Correctness and performance are important objectives. However, expert consultants and practitioners, who have broad and deep experience in ETL, have observed that in real-world projects additional objectives are important as well. As one practitioner said: "If I wanted better performance, I would ask for better hardware; unfortunately, I cannot buy a more maintainable or a more reliable system."

In fact, nowadays ETL designers do deal with a host of quality objectives besides performance, including reliability, recoverability, maintainability, freshness, scalability, availability, flexibility, robustness, affordability, and auditability. However, the modeling languages and ETL tools neither capture such quality objectives nor provide a formal mechanism to quantify, track, and measure them. Consequently, they are dealt with informally based on the best practices and experience of the ETL designer. Hence, in the translation from the high-level business and technical requirements to detailed ETL specifications, objectives may be dropped. This makes the final implementation sub-optimal with respect to

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

the project objectives. Correcting for this adds time, cost, and complexity to the engagement. Our goal is to reduce the time and cost of ETL engagements by facilitating the generation of optimal ETL designs that incorporate all of its objectives.

As an aside, we observe that as enterprises become more automated, data-driven, and real-time, Business Intelligence architectures are evolving to support operational decision-making. This imposes additional needs for next-generation ETL which include: a larger number and diversity of data sources and data types (less structured content, external data feeds, and streaming data), real time decision making, fast refresh cycles, more complex analytic and reporting tools, 24x7 availability, and so on. With these increasing demands on data warehouses, ETL design has become even more complex.

Consequently, we feel it is time to take a fresh, comprehensive look at ETL, or more generally, the problem of data integration flow design and implementation. We propose a novel, more comprehensive approach to ETL that considers additional quality metrics besides performance. We refer to these collectively as QoX. The QoX metric suite is incorporated at all stages of the design process, from high-level specifications to implementation. A major challenge is to identify the interrelationships and dependencies among the quality metrics that lead to tradeoffs for alternative optimizations of ETL processes. Another major challenge is to automate the optimization process. Hence, a systematic approach to ETL design based on QoX benefits the designer, implementer, and administrator.

We note that our approach applies not just to new integration projects. As is often the case in outsourcing, designers may be asked to evaluate or modify an existing ETL workflow for new requirements, sources, hardware, and so on. The techniques we describe here are just as applicable in understanding and improving pre-existing ETL as to new projects.

In the rest of this paper, first, we introduce a set of quality metrics, the QoX suite, which capture business and technical requirements for an ETL engagement. We also discuss how tradeoffs among QoX metrics can be identified and evaluated. Then, we present an example ETL workflow, which is used in various optimization scenarios for illustrating our ideas.

2. QoX METRIC SUITE

The purpose of the QoX metric suite is to capture all relevant quality metrics for an ETL engagement and incorporate these into an optimized design and implementation. By incorporating these metrics at all levels in the design process, the resulting implementation will better match the customer's expectations and objectives. This will reduce the time and cost of ETL projects.

2.1 Methodology

In our view, next-generation ETL projects require a layered methodology that proceeds in successive, stepwise refinements from high-level business requirements, through several levels of more detailed specifications, down to execution models (Figure 1). At each level of design, QoX metrics are introduced or refined from higher levels. In that sense, the layered approach presents opportunities for optimization at each successive level of specification. Optimizations at all design levels should be driven by QoX metrics. These metrics, in effect, prune the search space of all possible designs, much like cost-estimates are used to bound the search space in cost-based query optimization.

At each design level, the operators constituting the ETL flows are extended with specifications influenced by key quality metrics. That helps the automatic or semi-automatic transition among the levels. In other words, there may be several alternative *translations* from conceptual model to logical model and these alternatives can be driven by the QoX objectives and tradeoffs. Similarly, the translation from the logical model to the physical model enables additional types of optimizations. For example, a join operator at the conceptual level can be annotated with information stating high requirement for freshness. This information will eventually indicate, at the physical level, the choice of a physical implementation for join, suitable for real-time environment (e.g., the MeshJoin [9]).

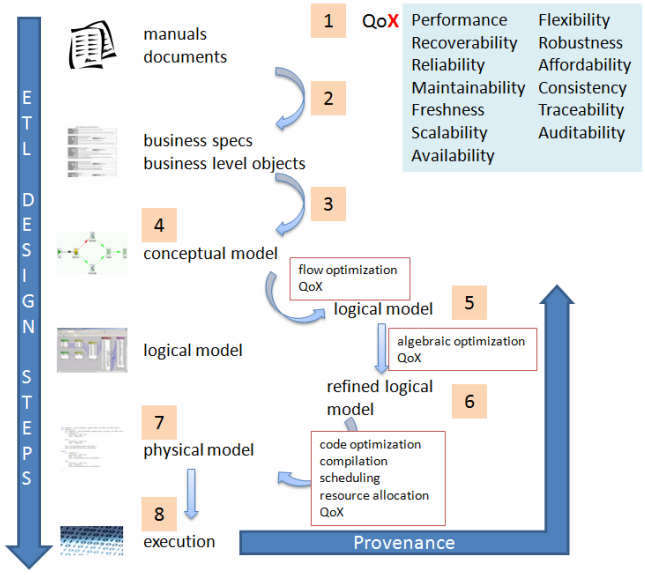

Figure 1. Layered approach for ETL design [3]

