Information Systems 38 (2013) 927–945

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Scheduling strategies for efficient ETL execution

Anastasios Karagiannis[a], Panos Vassiliadis[b], Alkis Simitsis[c,*]

[a] *Livemedia.gr, Inventics S.A., Thessaloniki, Hellas*[1]
[b] *University of Ioannina, Ioannina, Hellas*
[c] *HP Labs, Palo Alto, CA, United States*

ARTICLE INFO

Available online 25 December 2012

keywords:
Data warehouses
Extract-transform-load (ETL)
ETL optimization

ABSTRACT

Extract-transform-load (ETL) workflows model the population of enterprise data warehouses with information gathered from a large variety of heterogeneous data sources. ETL workflows are complex design structures that run under strict performance requirements and their optimization is crucial for satisfying business objectives. In this paper, we deal with the problem of scheduling the execution of ETL activities (a.k.a. transformations, tasks, operations), with the goal of minimizing ETL execution time and allocated memory. We investigate the effects of four scheduling policies on different flow structures and configurations and experimentally show that the use of different scheduling policies may improve ETL performance in terms of memory consumption and execution time. First, we examine a simple, fair scheduling policy. Then, we study the pros and cons of two other policies: the first opts for emptying the largest input queue of the flow and the second for activating the operation (a.k.a. activity) with the maximum tuple consumption rate. Finally, we examine a fourth policy that combines the advantages of the latter two in synergy with flow parallelization.

1. Introduction

Enterprise data warehouses (EDW or simply, DW) are complex systems serving as a repository of an organization's data. Apart from their role as enterprise data storage facilities, they include tools to manage and retrieve metadata, tools to integrate and cleanse data, and finally, business intelligence tools for performing analytical operations. Conceptually, data warehouses are used for the timely translation of enterprise data into information useful for analytical purposes. In doing so, they have to manage the flow of data from operational systems to decision support environments. The process of gathering, cleansing, transforming, and loading data from various operational systems that perform day-to-day transaction processing (hereafter, sources or source data stores) is assigned to the Extract-Transform-Load (ETL) processes.

ETL processes constitute the backbone of a DW architecture, and hence, their performance and quality are of significant importance for the accuracy, operability, and usability of data warehouses. ETL processes involve a large variety of activities (a.k.a. stages, transformations, operations) organized as a workflow. Typical activities are schema transformations (e.g., pivot, normalize), cleansing activities (e.g., deduplication, check for integrity constraint violations), filters (e.g., based on some regular expression), sorters, groupers, flow operations (e.g., router, merge), function application (e.g., built-in function, script written in a declarative programming language, call to an external library—hence, functions having 'black-box' semantics), text/data analytics and machine learning operations, and so on. Due to such a rich variety of operations and the typical structural complexity of ETL workflows, the ETL optimization problem goes beyond traditional query optimization and requires a fresher look, as discussed in the literature (e.g., [1,2,4,5,22]).

* Corresponding author.
E-mail addresses: tasos.karagiannis@outlook.com (A. Karagiannis), panos.vassiliadis@cs.uoi.gr (P. Vassiliadis), alkis@hp.com (A. Simitsis).
[1] Work performed while with University of Ioannina.

0306-4379/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2012.12.001

The ideal execution of ETL workflows suggests pipelining the flow of data all the way from the source to the target data stores. Typically, this cannot happen seamlessly due to the blocking nature of many ETL activities (e.g., sorters) and to the structural nature of the flow (e.g, activities with multiple input/output schemata). Appropriate scheduling strategies are required for orchestrating the smooth flow of data towards the target data stores. In this context, *workflow scheduling* during ETL execution involves the efficient prioritization of which activities are active at any time point, with the goal of minimizing execution time and/or memory consumption without any data losses.[2]

In available ETL engines – and in the corresponding industrial articles and reports, as well – the notion of scheduling refers to a higher design level than the one considered here, and specifically, to the functionality of managing and automating the execution of either the entire ETL workflow or fragments of it, according to the business needs, and based on time units manually specified by the designer. In this paper, we deal with the problem of scheduling ETL workflows at the data level and in particular, our first research contribution is an answer to the question: "what is the appropriate scheduling protocol and software architecture for an ETL engine in order to minimize the execution time and the allocated memory needed for a given ETL workflow?".

A commonly used technique for improving performance is parallelization, through either partitioning or pipeline parallelism. Typically, in the ETL context, the former refers to data partitioning. That is we split a dataset into *N* chunks and create *N* parallel subflows each working on a different chunk. Data partitioning works well with simple flow structures or the final data load to a target data store. However, when it comes to executing more complex ETL flows, data partitioning per se is not very effective without pipeline parallelism. Even with the best partitioning strategy, if the data is blocked at several points in the flow (e.g., in the very common cases of surrogate key generation, duplicate detection, pivoting, blocking joins or aggregations), we lose the advantage that partitioning parallelism may offer. Hence, it is important to group and execute operations in favor of pipeline parallelism too. Thus, here, a second important contribution is an answer to a rather neglected aspect in ETL execution, which is: "how to schedule flow execution at the operations level (blocking, non-parallelizable operations may exist in the flow) and how we can improve this with pipeline parallelization".

Although scheduling policies have been studied before in various research areas, in the context of ETL, the related work has only partially dealt with such problems so far. There are some first results in the areas of ETL optimization [1–3] and update scheduling in the context of near-real time warehousing [4,5,22]. The former efforts do not consider scheduling issues. The latter efforts are not concerned with the typical case of data warehouse refreshment in a batch, off-line mode; moreover, the aforementioned papers are concerned with the scheduling of antagonizing updates and queries at the data warehouse side (i.e., during loading) without a view to the whole process. We discuss related efforts in more detail in Section 2.

In this work, we experiment with scheduling algorithms specifically tailored for batch ETL processes and try to find an ETL configuration that improves two performance objectives: *execution time* and *memory requirements*. Based on our experience and publicly available documentation for ETL engines, state-of-the-art tools use two main techniques for scheduling ETL activities: the scheduling is relied on the operating system's default configuration or is realized in a round robin fashion, which decides the execution order of activities in FIFO order. Our implementations and experimentation showed that deadlocks are possible in the absence of low-level scheduling; hence, scheduling is necessary for an ETL engine. We demonstrate that three scheduling techniques, namely Minimum Cost Prediction, Minimum Memory Prediction, and Mixed Policy serve our goals better, as compared to a simple Round Robin scheduling. The first technique improves the execution time by favoring at each step the execution of the activity having more data to process at that given time. The second reduces the memory requirements by favoring activities with large input queues, thus, keeping data volumes in the system low. As our research shows, different workflow fragments may be executed best under different scheduling strategies. Therefore, we present an approach for splitting a workflow in parts that can be scheduled individually using a third scheduling strategy called Mixed Policy. Mixed Policy exploits parallelization opportunities and combines the benefits of Minimum Cost Prediction and Minimum Memory Prediction.

Moreover, we discuss how these results can be incorporated into an ETL engine and for that, we present our implementation of a generic and extensible software architecture of a scheduler module. We are using a realistic, multi-threading environment (which is not a simulation), where each node of the workflow is implemented as a thread. A generic monitor module orchestrates the execution of ETL activities based on a given policy and guarantees the correct execution of the workflow.

Finally, the evaluation of and experimentation with ETL workflows is not a trivial task, due to their large variety. An additional problem is that the research landscape is characterized by the absence of a commonly agreed framework for the experimentation with ETL flows. Hence, for evaluating our techniques against a realistic ETL environment, we have used a set of ETL patterns built upon a taxonomy for ETL workflows that classifies typical real-world ETL workflows in different template structures. By studying the behavior of these patterns on their own, we come up with interesting findings for their composition, and thus, for the scheduling of large-scale ETL processes.

[2] In a different context, stream processing frequently uses tuple shedding as a mechanism to improve execution time by reducing the amount of data being processed. However, in DWing all tuples should find their way into the target data stores, and thus, data losses are not typically allowed.

Contributions: Our main contributions are as follows.

- This paper is the first that deals with the problem of scheduling batch ETL processes, as we discuss in Section 2. We formally define the problem (Section 3) and discuss scheduling mechanisms that improve ETL execution in terms of execution time and memory requirements (Section 4).
- We present a novel approach for splitting an ETL workflow into fragments, each of which can run in parallel and exploit its own scheduling policy for better leveraging the effect of scheduling (Section 5).
- We propose a generic software architecture for incorporating a scheduler to an ETL engine and present its implementation as a proof of concept (Section 6).
- Finally, through a series of experiments, we demonstrate the benefits of the presented scheduling techniques. The findings are based on template ETL structures that constitute representative patterns of real-world ETL processes (Section 7).

2. Related work

Related problems studied in the past include the scheduling of concurrent updates and queries in real-time warehousing and the scheduling of operators in data streams management systems. However, since ETL workflows have some unique characteristics, we argue that a fresher look is needed in the context of ETL technology.

2.1. Scheduling for ETL

Related work in the area of ETL includes efforts towards the optimization of entire ETL workflows [1–3,6] and of individual operators, such as the DataMapper [7]. On the other hand, and despite the fact that ETL operations comprise a richer set than traditional relational operators, traditional query optimization techniques (e.g., as in [8,9]), although related in a broader sense (and indeed integrated in our work in terms of algebraic optimization; e.g., joins or sorts order) (a) are based on assumptions that do not necessarily hold for ETL workflows like left-deep plans (as opposed to arbitrary trees in ETL settings), and (b) are in general orthogonal to the problem of scheduling operators. For the same reasons, previous work on continuous queries (e.g., as in [10]) cannot be directly applied in our problem.

Thiele et al. [4] deal with workload management in real-time warehouses. The scheduler at the warehouse handles data modifications that arrive simultaneously with user queries, resulting in an antagonism for computational resources. Thomsen et al. discuss a middleware-level loader for near real time data warehouses [5]. The loader synchronizes data load with queries that require source data with a specific freshness guarantee. Luo et al. [11] deal with deadlocks in the continuous maintenance of materialized views. To avoid deadlocks, the paper proposes reordering of transactions that refresh join or aggregate-join views in the warehouse. Golab et al. [12] discuss the scheduling of the refreshment process for a real-time warehouse, based on average warehouse staleness i.e., the divergence of freshness of a warehouse relation with respect to its corresponding source relation.

Interestingly, despite the plethora of ETL tools in the market, there is no publicly available information for the scheduling of operators in their internals. Overall, related work for ETL scheduling has invested mostly on the loading part in the real-time context and specifically, on the antagonism between queries and updates in the warehouse. Here, we deal with workflows involving the entire ETL process all the way from the source to the warehouse and we consider the off-line, batch case.

2.2. Stream scheduling

The Aurora system can execute more than one continuous query for the same input stream(s) [13]. Every stream is modeled as a graph with operators (a.k.a. boxes). Scheduling each operator separately is not very efficient, so sequences of boxes are scheduled and executed as an atomic group. The Aurora stream manager has three techniques for scheduling operators in streams, each with the goal of minimizing one of the following criteria: execution time, latency time, and memory. The Chain scheduler reduces the required memory when executing a query in a data stream system [14]. This work focuses on the aspect of real-time resource allocation. The basic idea for this scheduler is to select an operator path which will have the greatest data consumption than the others. The scheduler favors the path that will remove the highest amount of data from the system's memory as soon as possible. Urhan and Franklin [15] present two scheduling algorithms that exploit pipelining in query execution. Both algorithms aim to improve the system's response time: the first by scheduling the stream with the biggest output rate and the second by favoring important data, especially, at joins.

