

ΠΟΛΙΤΙΚΕΣ ΡΥΘΜΙΣΗΣ ΤΗΣ ΔΙΑΧΕΙΡΙΣΗΣ  
ΤΗΣ ΕΝΗΜΕΡΩΣΗΣ ΑΠΟΘΗΚΩΝ ΔΕΔΟΜΕΝΩΝ

Η  
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης  
του Τμήματος Πληροφορικής  
Εξεταστική Επιτροπή

από τον

Αναστάσιο Καραγιάννη

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ  
ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούλιος 2007



## **DEDICATION**

---

This thesis is dedicated to my parents who have supported me all the way since the beginning of my studies, allowing me to achieve my goals.



## **ACKNOWLEDGMENTS**

---

I would like to record my gratitude to my supervisor Dr. Panos Vassiliadis for guiding, supporting and motivating me, as well as, for the patience he showed throughout this research work. I also thank Alkis Simitsis for his constructive comments during the course of my research. Also, I would like to express many thanks to Vasiliki Tziouvara for providing me the initial ETL scenarios.

At the end of my thesis I would like to thank all those people who made this thesis possible and an enjoyable experience for me; especially my colleagues and friends, Eutixia, Fotini, Giannis and Tzeni for their help and encouragement throughout this work.



# CONTENTS

---

Dedication	iii
Acknowledgments	v
Contents	vii
List of Tables	ix
List of Figures	xi
Abstract	xiii
Περίληψη	xv
CHAPTER 1. Introduction	1
1.1. Introduction	1
1.2. Thesis Structure	6
CHAPTER 2. Related Work	9
2.1. Data Warehouses and ETL	9
2.1.1. Commercial studies and tools.	10
2.1.2. Research Studies	10
2.2. General Theory on Scheduling	12
2.2.1. Types of processor scheduling	12
2.2.2. Criteria	14
2.2.3. Preemption	16
2.3. Scheduling in data stream systems	16
2.3.1. Aurora Data Stream Manager	17
2.3.2. Chain Scheduling Policy	18
2.3.3. Pipeline Scheduling	19
2.3.4. Summaries of the studied algorithms	20
CHAPTER 3. System Architecture	23
3.1. General Idea	23
3.2. Description and logical representation of an ETL Scenario	24
3.3. Logical and physical perspective of an ETL scenario	27
3.4. Execution model and requirements for the ETL engine	29
3.5. Scheduler	32
3.6. Implementation of the logical level	32
3.7. Implementation of the physical level	34
3.7.1. Execution Item functionality	35
3.7.2. Monitor functionality and messages	39
3.7.3. Unary activities	40
3.7.4. Binary activities	41
3.7.5. Sorter	42
CHAPTER 4. Scheduling Algorithms	43
4.1. Problem formulation	43
4.2. Categories of algorithms	45

4.3. Round Robin	46
4.4. Minimum Cost	47
4.5. Minimum Memory	48
CHAPTER 5. Experiments	51
5.1. Measures and Parameters	52
5.2. Datasets	54
5.2.1. TPC-H	54
5.2.2. TPC-DS	55
5.3. Scenarios and data sources	56
5.3.1. Data Sources	57
5.3.2. ETL Scenarios	58
5.4. Tuning scheduling policies	65
5.4.1. Tuning Round Robin	66
5.4.2. Tuning Minimum Cost	69
5.4.3. Tuning Minimum Memory	71
5.5. Line workflow	74
5.5.1. Effect of input size	74
5.5.2. Effect of workflow selectivity	76
5.6. Wishbone workflow	77
5.6.1. Effect of input size	78
5.6.2. Effect of workflow selectivity	79
5.7. Primary flow workflow	81
5.7.1. Effect of input size	82
5.7.2. Effect of workflow selectivity	83
5.8. Balanced butterfly workflow	85
5.8.1. Effect of input size	85
5.8.2. Effect of workflow selectivity	87
5.9. Tree workflow	89
5.9.1. Effect of input size	89
5.9.2. Effect of workflow selectivity	91
5.10. Fork workflow	93
5.10.1. Effect of input size	93
5.10.2. Effect of workflow selectivity	94
5.11. Observations deduced from experiments	96
CHAPTER 6. Conclusions And Future Work	99
6.1. Conclusions	99
6.2. Future Work	100
References	103
Appendix	107
Short Biography	111



## LIST OF TABLES

---

Table 2.1 Summary table of scheduling algorithms for microprocessors [Sched06]	15
Table 2.2 Summary table of all stream scheduling algorithms	22
Table 3.1 The <i>Execute()</i> function of the <i>Execution Item</i> class	36
Table 3.2 The <i>DataProcess()</i> function of a <i>Reader</i>	37
Table 3.3 The <i>DataProcess()</i> function of a <i>Filter</i>	38
Table 3.4 The message types of the ETL engine	40
Table 4.1 Categories of scheduling algorithms	45
Table 4.2 Scheduling steps of the studied scheduling policies	46
Table 4.3 Scheduling steps of <i>Round Robin</i>	47
Table 4.4 Scheduling steps of <i>Minimum Cost</i>	48
Table 4.5 Scheduling steps of <i>Minimum Memory</i>	49
Table 5.1 Development environment	52
Table 5.2 Configuration of RR	68
Table 5.3 Configuration of MC	70
Table 5.4 Configuration of MM	74
Table A.1 Experiments from the Aurora Scheduler [CCR+03]	107
Table A.2 Experiments from the Chain Scheduler [BBDM03]	108
Table A.3 Experiments from the X-Join Scheduler [UrFr01]	109