2.2 QoX Metrics

A non-exhaustive list of metrics that can be used to guide optimization include: *performance*, *recoverability*, *reliability*, *freshness*, *maintainability*, *scalability*, *availability*, *flexibility*, *robustness*, *affordability*, *consistency*, *traceability*, and *auditability*. An important research challenge is how to define these metrics precisely and how to measure them. Some metrics are quantitative (e.g., reliability, freshness, cost) while other metrics may be difficult to quantify (e.g., maintainability, flexibility). The software engineering community has measures for evaluating the quality of software design, so we adapted some to the quality of ETL designs. Due to space limitations, we focus here on a subset of the aforementioned metrics, but for a description of other metrics we refer the interested reader to [3].

Performance. Performance refers to the elapsed time to execute an ETL workflow. For a given workflow, there are several techniques to reduce the execution time that an optimizer might consider. An obvious one is to increase the resources allocated for the ETL execution (e.g., giving more memory or processing power). However, this process requires fine-grained tuning (e.g., some operations may need a greater extent of the available memory

than some others) and appropriate scheduling at both the data and the process level.

ETL workflows are much more complex than traditional relational queries, thus the well-known techniques for multi-query optimization are not enough in this context [e.g., 2, 10, 11]. Still, we can leverage knowledge acquired from query processing. For example, the rule that the most restrictive operations should be placed at the start of the flow, applies here as well. Such algebraic optimization can be done in several phases of the design: conceptual, logical, and physical. At the conceptual and logical levels, specific implementation details of the design may not be available. Still, the designer knows that a conceptual operator containing a join operation, such as surrogate key assignment, is more expensive than a filter operation that may reduce the moving data volume. Similarly, an effective technique is to gather pipelining and blocking operations separately from each other. For example, it would be more efficient to modify a sequence of the form {filter, sorter, filter, filter, function, grouper} to {filter, filter, filter, function, sorter, grouper}. The task of algebraic optimization is not trivial, and one must ensure the applicability and correctness of such modifications, and the correctness of the flow as well. As for now, only a few research attempts have tackled this issue [12, 14]. The commercial ETL software either do not support any automatic optimization capabilities or offer limited optimization functionality (e.g., the PushDown optimization that pushes, usually small, portions of the ETL workflow, –either its beginning or its end– to the DBMS trying to leverage its optimization power [5]).

Apart from the algebraic optimization, another promising technique is to partition the data using any of the broadly available methods –e.g., round-robin, hash-based, key-value, etc.– and to parallelize the flow (or some parts of it). However, this is not an automated task and should be designed manually. Additionally, parallelizing a flow is not a panacea for improving its performance. It is common knowledge that the parallelization is more useful when "smaller" data volumes are involved and this is mainly because the cost of merging back the partitioned data is not cheap. Interesting questions that arise here involve determining where to split and merge the parallel flows, the most appropriate partitioning method to be used, the number of parallel flows and processors to be used (the two numbers are not necessary equal), and so on.

Recoverability. ETL are software artifacts and as such they face the probability of suffering from errors. Recoverability reflects the ability of an ETL flow to resume after interruption and restore the process to the point at which a failure occurred within a specified time window. An error occurs when a tuple does not arrive at the target data warehouse at the expected time and form. Such an error can be a result of an *ETL operation failure* or *a system failure*. The former type of error is expected to be minimized after a thorough testing of the design. Usually, such errors may come up mostly due to the *evolution of the ETL design* [16] or the *modification of the underlying systems* (e.g., a software or hardware update). On the other hand, it is more difficult to restrain system failures. Typical errors of this category are due to *network, power, human,* r*esource* or *other (miscellaneous)* failures.

To deal with these errors, there are some options. The straightforward one is to restart the ETL process from scratch. This does not apply to most cases though, since usually the uninterrupted ETL execution nearly fits in the available time window. Another option is to insert recovery point at several places of the ETL workflow to make a persistent copy of the flow. Candidate places are just after the extraction or before the loading (as landing tables or files), within the transformations flow (as intermediate tables or files) or at any temporary storage points used by blocking operators like sorters. Then, in the presence of a failure, the system restarts from the closest recovery point that contains correct data (the task gets complicated if one considers the rollback of the already processed tuples or the incremental restart regarding only the previously unprocessed data). Obviously, the introduction of a recovery point adds an additional I/O cost for writing to disk, but this should be amortized appropriately over the operational cost of the covered operations (the ones that precede the recovery point) and taking also under consideration the failure probability at different places of the workflow.