Stream scheduling is very relevant to our problem since it involves the optimization of response time or memory consumption for flows of operations processing (consuming) tuples (possibly with a data shedding facility for voluminous streams). Still, compared to stream processing, ETL workflows present different challenges, since they involve complex processing (e.g., user-defined functions, blocking operations, cleansing operations, data and text analytics operations [16]) and they must respect a zero data loss constraint (as opposed to data shedding, which involves ignoring part of the data stream whenever the stream rate is higher than the ability of the stream management system to process the incoming data). Moreover, in the special case of off-line ETL, the goal is to minimize the overall execution time (and definitely meet a time-window constraint) instead of providing tuples as fast as possible to the end-users.

3. Problem formulation

Conceptually, an ETL workflow is divided into three generic phases. First, data are extracted from data sources (e.g., text files, database relations, XML files, unstructured documents, and so on). Then, appropriate transformation, cleaning, and/or integration *activities* are applied to the extracted data for making them free of errors and compliant

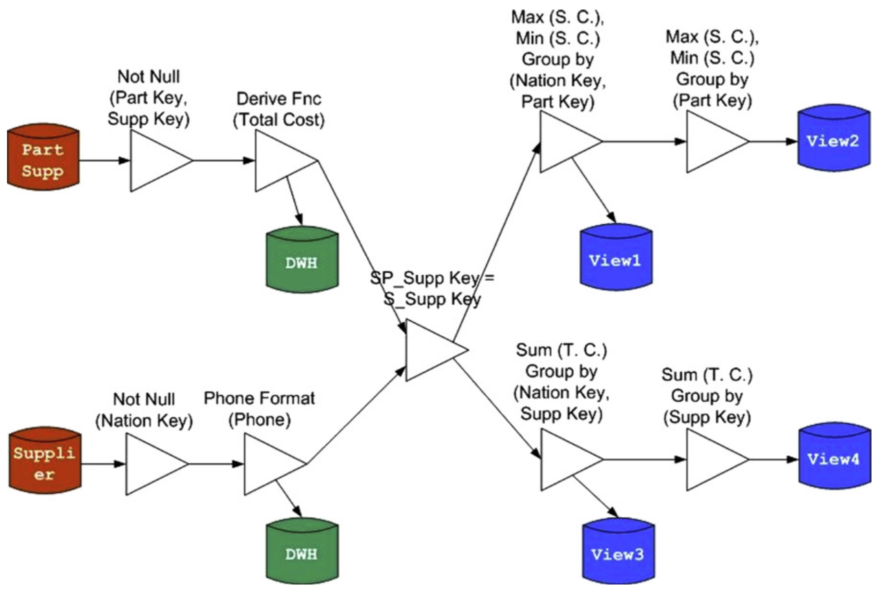
Part Supp
Not Null (Part Key, Supp Key)
Derive Fnc (Total Cost)
DWH
Max (S. C.), Min (S. C.) Group by (Nation Key, Part Key)
Max (S. C.), Min (S. C.) Group by (Part Key)
View2
View1
SP_Supp Key = S_Supp Key
Sum (T. C.) Group by (Nation Key, Supp Key)
Sum (T. C.) Group by (Supp Key)
View4
View3
Supplier
Not Null (Nation Key)
Phone Format (Phone)
DWH

Fig. 1. Example ETL workflow.

to target (e.g., data warehouse) schema. Other types of activities, like analytics operations, machine learning algorithms, and so on may exist as well. Finally, the data processed are loaded into the data warehouse. Note that different flavors of an information integration flow like ETL, ELT, ETLT, etc., fit seamlessly into our framework. Therefore, without loss of generality, in the rest of the paper, we use the term ETL in a broader sense that covers such structural variations.

The full layout of an ETL workflow, involving activities and recordsets may be modeled as a directed acyclic graph (DAG) [17]. ETL activities and recordsets (either relations or files) constitute the graph nodes. According to their placement into one of the three ETL phases, recordsets can be classified as source, intermediate (including any logging, staging or temporary data store), and target. The relationships among the nodes constitute the graph edges. Each node has input and output schemata, which can be empty as well; e.g., a source recordset has an empty input schema. The workflow is an abstract design at the logical level, which has to be implemented physically, i.e., to be mapped to a combination of executable programs/scripts that perform the ETL workflow. Typically, a logical-level activity is mapped to a concrete physical implementation, possibly, by picking one out of many candidates for this role; e.g., a logical join operation can be realized by a nested-loops, sort-merge, hash-join variation.

Example. Fig. 1 depicts an example ETL workflow starting with two relations *PartSupp* and *Supplier*. Assume that these relations stand for the differentials for the last night's changes over the respective source relations. The data are first cleansed and tuples containing null values in critical attributes are quarantined in both cases (*Not Null*). Then, two transformations take place, one concerning the derivation of a computed attribute (*Total Cost*) and another concerning the reformatting of textual attributes (*Phone Format*). The data are reconciled in terms of their schema and values with the data warehouse rules and stored in the warehouse fact and dimension tables. Still, the processing continues and the new tuples are joined in order to update several forms, reports, and so on, which we represent as materialized views $View_1, \ldots,$ and $View_4$ via successive aggregations.

Notation: We employ bold letters to refer to sets (e.g., of timepoints, nodes). We typically use v to refer to an arbitrary node of the workflow.

3.1. Workflow scheduling

First, we formally model the following problem: "Given a workflow deployed on a single server, what are the objectives and correctness guarantees for the scheduling of the workflow?"

Time: We consider $\mathbf{T}$ as an infinite countable set of timestamps. Time can be organized in time intervals. Hence, we divide $\mathbf{T}$ into disjoint and adjacent intervals $\mathbf{T} = \mathbf{T}_1 \cup \mathbf{T}_2 \cup \ldots$ with (see Fig. 2):

- $\mathbf{T}_i = [\mathbf{T}_i.first, \mathbf{T}_i.last]$,
- $\mathbf{T}_i.last = \mathbf{T}_{i+1}.first - 1$.

Graph: Formally, an ETL workflow comprises a directed acyclic graph $G(\mathbf{V},\mathbf{E})$, where $\mathbf{V} = \mathbf{V}_A \cup \mathbf{V}_R$. $\mathbf{V}_A$ denotes the activities of the graph and $\mathbf{V}_R$ the recordsets. The nodes of $\mathbf{V}$ can further be divided to two subsets with respect to their execution status: candidates (nodes that are still active and participate in the execution) and finished (nodes that have finished their processing), i.e., $\mathbf{V} = \mathbf{V}_{CAND} \cup \mathbf{V}_{FIN}$.

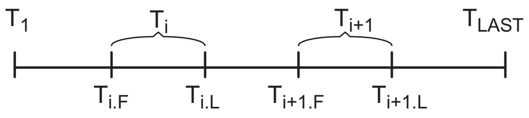

Fig. 2. Timestamps for scheduling.

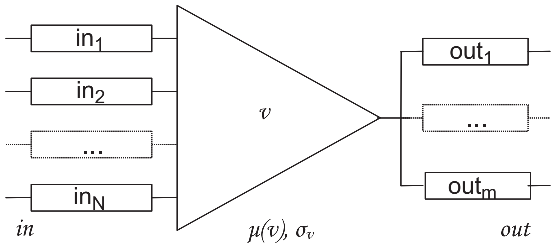

Fig. 3. ETL activity structure.

For a node v, we use the term *producers*(v) to denote all nodes u that populate node v with data (i.e., there is an edge (u,v) with v being the successor node. We will also use the term *consumers*(u) for the opposite edge direction.

Activities: A generic ETL activity is depicted in Fig. 3. For each activity node $v \in \mathbf{V}_A$ we define:

- $\mu(v)$, as the consumption rate of node v,
- σ_v, as the selectivity of node v,
- $\mathbf{Q}(v)$, as the set of all input queues of v,
- $queue_t^i$ as the size of the ith input queue of an activity v at the time point t,
- $queue_t(v)$ as the sum of all the input queue sizes of an activity v at t.

The consumption rate (measured in tuples per time unit) refers to the amount of memory gained per time unit due to the activation of a node of the workflow. Assuming that a node v is activated for a time interval $\mathbf{T}_v$, we measure *inputTuples* as the number of tuples within all queues of $\mathbf{Q}(v)$ and *outputTuples* the number of tuples in all the queues of the consumer nodes of v. The *consumption rate* $\mu(v)$ is the difference of these two measures divided by $\mathbf{T}_v$. The *selectivity* is the ratio *outputTuples* over *inputTuples* and shows how selective the node v is.

We define $size_t(q)$ as the memory size of a queue q at a time point t and $MaxMem(q)$, as the maximum memory size that the queue can obtain at any time point.

Recordsets: For each recordset node $v \in V_R$ we assume an activity responsible for reading or writing from node v and we define its consumption rate as the consumption rate of node v, $\mu(v)$. Furthermore, for each source recordset node we define $volume_t(v)$, as the size of the recordset at timepoint t.

Scheduler: The scheduler (equipped with a scheduling policy P) has to check which operator to activate and for how long. So, whenever a new interval $\mathbf{T}_i$ begins, (at timestamp $\mathbf{T}_i.first$) the scheduler has to decide on the following issues: (1) which is the next activity to run and (2) how long the time slot for this activity's execution will be. This is formally expressed as follows:

1. $v = active(\mathbf{T}_i)$. According to the scheduling policy P used, the scheduler has to choose the next activity to run: the function *active*() returns a node of the graph to be activated next. Note that in order to do this, every scheduling strategy implementing *active*( ) has to take into consideration the status of all queues, in order to avoid overloading queues that have reached their maximum capacity, thus resulting in data loss.
2. $\mathbf{T}_i.last$. This is the timestamp that determines when operator $active(T_i)$ will stop executing. (It also determines the scheduler time slot $\mathbf{T}_i.length()$.)

The node that has been selected and activated as a result of invoking $active(\mathbf{T}_i)$ will stop its processing at a time point t if one of the following occurs:

1. $t = \mathbf{T}_i.last$. In this case, the time slot is exhausted and it is time to reschedule the next node.
2. $queue_t(active(\mathbf{T}_i)) = 0$, for any time point t within $\mathbf{T}_i$. Then, the active operator has no more input data to process and so the scheduler must assign a new node to be activated.
3. There exists a node $u \in consumer(active(\mathbf{T}_i))$, such that for any of its queues, say q, $size_t(q) = MaxMem(q)$. In this case, one of the consumers of the active activity $active(T_i)$ has a full input queue and further populating it with more tuples will result in data loss.

At this point, we must check if the node v activated by $active(\mathbf{T}_i)$ should be moved to $\mathbf{V}_{FIN}$. In order for a node v to be moved to $\mathbf{V}_{FIN}$, either v is an empty source recordset, or both of the following conditions must be valid: (a) all the nodes feeding v with data have exhausted their input and (b) the queues of v have been emptied. Formally, this is expressed as follows:

$\mathbf{V}_{FIN} := \mathbf{V}_{FIN} \bigcup \{v\}$, $v = active(\mathbf{T}_i)$, if

1. $volume_t(v) = 0$, if v is a source recordset, or else
2. (a) $\forall u \in producers(v)$, it holds that $u \in \mathbf{V}_{FIN}$, and (b) $queue_{\mathbf{T}_i.last}(v) = 0$.

A workflow represented by a graph $G(\mathbf{V},\mathbf{E})$ ends when $\mathbf{V} = \mathbf{V}_{FIN}$. The interval during which this event takes place is denoted as $\mathbf{T}.last$.

Problem statement: Our goal is to decide a scheduling policy P for a workflow represented by a graph $G(\mathbf{V},\mathbf{E})$, such that:

- P creates a division of $\mathbf{T}$ into intervals $\mathbf{T}_1 \cup \mathbf{T}_2 \cup \cdots \mathbf{T}_{last}$.
- $\forall t \in \mathbf{T}$, $v \in \mathbf{V}$, $\forall q \in \mathbf{Q}(v)$, $size_t(q) \le MaxMem(q)$ (i.e., all data are properly processed without any data losses).
- One of the following objective functions is minimized:
  - $\mathbf{T}_{last}$ is minimized, where $\mathbf{T}_{last}$ is the interval where the execution of the workflow ends.
  - $\max \sum queue_t(v)$ is minimized, where $t \in \mathbf{T}$, and $v \in \mathbf{V}$.