## LIST OF FIGURES

---

Figure 1.1 Architecture of a Data Warehouse.	2
Figure 1.2 Extract - Transform - Load.	4
Figure 2.1 Typical template transformations provided by ARKTOS II.	12
Figure 2.2 An example of a data flow diagram [CCR+03].	16
Figure 3.1 Notation of the architecture graph [VSG+05].	24
Figure 3.2 Representation of an ETL scenario with the architecture graph.	26
Figure 3.3 Logical and physical level for the scenario elements.	28
Figure 3.4 Association of the logical and physical level [VSG+05].	29
Figure 3.5 Pipelined execution of an ETL scenario.	30
Figure 3.6 The class diagram of the logical level.	33
Figure 3.7 The class diagram of the physical level.	34
Figure 4.1 An example butterfly scenario.	46
Figure 5.1 The basic structure of a butterfly workflow.	54
Figure 5.2 The TPC-H relational schema.	55
Figure 5.3 The storage house relational schema.	57
Figure 5.4 The sales point relational schema.	58
Figure 5.5 A Line Scenario	59
Figure 5.6 A Wishbone Scenario	60
Figure 5.7 A Primary Flow Scenario	61
Figure 5.8 A Balanced Butterfly Scenario	62
Figure 5.9 A tree scenario	63
Figure 5.10 A fork scenario	64
Figure 5.11 Tuning DQS in the small line scenario (RR)	67
Figure 5.12 Tuning DQS in the butterfly scenario (RR)	68
Figure 5.13 Tuning RPS in the line scenario (RR)	69
Figure 5.14 Tuning RPS in the butterfly scenario (RR)	69
Figure 5.15 Tuning DQS in the butterfly scenario (MC)	70
Figure 5.16 Tuning RPS in the butterfly scenario (MC)	71
Figure 5.17 Execution time and $TmSl$ in the small line scenario (MM)	72
Figure 5.18 Execution time and $TmSl$ in the butterfly scenario (MM)	72
Figure 5.19 Max and avg memory and $TmSl$ in the small line scenario (MM)	73
Figure 5.20 Max and avg memory and $TmSl$ in the butterfly scenario (MM)	73
Figure 5.21 Execution time for a line scenario (Sel = 0.5)	75
Figure 5.22 Average memory for a line scenario (Sel = 0.5)	75
Figure 5.23 Maximum memory for a line scenario (Sel = 0.5)	76
Figure 5.24 Execution time for a line scenario (SF = 0.5)	76
Figure 5.25 Average memory for a line scenario (SF = 0.5)	77
Figure 5.26 Maximum memory for a line scenario (SF = 0.5)	77
Figure 5.27 Execution time for a wishbone scenario (Sel = 0.5)	78

Figure 5.28 Average memory for a wishbone scenario (Sel = 0.5)	79
Figure 5.29 Maximum memory for a wishbone scenario (Sel = 0.5)	79
Figure 5.30 Execution time for a wishbone scenario (SF = 0.5)	80
Figure 5.31 Average memory for a wishbone scenario (SF = 0.5)	81
Figure 5.32 Maximum memory for a wishbone scenario (SF = 0.5)	81
Figure 5.33 Execution time for a primary flow scenario (Sel = 0.5)	82
Figure 5.34 Average memory for a primary flow scenario (Sel = 0.5)	83
Figure 5.35 Maximum memory for a primary flow scenario (Sel = 0.5)	83
Figure 5.36 Execution for a primary flow scenario (SF = 0.5)	84
Figure 5.37 Average memory for a primary flow scenario (SF = 0.5)	84
Figure 5.38 Maximum memory for a primary flow scenario (SF = 0.5)	85
Figure 5.39 Execution time for a balanced butterfly scenario (Sel = 0.5)	86
Figure 5.40 Average memory for a balanced butterfly scenario (Sel = 0.5)	87
Figure 5.41 Maximum memory for a balanced butterfly scenario (Sel = 0.5)	87
Figure 5.42 Execution time for a balanced butterfly scenario (SF = 0.5)	88
Figure 5.43 Average memory for a balanced butterfly scenario (SF = 0.5)	88
Figure 5.44 Maximum memory for a balanced butterfly scenario (SF = 0.5)	89
Figure 5.45 Execution time for a tree scenario (Sel = 0.5)	90
Figure 5.46 Average memory for a tree scenario (Sel = 0.5)	90
Figure 5.47 Maximum memory for a tree scenario (Sel = 0.5)	91
Figure 5.48 Execution time for a tree scenario (SF = 0.5)	91
Figure 5.49 Average memory for a tree scenario (SF = 0.5)	92
Figure 5.50 Maximum memory for a tree scenario (SF = 0.5)	92
Figure 5.51 Execution time for a fork scenario (Sel = 0.5)	93
Figure 5.52 Average memory for a fork scenario (Sel = 0.5)	94
Figure 5.53 Maximum memory for a fork scenario (Sel = 0.5)	94
Figure 5.54 Execution time for a fork scenario (SF = 0.5)	95
Figure 5.55 Average memory for a fork scenario (SF = 0.5)	95
Figure 5.56 Maximum memory for a fork scenario (SF = 0.5)	96

