

# Macro-level Scheduling of ETL Workflows

Anastasios Karagiannis  
University of Ioannina  
Ioannina, Greece  
ktasos@cs.uoi.gr

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

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

## ABSTRACT

Extract-Transform-Load (ETL) workflows (a) extract data from various sources, (b) transform, cleanse and homogenize these data, and (c) populate a target data store (e.g., a data warehouse). Typically, such processes should terminate during strict time windows and thus, ETL workflow optimization is of significant interest. In this paper, we deal with the problem of scheduling the execution of ETL activities, with the goal of minimizing ETL execution time and allocated memory. Apart from a simple, fair scheduling policy we also experiment with two policies, the first aiming to empty the largest input queue of the workflow and the second to activate the activity with the maximum tuple consumption rate. We experimentally show that the use of different scheduling policies can improve ETL performance in terms of memory consumption and execution time.

## 1. INTRODUCTION

Extract-Transform-Load (ETL) processes constitute the backbone of a Data Warehouse (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., duplicate detection, 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) and so on.

One of the main practical problems involves the timely ETL-ing of large data volumes under pressing time constraints. To give an example, we mention a case study for mobile network traffic data, involving around 30 data flows, 10 sources, and around 2TB of data, with 3 billion rows [1]. In that case study, it is reported that user requests indi-

cated a need for data with freshness at most 2 hours. Since the data warehouse refreshment process must be completed within a short time window with completeness and accuracy guarantees for the data (and allowing some extra time for resumption processes in the case of failures) it is important to perform the ETL process as fast as possible.

The ideal execution of ETL workflows suggests pipelining the flow of tuples all the way from the source to the target data stores. Typically, this cannot happen 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). An appropriate scheduling policy is required for orchestrating the smooth flow of data towards the target data stores. In that sense, *activity scheduling* during the ETL workflow execution results in efficient prioritization of which activities are active at any time point, with the goal of minimizing the overall execution time without any data losses.

In available ETL tools –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, or parts of the ETL workflow, according to 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, we answer 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?”.

Although scheduling policies have been studied before, in the context of ETL workflows, related work has only partially dealt with such problems so far. There are some first results in the areas of ETL optimization [10, 12, 16] and update scheduling in the context of near-real time warehousing [13, 14]. 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 experiment with scheduling algorithms specifically tailored for batch ETL processes and try to find an ETL configuration that optimizes two performance objectives: *execution time* and *memory requirements*. To the best of our knowledge (based on our experience and publicly available documentation for ETL tools), 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 showed that deadlocks are possible in the absence of scheduling; hence, scheduling is necessary for an ETL engine. We demonstrate that two scheduling techniques, MINIMUM COST and MINIMUM MEMORY, 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.

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 representative 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, realistic framework for experimentation with ETL flows. That said, 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.

**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. First, we formally setup the problem (Section 3) and discuss scheduling mechanisms that optimize ETL execution in terms of execution time and memory requirements (Section 4).
- Then, we propose a generic software architecture for incorporating a scheduler to an ETL engine and present such an ETL engine as a proof of concept (Section 5).
- Finally, we demonstrate through a series of experiments the benefits of our scheduling techniques. The findings are based on template ETL structures that constitute representative patterns of real-world ETL processes (Section 6).

## 2. RELATED WORK

Related work revolves around the scheduling of concurrent updates and queries in real-time warehousing and the scheduling of operators in Data Streams Management Systems. In this section, we briefly present related work, and especially, we argue that the problem considered in this paper has not been tackled so far in the context of ETL workflows. We also discuss related work on stream scheduling, and we justify why a fresher look is needed for ETL.

### 2.1 Scheduling for ETL

Related work in the area of ETL involves efforts towards the optimization of entire ETL workflows [10, 11, 12, 16] and of individual operators, such as the DataMapper [4]. To forestall any possible criticism, we would like to mention that traditional query optimization, although related in a broader sense (and integrated in our work in terms of algebraic optimization) is (a) 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) orthogonal to the problem of scheduling operators.