The above problem formulation comes with different choices for the scheduler's strategy and specifically: (a) a strategy where the scheduler tries to minimize the execution time; or (b) a strategy where the scheduler tries to minimize the maximum memory consumption. We use this exclusive manner of presenting the choice of the scheduler based on the practical observation that the two goals are antagonistic as, typically, memory is the price to pay for speeding things up and vice versa; in practice, one can also think of the problem with a threshold in one of the two measures and the goal of minimizing the other. Of course, the list is not exhaustive. In fact, it is possible to devise combinations of goals in the scheduler's strategy (at the price of complicating and slowing down the scheduler's operation). However, an exhaustive exploration of all possible such goals is outside the scope of this paper.

3.2. Workflow fragmentation and stratification

Once the main problem of workflow scheduling has been dealt with, we can explore an extra alternative. The goal of the workflow fragmentation problem is to divide a large workflow (represented as graph $G(\mathbf{V},\mathbf{E})$) into appropriately connected subflows, such that each of these components recursively deals with the workflow scheduling problem. So, we can increase parallelism by dividing the larger problem of scheduling a large workflow into many smaller problems of activating many independent subflows simultaneously.

3.2.1. Workflow fragmentation

The problem of workflow fragmentation tries to fragment a large workflow into components, or subflows. Every subflow has the property that it allows the pipelining of intermediate results between its activities. As a side effect, we produce a zoomed-out version of the graph, with subflows as the nodes of the zoomed-out graph.

Problem statement: Given a graph $G(\mathbf{V},\mathbf{E})$ representing an ETL workflow, a fragmentation of the graph, $\mathcal{F}(G)$, is a pair comprising (a) a set of disjoint subflows $\mathbf{SF}=\{F_1,\ldots,F_k\}$, and, (b) a set of subflow connecting edges $\mathbf{E}_F$ such that the following hold:

- Each subflow F_i is a connected subgraph of G, $F_i(\mathbf{V}_i,\mathbf{E}_i)\subseteq G$, such that (a) all edges of $\mathbf{E}_i$ allow pipelining, (b) there is a single fountain and a single sink node of the subflow, F_i*.fountain* and F_i*.sink*.
- The edges of $\mathbf{E}_F$ are produced as $\mathbf{E}\text{-}\bigcup_i(\mathbf{E}_i)$ and we require that if an edge $e\in\mathbf{E}_F$, then e is of the form $e(F_i.sink,F_j.fountain)$, $i\neq j$.
- There is a full coverage of G, specifically $\bigcup_i(\mathbf{V}_i)=\mathbf{V}$ and $\bigcup_i(\mathbf{E}_i)\bigcup\mathbf{E}_F=\mathbf{E}$, with the extra constraint that subflows are mutually disjoint.

In practice, subflows are simple lines, within which the activities may pipeline intermediate results, without the need to store them at persistent storage. Note that the above definition allows blocking operations that require the existence of the entire input to be available (typically stored in a hard disk) to be a subflow of their own, comprising a single node, without edges. This is also why we do not require that the fountain and sink of the subflow are distinct nodes. Since subflows do not share edges, these blocking operators are the boundaries between the other subflows.

Subflow graph: We can produce a zoomed out version of the graph with subflows replacing its nodes. Specifically, given a graph $G(\mathbf{V},\mathbf{E})$ representing an ETL workflow, as well as a fragmentation of the graph G into a set of disjoint subflows $\mathbf{SF}=\{F_1,\ldots,F_k\}$ and a set of subflow connecting edges $\mathbf{E}_F$, we can produce the zoomed out version of the graph, or *subflow graph*, $G_S(\mathbf{V}_S,\mathbf{E}_S)$ by (a) replacing the nodes of $\mathbf{V}$ with the subflow to which they belong and (b) replacing each edge $e(F_i.sink,F_j.fountain)$ of $\mathbf{E}_F$, with an edge $e_S(F_i,F_j)$.

3.2.2. Workflow stratification

Once a subflow graph $G_S(\mathbf{V}_S,\mathbf{E}_S)$ has been obtained, we need to detect mutually independent subflows: mutually independent subflows can execute simultaneously (each with its own scheduling policy). We can recursively obtain a *stratification* of the subflow graph, i.e., an assignment of subflow nodes to subsequent layers of execution (or *strata*), as follows:

- Stratum S_0 comprises the fountains of the subflow graph, i.e., the sources of the data warehouse.
- Stratum S_{i+1} comprises the nodes of $\mathbf{V}_S$ that fulfill the following:
  - they have at least one incoming edge from S_i,
  - they may have incoming edges from any stratum S_j as long as $j<i+1$,
  - they have no other incoming edges.

Practically, a modified topological sorting algorithm can obtain such a stratification (cf. Section 5). We call the flows of a stratum a *Stratified Subflow Independent Set*, as they are mutually independent (i.e., there is no data path, neither direct, nor transitive, between any of them). As already mentioned, once the strata of the subflow graph have been identified, we can activate each stratum in its own turn. Subflows within the same stratum can execute independently and with different scheduling policies if necessary.

4. Scheduling algorithms for ETL

Related work on scheduling suggests four generic categories of scheduling algorithms based on the goal they try to achieve: (a) token-based algorithms (e.g., round robin) used mostly as a baseline for evaluating more sophisticated algorithms, and then algorithms that opt for improving (b) the total execution time (when the last tuple lands on the target data stores), (c) the response time (how fast the first tuple arrives at the target), and (d) the required memory during the execution. Since in our context the response time is an issue of secondary importance (see Section 2 and our differences with streams and real-time processing), we investigate scheduling policies belonging to the other three categories. First, we explore three generic algorithms: Round Robin, Minimum Cost Prediction, and Minimum Memory Prediction belonging to one of the aforementioned categories.

Table 1
Decision criteria for scheduling algorithms.

	Pick next	Reschedule when
RR	Operator id	Input queue is exhausted
MC	Max size of input queue	Input queue is exhausted
MM	Max consumption rate	Time slot

Table 1 shows the different criteria of the three algorithms concerning the decision on (a) which activity is favored each time the scheduler is called, and, (b) for how long the selected activity will continue to operate until the scheduler makes a new decision.

4.1. Round robin

The Round Robin (RR) scheduling algorithm handles all activities without any particular priority. It assigns time slices to each activity in equal portions and in an order based on a unique identifier that every activity has. Assuming a list V_{CAND} containing activities to be scheduled, at each iteration the algorithm picks the first activity from V_{CAND}. Its main advantages are as follows: every activity gets the same chances to run (fairness) and the system always avoids starvation.

Algorithm Round Robin.

```
In: A list $\mathbf{V}_{CAND}$ containing activities
Out: The next activity RR_next
1  begin
2      return $\mathbf{V}_{CAND}.pop$;
3  end
```

4.2. Improving cost

The Minimum Cost Prediction (MC) scheduling algorithm opts for reducing the execution time of ETL workflows. Therefore, the overhead imposed by the scheduler (e.g., the communications among the activities) is minimized. The selected activity should have data ready for processing, and typically, this activity is the one having the largest volume of input data. Since there are no time slots, the selected activity processes all data without any interruption from the scheduler. For the sake of simplicity, in our implementation we have considered that all activities that read data from an external source are always available for execution.

Algorithm Minimum Cost Prediction.

```
In: A list $\mathbf{V}_{CAND}$ containing activities
Out: The next activity MC_next
1  begin
2      $MaxInput = -1$;
3      for $v \in \mathbf{V}_{CAND}$ do
4          if ($MaxInput < v_Q$) then
5              MC_next = $v$;
6              $MaxInput = v_Q$;
7
       return MC_next;
8  end
```

4.3. Improving memory

The Minimum Memory Prediction (MM) algorithm schedules the flow activities in a way that improves the system memory required during the workflow execution. In each step, MM selects the activity that will consume the biggest amount of data. We can compute the consumption rate directly, considering the number of tuples consumed (input data–output data) divided by the processing time of the input data. This fraction shows the memory gain rate throughout the execution of activity so far. Given a specific time interval (which is the same for all candidates), multiplying this fraction by the input size of the candidates returns a prediction for the one that will reduce the memory most in absolute number of tuples. Thus, the overall memory benefit is

$$MemB(p) = ((In(p) - Out(p))/ExecTime(p)) \times Queue(p)$$

where $In(p)$ and $Out(p)$ denote the number of input and output tuples for activity p, $ExecTime(p)$ is the time that p needs for processing $In(p)$ tuples, and, $Queue(p)$ is the number of tuples in p's input queues. MM selects the activity with the biggest $MemB()$ value at every scheduling step.

In practice, the amount of data that an activity consumes is the data that the activity removes from memory, either by rejecting tuples or writing them into a file, for a specific portion of time. Small selectivity, large processing rate, and input size help an activity to better exploit this scheduling. Small selectivity helps an activity to consume large portion of its input tuples. Large input size helps an activity to process and possibly, reduce the in-memory data. Finally, large processing rate expedites data consumption.

Note that when the workflow execution starts no activity has processed any data, so the above formula cannot be applied. In this case, resembling Minimum Cost Prediction, the activity with the biggest input size is selected.

Algorithm Minimum Memory Prediction.

```
In: A list $\mathbf{V}_{CAND}$ containing activities
Out: The next activity MM_next
1  begin
2      $MaxInput = -1$;
3      $MMem = -\infty$;
4      for $v \in \mathbf{V}_{CAND}$ do
5          if ($MMem < v_{mem}$) then
6              MM_next = $v$;
7              $MMem = v_{mem}$;
8          if ($MaxInput < v_Q$) then
9              MC_next = $v$;
10             $MaxInput = v_Q$;
11
12     if ($MMem \leq 0$) then
13         MM_next = MC_next;
       return MM_next;
14 end
```

4.4. Mixed policy

Our experiments have indicated that on average, MC performs better than RR, in terms of execution time, while MM is slower than the other two (Fig. 4). However,

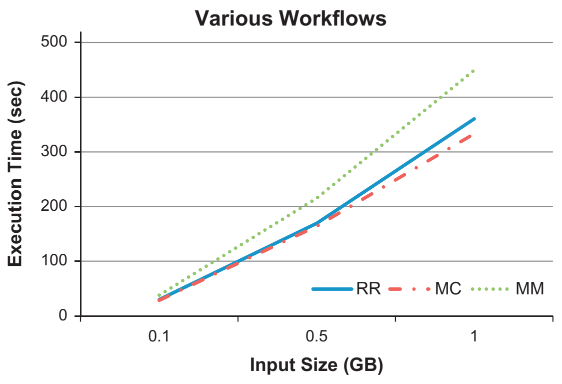

Fig. 4. Avg execution time for various workflows and data sizes.

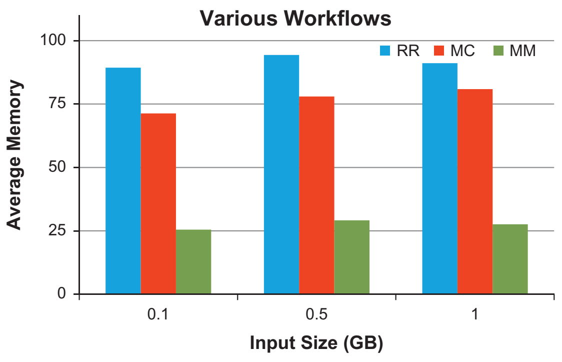

Fig. 5. Avg memory consumption (# row packs) for various workflows and data sizes.