## ABSTRACT

---

Karagiannis Anastasios. MSc, Computer Science Department, University of Ioannina, Greece. July, 2007. Scheduling policies for the refresh management of Data warehouses. Thesis Supervisor: Panos Vassiliadis.

Data Warehouses are collections of data coming from different sources, used mostly to support decision making and data analysis in an organization. To populate a data warehouse with up-to-date records that are extracted from the sources, special tools are employed, called *Extraction – Transform – Load* (ETL) tools, which organize the steps of the whole process as a workflow. An ETL workflow can be considered as a directed acyclic graph (DAG) used to capture the flow of data from the sources to the data warehouse. The nodes of the graph are activities that apply transformations or cleansing procedures on data or recordsets used for storage purposes. The edges of the graph are input/output relationships between the nodes. 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. Each activity of the workflow can be implemented physically, to be mapped to a set of software modules that can execute the ETL workflow.

This thesis proposes the design of an ETL workflow engine, in which all logical-level activities can be implemented with various algorithmic methods; every one with different cost in terms of time or system resources (e.g., main memory, disk usage). The system is easily expanded to support any possible activities. Another contribution of this thesis is the systematic study of tuning the execution of a workflow concerning its logical and physical characteristics; the size of the input data, the workflow complexity and selectivity, etc. Lacking of related research methodology the workflows that are used in the experimental methodology are grouped into fiducial

structures. Finally, the third contribution of this thesis is the suggestion of a well organized set of experimental scenarios is.

## ΠΕΡΙΛΗΨΗ

---

Αναστάσιος Καραγιάννης, του Γεωργίου και της Μαρίνας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος, 2007. Πολιτικές ρύθμισης της διαχείρισης της ενημέρωσης αποθηκών δεδομένων. Επιβλέπων: Παναγιώτης Βασιλειάδης.

Οι Αποθήκες Δεδομένων είναι συλλογές δεδομένων που προέρχονται από διαφορετικές πηγές και χρησιμοποιούνται κυρίως για τη λήψη αποφάσεων σε ένα οργανισμό. Για να τροφοδοτηθεί μια αποθήκη με νέα δεδομένα, όπως αυτά παράγονται στις πηγές, χρησιμοποιούνται εργαλεία Εξαγωγής – Μετασχηματισμού – Φόρτωσης δεδομένων (Extract – Transform – Load εργαλεία, ETL), τα οποία οργανώνουν τα επί μέρους βήματα της όλης διαδικασίας σαν μια ροή εργασίας. Μια ροή εργασίας ETL μπορεί να θεωρηθεί ως ένας κατευθυνόμενος ακυκλικός γράφος που χρησιμοποιείται για να αναπαραστήσει τη ροή δεδομένων από τις πηγές δεδομένων προς την αποθήκη δεδομένων. Οι κόμβοι του γράφου είναι διαδικασίες καθαρισμού/ μετασχηματισμού δεδομένων ή σύνολα εγγραφών και οι ακμές σχέσεις εισόδου/εξόδου μεταξύ των κόμβων. Η ροή εργασίας είναι ένα αφηρημένο σχήμα σε λογικό επίπεδο, το οποίο πρέπει να υλοποιηθεί σε φυσικό επίπεδο, δηλαδή να αντιστοιχηθεί σε ένα συνδυασμό από εκτελέσιμα προγράμματα που εκτελούν την ETL ροή εργασίας.

Στην εργασία αυτή, κατασκευάστηκε ένα σύστημα εκτέλεσης ροών εργασίας ETL, στο οποίο οι λογικού επιπέδου διαδικασίες της ροής εργασίας μπορούν να υλοποιηθούν με ποικίλες αλγοριθμικές μεθόδους, καθεμιά με διαφορετικό κόστος όσον αφορά απαιτήσεις σε χρόνο ή πόρους συστήματος (π.χ., μνήμη, χώρο στο δίσκο, κλπ.). Το σύστημα είναι εύκολα επεκτάσιμο σε σχέση με τις διαδικασίες που μπορεί να υποστηρίξει. Η αρχιτεκτονική του συστήματος είναι σχεδιασμένη με τέτοιο τρόπο,

ώστε να γίνεται αποδοτική χρήση των ενδιάμεσων δεδομένων, κάνοντας χρήση της τεχνικής της διοχέτευσης, όπου αυτό είναι εφικτό.