Thiele et al. deal with workload management in real time warehouses [13]. 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 [14]. The loader synchronizes data load with queries that require source data with a specific freshness guarantee. Luo et al. deal with deadlocks in the continuous maintenance of materialized views [8]. To avoid deadlocks, the paper proposes reordering of transactions that refresh join or aggregate-join views in the warehouse. Golab et al. discuss the scheduling of the refreshment process for a real time warehouse [6], 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 [3] can execute more than one continuous query for the same input stream(s). 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: (a) execution time, (b) latency time, and (c) memory. The Chain scheduler reduces the required memory when executing a query in a data stream system [2]. 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 present two scheduling algorithms that exploit the pipelining in query execution [17]. 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 volumi-

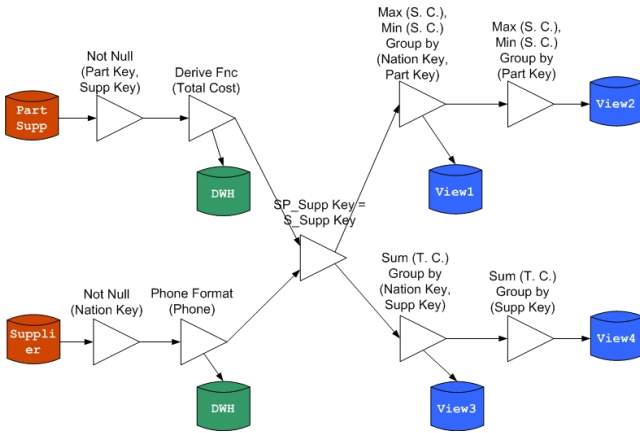


Figure 1: Example ETL workflow

nous streams). Still, compared to stream scheduling, ETL flows present different difficulties, since they involve complex processing (e.g., user-defined functions, blocking operations, cleansing operations, data and text analytics operations [5]) and they must respect a zero data loss constraint (as opposed to stream shedding). 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, 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 to target (e.g., data warehouse) schema. Finally, the processed data are loaded into the data warehouse relations.

The full layout of an ETL workflow, involving activities and recordsets can be modeled as a directed acyclic graph (DAG) [19]. 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. Being a workflow, each node has input and output schemata, which can be empty as well; e.g., a source recordset has an empty input schema. A *producer* node feeds its successor node, which in its turn, is the *consumer* of the first one. 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.

**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*

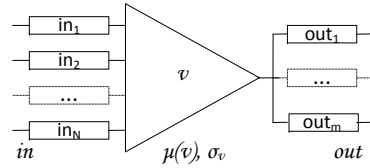


Figure 2: ETL activity structure

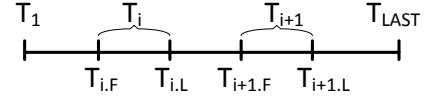


Figure 3: Timestamps for scheduling

*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  $View_1, \dots, View_4$  via successive aggregations.

**Definitions.** An ETL workflow comprises a 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. Let  $\mathcal{V}$  be a set containing nodes classified regarding their execution status: candidates (nodes that are still active and participate in the execution) and finished (nodes that have finished their processing), i.e.,  $\mathcal{V} = \mathcal{V}_{CAND} \cup \mathcal{V}_{FIN}$ . Obviously,  $\mathcal{V} = \mathbf{V}$ .

A generic ETL activity is depicted in Fig. 2. For each activity node  $v \in \mathbf{V}_A$  we define:

- $\mu(v)$ , as the consumption rate of node  $v$ ,
- $\mathbf{Q}(v)$ , as the set of all input queues of  $v$ ,
- $queue(v)$ , as the sum of all input queue sizes (not capacity) of node  $v$  (of course, unary activities have only one input queue), and  $queue_t(v)$  is the  $queue(v)$  at a certain time interval  $t$ .
- $\sigma_v$ , as the selectivity of node  $v$ .

For each queue  $q$ , we define  $size(q)$  as the memory size of  $q$  at a given time point and  $MaxMem(q)$ , as the maximum memory size that the queue can obtain at any time point.

For each recordset node  $v \in \mathbf{V}_R$  we define:

- $\mu(v)$ , also as the consumption rate of node  $v$ .

Furthermore, for each source recordset node we define:

- $volume(v)$ , as the size of the recordset.

We consider  $\mathbf{T}$  as an infinite countable set of timestamps and a scheduler with policy  $P$ . The scheduler divides  $\mathbf{T}$  into disjoint and adjacent intervals  $\mathbf{T} = \mathbf{T}_1 \cup \mathbf{T}_2 \cup \dots$  with:

- $\mathbf{T}_i = [\mathbf{T}_i.first, \mathbf{T}_i.last]$
- $\mathbf{T}_i.last = \mathbf{T}_{i+1}.first - 1$  (see also Fig. 3).