Fig. 5 shows that MM is clearly superior to the other too in terms of memory gain. This observation motivates us to explore the opportunity of exploiting the memory gains of MM for parallelizing the workflow execution (or of parts of it). Hence, this creates the challenge of finding subflows of the original workflow that can be executed using different scheduling policies so that (a) simple subflows can use a faster policy as the MC and (b) more complicated subflows that involve memory consuming tasks and blocking operators can use the MM policy for gaining in memory, and thus, for freeing some resources. Then, the memory gained can be reused for boosting the faster workflows with parallelization. We discuss this challenge in detail in the next section.

5. Segmentation of a workflow to subflows

This section describes the steps required for using Mixed Policy (MP). The motivation for segmenting a workflow to fragments (or, *subflows*) and deploying each of them to a single CPU lies on the observation that we can exploit memory and pipelining to speed up the whole process, by dividing the workflow to fragments that can be executed in parallel and with different scheduling policies. Assuming that several sources populate a warehouse, the respective ETL workflow can be split into fragments that have no dependencies with each other—practically corresponding to different sources. The same applies to the part of the workflow that populates different materialized views, which can be populated in parallel from different threads. Overall, the principles that guide the segmentation of a workflow to subflows that can be executed in parallel are:

- Activities that feed each other in a pipelined manner should be placed in the same subflow to increase the benefit of pipeline parallelism.
- Subflows with no dependencies to each other may be executed in parallel.
- Blocking activities (i.e., activities that produce output only after they have consumed their entire input) split the overall workflow in two parts that have to be synchronized: unless the first part is completed, there is no use of allocating resources to the subsequent part.

Hence, there is an opportunity of splitting an ETL workflow into groups of subflows. Each such a group should contain subflows that can be executed in parallel. Modern ETL tools offer the functionality for parallelizing the workflow. However, to the best of our knowledge, none of them automates the procedure of identifying the parts of the workflow that can be executed in parallel; this task is performed manually by the designer according to her experience [6,16].

5.1. Subflow creation

An ETL workflow may be seen as a directed acyclic graph $G(\mathbf{V},\mathbf{E})$, where its nodes are activities or recordsets and the edges show the data flow among the nodes (see Section 3). We extend G by adding two types of nodes: a *reader* for each edge feeding an activity from a recordset and a *writer* for each edge staging data to a recordset.

A *subflow* is a connected component of a subset of the original graph, which allows data to flow from one activity to another without being (intermediately) saved at persistent storage from its starting nodes towards its ending node (*data pipelining*). A node v is a *boundary* of two subflows if it belongs to one of the following categories:

- v is a recordset. Recordsets are staging points and, moreover, they are not to be scheduled.
- v is a blocking activity (for the reasons we discussed earlier in this section).
- v is a router activity. Since many flows can stem out of a router, we may split them a priori into different subflows and reserve the right to merge the anteceding subflow with one of the subsequent subflows later.
- v is a binary activity with two blocking inputs. Remember that typically binary activities are join variants (e.g., join, difference, sorted union). A binary activity may be a boundary in the case of two pipelines that need to be staged and sorted before joined or two recordsets as inputs. On the other hand, the case of one activity and one recordset as inputs is not necessarily blocking, if the activity is a 'nested-loops' variant.

Two nodes are *mergeable* and can be merged in the same subflow if they are adjacent in the graph and none of them is a boundary. Boundaries form subflows of their own. Assuming we remove boundaries from the graph, the components of the graph that remain produce the subflows of the overall workflow.

Algorithm BreakIntoSubflows (BIS) produces the subflows of a workflow. First, it puts each node into a subflow. Then, it progressively merges subflows whenever they contain nodes that can be merged. Two subflows s_1 and s_2 can be merged, if they contain two activities a_1 and a_2 that can be merged. In this case, it is possible to pipeline data from one to another. Therefore, the two subflows can be merged. The following property is easy to prove.

Algorithm BreakIntoSubflows (BIS).

```
In: an ETL workflow $G(\mathbf{V},\mathbf{E})$
Out: a set of subflows $S_F$
1   begin
2     for all $v \in \mathbf{V}$ do
3       create a new subflow $s(v)$;
4       $S_F = S_F \cup s(v)$;
5     while no merge happens do
6       for all edges $(v,u \in \mathbf{E}$, s.t. $(v,u) \notin$ visited do
7         if mergeable$(v,u)$ then
8           merge $s(v)$ and $s(u)$ into $s(v)$;
9           remove $s(u)$ from $S_F$;
10          add $(v,u)$ in visited;
11  end
```

Property. *If the numbering of the nodes in the adjacency matrix corresponds to their topological sorting, then the algorithm BreakInSubFlows runs in linear time.*

5.2. Subflow parallelization

Once a graph is split in subflows, the next step is to decide which of the subflows can be deployed to different CPUs at the same time.

We use the term *stratified independent subflow set* (SISS) being close to the traditional graph theory terminology. In graph theory, an independent set is a set of nodes that are mutually non-adjacent. Our SISS differs from the independent sets of graph theory in two ways: (a) we use subflows instead of nodes, and, (b) non-adjacency is a weak constraint; in our setting, there is no path between the members of an SISS. The latter requirement explains the term "stratified" in the name: we practically split the workflow in strata of subflows that are mutually independent. Hence, in our context, an SISS is defined as a set of subflows that can be executed in parallel over different CPUs. SISS's can be derived via the following method.

Each subflow is characterized by its last node (which is typically a blocking or a routing activity or a persistent recordset). We can construct a zoomed-out graph of the ETL workflow comprising the subflows as its nodes. For each edge connecting nodes of two subflows in the graph of the original workflow, we add an edge between the two subflows in the zoomed-out graph.

Two subflows s_a and s_b belong in the same SISS if there is no path from s_a to s_b. Assuming each SISS has a unique integer as *id*, for finding SISS's we work as follows:

- Sources belong to SISS 0.
- Every node belong to the SISS with id equal to the highest of id of all its ancestors, augmented by 1: SISS$(v) = 1 + \max(\text{SISS}(u))$, $\forall u$ s.t. $\exists$ an edge (u,v) in the zoomed-out graph.

Formally, this task is performed by a modified topological sort algorithm that annotates each node with the SISS id of the node responsible for pushing it in the stack. Algorithm DeriveSISSets (DSISS) performs this task.

Algorithm DeriveSISSets (DSISS).

```
In: a graph $G_s(\mathbf{V}_s,\mathbf{E}_s)$ with subflows being the nodes of the graph
Out: a set of SISS's $\mathbf{W} = W_1,\ldots,W_k$
Variables: a counter current counting which SISS is currently
produced, and an array s characterizing the SISS of each node v
1   begin
2     $current = 0$; $s(v) = -1$, $\forall v \in V_s$;
3     while $\exists v \in V_s$ s.t. $s(v) = -1$ do
4       for all $v \in \mathbf{V}_s$, s.t., $s(v) < 0$ do
5         if in-degree$(v) == 0$ then
6           $s(v) = current$;
7           remove all edges from $v$ to other nodes in $\mathbf{V}_s$;
8           add $v$ to $W_{current}$ and remove it from $\mathbf{V}_s$;
9         current++;
10  end
```

5.3. Subflow execution

Working as in BIS and DSISS, we determine a set of SISS **W** for a graph G. Next, we execute the subflows s_i^j within an SISS W_j based on the following observations:

- Subflows constitute local units of scheduling where pipelining takes place and time is the scheduling criterion.
- When subflows are independent to each other, i.e., within the same SISS, they can exploit parallelization. If there are more CPUs than subflows within an SISS, they can all be used for executing the subflows in parallel (assuming memory is readily available).
- SISS's provide boundaries for ETL workflows where the scheduling policy can change. Specifically, lightweight subflows can be scheduled with a policy that favors fast delivery of tuples (like MC), whereas areas with memory-intensive blocking or binary operators can utilize scheduling policies that reduce the antagonism for memory (like MM).

Algorithm ExecuteSISS (EIS) describes how we run subflows within SISS's. SISS's are explored according to a topological sort. The subflows contained in a given SISS W_j may run in parallel; each subflow runs as a single thread. In our implementation, if the number of subflows is smaller or equal to the number of available CPUs, then each subflow thread is assigned to a different CPU (often, the OS does this by itself; we supervise the process to ensure that this is the case and fix it if needed). Otherwise, we let OS take over, since we observed that typically, the OS may handle adequately the communication cost and design complexity in such a case.

Within a subflow s_i^j, the activities are executed according to a scheduling policy P. P is determined based on the nature of activities in s_i^j. If s_i^j contains memory-intensive activities (e.g., blocking operations) then as a heuristic, we favor using the MM policy to enable better buffer management. Otherwise, we boost pipelining using Minimum Cost Prediction. In general, we tune the choice on the scheduling policy with a threshold parameter θ showing the number of memory-intensive activities that should be contained in a subflow before we choose to use the MM policy.

Algorithm ExecuteSISS (EIS).

```
In: a set of SISS's $\mathbf{W}$
Out: a plan for running subflows $S$ using a scheduling policy $P$
1 begin
2   for all $W_j \in \mathbf{W}$ do
3     for all $s_i^j \in W_j$ do
4       if $card(u) > \theta$, where $u \in s_i^j$
        and $u$ is a memory − intensive operation then
5         $P = $ MM;
6       else
7         $P = $ MC;
8       run $s_i^j$ as a thread using $P$;
9 end
```

In practice, as we discuss in Section 7 too, we consider a set of pattern ETL workflows for which we have ran extensive micro-benchmarks for understanding their behavior under different settings (e.g., various data sizes, selectivity). We have identified what policy works best for each pattern in a given setting. Therefore, we leverage these observations and use them as heuristics in the choice of a scheduling policy. We further discuss these issues in our experimental findings.

6. Software architecture

We have implemented a scheduler and a generic ETL engine in a multi-threaded fashion. One reason for our choice is that, in general, commercial ETL tools are not amenable to modification of their scheduling policy. Our software architecture is generic enough to be maintained and extended. Each workflow node is a unit that performs a portion of processing; even if that is simply reading or writing data. Hence, we consider every node (either activity or recordset) as an *execution item* or *operator* and represent it as a single thread. (Representing a recordset as a thread means that a dedicated thread is responsible for reading from or writing data to the recordset.) A messaging system facilitates communication among threads.

Function Execute().

```
1 begin
2   while (execution item not finished) do
3     check inbox for scheduler messages;
4     if (stalled) then
5       thread sleep;
6     else DataProcess();
7 end
```

All intermediate data processed by the various operators are stored in *data queues* and thus, we enable pipelined execution. Processing every tuple separately is not efficient (see also [13]), so data queues contain and exchange blocks of tuples, called *row packs*. Each operator (a) has a mailbox for supporting the messaging system, (b) knows the mailbox of its producers and consumers, and (c) knows the monitor's mailbox. The *monitor* is a system component that supervises and directs workflow execution.

Fig. 6 shows the class diagram of our system architecture. Next, we elaborate on two core system components, namely the *execution item* and *monitor*.

6.1. Execution item

When flow execution starts, a function called *Execute()* is called for each operator. An operator's execution completes when the respective *Execute()* terminates. For a short period, an operator may not have data to process. Then, for performance reasons, we stall that operator for a small time fragment (every thread sleeps for a while).

Execute() implements a loop in which (a) the respective operator checks its inbox regularly for messages either from the monitor or some other operator and (b) decides whether to process some data or to stall for a small time fragment. Each operator has two flags: *status* indicates whether it must process data or not and *finished* indicates whether the operator should terminate *Execute()*. The *DataProcess()* function is implemented independently of the *ExecutionItem*; therefore, in our extensible and customizable scheduling and data processing framework, each operator implements its own data processing algorithm.

An operator's inbox receives messages from the monitor with directives on when the current execution round completes (and hence, another operator should be activated). For relating an operator to such notifications, *DataProcess()* processes 'small' data volumes, which are small enough, so that their processing has been completed before the designated deadline arrives. In addition, *DataProcess()* respects the constraint that whenever the output queue is full, the operator must be stalled; hence, it does not allow data loss.

