Partitioning Real-Time ETL Workflows

Alkis Simitsis[#1], Chetan Gupta[*2], Song Wang[*3], Umeshwar Dayal[#4]

[#]*HP Labs*
Palo Alto, CA, USA
[1]alkis@hp.com, [4]umeshwar.dayal@hp.com

[*]*HP Labs*
Austin, TX, USA
[2]chetan.gupta@hp.com, [3]songw@hp.com

Abstract—**Many organizations are aiming to move away from traditional batch processing ETL to real-time ETL (RT-ETL). This move is motivated by a need to analyze and take decisions on as fresh a data as possible. The RT-ETL engines operate on the abstraction of data flow executed on parallel architectures. For high throughput and low response times, there is a need for partitioning the data over the large number of nodes in the engine. In this paper, we consider the problem of partitioning real-time ETL flows and we propose a high level architecture for that.**

I. Introduction

Businesses are increasingly demanding an ability to make decisions at any time and over any time scale, i.e., they are no longer patient for a traditional Extract-Transform-Load (ETL) process to load data once a day and wait for it to become available for analysis and decision making. One such example is the transportation networks in world's major cities such as London which collect data from its users. They want to use this data to make real time decisions on congestion, rerouting and so on, rather than wait a day, in which case the data is useful mostly for record keeping and auditing. In such a scenario, the use of real-time ETL (RT-ETL) becomes critical. An additional reason that enterprises tend to move towards RT-ETL is to reduce management and labor costs.

The main difference between RT-ETL and traditional, batch processing ETL is that an RT-ETL process is responsible for keeping the target data stores – e.g., data warehouse (DW) tables – as fresh as possible. In general, RT-ETL processes are executed more frequently, run in streaming fashion, and typically, involve smaller data volumes [15]. For the successful design and execution of RT-ETL processes, progress needs to be made at both the workflow design and execution levels.

The problem of ETL workflow design is an optimization problem [10]. It is already challenging in the traditional batch processing mode, but it becomes even more challenging in RT-ETL. For example we have to consider not only queries over stored data, but also queries over streaming data and hybrid queries that need to access both streaming and stored data. Therefore, the problem of workflow design for RT-ETL can be seen as a superset of the ETL workflow design problem.

The RT-ETL execution problem is a problem of autonomic management. Streaming data behaves much more erratically than persistent data does. Furthermore, we often do not know much about the properties of streaming data that can help us design an optimal execution strategy a-priori. The total costs of enterprise data warehouses are already large, and we need to design new strategies that automate the decision making as much as possible. This not only helps with reduction in costs but can lead to quicker reaction times as compared to a human operator – an ability that is useful especially with real time systems such as a RT-ETL.

To address the problem of design and execution of ETL workflows we have proposed a framework based on a set of quality metrics, which we collectively call QoX metrics [4]. Our QoX-driven design framework assists in designing workflows such that the various competing objectives, like performance, freshness, reliability, recoverability, fault-tolerance, maintainability, auditability, and scalability, can be optimised all together. This leads to a choice between a variety of tradeoffs [11]. An example tradeoff is between freshness and recoverability. For high speed data flows keeping the data fresh might mean that business requirements for certain values of MTTR (mean-time-to-recover) may not be satisfied, and we may use alternative means for ensuring fault-tolerance [12]. In this paper, we discuss a thread of our on-going work on using the QoX-driven approach to design and automatically tune ETL workflows to deal with QoX tradeoffs in real time.

Although, there are several challenges and problems that RT-ETL needs to tackle, here we only focus on issues related to partitioning RT-ETL flows, which is a key technique for improving performance and thus, latency, especially on high parallel systems [12]. In this context, several parameters need to be considered as flow throughput, memory requirements, response time, load of each partition and each operator within a partition, synchronization, and so on. Such issues drive the choice of a partitioning policy, which determines how the data are distributed across partitions, and of an appropriate placement of the partitioning points.