The scheduler 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.  $active(\mathbf{T}_i)$ . According to the scheduling policy  $P$  used, the scheduler has to choose the next activity to run.
2.  $\mathbf{T}_i.last$ . This is the timestamp that determines when operator  $active(\mathbf{T}_i)$  will stop executing. (It also determines the scheduler time slot  $\mathbf{T}_i.length()$ .)
3. Status of all queues at  $\mathbf{T}_i.last$ , in order to highlight queues that have reached their maximum capacity.

The operator  $active(\mathbf{T}_i)$  will stop its processing if one of the following occurs:

1.  $clock = \mathbf{T}_i.last$ . Then, the time slot is exhausted.
2.  $queue(active(\mathbf{T}_i)) = 0$ . Then, the active operator has no more input data to process.
3.  $\exists v \in consumer(active(\mathbf{T}_i))$  such that for any of its queues, say  $q$ ,  $size(q) = MaxMem(q)$ . Then, one of the consumers of the active activity  $active(\mathbf{T}_i)$  has a full input queue and further populating it with more tuples will result in data loss.

At this point we must check if  $active(\mathbf{T}_i)$  should be moved to  $\mathbf{V}_{FIN}$ . In order for an operator  $v$  to be moved to  $\mathbf{V}_{FIN}$ , both of the following must be valid.

- $\forall v \in producer(active(\mathbf{T}_i)), v \in \mathbf{V}_{FIN}$ , and
- $queue(active(\mathbf{T}_i)) = 0$  or  $volume(v) = 0$ , if  $v$  is a source recordset.

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  $\mathbf{T}_1 \cup \mathbf{T}_2 \cup \dots \cup \mathbf{T}_{last}$
- $\forall t \in \mathbf{T}, v \in \mathbf{V}, \forall q \in \mathbf{Q}(v) \ size(q) \leq MaxMem(q)$  (i.e., all data are properly processed).
- One of the following objective functions is minimized:
  - $\mathbf{T}_{last}$  is minimized, where  $\mathbf{T}_{last}$  is the interval where  $G$  stops
  - $max \sum_{v \in \mathbf{V}} queue_t(v)$  is minimized, where  $t \in \mathbf{T}$ , and  $v \in \mathbf{V}$ .

## 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, (c) the response time, and (d) the required memory during the execution. Since in our context the response time is an issue of secondary importance (see Section 2), we investigate scheduling policies belonging to the other three categories. We explore three generic algorithms: ROUND ROBIN, MINIMUM COST, and MINIMUM MEMORY, belonging to one of the aforementioned categories.

	pick next	reschedule when
RR	operator id	input queue is exhausted
MC	max size of input queue	input queue is exhausted
MM	max tuple consumption	time slot

**Table 1: Decision criteria for scheduling algorithms**

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 is simple and easy to implement. It assigns time slices to each operator in equal portions and in an order based on a unique identifier that every operator has. Assuming a list  $V_{CAND}$  containing activities to be scheduled, each time, the algorithm picks the first activity from  $V_{CAND}$ . Its main advantages are as follows: every operator gets the same chances to run (fairness) and the system always avoids starvation.

### 4.2 Minimum Cost

The MINIMUM COST (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. Each time, the selected operator should have data ready for processing, and typically, this operator is the one having the largest volume of input data. Since there are no time slots, the selected operator processes all data without any interruption from the scheduler. Without loss of generality, in our implementation we have considered that all operators that read data from an external source are always available for execution.

---

#### Algorithm MINIMUM COST

---

**Input:** A list  $V_{CAND}$  containing activities  
**Output:** The next activity  $MC_{next}$

```

1 begin
2    $MaxInput = -1$ ;
3   for  $v \in V_{CAND}$  do
4     if (  $MaxInput < v_Q$  ) then
5        $MC_{next} = v$ ;
6        $MaxInput = v_Q$ ;
7   return  $MC_{next}$ ;
8 end
```

---

### 4.3 Minimum Memory

The MINIMUM MEMORY (MM) scheduling algorithm schedules the operators in a way that minimizes the system memory required during the workflow execution. In each step, MM selects the operator 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 operator 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 operator  $p$ ,  $ExecTime(p)$  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 operator with the biggest  $MemB()$  value at every scheduling step.