The *Execution Item* class is extended to the *Execution Recordset* and *Execution Activity* abstract classes, and can be appropriately specialized depending on the functionality of the recordset or activity represented by this execution item. For an *Execution Activity*, *DataProcess()* (a) reads from its data queues, (b) processes the tuples,

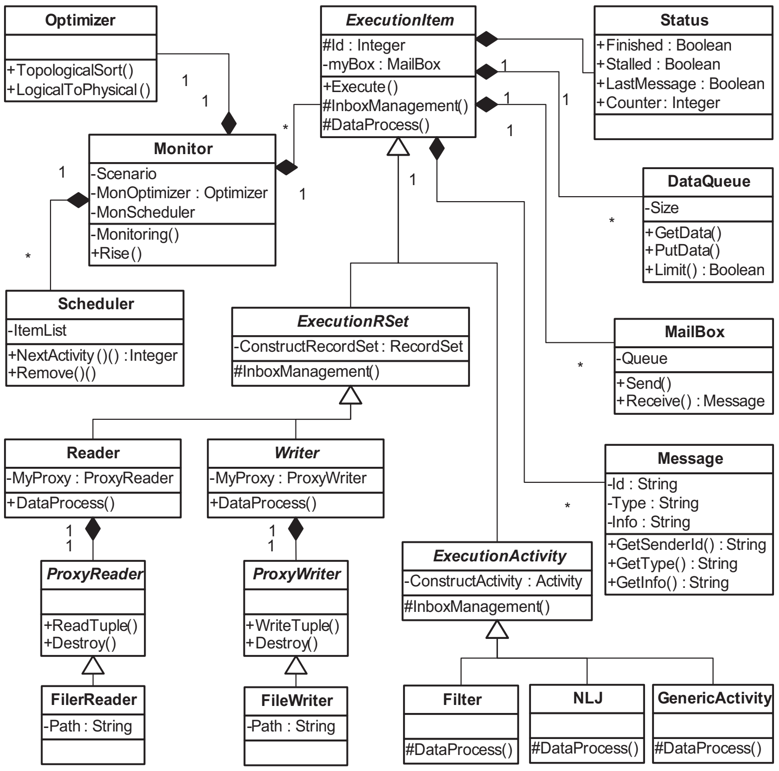

Fig. 6. Software architecture of the scheduler.

and then, (c) forwards them to its producers. As an example, we present an abstract implementation of *DataProcess()* for a *Filter*. The operator checks the status of the consumer's queue, and if it is full the data processing temporarily stops. For more complex activities, the logic of *DataProcess()* remains the same, although its implementation is more complicated. In all cases, *DataProcess()* processes small batches of input data, so that the operator can check its inbox frequently. At the same time, *Execution Recordsets* are instantiated as *Readers* or *Writers*, each of which implements *DataProcess()* and operates as a dedicated thread to read from (or write to) the appropriate text files.

Every *Execution Item* has a *Status* that keeps track of the status of an operator, which can be one of the following:
(a) *Stalled* allows the operator to call the *DataProcess()*, (b) *LastMessage* indicates whether the operator will receive or not more messages from its producers, and, (c) *Finished* indicates whether the operator's execution is complete.

6.2. Monitor

The *Monitor* is responsible for the correct initialization and execution of the workflow. It initiates a thread for every operator by calling the *Execute()* function and starts monitoring the entire process. *Monitor* uses a *Scheduler* to select the next thread to activate. The interface of *Scheduler* is as follows: (a) on creation it creates a list with all threads, (b) a *NextActivity()* function returns the id of the selected thread, and, (c) a *Remove(Id)* function removes a thread from the list whenever this thread has finished its operation. When the monitor needs to activate and execute a thread, it uses *NextActivity()* for selecting the best operator according to the scheduling policy enforced.

Table 2
Example message types.

Message type	Receiver's reaction
MsgEndOf Data	Receiver knows that its producer has finished producing data.
MsgTerminate	Receiver terminates even if its processing is not complete. If sent to the monitor, it signifies that the sender has terminated.
MsgResume	Receiver resumes the data processing by switching the flag *Stalled* to false.
MsgStall	Receiver temporarily stops processing data by switching the flag *Stalled* to true.
MsgDummy-Resume	Used to force all operators to execute *DataProcess()* once. This is used only when the scheduler cannot select the next thread and it gives the chance to operators to update some flags used internally.

Function DataProcess() for *Reader*.

```
begin
    for all tuples t in current pack do
        read tuple t;
        if t is NULL then
            status = finished;
        else
            status = forwardToConsumers(t);
            if status = false then stall thread;
end
```

Function DataProcess() for a *Filter*.

```
begin
    if no pack then
        if last message then status = finished;
        else stall thread;
    else
        while exists next tuple in input do
            if can process current tuple then
                status = status & forwardToConsumers(t);
        if status = false then stall thread;
end
```

Once initialized, the monitoring process is a loop in which the monitor thread checks its mailbox and gathers statistics for the required memory during flow execution. The monitor checks whether an operator has stalled or finished its execution and acts accordingly. Each operator has a mailbox and knows also the mailbox of the monitor and of its neighbors. All these objects communicate by sending messages. Table 2 lists the most important of them.

7. Experiments

In this section, we report on the experimental assessment of our algorithms. We start with presenting a principled set of experimental configurations for ETL workflows, which we call butterflies due to their structure. Then, we compare the various algorithms for their performance with respect to memory consumption and efficiency. Finally, we demonstrate that a mixed scheduling policy provides improved benefits.

7.1. Archetype ETL patterns

We have experimented with a set of ETL workflows described in a benchmark comprising characteristic cases of ETL workflows [18]. The main design artifact used for the workflow construction is the *butterfly*, which is an archetype ETL workflow composed of three parts: (a) the *left wing*, which deals with the combination, cleaning and transformation of source data; (b) the *body* of the butterfly, which involves the main points of storage of these data in the warehouse; and (c) the *right wing*, which involves the maintenance of data marts, reports, etc., after the fact table has been refreshed—all are abstracted as materialized views that have to be maintained. A butterfly workflow can be recursively decomposed to components that have an archetype structure themselves (e.g., surrogate key assignment,

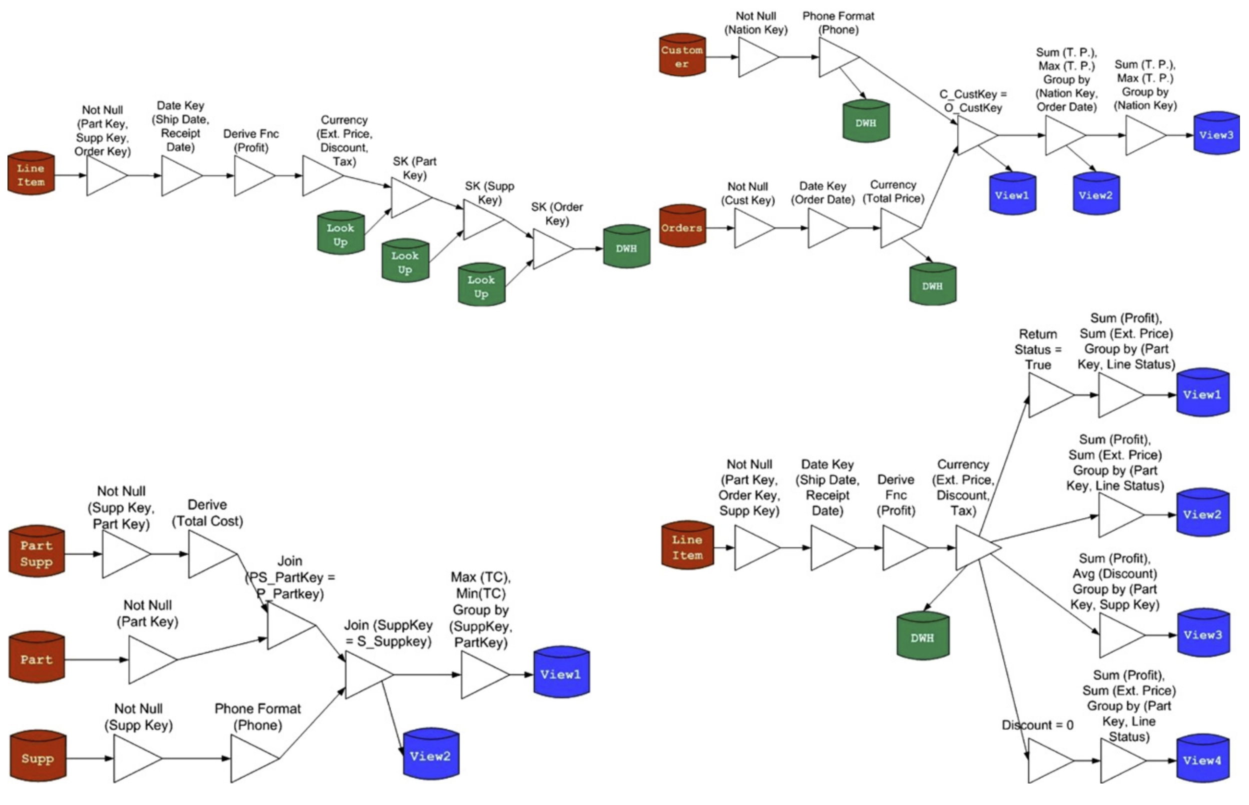
Line Item
Not Null (Part Key, Supp Key, Order Key)
Date Key (Ship Date, Receipt Date)
Derive Fnc (Profit)
Currency (Ext. Price, Discount, Tax)
SK (Part Key)
SK (Supp Key)
SK (Order Key)
Look Up
Look Up
Look Up
DWH
Custom er
Not Null (Nation Key)
Phone Format (Phone)
DWH
C_CustKey = O_CustKey
Sum (T. P.), Max (T. P.) Group by (Nation Key, Order Date)
Sum (T. P.), Max (T. P.) Group by (Nation Key)
View3
View1
View2
Orders
Not Null (Cust Key)
Date Key (Order Date)
Currency (Total Price)
DWH
Line Item
Not Null (Part Key, Order Key, Supp Key)
Date Key (Ship Date, Receipt Date)
Derive Fnc (Profit)
Currency (Ext. Price, Discount, Tax)
DWH
Return Status = True
Sum (Profit), Sum (Ext. Price) Group by (Part Key, Line Status)
View1
Sum (Profit), Sum (Ext. Price) Group by (Part Key, Line Status)
View2
Sum (Profit), Avg (Discount) Group by (Part Key, Supp Key)
View3
Discount = 0
Sum (Profit), Sum (Ext. Price) Group by (Part Key, Line Status)
View4
Part Supp
Not Null (Supp Key, Part Key)
Derive (Total Cost)
Join (PS_PartKey = P_Partkey)
Part
Not Null (Part Key)
Join (SuppKey = S_Suppkey)
Max (TC), Min(TC) Group by (SuppKey, PartKey)
View1
View2
Supp
Not Null (Supp Key)
Phone Format (Phone)

Fig. 7. Selected workflows used in our experiments: (clockwise from top left) primary flow, wishbone, fork, and tree.

slowly changing dimensions). More details can be found in [18]. Fig. 7 illustrates some example workflow archetype patterns.

The *line* workflow has the simplest form of all since it linearly combines a set of filters, transformations, and aggregations over the data of a single table on their way to a singe warehouse target. A *wishbone* workflow joins two parallel lines into one and refers, for example, (a) to the case when data from two lines, stemming from the sources should be combined in order to be loaded to the data warehouse, or, (b) to the case where we perform similar operations to different data that are later "joined" (possibly via a sorted union operation). The *primary flow* is a common archetype workflow in cases where the source table must be enriched with several surrogate keys; therefore, source data pass via a sequence of surrogate key assignment activities which use lookup tables to replace production keys with surrogate keys. The *tree* workflow joins several source tables and applies aggregations to the result recordset. The joins are performed over either heterogeneous relations, whose contents are combined, or homogeneous relations, whose contents are integrated into one unified (possible sorted) data set. The *fork* workflow is an archetype heavy on the right wing and is used to apply a large set of different aggregations over the data arriving to a warehouse table.