To address the partitioning problem we need to introduce two generic operators: first, a *router* operator to split the data into multiple flows and then, to merge back the data into one flow we need a *merger* operator. Existing techniques and tools are not specifically tailored to accommodate the needs of a real time environment as we discuss in Section II. In Section III, we discuss the characteristics of the real time case, present partitioning and merging policies, and propose a high level architecture for routing and merging operations that fits in the real time context.

II. Background

Partitioning in ETL. Today's commercial or open source ETL tools provide the designer with functionality for partitioning an integration flow. What is not yet provided is an *automatic* and *systematic* method to partition ETL designs. Hence, the designer has to decide how to partition based on his/her experience and previous practice.

On the other hand, as far as we are aware, research efforts have not dealt so far with the partitioning problem in the ETL context. Past work has demonstrated results on applying partitioning in synergy with other quality objectives [11, 12]. These results show that by simply partitioning a flow (and for example by just increasing the degree of parallelism) we do not necessarily satisfy our requirements. Hence, efficiently partitioning a flow is a complex flow optimization problem and deserves special treatment. There exist efforts focusing on the optimization of ETL flows (without considering partitioning though) considering the conventional batch processing of ETL flows [10]. In the context of RT-ETL [15], researchers have focused on optimizing specific parts of the flow [e.g., 9, 14] and on scheduling issues [e.g., 7]. None of these approaches have studied partitioning in an adequate extent.

In the RT-ETL case, due to the streaming nature of the processing data, several of the techniques and assumptions made for the conventional batch-mode ETL cannot be used. In addition, the fact that we see partitioning as a means to optimize a data flow for a variety of quality objectives (QoX) adds a new perspective to the ETL optimization.

Partitioning in Streams. Distributed stream processing has been the focus of many recent research efforts, such as [3] for distributed Aurora, [2] for Borealis, and so on. Many aspects of the parallel processing stream engine have been tackled, such as pipelined and partition parallel execution. However, RT-ETL processing differs from stream processing in the following aspects: (a) the processing time variance for RT-ETL is much larger than for stream processing; (b) popular stream processing approaches like load shedding [13] and approximation are not generally acceptable in RT-ETL design; and (c) RT-ETL workflows involve complex data analytics and face various quality requirements and resource constraints. Because of these differences, partitioning for RT-ETL involves additional choices. For example, beyond the quality aware scheduling and load balancing, RT-ETL also need to handle QoX aware adaptive partitioning, data flow migration and other end-to-end optimizations at running time.

Moreover, most of the partitioning techniques deployed by the state-of-the-art stream engines are operation specific. For example, in [8] the authors proposed special approaches for pipelined and partition stream join. Such approaches are not directly applicable and useful for RT-ETL, which may include operations like joins between very fast and also very slow streams or joins between streams and historical data, and so on. Also, these works concern parallelization of single operators and are not aware of end-to-end quality requirements and global constraints.

Partitioning in DBMSs and Workflows. There are many works for parallel processing in relational DBMS and they have performance as their main concern [e.g., 5, 6]. Their main difference compared to our work is that ETL processes contain operators that are not typically handled by the optimizer of relational DBMS (e.g., a plethora of functions, user-defined functions or calls to external dlls that would be treated as black boxes by relational database systems). In addition, we consider processing of both persistent and non-persistent data at run time and our focus is dynamic and adaptive partitioning at query time and such partitioning must be QoX aware. Thus, existing techniques are not directly applicable to our context. Similar differences exist when we compare RT-ETL with shared-nothing massive parallel processing (MPP) techniques and the recently proposed HadoopDB [1]. However, the latter does share with RT-ETL some similar qualities beyond performance, such as scalability, fault tolerance, and flexibility. Hence, we are interested in adapting some functionality of HadoopDB in the near future.

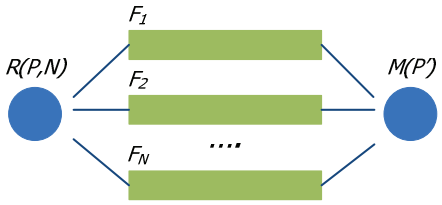

Fig. 1. Abstract partitioning of a flow F into N flows

Partitioning has been considered for workflows in general. Related work on this topic has focused on different aspects like concurrency; e.g., how to assign resources to nodes. As far as we are aware of, partitioning of workflows as a method for optimizing for various qualities has not been considered.