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

Note that when the workflow execution starts no operator has processed any data, so the above formula cannot apply. In this case, resembling MINIMUM COST, the operator with the biggest input size is selected.

---

#### Algorithm MINIMUM MEMORY

---

**Input:** A list  $V_{CAND}$  containing activities  
**Output:** The next activity  $MM_{next}$

```

1 begin
2    $MaxInput = -1$ ;
3    $MMem = -\infty$ ;
4   for  $v \in V_{CAND}$  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  if ( $MMem \leq 0$ ) then
12     $MM_{next} = MC_{next}$ ;
13  return  $MM_{next}$ ;
14 end

```

---

## 5. 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, industrial 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.

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 [3]), so data queues contain and exchange blocks of tuples, called *row packs*. Each operator (a) has a mailbox

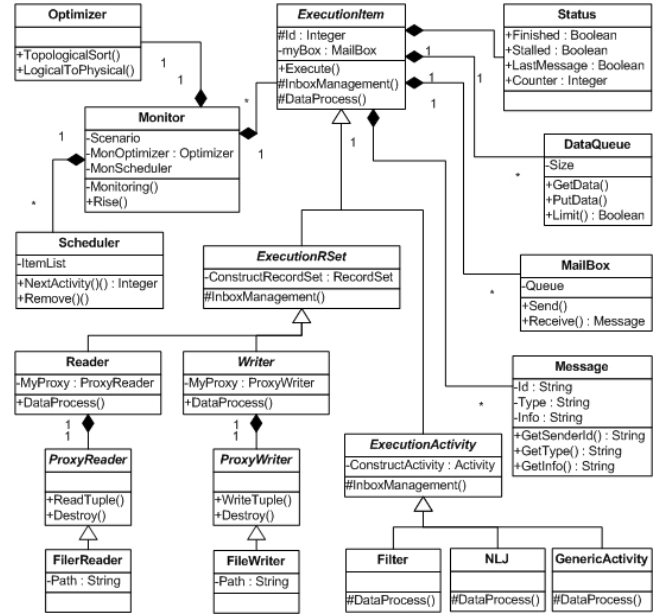


Figure 4: Software architecture of the scheduler

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. 4 shows the class diagram of our system architecture. Next, we elaborate on two core system components, namely the *Execution Item* and *Monitor*.

### 5.1 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

---

**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
7 | | | DataProcess();
8 end
```

---

be appropriately specialized depending on the functionality of the recordset or activity represented by this execution item. Next, we discuss these two as an example for illustrating the mechanics of *DataProcess()*.

**Recordsets.** Recordsets are instantiated as *Readers* or *Writers*. These classes are responsible for feeding the workflow with input data or storing the output data, respectively. Each *Reader* or *Writer* uses a proxy inside *DataProcess()*. The proxy is a wrapper for objects that read from (or write to) text files, XML files, database tables, and so on. Depending on whether the recordset is used for reading, writing or both, we define the correct proxy to instantiate; e.g., a *FileReader* or *FileWriter* class. Next, we present an abstract implementation of *DataProcess()* for *Reader*. The *Status* variable keeps track of the consumer's data queues; if these queues are full, the operator must stop processing data.

---

**Function DATAPROCESS() for Reader**

---

```
1 begin
2 | for all tuples t in current pack do
3 | | read tuple t;
4 | | if t is NULL then
5 | | | status = finished;
6 | | else
7 | | | status = forwardToConsumers(t);
8 | | | if status = false then stall thread;
9 end
```

---

**Activities.** For an activity, *DataProcess()* (a) reads from its data queues, (b) processes the tuples, and then, (c) forwards them to its producers. Next, 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.

**Status.** Every *Execution Item* has a *Status* that keeps track of the status of an operator, which can be one of the following:

- *Stalled* allows the operator to call the *DataProcess()*.
- *LastMessage* indicates whether the operator will receive or not more messages from its producers.
- *Finished* indicates whether the operator's execution is complete.

---

**Function DATAPROCESS() for a Filter**

---