Having such archetypes in hand, one may compose them for producing large-scale ETL workflows according to the desired characteristics. For example, a number of primary flows can be combined with a number of trees for producing an ETL workflow having a really heavy load among the sources and the data warehouse. Clearly, such a workflow offers opportunities for parallelization.

7.2. Experimental setting

The experimental assessment of the constructed scheduling and the proposed scheduling policies aims at the evaluation of two metrics: (a) *execution time* that measures the necessary time for the completion of each workflow and (b) *memory consumption* that measures the memory requirements of every scheduling policy during execution.

In terms of time, we focus on the execution time and not on response time, which is typically targeted by streaming systems, because, as discussed, it is quite irrelevant to traditional ETL workflows (see also Section 2). In addition, in the ETL context we do not normally have the luxury of data shedding. In terms of memory, we are interested in both, average and maximum memory requirements. The assessment of memory requirements has been performed as follows: in regular time intervals, we get a snapshot of the system, keeping information for the size of all queues. We keep the maximum value and a sum, which eventually gives the average memory per experiment.

Important parameters that affect the performance of the alternative scheduling policies are: (a) the *size* and *complexity* of a workflow, (b) the *size of data* processed by a workflow, and (c) the *selectivity* of a workflow (or, in other words, the degree of cleansing performed due to the 'dirtiness'of source data). Workflow complexity is determined via the variety of butterfly workflows used. Our test data have been generated with the TPC-H [19] generator, and all scale factors refer to TPC-H numbers too. With respect to the data sizes used, we have experimented with various data sizes and we report here some indicative results. Although the overall data volumes processed by ETL processes are quite large, usually, these are not processed in a single batch. In fact, when freshness is desirable, smaller batches are processed each time – and more frequently – for optimizing parts of the process like the load. In this sense, due to space consideration, we report here on results regarding scale factors up to 1 GB, which is a fairly realistic size for micro-batch ETL processing.

Workflow execution requires the fine-tuning of both the engine and scheduler. Hence, we need to tune: (a) the *stall time*, i.e., the duration for which a thread will remain stalled; (b) the *time slot* (*TmSl*) given each time to an activated operator; (c) the *data queue size* (*DQS*), which gives the maximum size of the system's data queues; and (d) the *row pack size* (*RPS*), i.e., the size (number of tuples) of every row pack.

Stall time is used as parameter for the system command *Thread.Sleep(Engine StallTime)*. This parameter should be kept small enough, as large values lead the system to an idle state for some time. (Large values make operators idle for a long period of time and also, make them read their messages long after they are sent.) Other techniques can be used for stalling threads, as well. However, since each activity runs as a different thread and each queue is connected with a single provider, there is no concurrent access to write in a queue. Thus, after executing extensive micro-benchmarks on stall time, and on the aforementioned parameters too, we tuned the sleeping period in a reasonably small value of 4 ms and used that value for all experiments.

Tuning the *time slot* depends on the policy tested. Using time slots in the RR and MC scheduling policies would lead to more communication and scheduling overhead and finally to a longer execution time. In MC, consider for example an operator *p* that needs 150 ms to empty its data queue. If the time slot is 50 ms, the scheduler will interrupt *p* two times before its queue is empty. These two interrupts are unnecessary and add additional cost to the execution. Since our concern is to minimize execution time, we avoid such unnecessary scheduling interrupts by not using time slots.

For all remaining parameters requiring tuning, we have experimented with different values for various ETL archetypes and found a stable region of values that performs best. Table 3 depicts both, the chosen values and good value

Table 3
Fine tuning for different scheduling policies.

	RR	**MC**	**MM**
TmSl (ms)	0	0	70 (60–70)
DQS (row packs)	100 (30–150)	100 (80–150)	100
RPS (tuples)	400 (200–500)	400 (200–450)	400

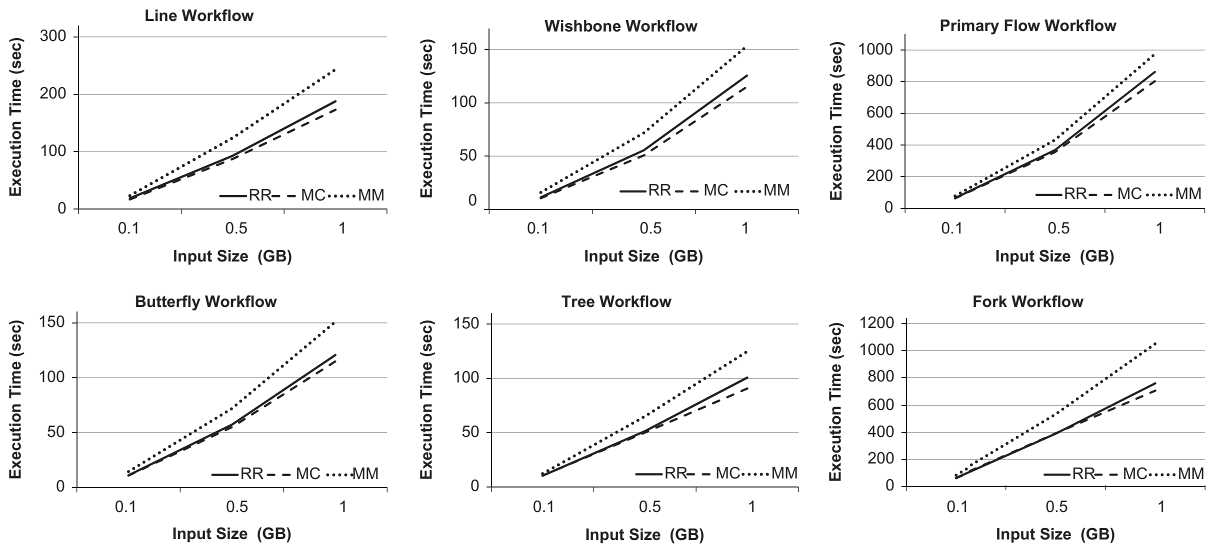

Fig. 8. Effect of data size and scheduling policy to execution time.

ranges (inside the parentheses) for time slot, queue, and row pack sizes. For additional details and a complete listing of experiments and measurements regarding the tuning of our system, we refer the interested reader to a document describing our system in more detail [20].

All experiments have been conducted on a Dual Core 2 PC at 2.13 GHz with 1 GB main memory and a 230 GB SATA disk. All findings reported here are based on actual executions and not simulations. (Using additional nodes would implicate our analysis, as we would need to consider network issues like availability, bandwidth, and so on. However, this is a topic for future work that builds on the current findings.)

7.3. Experimental results with single policies

Next, we report on our findings on the behavior of the measured scheduling policies with respect to their execution time and memory requirements when varying data size, selectivity, and structure of the flow. In these experiments, a single scheduling policy was used for each single scenario. Detailed results on selectivity impact are omitted for lack of space; still our findings are consistent with the effect of data size.

Effect of data size and selectivity on execution time: The effect of data size processed by the scheduling algorithms to the total execution time is linear in almost all occasions (Fig. 8). Typically, the MM algorithm behaves worst of all the others. MC is slightly better than the RR algorithm. Still, the difference is small and this is mainly due to the fact that the RR scarcely drives the execution to a state with several idle activities; therefore, the pipelining seems to work properly. The effect of selectivity to the execution time is similar (Fig. 11 shows representative results for selectivity and two workflow types: butterfly and primary flow, for data with scale factor 1). However, each workflow type performs differently. Workflows with heavy load due to blocking and memory-consuming operators, as the Primary Flow and the Fork, demonstrate significant delays in their execution.

Effect of data size and selectivity on average memory: The average memory used throughout the entire workflow execution shows the typical requirements of the scheduling protocol normalized over the time period of execution. In all occasions, the MM algorithm significantly outperforms the other two, with the RR algorithm being worse than MC. The effect is the same if we vary data size (specifically, the TPC-H scale factor—cf. the *x*-axis in Fig. 9) or the selectivity of the workflow (see Fig. 11 for butterfly and primary flow). Workflows containing a large number of activities, especially the ones with a right butterfly wing (e.g., fork) necessarily consume more memory than others. Still, the benefits of MM are much more evident in these cases (bottom three graphs of Fig. 9), as this algorithm remains practically stable to its memory requirements independently of workflow type.

Effect of data size and selectivity on maximum memory: The maximum memory measure tries to capture the possible peaks in memory consumption that occur throughout the execution of a workflow. Again, we vary the data size (Fig. 10) and the selectivity of the workflow (see Fig. 11 for butterfly and primary flow) and assess the impact to the maximum memory on different workflows. The workflows with a large number of activities or with memory-intense activities (like primary flows for example) result in much higher peaks that are practically indifferent to the affecting factors like data size and selectivity of the workflow. In most cases, the MC algorithm provides the lowest peaks; moreover, it constantly gives impressively smaller peaks in memory intensive workflows than the other two algorithms (see for example the fork and the primary flow cases in Figs. 10 and 11). In cases with low percentage of joins in the workflow, or, small data sizes, the MM algorithm prevails.

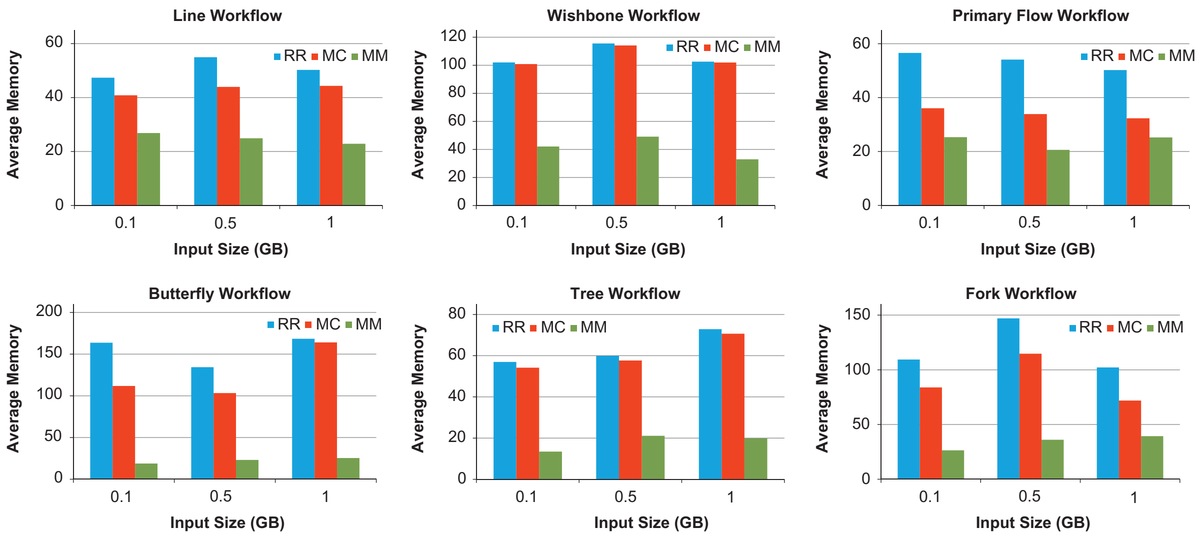

Fig. 9. Effect of data size and scheduling policy to avg memory consumption (♯ row packs).