Interesting challenges that need to be tackled include how many recovery points to use and at which positions on the workflow. For example, a simple flow may have none if it is faster to completely rerun it. On the other hand, for more costly flows, it makes sense to add a recovery point after extraction (to reduce overhead on the source systems) or following an operation that is costly or difficult to undo (e.g., a sort). A common guideline is to store between every phase (e.g., after the extraction and after the transformations). In general, practice shows that after a failure the design should allow restart from a previous recovery point or a previous join point (when more than one flow are merged) or from the beginning of the flow if all else fails. In any case, restartability gains if we land to disk when data are in a good state. Finally, as a rule of the thumb, to leave time for potential recovery, the design should contain operations that each run for less than: (ETL_time_window – ETL_total_execution_time).

Reliability. Ideally, in the presence of a failure, the process should either resume accordingly or should be immune to the error occurred. The use of recovery points aims at achieving the first goal. However, if the available execution time window does not allow having recovery points (mainly due to the I/O cost needed for their maintenance) or if the business requirements include a demand for high reliability, an alternative to cope with system failures is to increase the software reliability or in other words, to improve the ETL workflow performance over time.

ETL reliability represents the probability that an ETL design will perform its intended operation during a specified time period under given conditions. In general, fault-tolerance can be achieved by either *replication* (running in parallel multiple identical instances of a flow), *redundancy* (providing multiple identical instances of a flow and switching to one of the remaining instances in case of a failure) or *diversity* (providing multiple different implementations of a flow, and using them like replicated systems to cope with errors in a specific implementation). There are various qualitative measures related to reliability, such as computation availability (i.e., the expected computation capacity of the system at given time t) or computation reliability (i.e., the failure-free probability that the system will execute a task of length x initiated at time t without an error). Such measures that usually are quantified during the gathering of business requirements are expressed in terms of more fine-grained quantitative measures as the mean time to failure, mean computation before failure, capacity threshold, computation threshold, and so on.

Freshness. This metric concerns the latency between the occurrence of an event at an operational system (or data source) and its appearance in the data warehouse. Better freshness (reduced latency) requires either performance improvements or alternative design (or both). The former can affect the resources allocation (e.g., more memory and processing power should be assigned to that flow) or techniques related to performance improvement (e.g., re-arranging operators). The latter involve design decisions like whether to use parallelism or instead of using recovery points, one may consider using replication or redundancy. Also, depending on the incoming rate of tuples, alternative implementation techniques more suitable for faster incoming rates should be considered at the physical level. Recently, implementation algorithms specifically tailored for the real time ETL have been proposed either for the transformation [e.g., 9] or the loading phases [e.g., 13]. It is imperative that lightweight ETL flows should be used in such cases, which should avoid using blocking operations, where possible. In that sense, scheduling of both the data flow and execution order of transformations becomes crucial [4].

Maintainability. Some hard-to-quantify measures, such as maintainability, are often overlooked when designing an ETL workflow. That increases at a later point the development cost, the project overall cost (e.g., in terms of people engaged in it), and the performance cost (e.g., as a result of "spaghetti" coding). Consider the following two cases. First, when an expert needs to modify the design, his/her task would be easier if the design is readable and well documented; especially, when this expert is not the original designer. Second, when a change occurs at the source or even the target schemas (e.g., insert of a new attribute, modification of a data type, drop/rename of a table/view, and so on), then the workflow should easily adapt to that change; however, this task is not straightforward [8]. In that sense, maintainability is an important measure that should drive the ETL design. Typical metrics for the maintainability of a flow are its size, length, modularity (cohesion), coupling, and complexity [16]. Unfortunately, as far as we are aware, current ETL tools do not provide the functionality for considering maintainability during the design.

Cost. The abovementioned measures have a common reference point: the overall cost. This can be expressed in either financial units, time units, personnel required, hardware needed, and so on. The QoX metrics span different design levels. However, since we refer to software artifacts, the QoX metrics can be expressed in terms of resources needed for the ETL execution, such as memory, disk, processing power, network availability and speed, and other hardware and software resources. For example, the cost of buying an ETL tool should be balanced with the actual needs of the project and the prospect for future scaling. A similar decision is whether to choose a commercial product, an open-source tool or an in-house developed solution. Finally, for more accuracy, the total costs of owing, developing, using, and maintaining the ETL software and training and employing personnel for operating it should be added as well to the cost model.

2.3 Working with Tradeoffs

A major challenge is to identify dependencies and relationships among the metrics that cause tradeoffs in optimizations of flows. For example, a design may sacrifice performance for maintainability. Alternatively, in some cases, techniques for improving performance like partitioning and parallelization may increase freshness but on the other hand, may hurt maintainability and robustness. An inherent difficulty is that different metrics come into play at different levels of the methodology. For example, freshness and reliability can be evaluated at the physical level, while their implication at the conceptual or logical levels is not clear. On the other hand, maintainability and robustness can drive conceptual and logical modeling. Scalability and performance span the conceptual, logical, and physical levels. Another dimension of metrics is that some reflect characteristics of the data, such as freshness, consistency, traceability, while others are characteristic of the workflow, such as maintainability, recoverability, robustness.