```
1 begin
2 | if no pack then
3 | | if last message then
4 | | | status = finished;
5 | | else
6 | | | stall thread;
7 | else
8 | | while exists next tuple in input do
9 | | | if can process current tuple then
10 | | | | status = status & forwardToConsumers(t);
11 | | | if status = false then stall thread;
12 end
```

---

## 5.2 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.

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.

Note that the scheduler exploits queues and the DAG nature of the graph for *avoiding starvation*. In general, an activity may starve when its output queue is always full. However, this is infeasible in our approach, since the graph structure is such that writers eventually empty the final queues and that propagates the activation of the appropriate activities all the way toward the end. Starvation might happen in a distributed setting when some source disappears; this case is out of the context of this paper.

## 5.3 Extensibility of the architecture

Extensibility, has been an important design goal for our architecture and involves (a) the engine, (b) the scheduler, and, (c) a variety of supported activity types. The last part is facilitated via an extensible approach to implementing *Execution Activity* and *DataProcess()*. Therefore, our scheduler supports an extensible library of ETL activities and it can be smoothly linked to an open-source ETL via the appropriate wrapping of the ETL engine's operations within the *DataProcess()* functions of the respective classes.

Our implementation classifies operators into different categories. *Pipelining* operators (e.g., filters and functions), are applied over individual records having only one input edge and a simple task to perform over each input tuple in isolation. A simple hierarchy of classes overloads the data

Message type	Receiver’s reaction
MsgEndOfData	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 <i>Stalled</i> to false.
MsgStall	Receiver temporarily stops processing data by switching the flag <i>Stalled</i> to true.
MsgDummy-Resume	Used to force all operators to execute <i>DataProcess()</i> 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.

Table 2: Example message types

process methods with the appropriate semantics and performs the checks and transformations needed. *Binary* and *blocking* activities constitute two other different categories. As an example, representative, proof of concept, operators implemented in our system include (a) the unary *Aggregator* class and (b) *Join*, *Surrogate Key*, *Diff* (all three with sort-merge and nested-loops variants, with the latter being blocking only for their right input). As with traditional relational engines, whenever these operators are blocking, they have (a) to gather all input data to text files and prepare them (e.g., sort them) and (b) to perform the appropriate data processing (e.g., the matching of the two inputs or the aggregation, depending on the operator’s class).

## 6. EXPERIMENTS

In this section, we report on the experimental assessment of the proposed 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.