Μια περαιτέρω συμβολή της εργασίας είναι η συστηματική μελέτη της ρύθμισης της λειτουργίας μιας ροής εργασίας σε σχέση με λογικά και φυσικά χαρακτηριστικά της: όγκο πηγαίων δεδομένων, πολυπλοκότητα της δομής της ροής, επιλεκτικότητα στον όγκο των τελικών δεδομένων κλπ. Τρεις πολιτικές ρύθμισης προτείνονται σε αυτή την εργασία. Η μία είναι πολιτική Round Robin, γνωστή από το τομέα των λειτουργικών συστημάτων. Η δεύτερη είναι η *Minimum Cost*, όπου έχει ως στόχο τη μείωση του χρόνου εκτέλεσης. Τέλος, η τρίτη πολιτική ρύθμισης, *Minimum Memory*, μειώνει τις απαιτήσεις του συστήματος για μνήμη κατά τη διάρκεια εκτέλεσης ενός ETL σεναρίου.

Ολοκληρώνοντας, ελλείψει σχετικής πειραματικής μεθοδολογίας στη βιβλιογραφία, οι ροές που χρησιμοποιούνται στην πειραματική μελέτη της εργασίας οργανώνονται σε πρότυπες δομές, τύπου πεταλούδας. Η πρόταση ενός καλά σχεδιασμένου συνόλου πειραματικών σεναρίων για την μελέτη ροών εργασίας ETL είναι η τρίτη συμβολή της εργασίας.



# CHAPTER 1. INTRODUCTION

---

1.1 Introduction

1.2 Thesis Structure

---

## 1.1. Introduction

A Data Warehouse (DW) is an information infrastructure that collects, integrates and stores an organization's data. The most important feature of a Data Warehouse is that it produces accurate and timely management information, so companies utilize data warehouses to enable their employees (executives, managers, analysts, etc.) to make better and faster decisions. Furthermore, data warehouses can be used to support complex data analysis. According to Inmon [Inmo02], a DW is “a collection of *subject-oriented, integrated, non-volatile* and *time-variant* data in support of management decisions”.

W. H. Inmon [Inmo02] presents a formal definition of a data warehouse as a database consisting of computerized data that is organized to most optimally support reporting and analysis activity. According to Inmon, a data warehouse has four characteristics:

1. It is *subject-oriented*, meaning that the data in the DW is organized so that all data elements relating to the same real-world event or object are linked together.
2. *Integrated*, meaning that the database contains data from most or all of an organization's operational applications, and that this data is gathered in a single location to be made consistent.
3. *Non-volatile*, meaning that data in the database is never over-written or deleted, but retained for future reporting.

4. *Time-variant*, meaning that the changes to the data in the database are tracked and recorded so that reports can be produced showing changes over time.

There are many advantages of using a data warehouse. First of all, a data warehouse is able to combine a variety of data from different sources in a single location. Interesting information is extracted from various distributed sources, which are usually *heterogeneous*. This means that the same data is represented differently at the sources, for instance through different database schemata. The data warehouse has to identify same entities, represented in different ways at the sources, and model it under a unique database schema. This means that data in a data warehouse have to go through a series of transformations to be made consistent and up-to-date. This process is often referred to as *semantic reconciliation* and is an important property of the data warehouse. Another advantage of a data warehouse is that it can support changes to data, since modifications to the data in a data warehouse are tracked and recorded. The data warehouse also keeps a historical record of the loaded data.

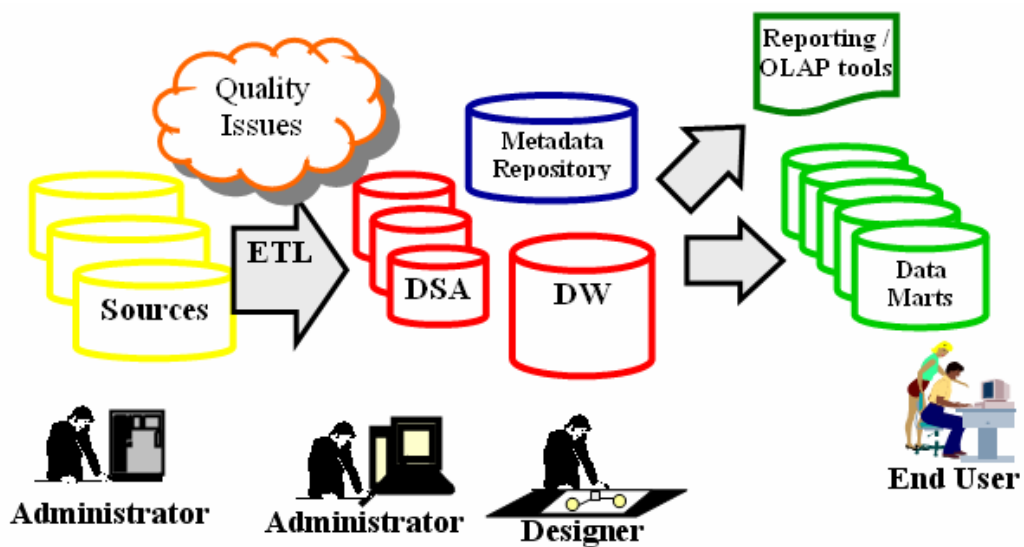


Figure 1.1 Architecture of a Data Warehouse.