A systematic approach to design or evaluate a design based on QoX tradeoffs benefits both the flow designer and the administrator. A great challenge is to devise a method for enabling comparison and tradeoffs of the different metrics. To incorporate the semantics that each metric has into a common design space, we consider two classes of metrics: the *qualitative* and the *quantitative*. The former contains "higher level" QoX metrics that can be seen as *soft-goals*; e.g., "The ETL process should be *reliable*." The latter contains "lower level" metrics that are functional parameters of the system; e.g., time window, execution time, recoverability time, arrival time, number of failures, latency of data updates, memory, space, CPU utilization uptime, throughput, number of processors, and so on. With this modeling approach, we are able to correlate the QoX metrics and enable comparison among them. For example, the notion of "*reliable*" can be expressed as: "the *mean time between failures (MTBF)* should be greater than x time units". Another example could be "the *uptime* should be more than y time units." Working like this, we get a means for associating reliability with availability. Going from higher to lower, more detailed, design levels, these associations become more concrete and are represented by objective functions.

For supporting the systematic modeling of the design, soft-goal interdependency graphs can be used [1]. Consider the case of a design that should balance requirements for reliability, maintainability, performance, and freshness. Figure 2 shows the respective interdependency graph. These three soft-goals, expressed in the form of type[topic], are refined as soft-sub-goals and are based either on topic or on type. The graph shows the relationships among the soft-goals and the quantitative measures. For example, Figure 2 illustrates that the degree of parallelism contributes extremely positively (++) to the fulfillment of the reliability[software] soft-goal, since it can be seen as a form of redundancy. It also affects positively freshness and performance. On the other hand, parallelism affects negatively (-) the reliability of hardware (more devices increase the probability of failure).

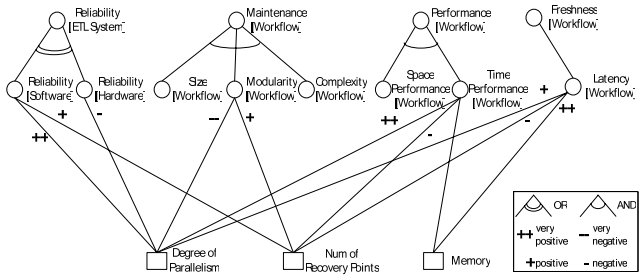

Figure 2. Example soft-goal interdependency graph

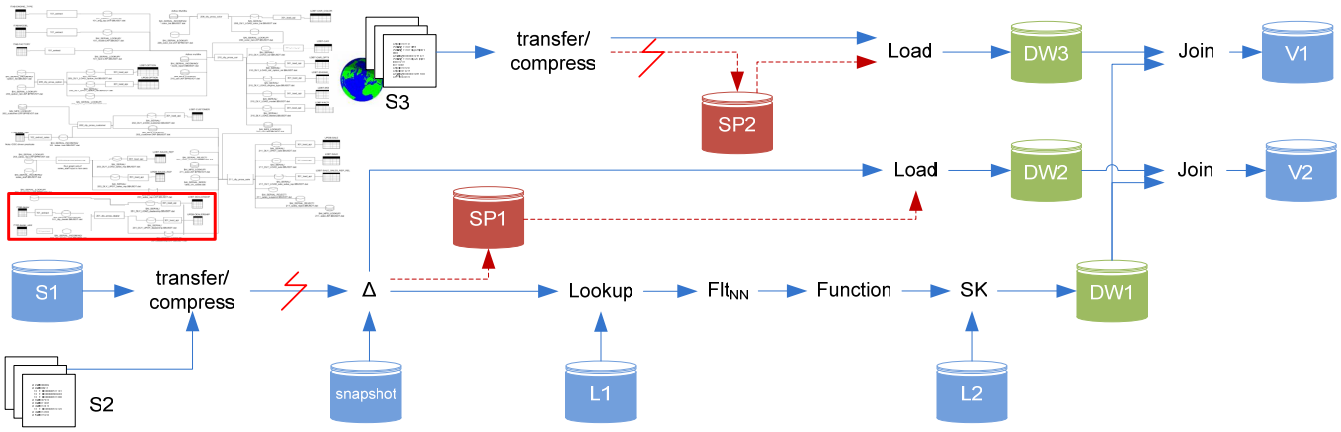

Figure 3. Example ETL workflow

Such interdependency graphs can be used for facilitating the understanding of the requirements by all the parties involved in the ETL projects and for visualizing tradeoffs among multiple design QoX objectives. For example, they are a useful tool for allowing designers to demonstrate to business people the implication of the objectives to the design and how the different objectives affect each other. Ultimately, they can be used to drive the search space for the design optimization or even for evaluating the cost of a given design regarding a set of QoX metrics.