### 6.1 Archetype ETL patterns

A particular problem one has to resolve when working with ETL workflows is to decide what workflows to use for the experimental assessment of any suggested methods. To contribute with a solution to the problem, we have proposed a benchmark with characteristic cases of ETL workflows [9]. The main design artifact upon which we base the workflow construction is the notion of *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 on their way to the warehouse; (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, and so on, 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, slowly changing dimensions). The internal structure of the butterfly ultimately results in composing a left wing as a tree of subflows converging to-

ward the body (with the tree’s root at the body), whereas the right wing can be viewed as the inverse tree. The details of the employed workflow patterns can be found in [9]. Here, we briefly sketch some example workflow archetype patterns (see Fig. 5). 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 single 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 on the result recordset. The join can be performed over either heterogeneous relations, whose contents are combined, or homogeneous relations, whose contents are integrated into one unified (possibly 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 workflow offers opportunities for parallelization as well.

### 6.2 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.

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. In the next subsection, we demonstrate results regarding the other parameters. Our test data has been generated with the TPC-H [15] generator.

Workflow execution requires the fine-tuning of both the engine and scheduler. Specifically, 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

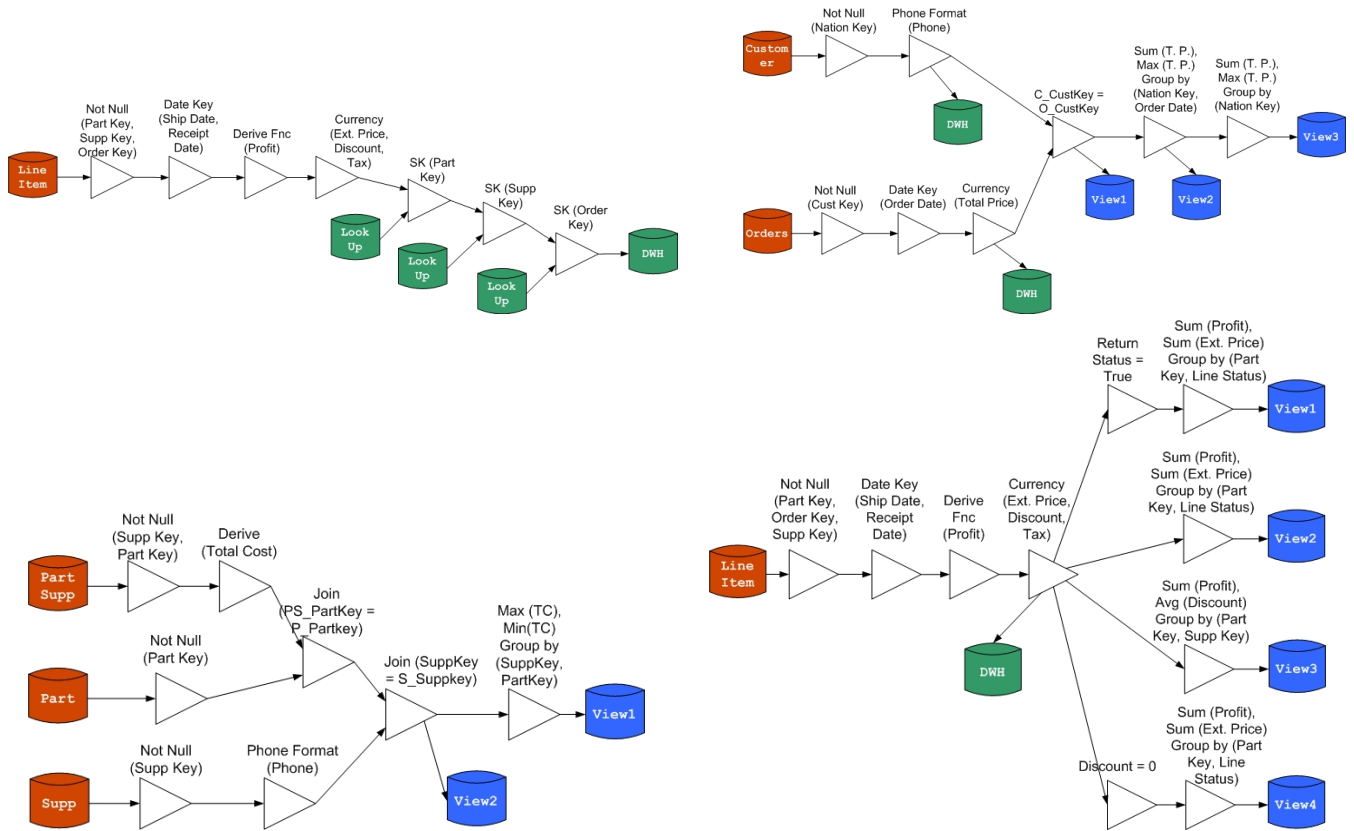


Figure 5: Selected workflows used in our experiments: (clockwise from top left) primary flow, wishbone, fork, and tree

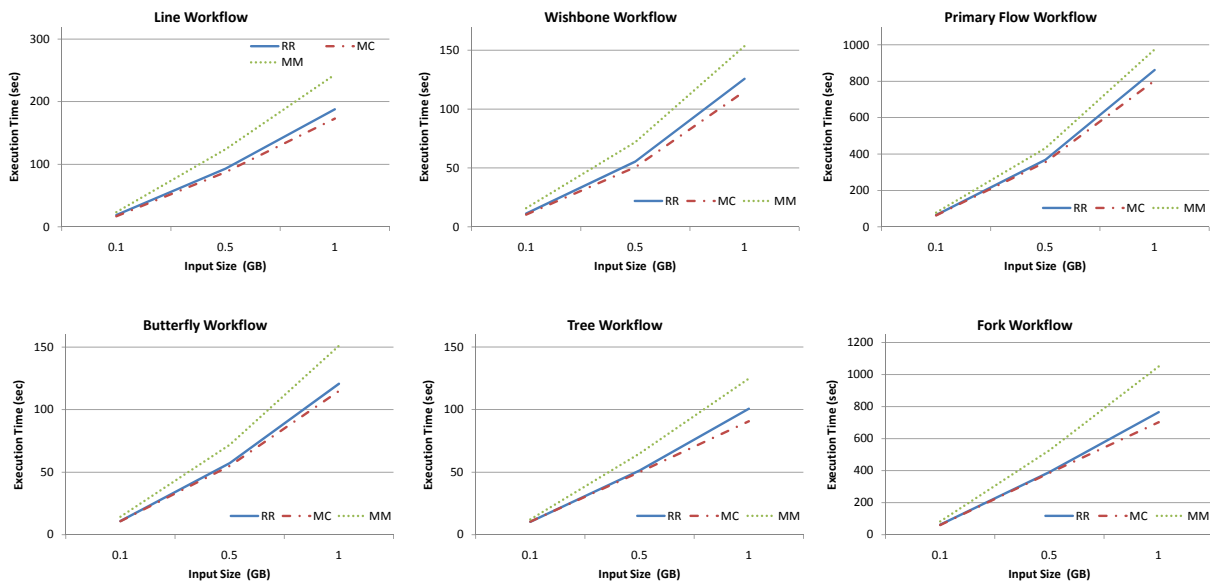


Figure 6: Effect of data size and scheduling protocol to execution time for different workflows

	RR	MC	MM
<b>TmSl</b> (ms)	0	0	70 (60-70)
<b>DQS</b>	100 (30-150)	100 (80-150)	100
<b>RPS</b>	400 (200-500)	400 (200-450)	400

**Table 3: Fine tuning for different scheduling policies**

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(EngineStallTime)*. 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 various micro-benchmarks on stall time, and on the aforementioned parameters too, we tuned the sleeping period in a reasonably small value of 4ms and used that value for all experiments.

The treatment of *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* than needs 150 msec to empty its data queue. If the time slot is 50 msec, 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. The results show that we can always find a stable region of values that perform best and thus, we used such stable values for the assessment of data size and selectivity effects. Table 3 depicts the values used and also, good value ranges (inside the parentheses) for time slot, queue, and row pack sizes. For additional details, we refer the interested reader to the long version of the paper [7].