III. QoX-Driven Approach to Partitioning

A. Preliminaries

Assume an ETL flow F containing K operators. Such operators span a large variety of schema and cleansing transformations, filters, functions (e.g., string manipulation and date functions, user-defined functions, calls to external dll's), sorters, aggregators, (un-)pivot, and so on. These operators may be either blocking or pipeline operators. Parallelizing F involves pipelining and partitioning parallelization. Elsewhere, we have discussed how to group together pipeline operators for enabling large pipeline chains and boost pipeline parallelization [12]. Here, we focus on *partitioning parallelism*.

To enable partitioning of a flow F, we first need a router operator $R(P,N)$ that splits the flow in N *partitions* [1] (or branches) using a partitioning policy P. To merge back the partitions into one flow, we need a merger operator $M(P')$ that merges the incoming partitions using the merging policy P'. Fig. 1 shows the usage of router and merger operators in an example flow F involving N partitions.

For ensuring correctness, we need to guarantee that the semantics of the flow F are retained after partitioning. This means that whatever semantics the flow F has before the routing takes place, these should be preserved after the merging. For achieving that, the merging policy P' needs to take into account the partitioning policy P.

Formally, we denote as $pre_{\Phi}(F)$ and $post_{\Phi}(F)$ the pre- and post-conditions, respectively, of the flow F over a set of semantics Φ. Then, we say that the partitioning is correct iff:

$$post_{\Phi}(F) = pre_{\Phi}(F) \quad (1)$$

[1] Observe that in our context 'partition' indicates a branch of the flow that processes a subset of the incoming data, whereas in databases 'partition' typically indicates a certain table structure.

Note that it is not always necessary to ensure correctness as described by the equality 1. There are cases where we might not be interested in preserving the flow semantics. Typically, restricting partitioning to respect the equality 1 forces us to use a more expensive merging policy P'. Example semantics of a flow F may determine a certain order for the data processed by F. If we do care about preserving this order after the merging, then equality 1 should hold and the merge costs; otherwise, we may choose a cheaper merging policy.

B. Partitioning and Merging Policies

Various partitioning policies are widely supported by commercial ETL tools. We do not restrain ourselves to any particular partitioning method. Our approach is generic and uses such methods as variants. Example policies include round robin (RR), hash (HS), range (RG), random, modulus, same (existing partitioning remains unchanged), copy (send all rows to all partitions), and so on [15]. We complement these policies by considering a novel partitioning policy termed QoX-based (QB). The need for such policy is based on the frequent demand of ETL consultants and practitioners for having a means to represent business requirements in all phases of ETL design and in doing so, such requirements will not be missed somewhere in the design. Hence, the designers will be able to satisfy such requirements to a greater extent.

QB partitions data according to QoX parameters. For example, if we know that a certain view/table/report has greater priority than others based on service level agreements (SLAs), then, QB will prioritize data populating such objects. An example criterion for QB is the data origin. Having stored provenance information for views/tables/reports, we may know their respective sources. Hence, data coming from certain sources should be favored against data stemming from other sources. Of course, QB may handle more complex cases as well. For example, it can discriminate rows coming from the same source according to some other criteria; e.g., using a filter or a more complex operation.

Each partition policy has several characteristics that help us decide upon its use. For example, RR is a fair policy that distributes data equally among partitions. When we use RR, each operator i belonging to a partition F_j has on average the same processing cost c_i in all partitions (assuming a uniform over time data distribution). (A discussion about a relevant cost model can be found in [12].) In fact, c_i's may vary since they depend on the availability of resources assigned for the execution of operator i (each partition may have different workload or different resources may be assigned to it) and on the actual selectivity of i too (due to skew in data distribution). Hence, RR does not leave room for much optimization as for example other policies like HB do. In HB (similarly, in QB too) we expect different data volumes in each partition determined by the hash function used. Thus, we have significantly different c_i's. One can go further and create partitions containing different set of operators i, since the properties of the hash function gives us some knowledge on the data processed by each partition. This enables various optimization opportunities.