3. OPTIMIZATION TRADEOFFS

In this section, we elaborate on the necessity of using the QoX metric suite by means of an ETL application from the enterprise domain (here we discuss a simplified version of it). The respective ETL workflow is depicted in the upper left corner of Figure 3. For ease of presentation, we focus only on the highlighted part of the scenario (enclosed by the rectangle at the bottom left corner of the large workflow), which involves the propagation of sales data to the target data warehouse. The scenario is depicted in Figure 3. For now, consider the entities SP1 and SP2 as placeholders for recovery (or save) points; we discuss their use later on. The actual names of the involved entities are as follows:

S1: SALES_TRAN, S2: SALES_STAFF, S3: CUSTWEB_CS,
L1: STORE_DT, L2: PRODUCT,
DW1: SALES, DW2: SALES_REP, DW3: CUSTOMER,
V1: CUSTOMER_SALE_RELS, V2: SAL_SALES_REP_RELS.

Conceptually, this scenario can be divided in three main parts: the sources, the data transformation area (a.k.a. data staging area, DSA [7]), and the data warehouse site.

The source site comprises three source data stores that follow different schemas and have different formats. Source S1 is a relational table storing transactional data about sales and source S2 consist of a set of files (log-sniffer dumps) containing information about sales staff (e.g., status, branch, and working hours). Data extracted from both sources use the same transfer channel for populating the data transformation area. Source S3 handles data from the enterprise's web portal. Usually, their propagation should be done in a streaming fashion, but at different moments depending on system's workload and business requirements it can be done through batches of small files as well.

The transformation part consists of three conceptual flows; each one is responsible for the population of a single data warehouse table. The top flow essentially populates the customer table in the data warehouse with click-stream data captured at the web portal. This flow has a pressing requirement for freshness and also, in its most frequent configuration, has to deal with streaming input. The middle flow depends on data from source S2. Following the landing of that data, the newly inserted and updated records are loaded to DW2, which stores information about sales representatives. Both the top and middle flows contain a few transformations hidden under the load task, which are not discussed here for space limitations. The bottom flow that populates the sales fact table is fed from both S1 and S2. The data after their landing to the transformation area are compared (Δ transformation) against the previous landing (snapshot table) for identifying the changed tuples. Then, four transformations are applied: a lookup operation (for finding corresponding codes from store sites and for verifying the moving information as well), a filter (for rejecting tuples containing null values) and a function operation (for modifying the schema), and finally, a surrogate key assignment that replaces the transactional keys with surrogate keys. Next, the data populate the DW1 table.

Finally, at the data warehouse site, there exist (in our example workflow) three tables and two views defined on top of them. The first view, V1, relates customer and sales information (e.g., for identifying customers' status: platinum, gold, and silver). The second view, V2, relates sales representatives and sales (e.g., for categorizing staff and branches based on their performance).

Although, at a first glance the above design seems reasonably adequate, looking at it from different perspectives reveals various optimization opportunities. In the rest, we use the above example for a discussion on design tradeoffs among different optimization objectives. For assisting the presentation, we present a few illustrative graphs produced for different configurations of the example workflow. (We implemented the same workflow in different open source and commercial ETL tools; although the behavior and performance of these tools differ, the trends discussed in this study are alike in the tools we used.) All graphs depict average values concerning the best possible configuration in each case, unless otherwise stated. We stress that our goal here is to indicate the tradeoffs in using different QoX metrics, and not to focus on individual numbers.

3.1 Optimization for Performance

Although performance is the typical optimization objective, still with current ETL technology, designs must be optimized in a manual, rather ad-hoc way, based on previous practices known to the designer. As we discussed, one solution is algebraic optimization of the design. Following the idea of moving the most restrictive operators to the start of a flow, for the bottom flow of Figure 3, an option for reducing the data volume will be to move the Flt_{NN} before the lookup operation; of course the move must be valid (the filter does not depend on the lookup) and offer some gain (the data do contain null values).

Another optimization opportunity concerns the parallelization of the ETL workflow. Conceptually, this implies two actions: (a) assigning more CPU's (at least more than one) to the ETL execution and (b) creating multiple flows that run in parallel. Usually, both actions should be performed together and that complicates the task of automatically tuning this process. In addition, one important tuning decision involves whether to parallelize the whole ETL flow or parts of it.