Finally, *data quality* is an important issue, since data arriving at the data warehouse are in most cases inconsistent. The above features of a data warehouse show that a data warehouse is always expected to contain up-to-date, consistent and integrated

data in order to support decision making and data analysis. Figure 1.1 presents the architecture of a data warehouse.

1. The primary components of a data warehouse are Data Sources, Data Staging Area, Data Marts, the Metadata Repository, ETL and other reporting and OLAP applications.
2. *Data Sources* or *Operational Databases* are databases that store structured or unstructured data as part of the operational environment of a company or an organization. Data Sources supply the data warehouse with operational data. Data derived from various Sources are usually heterogeneous.
3. The *Data Staging Area (DSA)* is a smaller database used to store intermediate results produced by the application of cleansing techniques or transformations to the source data.
4. The *Data Warehouse* and the *Data Marts* are systems that store data provided to the users. The data in the warehouse are organized in *fact* and *dimension* tables. Fact tables contain the records with the actual information in terms of measured values, whereas dimension tables contain reference values for these facts. For example, assuming that a customer purchases a part for a certain price, the reference values for the customer and the part are stored (along with all their extra details) in the dimension tables, and the fact table records the references to these records (through foreign keys) along with the price paid. Data marts focus on a single thematic area and usually contain only a subset of the enterprise information. For example, a data mart may be used in a single department of the company and may contain only the data that is available to this department.
5. The *Metadata Repository* is a subsystem that stores information concerning the structure and the operation of the system. This information is called Metadata and concerns the ETL design and runtime processes.
6. *ETL (Extraction - Transformation - Loading)* applications extract the data from the sources, clean it and apply transformations over it before the loading of data to the data warehouse.
7. Finally, *reporting and OLAP tools* are reporting applications that perform OLAP and Data Mining tasks. OLAP tools form data into logical multi-dimensional structures and allow users to select which dimensions to view

data by. On the other hand, Data mining tools allow users to perform detailed mathematical and statistical calculations on data to detect trends, identify patterns and analyze data.

The process of moving data from the sources into a warehouse is performed in three steps:

- *Extraction* – is the process used to determine which data stored in the sources should be further processed and ultimately loaded to the data warehouse.
- *Transformation* – is the step in which data are adapted into the format required by the warehouse.
- *Loading* – is the process of populating the data into the warehouse.
- This process is normally abbreviated *ETL*. Figure 1.2 presents these three steps of an ETL process.

In order to manage the data warehouse operations, specialized tools are available in the market, called ETL tools. *ETL (Extraction-Transformation-Loading) tools* are a category of software tools responsible for the extraction of data from distributed sources, their cleansing and customization and finally their loading to the data warehouse ([VaSS02]).

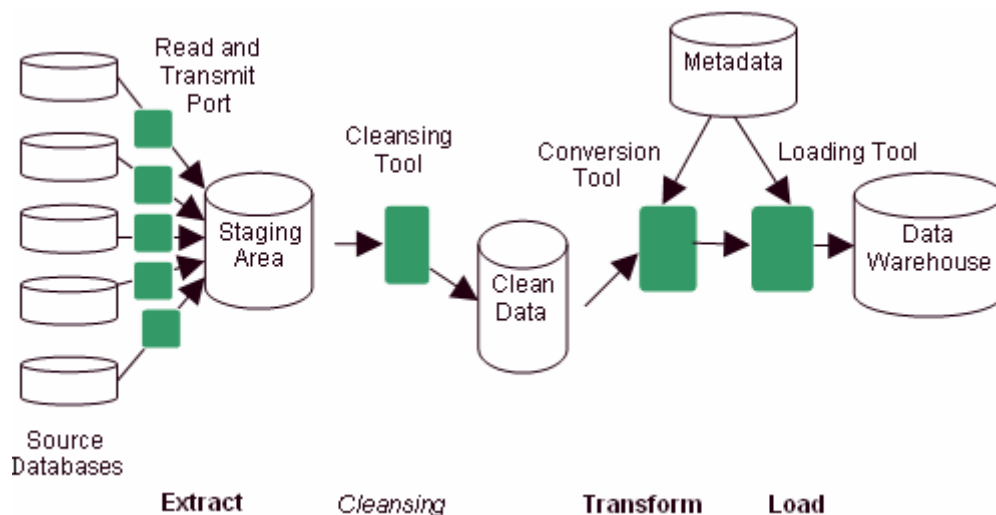


Figure 1.2 Extract - Transform - Load.

Their basic tasks are:

- the identification of relevant information at the source side
- the extraction of this information
- the customization and integration of the information coming from multiple sources into a common format
- the cleansing of the resulting data set, on the basis of database and business rules
- the propagation and loading of the data to the data warehouse and/or data marts.