In addition, the flow semantics Φ may drive our decision to a partitioning policy. If we need to preserve the schema of data, then we need to consider one of the abovementioned policies for *horizontal* partitioning (assuming of course that the operators contained in F will do the same). If preserving the schema is not as important as boosting performance, we may need to consider *vertical* partitioning. Then, we can focus on a subset of the columns that each tuple has, process only that information (possibly along with horizontal partitioning too) and at some point (e.g., at the merging) we can reconstruct the original schema. In doing so, we get a number of advantages as for example using the same amount of memory for processing additional number of tuples. Saving memory is of great importance in real-time environments.

When the data process finishes, we need to merge back the partitioned data. For doing so, we use a merging policy (a.k.a. data collecting policy). Typical merging policies include round robin, sequential (or ordered; reads all rows from first partition, then continues with the second and so on), sort merge (produces a globally sorted sequential stream from within partition sorted rows; typically, it picks the partition that produces the row with the smallest key value), and so on. As with the partitioning policies, each merging policy has its own characteristics. Choices of merging policy are not arbitrary. According to the flow semantics and the partitioning policy used, we choose an appropriate merging policy. For example, if preserving the original order is required, then sort merge policy (or a similar one) should be preferred despite its substantial processing cost.

Automating the design of RT-ETL flows and taking under consideration the abovementioned techniques is one thread of our on-going work. In the rest, due to space consideration, we focus on horizontal partitioning and briefly present the complexity of the problem and one way to deal with it.

C. Generic Architecture for Partitioning

In general, the case when we need to guarantee that the semantics Φ of F are kept intact after we partition a RT-ETL flow F, is more challenging. Without loss of generality, let's assume that Φ stipulates preserving the order of tuples in F. For preserving the order there are a few challenges:

- We need to know the original order of tuples or to be able to reproduce it.
- We need to merge using a policy P' that takes into account the partitioning policy P.
- We need to do so for varying stream rates (i.e., we need to keep up with the speed/rate of incoming tuples).

Dealing with the first challenge depends on the partitioning policy P. If we use a policy without memory as the round-robin, then we need to enrich data with additional information, which would assist us reconstructing the dataset after the merging. Such information could be an appropriate *timestamp* that can be added before we split the data. Of course, depending on the application, we may use an existing (unique) key as well for the same purpose; however, in this case, the data should be sorted on that key.

Fig. 2(a) shows a generic architecture for a router operator *R(P,N)*. Essentially, R is a composite operator comprising a function *add_time* that adds a new attribute value (e.g., a timestamp as '*currval*') to each tuple and a *fork* operation that

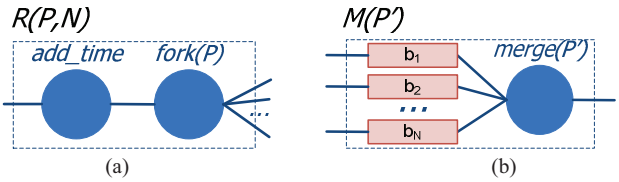

Fig. 2. A generic architecture for (a) router and (b) merger operators

sends data to N destinations using the partitioning policy P. Observe that both operations enable pipelining.

For ensuring the second challenge, when we merge back tuples from different partitions, we need to use an appropriate merging policy P' that considers P.

If we use round-robin as P, then ideally, we could synchronize the merging and use round-robin again for producing the same order. However, in reality this is not always feasible for various reasons. First, achieving perfect synchronization and expecting that all tuples will arrive at the merging point in the same order as they left from the fork is rather rare. Delays may occur in each partition due to several unpredictable reasons (e.g., system utilization or network problems) or even due to the different processing time each tuple may need. Assume for example a cleaning task i that has to check and clean problematic values in two tuples t_1 and t_2 running in two different partitions; if tuple t_1 is already cleansed and tuple t_2 needs to be cleansed, then clearly t_2 will face a greater delay than t_1. Another reason, for which a round robin merging technique cannot guarantee the order, is that a flow F may contain operations (e.g., filters) that project out tuples. Therefore, since the cardinality in each partition may change, we cannot reproduce the same order by simply using a synchronized round-robin merging technique. We face similar problems when we choose other partitioning policies.