For demonstrating these tradeoffs, we experimented with the example of Figure 3 using different execution scenarios involving multiple CPUs (from 1 to 8) and different parallelization options (see Figure 4). Specifically, we tested 4 configurations: (a) 1PF, in which we did not parallelize the ETL process (i.e., this is the normal configuration); (b) 4PF-p, in which we parallelized parts of the original flow using 4 parallel branches; (c) 4PF-f, in which we run the whole ETL flow in 4 branches; and (d) 8PF-p, in which we parallelized parts of the flow using 8 parallel flows. Some interesting tradeoffs are depicted in Figure 4. The part of each bar filled with strong color represents extraction times, whereas the part filled with faded color represents transformation times. It is not only that extraction dominates the ETL execution in this case; one may observe that the parallelization improves more the transformation part in almost all the settings. Also, it seems that increasing the processing power does not improve performance linearly. Running the whole workflow in parallel is not the best solution either. Possible tunings involve the number of partitions and which part(s) of the flow run in parallel, as well.

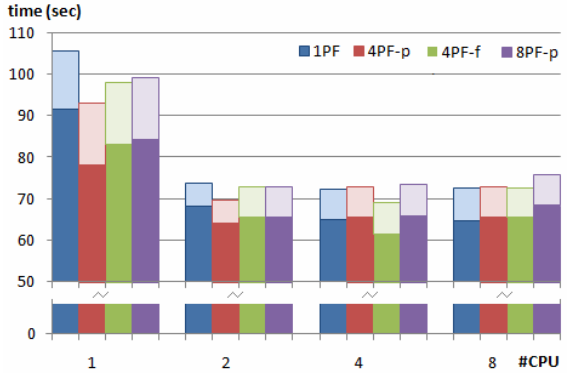

Figure 4. Parallelization effects on ETL design

Notice that just assigning more processors to an ETL workflow (without performing any parallelization) is not optimal. Observe, also, Figure 5 (the w/o RP case) where the normal execution of a part of the example scenario is not much affected by simply assigning more processors; an appropriate parallelization policy is required.

3.2 Optimization for Recoverability and Performance

As we have discussed, the use of recovery points (RP) –placed in various points of the workflow– significantly increases the total execution cost since it adds an additional I/O cost. We show this by experimenting with the example workflow.

Figure 5 shows the behavior of the design when we use recovery points (w/ RP case) and when we don't (w/o RP case). In particular, w/ RP (b) and w/ RP (w) are results for the best and worst possible configuration, respectively. In this experiment, for isolating the impact of recoverability, we did not parallelize the flow; we tested the performance of a single flow varying the number of processors. The results show that using recovery points increases significantly the total cost. In fact, even when the processing power increases the w/ RP (b) case is getting much worse than the normal case (which performs slightly better), because the threads responsible for I/O operations do not exploit caching benefits to an adequate extent.

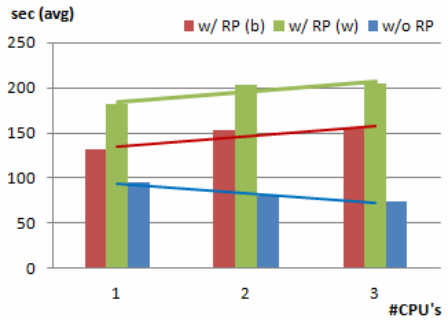

Figure 5. Cost imposed by the use of recovery points

However, when a failure occurs, the use of recovery points improves performance. Observe Figure 6, where the normal (w/o RP) and the w/RP(b) cases of Figure 5 (without failures, w/o f) are examined again, this time in the presence of a failure (with failure, w/ f). Then, the performance of normal case (w/f, w/o RP) is worse than the performance when recovery points are used. Still, if a failure occurs near the previous recovery point (w/f, w/ RP(b)-n), then, the recovery performs well. Otherwise, if a failure occurs at a point far from the previous recovery point (w/f, w/ RP(b)-f), then, the recovery does not perform well. (Of course, this result depends on the cost of transformations existing between the failure and the previous recovery points; if the transformations' cost is fairly small, then the recovery performs better.)

Figure 6. Cost in the presence of a failure

Therefore, interesting decision points include the number and the placement of the recovery points on the workflow. As we have seen, different design choices result in different results (observe best and worst cases in Figure 5 and the scenarios of a failure occurred near and far from a recovery point in Figure 6).

However, some heuristics can be used for facilitating such decision. One may use a recovery point –e.g., SP1 and/or SP2 in Figure 3– just after extraction when the data reach the transformation area or after resolving the changes (the exact point is a matter of tuning and depends on the specific scenario details). Recall the results of Figure 4 where the extraction dominates the execution time; when the extraction is that costly, then it definitely makes sense to add a recovery point close to it. Using recovery points after the extraction phase assists in more than one way beyond recovery; e.g., as a synchronization point since not all sources or source data are available at the same time. Additionally, since the network channels used between the source sites and the transformation area inflict a notion of unpredictability on the workflow and increase the risk of having a failure, one may want to store just after the data transfer depending on the reliability of his/her network. On the other hand, regarding performance, it is often faster to store first to a flat file and later populate a table, instead of hitting the table directly.

3.3 Optimization for Reliability, Recoverability, and Performance

An alternative to recovery points is to have ETL designs that can tolerate such failures; in other words, to design reliable ETL designs.