As we mentioned earlier, in data warehousing, data are extracted from various sources and have to go through a set of transformations and cleansing procedures before they reach their destination, usually a data warehouse and/or data marts. Typical data transformations are data conversions (e.g., conversions from European formats to American and vice versa), orderings of data, generation of summaries of data (in other words groupings), etc. Finally, data are loaded into the data warehouse. A typical load of data involves processing *large volumes of data* (e.g., several GBs of data) and requires many complex transformations of data. This means that this process is *time-consuming* (often takes many hours or even days to complete) and usually takes place during the night, in order to avoid overloading the system with extra workload. Moreover, in many systems, the warehouse load must be completed within a certain time window, which means that the request for performance is pressing. Based on the above, we can summarize the main problems of ETL tasks: (a) the enormous volumes of data for processing, (b) performance, since all operations must be completed within a specific period of time, (c) quality problems, since data usually have to be cleansed. Furthermore, (d) failures during the transformation process or the warehouse loading process, cause significant problems to the warehouse operation and finally, (e) the evolution of the sources and the data warehouse can lead to daily maintenance operations. Under these conditions, we see that we can overcome the problems of ETL tasks by designing and managing ETL tasks efficiently.

This thesis makes the following three contributions in the research area of ETL tools:

- Our first contribution concerns the design and implementation of an execution engine of ETL scenarios. The elements of the ETL scenario are mapped from a logical level to a physical level. In other works, all logical-level activities and recordsets are mapped to the appropriate physical-level software modules. The execution engine provides the software components that a scenario needs. The ETL engine guarantees that all source data will be produced and there will be no data loss.
- We have designed and studied three scheduling algorithms. A scheduling mechanism is necessary to lead the execution towards optimizing a measure such as execution time or low memory requirements. The measures that are of interest in the case of ETL are execution time or low memory requirements. Our algorithms are the following:
  - *Round Robin*: A simple scheduling algorithm that assigns the activities to execute in FIFO order.
  - *Minimum Cost*: This algorithm improves the execution time of the execution by assigning for execution the activity that has more data to process (at the time of scheduling).
  - *Minimum Memory*: This algorithm reduces the execution's requirements for memory. At every time the activity that will consume the largest number of tuples.
- Our experiments suggest that *Minimum Cost* performs better than *Round Robin* in all cases; at the same time, *Minimum Memory* though is the most time consuming policy of all three policies, still *Minimum Memory* is the most efficient policy when it comes to average memory requirements. In most cases *Minimum Cost* has less average memory requirements than *Round Robin*.

## 1.2. Thesis Structure

This thesis consists of 6 chapters. Chapter 2 presents related research in the area of ETL tools. Also, we discuss the related work on scheduling in data stream systems, among with some basic principles in scheduling in the same chapter. In chapter 3, the architecture of the ETL engine is explained in detail among with the class diagrams of

the implementation. In chapter 4 the implemented scheduling algorithms are explained, along with examples of how they apply on a specific scenario. In chapter 5 we experimentally assess the studied scheduling algorithms. Finally in chapter 6 all results and conclusions are summarized and there is a discussion for future work.





## CHAPTER 2. RELATED WORK

---

### 2.1. Data Warehouses and ETL

### 2.2. General Theory on Scheduling

### 2.3. Scheduling in Data Stream Systems

---

The related work that concerns us is research on systems that process a great amount of data. Such systems are traditional ETL engines and data stream systems. It is common to data stream system to have a scheduler that will coordinate the query execution. In such systems we will emphasize in this chapter, since the aim for this thesis is the design for a scheduler for the Arktos project.

### **2.1. Data Warehouses and ETL**

Due to their importance and complexity, ETL tools constitute a multi-million market. There is a plethora of commercial ETL tools available. The traditional database vendors provide ETL solutions built in the DBMS's. In [SVSS07] and [SiVS05] there is a list with the most popular ETL market tools; we briefly mention them in the following section. Also, there have been research efforts towards the design and optimization of ETL tasks. We mention three research prototypes: (a) AJAX [GFSS00], (b) Potter's Wheel [RaHe01], and (c) ARKTOS II [VSG+05]. The first two prototypes are based on algebras, which are mostly tailored for the case of homogenizing web data; the latter concerns the modeling of ETL processes in a customizable and extensible manner, without the support, though, of an execution engine.

### *2.1.1. Commercial studies and tools.*

In terms of technological aspects, the main characteristic of the area is the involvement of traditional database vendors with ETL solutions built in the DBMS's. The three major database vendors that practically ship ETL solutions "at no extra charge" are pinpointed: Oracle with Oracle Warehouse Builder [Oracle07], Microsoft with Microsoft with SQL Server 2005 Integration Services (SSIS) (the next version of Data Transformation Services in MS-SQL Server 2000) [SSIS07] and IBM with the Data Warehouse Center [IBM07]. Still, the major vendors in the area are Informatica's Powercenter 8 [Infrm07] and Ascential's DataStage suites [Asc03] (the latter being part of the IBM recommendations for ETL solutions). As a general comment, we emphasize the fact that the former three tools have the benefit of the minimum cost, because they are shipped with the database, while the latter two have the benefit to aim at complex and deep solutions not envisioned by the generic products. The aforementioned discussion is supported from a second recent study [Gart03], where the authors note the decline in license revenue for pure ETL tools, mainly due to the crisis of IT spending and the appearance of ETL solutions from traditional database and business intelligence vendors. The Gartner study discusses the role of the three major database vendors (IBM, Microsoft, Oracle) and points that they slowly start to take a portion of the ETL market through their DBMS-built-in solutions.