All experiments have been conducted on a Dual Core 2 PC at 2.13 GHz with 1GB main memory and a 230Gb SATA disk. All findings reported here are based on actual executions and not simulations.

### 6.3 Experimental results with single policies

In this subsection, 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. 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. 6). Typically, the MINIMUM MEMORY algorithm behaves worst of all the others. MINIMUM COST is slightly better than the ROUND ROBIN algorithm. Still, the difference is small and this is mainly due to the fact that

the ROUND ROBIN 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. 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 MINIMUM MEMORY algorithm significantly outperforms the other two, with the ROUND ROBIN algorithm being worse than MINIMUM COST. The effect is the same if we vary data size (Fig. 7) or the selectivity of the workflow. 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 MINIMUM MEMORY are much more evident in these cases (bottom three graphs of Fig. 7), as this algorithm remains practically stable to its memory requirements independently of workflow type.

**Observations.** We used a real implementation –not a simulation– to evaluate three scheduling policies with respect to execution time and memory requirements. We used several ETL workflows as our experimental platform for assessing the effect of data size and workflow selectivity to the aforementioned measures. Overall, we argue that ROUND ROBIN does not perform satisfactory compared to the other two. In all cases, ROUND ROBIN behaves worse than MINIMUM COST in terms of memory consumption (frequently with significant differences), although it is quite close to the best values in terms of execution time. MINIMUM MEMORY manages to outperform the other two, when it comes to average memory requirements. MINIMUM MEMORY 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. Finally, the MINIMUM COST scheduling policy outperforms the other two policies concerning the execution time metric, in all cases.

Lessons learned:

- The state-of-practice tactics of round robin scheduling is quite efficient in terms of time behavior, but lags in memory consumption effectiveness.
- It is possible to devise a scheduling policy (i.e., MINIMUM COST) 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., MINIMUM MEMORY) can give significant earnings in terms of average memory consumption that range between 1/2 to 1/10 of the memory used by the other policies.

## 7. 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 losses. We have proposed an extensible architecture for implementing an ETL scheduler based on the pipelining of results produced by ETL operations. We have assessed a set of scheduling policies for the execution of ETL flows and

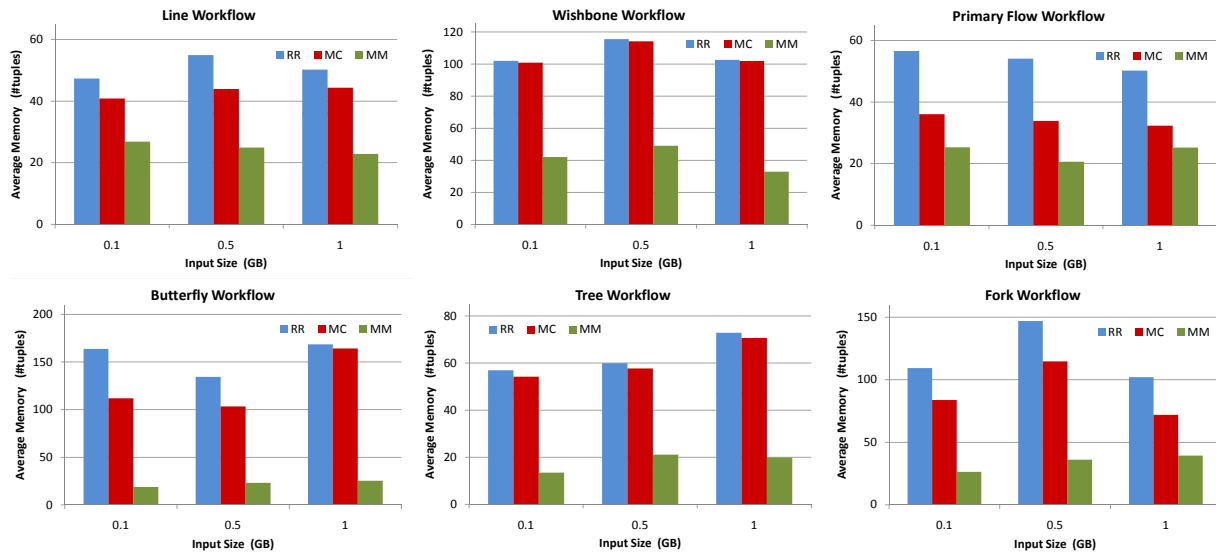