An obvious tradeoff involves the reliability versus the performance. As the execution time window decreases and especially, as it reaches the near real-time case, the option for recovery diminishes. Consider the top flow of our example: the data are coming in fast rates, and since we cannot afford to lose any tuple (this is a strict requirement in the ETL context [15]), we need either to store them in a persistent disk-based data store (e.g., at SP2) or to use a fault-tolerant method.

For ensuring the reliability of the process, we may need to relax the other objectives' expectations. If we choose to keep the cost intact, we can retain the ETL budget by accepting lower performance standards. For example, we can use the same resources and instead of parallelizing the ETL process (multiple threads handle subsets of the data volumes) we can replicate the ETL process (multiple threads handle the whole data volume). In that sense, we follow a fail-safe design that allows the system to continue operation at a reduced level (graceful degradation) and avoids a potential crash of the process, when a certain component fails. Typical consequences are a reduction in throughput and an increase in execution time. Observe Figure 7 where we measured the additional costs that recovery points and redundancy impose to the normal execution of the workflow. Clearly, redundancy guarantees better performance than recovery but the relative improvement depends on the redundancy type used. Figure 7 shows average values for n-modular redundancy (NMR) that vary from 14% (for triple modular redundancy, TMR) to 58% (5-modular redundancy). Of course, as the number of redundant flows increases the reliability of the system increases too (the failure probability decreases), but then the system resources are shared among a larger number of flows.

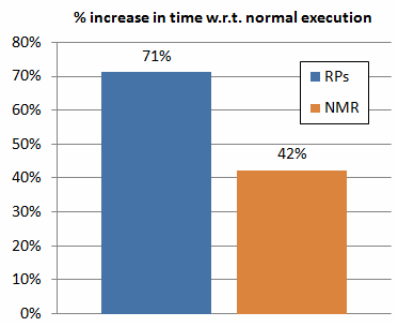

Figure 7. Use of recovery points vs. NMR

An alternative for honoring the performance requirements affects the total cost of the process. For example, we can maintain the execution of the ETL process to achieve the performance goals, but then, we should increase the resources (e.g., number of machines) used for executing concurrently the same process multiple times.

3.4 Optimization for Freshness, Reliability, Recoverability, and Performance

The top flow of Figure 3 needs special care when it involves streaming data and the process is realized in a (near) real-time fashion. At the physical level, as we discussed, possible solutions include changing the resource allocation, using implementation techniques suitable for streaming data, and parallelizing the flow. Since in this case the goal is to minimize the latency for updating the target data store, we need to increase the frequency of loads (i.e., the frequency of ETL executions) in the course of time too.

However, a requirement for freshness may change the design at higher levels as well. Considering the bottom flow, assume that a service level agreement requires that the freshness of data in table DW1 or in view V1 should be no less than t time units and that at the same time, the sources S1 and S2 feed data to the flow at different rates (rows/sec) (S1 provides data faster than S2). Having that knowledge, a possible solution is to create three different flows for populating the middle and bottom flows. The middle flow should have its own link to the source S2 to avoid further delaying the bottom flow. Then, the bottom flow can be replaced by two new flows, a faster and a slower, each one corresponding to a different source. With such change that can be captured even at the conceptual or logical levels, we can improve the freshness of data in DW1 and in V1 (the other branch that feeds it is the real-time one). Similarly, we can work for the recoverability of the process; the flow having more pressing requirement for freshness may use replication, while the other may use recovery points.

Observe Figure 8 that shows the propagation of a certain data volume to the target data store using various design configurations. The configurations tested involve running the process in 2 parallel flows without any recovery (w/o RP, 2PF), using redundancy (TMR), using a small (RP+) and a larger (RP++) number of recovery points, and executing the process without recovery points, redundancy and parallelization (w/o RP, 1F). The y-axis in Figure 8 shows the time needed for propagating a change occurred at the source to the target data store. As the number of loads per time unit (e.g., per day) increases (and the data are processed in batches of smaller size) the freshness of the data warehouse gets improved. However, each configuration shows

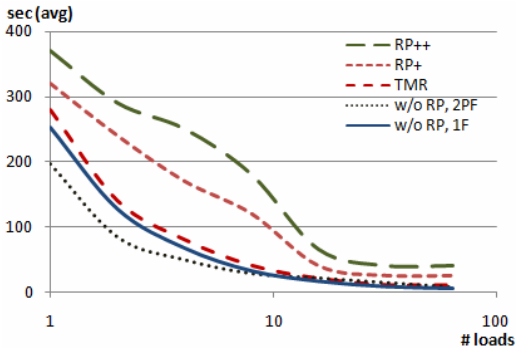

Figure 8. Freshness of data vs. frequency of ETL execution

different results. Therefore, the designer should choose according to the requirements for the design; for example, the parallelized version should be preferred when only performance matters, but to cope with failures the TMR is more appealing.