### *2.1.2. Research Studies*

The AJAX [GFSS00] system deals with typical data quality problems, such as the object identity problem, errors due to mistyping and data inconsistencies between matching records. This tool can be used either for a single source or for integrating multiple data sources. AJAX provides a framework wherein the logic of a data cleaning program is modeled as a directed graph of data transformations that start from some input source data. AJAX also provides a declarative language for specifying data cleaning programs, which consists of SQL statements enriched with a set of specific primitives to express mapping, matching, clustering and merging

transformations. Finally, an interactive environment is supplied to the user in order to resolve errors and inconsistencies that cannot be automatically handled and support a stepwise refinement design of data cleaning programs.

The Potter's Wheel [RaHe01] system is targeted to provide interactive data cleaning to its users. The system offers the possibility of performing several algebraic operations over an underlying data set, including format (application of a function), drop, copy, add a column, merge delimited columns, split a column on the basis of a regular expression or a position in a string, divide a column on the basis of a predicate (resulting in two columns, the first involving the rows satisfying the condition of the predicate and the second involving the rest), selection of rows on the basis of a condition, folding columns (where a set of attributes of a record is split into several rows) and unfolding. Optimization algorithms are also provided for the CPU usage for certain classes of operators. The general idea behind Potter's Wheel is that users build data transformations in an iterative and interactive way; thereby, users can gradually build transformations as discrepancies are found, and clean the data without writing complex programs or enduring long delays.

Arktos II [VSG+05] is a coherent framework for the conceptual, logical, and physical design of ETL processes. The uttermost goal of this line of research is to facilitate, manage and optimize the design and implementation of the ETL processes both during the initial design and deployment stage, as such during the continuous evolution of the data warehouse. To this end, in [VaSS02] and [SVSS03] a conceptual model is proposed. Further, in [SVSS03] a logical model is presented. The proposed models, conceptual and logical, are constructed in a customizable and extensible manner, so that the designer can enrich them with his own re-occurring patterns for ETL processes. Therefore, Arktos II offers a palette of several templates, representing frequently used ETL transformations along with their semantics and their interconnection (Figure 2.1). In this way, the construction of ETL scenarios, as a flow of these transformations, is facilitated.

<b>Filters</b>	<b>Unary transformations</b>	<b>Binary transformations</b>
Selection ( $\sigma$ )	Push	Union (U)
Not null (NN)	Aggregation ( $\gamma$ )	Join ( $\bowtie$ )
Primary key violation (PK)	Projection ( $\pi$ )	Diff ( $\Delta$ )
Foreign key violation (FK)	Function application (f)	Update Detection ( $\Delta_{\text{UPD}}$ )
Unique value (UN)	Surrogate key assignment (SK)	
Domain mismatch (DM)	Tuple normalization (N)	<b>Composite transformations</b>
	Tuple denormalization (DN)	Slowly changing dimension (Type 1,2,3)(SDC-1/2/3)
<b>Transfer operations</b>		Format mismatch (FM)
Ftp (FTP)	<b>File operations</b>	Data type conversion (DTC)
Compress/Decompress (Z/dZ)	EBCDIC to ASCII conversion (EB2AS)	Switch ( $\sigma^*$ )
Encrypt/Decrypt (Cr/dCr)	Sort file (Sort)	Extended union (U)

Figure 2.1 Typical template transformations provided by ARKTOS II.

## 2.2. General Theory on Scheduling

This section contains some general theory about scheduling, which derives from the operating system research. Also the terminology that is used is very close to operating system theory; in operating systems scheduling is among processes and not activities. We discuss the basic types of processor scheduling, fundamental principles and criteria that characterize these algorithms. Moreover we mention a few well known simple algorithms such as FIFO, Round Robin etc. There is a brief description of these algorithms in Table 2.1 [Sched06], at the end of this section.

### 2.2.1. Types of processor scheduling

There are three different types of scheduling [UnSched07], identified by the size of the time fragment that the scheduler provides to each process.

- **Long-term scheduling** is performed to decide if a new process is to be created and be added to the pool of processes. Long-term scheduling controls the degree of multiprogramming. The more processes that are created, the smaller is the percentage of time that each process can be executed. Thus, the

long term scheduler may limit the degree of multiprogramming to provide satisfactory service to the current set of processes. Whenever a process terminates, or the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked. The decision may be made on a first-come-first-served basis or it can be a tool to manage system performance. For example, if the suitable information is available, the scheduler may attempt to keep a mix of processor-bound and I/O-bound processes. A processor-bound process is one that mainly performs computational work and occasionally uses I/O devices, while an I/O-bound process is one that uses I/O devices more than the microprocessor.