For all these reasons, merging is not trivial and to deal with it, we need a feedback based partitioning technique. In general, we cannot simply decide what to do with a certain tuple the moment it arrives at the merging point. We need to keep it on hold until the merger M is ready to extract it. Hence, we need to accompany M with a buffer mechanism for temporarily holding the incoming tuples. This feature is abstractly presented in Fig. 2(b), where each partition j ends at a buffer b_j. Depending on the merging policy used, the merger may work either as a blocking or a pipeline operator. Clearly, varying buffer sizes may reduce in some cases its blocking nature.

Without loss of generality, we can assume that the router acts as a pipeline operator and thus, may handle much larger data volumes than the merger. Therefore, to keep up with the stream rate, which is the third challenge, merger becomes the bottleneck. (We may have blocking operators in the partitions, but we do not focus on that at this point.) Thus, for increasing merger's throughput, we exploit again the merger buffers. If all partitions have the same processing rate (throughput), then all buffers should be of equal size. However, if each partition has different throughput (which typically happens for the reasons we explained), then we need to allocate different memory size for each buffer. For a specific stream rate, this can be done offline. But in general, we need to dynamically tune buffer sizes to keep up with a continuously changing stream rate.

IV. Conclusions

We have discussed a thread of our on-going work at HP Labs for enabling QoX-driven optimization of RT-ETL workflows. Modern business needs advocate for a move towards real-time environments. In doing so, we need to take into account not only performance, but other quality objectives as well. Having difficulties to represent SLAs at all design phases is a common problem in practice. Our QoX-driven approach to ETL optimization tackles this problem.

In particular, this paper has focused on partitioning RT-ETL workflows, which we believe is a key technique to reduce latency in decision making. In our work, we treat partitioning and merging policies as variants, and we complement them by adding a QoX-driven partitioning policy. This assists our task of automating the representation and satisfaction of SLAs at lower design levels and not only at the business level. Choosing appropriate partitioning techniques is a task performed by our QoX-driven optimizer, which is under development and we continuously add features to it. Currently, it supports optimization of ETL workflows for performance, freshness, recoverability, and fault-tolerance by considering various techniques, partitioning included. Preliminary results have been already published elsewhere [e.g., 11, 12]. We are looking forward to having a fruitful discussion and getting feedback from the research community.

References

1. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi1, A. Silberschatz, A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In VLDB, 2009.
2. Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, S. B. Zdonik. Distributed operation in the Borealis stream processing engine. In SIGMOD, pp. 882-884, 2005.
3. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, S. B. Zdonik. Scalable Distributed Stream Processing. In CIDR, 2003.
4. U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson. Data Integration Flows for Business Intelligence. In EDBT, pp. 1-11, 2009.
5. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In VLDB '86, 1986.
6. S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of the System Software of a Parallel Relational Database Machine. In VLDB '86, 1986.
7. L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling Updates in a Real-Time Stream Warehouse. In ICDE, pp. 1207–1210, 2009.
8. X. Gu, P. S. Yu, H. Wang. Adaptive Load Diffusion for Multiway Windowed Stream Joins. In ICDE, pp. 146-155, 2007.
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. Simitsis, P. Vassiliadis, T. K. Sellis. Optimizing ETL Processes in Data Warehouse Environments. In ICDE, pp. 564-575, 2005.
11. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal. QoX-driven ETL design: Reducing the Cost of ETL Consulting Engagements. In SIGMOD, pp.953-960, 2009.
12. Simitsis, K. Wilkinson, U. Dayal, M. Castellanos. Optimizing ETL Workflows for Fault-Tolerance. In ICDE, 2010.
13. N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In VLDB, 2003.
14. C. Thomsen, T. B. Pedersen, and W. Lehner. RiTE: Providing On-Demand Data for Right-Time Data Warehousing. In ICDE, pp. 456–465, 2008.
15. P. Vassiliadis, A. Simitsis. Near Real Time ETL. In Springer's Annals of Information Systems, vol. 3, 2008.