Observations so far: In general, the state-of-practice tactics of round robin scheduling is efficient in terms of time behavior, but lags in memory consumption effectiveness. It is possible to devise a scheduling policy (i.e., MC) with time performance similar (actually: slightly better) to the round robin policy and observable earnings in terms of average memory consumption. A slower policy (i.e., MM) can give significant earnings in terms of average memory consumption that range between 1/2 and 1/10 of the memory used by the other policies. MM can be used in an environment where more than one concurrent operations run, being memory efficient is important, and memory has to be available at peak times. On the other hand, if maximum memory is really important, then for certain classes of workflow structure it is possible to derive even better achievements by picking the appropriate policy. In most cases, although outperformed with respect to average memory requirements, MC provides the solution with the less peaks for memory consumption in most of the experiments conducted (however, in few other cases it is the MM that achieves this property). Hence, overall MC provides a good equilibrium for both the antagonizing measures of time and resource efficiency.

7.4. Mixture of scheduling policies

Next, we experiment with Mixed Policy (MP) that builds on top of our findings for RR, MC, and MM and aims at combining their best characteristics. Having tested each policy on entire workflows, we use MP to separately schedule the execution of different workflow fragments.

As we discussed in Section 5, MP may schedule arbitrary workflow fragments with different policies. The criteria of such a choice involve the structure of the workflow, like for example, how many memory-intensive operations it involves (this is regulated by the θ parameter). Our experiments show that as the data volume increases, a smaller value of θ may be used (e.g., for scale factor 1, θ may be as low as 4 to 6; for scale factor 10, even a $\theta = 2$ makes a difference). Obviously, θ may need to take different values with different settings (e.g., larger available physical memory), but the trend remains: the choice of θ should largely depend on the data volume.

Having tuned θ, we tested MP on 20 workflows containing a varying number of nodes between 20 and 45. Overall, MP improves memory utilization and may result in an average 35–53% improvement in execution time. MP decomposes a workflow into subflows that in large fall into the archetype workflow categories discussed earlier in this section (examples are depicted in Fig. 7) possibly extended with longer lines (e.g., subflows containing a series of pipeline operations). MP schedules the lines with the MC scheduling policy to boost pipelining and to achieve a better execution time (see the discussion in Section 5). The “core” of the workflows (e.g., the body of a butterfly) and subflows containing memory-intensive operations (like blocking operators as joins, sorted unions, etc.) are executed using the MM policy. As an example, in the ‘tree’ workflow depicted in Fig. 7, the two joins constitute the core of the workflow. Also, in cases as the ‘butterfly’ workflow of Fig. 1, there are four extremes that can pursue speed whereas the central join with its blocking activities constitute the core. Finally, the nodes next to the router of the ‘fork’ scenario (bottom right in Fig. 7) along with the router also constitute the core of the butterfly.

Fig. 12 illustrates example results of using the Mixed Policy strategy for executing three workflows. The first workflow corresponds to the extraction phase of an ETL scenario. It involves a tree archetype having five joins as a core component and it is extended by three long (8–12 pipeline operations each) and two short (3–6 pipeline operations each) lines from the left and one short line (four operations) from the right. The lines contain data computation, filter, and schema modification operations, which are not memory-intensive. The second workflow corresponds to the transformation and cleansing phase of the same ETL scenario. It may be seen as a butterfly

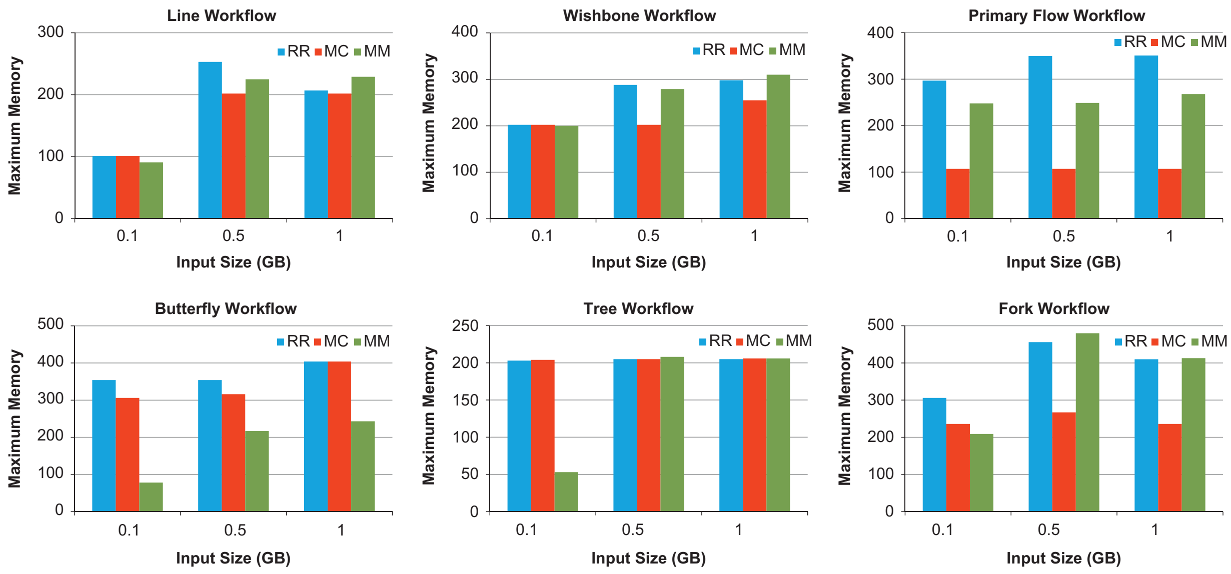

Fig. 10. Effect of data size and scheduling policy to max memory consumption (♯ row packs).

pattern extended with four lines (3–7 cpu-intensive operations each). The butterfly body has a sorted union and two blocking aggregate operations. The third workflow describes the load phase. It involves three fork-like structures that load 2, 3, and 6 target data stores, respectively, through 14 lines in total connected before and after the fork structures.

Observe that in all cases the savings can go up to 40% of time as the amount of data increases. The reason for these savings has to do with the fact that the scheduling exploits both the SISS's and the knowledge on the archetype patterns' behavior. Using SISS's, we identify the boundaries where staging takes place and so there is an opportunity for a change in the scheduling policy. Then, we assign an appropriate scheduling policy to each subflow based on the lessons learned from the extensive set of micro-benchmarks we conducted (similar to the experiments on different archetypes shown in Section 7.3) on various workflow structures like those discussed in Section 7.1. Hence, we speed up fast workflow fragments with MC and we switch to a different policy, MM, when there is a pressing need for memory.

Note that the overall gain is achieved by using the exact same system resources (the allocation of additional resources favors Mixed Policy, since the single scheduling policies do not "know" how to efficiently handle them). This observation is important, since a common problem that real-life warehouse administrators often have is that they cannot improve the ETL execution (or equally, they cannot meet the strict restrictions in the desired execution time window) by simply adding more resources [6]. ETL optimization techniques are needed, and currently, such techniques are very limited in ETL tools.

8. Conclusions

In this paper, we have dealt with the problem of scheduling off-line ETL scenarios, aiming at improving both the execution time and memory consumption, without allowing data shedding. We have proposed an extensible architecture for implementing an ETL scheduler based on the pipelining of results produced by ETL operations. We have used a real implementation – not a simulation – to evaluate our techniques. We have assessed a set of scheduling policies for the execution of ETL flows and have shown that a Minimum Cost Prediction - policy that aims at emptying the largest input queue of the workflow, typically performs better with respect to execution time, whereas a Minimum Memory Prediction policy that favors activities with the maximum tuple consumption rate, is better with respect to the average memory consumption. Apart from a single policy scheduling, we have also proposed a principled method for segmenting ETL workflows in parts that can be executed with different scheduling policies. This way, memory intensive parts of a workflow can benefit from less antagonism for resources. Our Mixed Policy strategy exploits that to achieve better time performance.

Future work can be directed to other prioritization schemes (e.g, due to different user requirements) and the encompassing of active data warehouses (a.k.a. real-time data warehouses) in the current framework. So far, in the context of our work, the related research on active warehouses has focused mostly on the loading part. Still, there are several open problems concerning the orchestration of the entire process from the sources all the way to the final data marts [21].

In addition, although our implementation does support and favor pipeline parallelism (parallelism within a subflow), we do not consider partitioning parallelism (parallelism within ETL activities) yet. In theory, the most obvious consequence would be a reduction of the data volume processed by a single workflow, without differentiating much the overall analysis. However, this would add an interesting perspective to our problem (mostly, to the workflow segmentation).

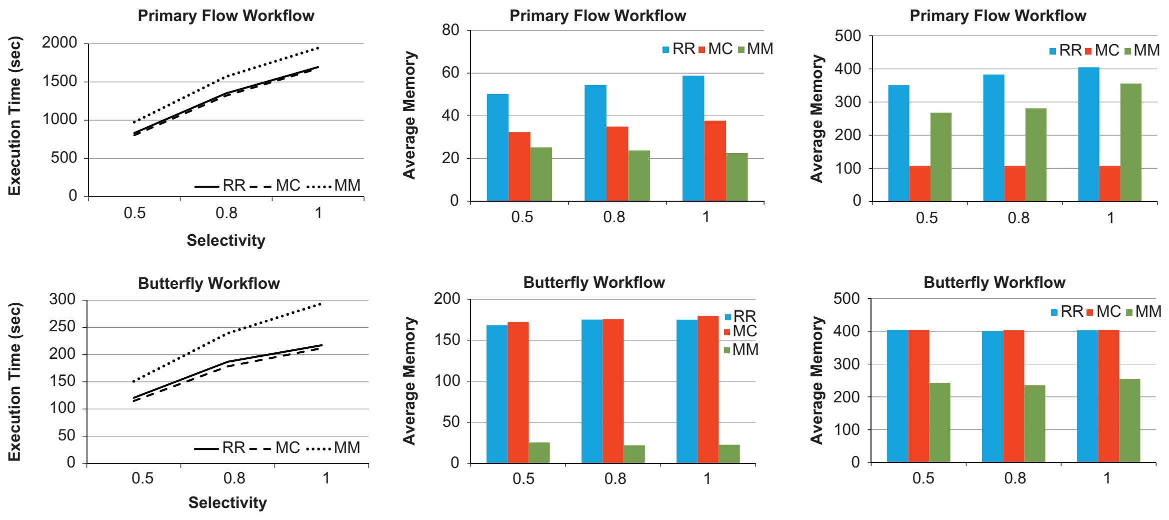

Fig. 11. Example selectivity and scheduling policy effect for two workflows.

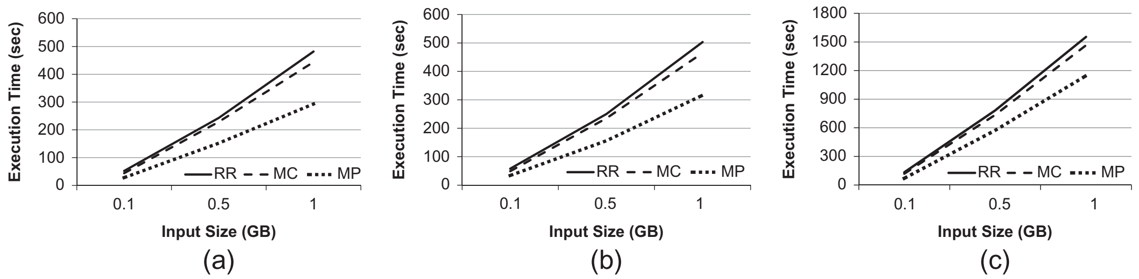

Fig. 12. Mixed Policy behavior for workflows containing combinations of multiple line archetypes with (a) tree, (b) butterfly and (c) fork archetypes.

Appendix A. An example ETL scenario on X-rays

To further illustrate the motivation behind our work, we present in detail the execution of the balanced butterfly ETL scenario shown in Fig. 1, which involves (a) a left wing of the butterfly, with checks and value computations and function applications, and, (b) a right wing, with the population of two target tables and four materialized views via a central join operation that combines the two sources. The left wing has two parallel lines, one for loading table *PartSupp* (along with isolating *NULL* values and deriving some statistics) and another for loading table *Supplier* (along with not null checks isolation and some formatting of strings). The right wing groups by (i) nation and part, (ii) part, (iii) nation and supplier, and, (iv) supplier. In the body of the butterfly, the join is a sort-merge join and the results are stored in a target table too.