3.5 Optimization for Other QoX Metrics

In addition, the maintainability of the flow depicted in Figure 3 can be improved. An ETL workflow can be represented as a directed graph; its nodes are the data stores and ETL operations of the workflow. Observe that most of the nodes (e.g., ETL operations) of our example workflow depend on one node and feed another one. However, the Δ transformation depends on three nodes (i.e., the three sources) and many nodes depend on it (i.e., observe the two flows starting from Δ and the SP1 recovery point). That makes the Δ transformation a *vulnerable* point of the design [16]. If we follow the suggestion of replacing the middle and bottom flows with three new flows starting from the respective sources, then this problem will be resolved. In addition, the workflow complexity gets improved, but the modularity and size of the workflow are affected negatively (e.g., parts of the workflow perform identical tasks).

From another perspective, one may choose to increase the workflow complexity and the data volumes by enriching the data flow with extra information useful for provenance purposes. In doing so, at least the performance, freshness, complexity, and in some extent the auditability of the system are hurt, but the *traceability* gains ground.

4. CONCLUSIONS

In this paper, we have addressed the problem of dealing with multiple quality and optimization objectives in an ETL design. Existing research and commercial solutions focus mainly on performance, while practice shows that other metrics like recoverability, reliability, maintainability, freshness, scalability, availability, flexibility, robustness, and so on, are also of great interest. In fact, since current solutions neither capture nor track such qualities, the consulting and designing teams have to revisit and complement the designs at later points. Naturally, this process increases the consulting and thus, the overall cost of the ETL design. We have presented the QoX metric suite that aims at handling such metrics in all the ETL design levels. We have discussed the interrelationships and dependencies among the metrics that lead to tradeoffs for alternative optimizations of ETL processes. Another challenge is creating tools to automate the optimization, which is a topic we are working on.

The QoX metric suite is extensible to other metrics. It can work on top of any ETL engine that provides export and import capabilities (e.g., the metadata of an ETL workflow can be exported as or imported from an XML file). Additionally, our approach is agnostic to any particular implementation style, e.g., ETL, ELT, ELTL, ETLT. We are not tying ourselves to a specific product; rather, we are creating a consulting solution for creating ETL designs that are optimal for the customer's QoX requirements, for optimizing existing ETL designs for QoX metrics, and for evaluating the cost of a given ETL design.

5. ACKNOWLEDGMENTS

Our thanks to Paul Watson, Paul Urban, John Bicknell, Rom Linhares, Heather Wainscott, Tom Harrocks, Werner Rheeder, and Blair Elzinga for sharing with us their insights and experience regarding the ETL practice and needs in real-world, large-scale projects.

6. REFERENCES

1. L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos. Non-Functional Requirements in Software Engineering. Kluwer Academic Publishing, 1999.
2. N.N. Dalvi, S.K. Sanghai, P. Roy, S. Sudarshan. Pipelining in Multi-Query Optimization. In PODS, 2001.
3. U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson. Data Integration Flows for Business Intelligence. In EDBT, 2009.
4. L. Golab, T. Johnson, V. Shkapenyuk. Scheduling Updates in a Real-Time Stream Warehouse. In ICDE, 2009.
5. Informatica. How to Achieve Flexible, Cost-effective Scalability and Performance through Pushdown Processing. White paper, November 2007.
6. W.H. Inmon. Building the Data Warehouse. John Wiley, 1993.
7. Kimball, R., et al.: The Data Warehouse Lifecycle Toolkit. John Wiley & Sons, 1998.
8. G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou. Policy-Regulated Management of ETL Evolution. In Springer JoDS, Vol. XIII, pp. 146–176, 2009.
9. N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, N.-E. Frantzell. Supporting Streaming Updates in an Active Data Warehouse. In ICDE, pp. 476-485, 2007.
10. P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In SIGMOD, pp. 249-260, 2000.
11. T.K. Sellis. Multiple-Query Optimization. In ACM Trans. Database Syst. 13(1), pp. 23-52, 1988.
12. A. Simitsis, P. Vassiliadis, T.K. Sellis. Optimizing ETL Processes in Data Warehouses. In ICDE, pp. 564-575, 2005.
13. C. Thomsen, T.B. Pedersen, W. Lehner. RiTE: Providing On-Demand Data for Right-Time Data Warehousing. In ICDE, pp. 456-465, 2008.
14. V. Tziovara, P. Vassiliadis, A. Simitsis. Deciding the Phys-ical Implementation of ETL Workflows. In DOLAP, pp. 49-56, 2007.
15. P. Vassiliadis, A. Simitsis. Near Real Time ETL. In Springer Annals of Information Systems, Vol. 3, pp. 19-29, 2008.
16. P. Vassiliadis, A. Simitsis, M. Terrovitis, S. Skiadopoulos. Blueprints and Measures for ETL Workflows. In ER, pp. 385-400, 2005.