Figure 7: Effect of data size and scheduling protocol to average memory consumption

shown that a MINIMUM COST policy that aims at emptying the largest input queue of the workflow, typically performs better with respect to execution time, whereas a MINIMUM MEMORY policy that favors each time activities with the maximum tuple consumption rate, is better with respect to the average memory consumption.

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, 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 [18].

## 8. REFERENCES

- [1] J. Adzic and V. Fiore. Data warehouse population platform. In *DMDW*, 2003.
- [2] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 253–264, 2003.
- [3] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
- [4] P. J. F. Carreira, H. Galhardas, J. Pereira, and A. Lopes. Data Mapper: An Operator for Expressing One-to-Many Data Transformations. In *DaWaK*, pages 136–145, 2005.
- [5] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. Data integration flows for business intelligence. In *EDBT*, pages 1–11, 2009.
- [6] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *ICDE*, pages 1207–1210, 2009.
- [7] A. Karagiannis. Scheduling policies for the refresh management of data warehouses. Master’s thesis, Available at: <http://www.cs.uoi.gr>, 2007.
- [8] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Transaction reordering and grouping for continuous data loading. In *BIRTE*, pages 34–49, 2006.
- [9] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziouara. Benchmarking ETL workflows. In *TPCTC*, pages 199–220, 2009.
- [10] A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-Space Optimization of ETL Workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.
- [11] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL Design: Reducing the Cost of ETL Consulting Engagements. In *SIGMOD Conference*, pages 953–960, 2009.
- [12] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing ETL Workflows for Fault-Tolerance. In *ICDE*, pages 385–396, 2010.
- [13] M. Thiele, U. Fischer, and W. Lehner. Partition-based Workload Scheduling in Living Data Warehouse Environments. In *DOLAP*, pages 57–64, 2007.
- [14] C. Thomsen, T. B. Pedersen, and W. Lehner. RiTE: Providing On-Demand Data for Right-Time Data Warehousing. In *ICDE*, pages 456–465, 2008.
- [15] TPC. The TPC-H benchmark. Technical report, Transaction Processing Council. Available at: <http://www.tpc.org/tpch/>, 2007.
- [16] V. Tziouara, P. Vassiliadis, and A. Simitsis. Deciding the Physical Implementation of ETL Workflows. In *DOLAP*, pages 49–56, 2007.
- [17] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, 2001.
- [18] P. Vassiliadis and A. Simitsis. *Annals of Information Systems: New Trends in Data Warehousing and Data Analysis*, chapter Near Real Time ETL. Springer, 2009.
- [19] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos. A generic and customizable framework for the design of ETL scenarios. *Inf. Syst.*, 30(7):492–525, 2005.