Next, we show what happens in the internals of the scheduler and execution engine, and why it pays off to improve the scheduling compared to a simple round-robin mechanism. To this end, we have heavily monitored the execution of the scenario for different scheduling policies. Our logging traces the choices of the scheduler, as well as, the memory load, by inspecting the queues of the activities approximately every 100 ms. The results of our analysis are shown in Fig. A1. Before proceeding, two notes on the experimental validity:

- We have executed each scenario 10 times, without observing noticeable differences. Therefore, the reported executions are characteristic of their category.
- Due to heavy monitoring and logging, the execution times are simply dominated by the logging and irrelevant in this discussion. The reader should refer to the experimental section of the paper for the behavior of the engine in terms of time. At the same time, the scheduling order and the memory consumption are accurately monitored, and they constitute the focus of our deliberations.

First, we start with the RR execution, which gives a fair chance to all parts of the scenario. This results in useless calls to the right wing early enough. RR comes with a heavy price on main memory: as data are loaded, queues start to fill up and, as the activities that will consume these data are not fired as soon as possible, the data reside in the main memory for long. Whenever activities storing data to their ultimate targets are fired, the memory used drops significantly (see MC for explanations). The visual representation of activity executions and memory

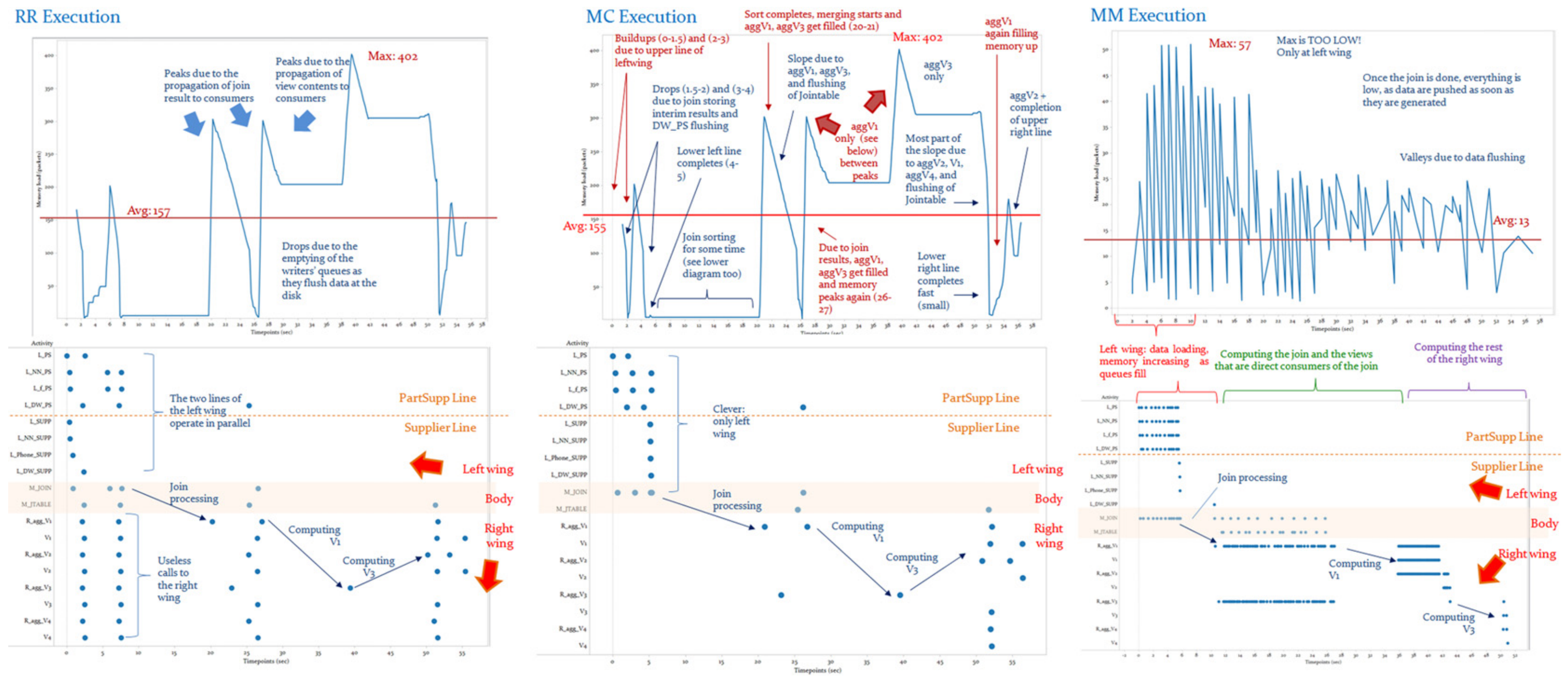

Fig. A1. RR, MC, and MM behavior. Top: memory consumption over time. Bottom: timepoints when an activity (including readers and writers for data stores) is invoked; durations are not depicted, however, long-lasting executions are highlighted with arrows.

consumptions shows that after the sort phase of the join in the body has been executed we have the entire right wing alive and this results in keeping large amounts of data in main memory (join results) before being aggregated and stored at their target materialized views.

Thus, it is evident that we can do better than RR in many ways: (a) reduce the calls to activities that should not be invoked, (b) reduce the switching of activities by keeping “hot” activities active, and (c) save memory by prioritizing memory emptying activities. These opportunities are depicted in the strategies that we have implemented.

The MC scheduling mechanism is much more compact in execution: the early stages are also characterized by high peaks of memory, yet activity switching is reduced. As in the case of Round Robin, the three periods of execution are evident in the diagrammatic layout (observe also that when the left part is done, its activities are removed from the task list). We use MC as an opportunity to explain how the internals of memory management work. At the early stages, as data are retrieved from the sources, the input queues of the left wing start to fill up. They are temporarily relieved (see the slopes in the diagram) whenever the join is awaken or some data reach the two target tables of the left wing. Since the join is a sort-merge join, the early wake-ups of the join push data to the temporary cache files of the sorters that await to receive all input before they start sorting. When sorting starts, all is quiet (as we use an external sorter for the task). When the sorting of the sort-merge join is over and merging starts, join results start to be pushed towards the join consumers (the first two aggregators plus the Jointable writer). This builds up memory usage. The operation of the right wing is memory intensive as the aggregators use memory to produce the results of the two large views ($V1$ and $V3$); when these views are done, the flushing of the results and the computation of the smaller views takes place fast and with low memory consumption. Overall, as the memory “cardiograms” show, the differences from RR are not many, mainly concern the smaller amount of activity switching, and, as demonstrated in the experiments, typically make MC present slightly better times and memory than RR.

A drastic difference comes with MM. MM suffers from a couple of problems, namely (a) the early wake-ups of the body and (b) late prioritization of the loading of the second source table (Supplier), due to its small size. The frequent wake-ups of the join make us pay in execution time (as they flush interim results) but prevent the system from reaching the memory heights of the other two methods. At the same time, we observe a very different pattern of execution. In the beginning, we have first the upper line of the left wing, and then the lower line of the left wing (which is not so good, esp. since the lower line involves small data amounts). Then, we have a mixed execution of the join activity along with its immediate right wing consumers, since, when the sorting is done the consumers can consume join results. Finally, as data are produced in the right wing, the execution follows a practically RR fashion, where data are consumed and pushed immediately to their destinations, resulting to lower memory usage—unfortunately with longer periods of execution. As our experiments have shown, the activity switching that MM does as well as the fact that the writers are activated frequently eventually degrades performance (remember: writers are the ones who eventually gain memory for us, albeit at the price of the increased I/O cost). However, this comes at highly significant gains in terms of memory consumption: as data get unblocked, they are quickly pushed towards their target, keeping memory low, not only on average, but in terms of peaks too (compare the peak of MM with the averages and the peaks of the two other methods to see the extent of the difference). In fact, the gains from the MM policy cannot be underestimated: assuming that a server has memory constraints (due to the data sizes or the existence of other ETL tasks running simultaneously) the MM strategy is by far the method of choice.

References

[1] A. Simitsis, P. Vassiliadis, T.K. Sellis, State-space optimization of ETL workflows, IEEE Transactions on Knowledge and Data Engineering 17 (10) (2005) 1404–1419.
[2] A. Simitsis, K. Wilkinson, U. Dayal, M. Castellanos, Optimizing ETL workflows for fault-tolerance, in: ICDE, 2010, pp. 385–396.
[3] V. Tziovara, P. Vassiliadis, A. Simitsis, Deciding the physical implementation of etl workflows, in: DOLAP, 2007, pp. 49–56.
[4] M. Thiele, U. Fischer, W. Lehner, Partition-based workload scheduling in living data warehouse environments, in: DOLAP, 2007, pp. 57–64.
[5] C. Thomsen, T.B. Pedersen, W. Lehner, Rite: providing on-demand data for right-time data warehousing, in: ICDE, 2008, pp. 456–465.
[6] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, QoX-driven ETL design: reducing the cost of ETL consulting engagements, in: SIGMOD Conference, 2009, pp. 953–960.
[7] P.J.F. Carreira, H. Galhardas, J. Pereira, A. Lopes, Data mapper: an operator for expressing one-to-many data transformations, in: DaWaK, 2005, pp. 136–145.
[8] R. Avnur, J.M. Hellerstein, Eddies: Continuously adaptive query processing, in: SIGMOD Conference, 2000, pp. 261–272.
[9] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73–170.
[10] M.A. Sharaf, P.K. Chrysanthis, A. Labrinidis, K. Pruhs, Algorithms and metrics for processing multiple heterogeneous continuous queries, ACM Transactions on Database Systems, 33 (1) (2008).
[11] G. Luo, J.F. Naughton, C.J. Ellmann, M. Watzke, Transaction reordering and grouping for continuous data loading, in: BIRTE, 2006, pp. 34–49.
[12] L. Golab, T. Johnson, V. Shkapenyuk, Scheduling updates in a real-time stream warehouse, in: ICDE, 2009, pp. 1207–1210.
[13] D. Carney, U. Çetintemel, A. Rasin, S.B. Zdonik, M. Cherniack, M. Stonebraker, Operator scheduling in a data stream manager, in: VLDB, 2003, pp. 838–849.
[14] B. Babcock, S. Babu, M. Datar, R. Motwani, Chain: operator scheduling for memory minimization in data stream systems, in: SIGMOD, 2003, pp. 253–264.
[15] T. Urhan, M.J. Franklin, Dynamic pipeline scheduling for improving interactive query performance, in: VLDB, 2001, pp. 501–510.
[16] U. Dayal, M. Castellanos, A. Simitsis, K. Wilkinson, Data integration flows for business intelligence, in: EDBT, 2009, pp. 1–11.
[17] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, S. Skiadopoulos, A generic and customizable framework for the design of ETL scenarios, Information Systems 30 (7) (2005) 492–525.
[18] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, V. Tziovara, Benchmarking ETL workflows, in: TPCTC, 2009, pp. 199–220.
[19] TPC, The TPC-Hbenchmark, Technical Report, Transaction Processing Council, 2007. Available at: 〈http://www.tpc.org/tpch/〉.
[20] A. Karagiannis, Scheduling policies for the refresh management of data warehouses, Master's Thesis, 2007. Available at: 〈http://www.cs.uoi.gr〉.
[21] P. Vassiliadis, A. Simitsis, Near real time ETL, in: New Trends in Data Warehousing and Data Analysis, 2009, pp. 1–31.
[22] X. Liu, C. Thomsen, T.B. Pedersen, ETLMR: a highly scalable dimensional ETL framework based on MapReduce, in: DaWaK, 2011, pp. 96–111.