- **Medium-term scheduling** is a part of the swapping function of the operating system. In operating systems, in order to increase the amount of total memory the idea of virtual memory is used. This technique increases the resources of a computer in main memory by using some disk space also. When a process is idle there is no use to keep it loaded in the main memory. So, the process is copied to a file (swap file) and the freed space in memory is then available to the system. The way virtual memory is handled can affect the performance of a system. The scheduler can decide if a process should be loaded into the main memory either completely or partially so as to be available for execution and improve the system's performance.
- **Short-term scheduling** is the most common use of the term scheduling, i.e. deciding which ready process to execute next. The short-term scheduler, also known as the dispatcher, is invoked whenever an event occurs that may lead to the suspension of the current process or that may provide an opportunity to preempt a currently running process in favor of another. Examples of such events include:
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals

### 2.2.2. Criteria

There are various algorithms available for the short-term scheduling work. Each scheduling algorithm is built in such a way that one or more fundamental criteria are best served by it. The major criteria relating to processor scheduling are as follows:

- **Turnaround time** is the interval of time between the submission of a process and its completion. This is an appropriate measure for a process in a batch operating system.
- **Response time** is the elapsed time between the submission of a request and the moment the response appears.
- **Throughput** is the rate at which processes are completed. The scheduling policy should attempt to maximize the throughput so that more tasks could be performed.
- **Processor utilization** is the percentage time that the processor is busy. For a shared system, this is a significant criterion, while in single-user systems and real-time systems, this criterion is less important than some of others.
- **Fairness** addresses whether some processes suffer starvation. Fairness should be enforced in most systems.

These criteria may be categorized into two groups: user-oriented and system-oriented. The former group focuses on the properties that are visible and of interest to the users. For example, in an interactive system, a user always wishes to get response as soon as possible. This may be measured by response time. Some criteria are system oriented, focusing on effective and efficient utilization of the processor, such as throughput. System-oriented criteria are usually important on multi-user operating systems, while on the single-user system, it is probably not important to achieve high processor utilization or high throughput as long as the single user's need is fully met. It is obvious that the above criteria are interdependent and cannot be optimized simultaneously. For example, providing good response time may require a scheduling algorithm that switches between processes frequently, which increases the overhead of the system, reducing throughput. In a particular operating system, some criteria

may be of more importance than others, thus the designer of the operating system may simply focus on improving those concerned aspects.

Table 2.1 Summary table of scheduling algorithms for microprocessors [Sched06]

<b>FCFS</b> (First Come First Served, also known as FIFO)	The first ready task is executed first until it is done. The next one is the second ready task and so on
<b>RR</b> (Round Robin)	Every ready task is kept in a queue and they take control of the CPU for a while. Another version is VRR (Virtual Round Robin), where blocked tasks from I/O are put in another queue and the system gives them the remaining time of their time slice.
<b>SPN</b> (Shortest Process Next)	In this algorithm every task has a priority. The one that is expected to need the least CPU time to finish has the bigger priority. It is not easy to tell the remaining time of a task. There are not time slices here
<b>SRT</b> (Shortest Remaining Time)	This algorithm is similar to SPN but the running task might be interrupted when a new task is ready for execution and the new task will finish sooner than the running task.
<b>HRRN</b> (Highest Response Ratio Time)	This algorithm has a simple formula calculating the priorities of all tasks, favoring those that have the smallest remaining execution time. It seems better than the two above because in the formula there is estimated the time a task waits to get the CPU. This way starvation is avoided. There are no time slices, a task gets the CPU only when the active has finished or blocked (due to I/O).
<b>Feedback</b> (with q as the number of priority queues)	This algorithm keeps a number of priority queues and places tasks to one of these queues. The new tasks are put in first queue, which is the one with the higher priority. This algorithm is preemptive and uses time slices. When the time slice is finished, the scheduler picks one task from the first queue (biggest priority), and if it is empty it goes to the next queue. The task that has been interrupted gets a lower priority and is put to the appropriate queue. For example a new task is put at first to the first queue, and the second time (when its time slice was finished) will be put to the second queue and so on. This algorithm could possibly lead some tasks to starvation

### 2.2.3. Preemption

Another issue relating to scheduling is whether a running process could be preempted or not. There are two categories:

- **Non-preemptive:** In this case, a running process continues to execute until (a) it terminates or (b) blocks itself to wait for I/O or to request some operating system service.
- **Preemptive:** The currently running process may be interrupted and moved to the "ready" state by the operating system. The preemption may possibly be made due to the arrival of a new process, or the occurrence of an interrupt that places a blocked process in the "ready" state.

Preemptive policies incur greater overhead than non-preemptive ones but may be preferred since they prevent some processes from monopolizing the processor for a long time.

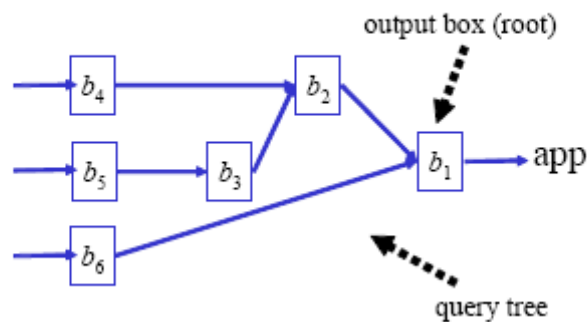


Figure 2.2 An example of a data flow diagram [CCR+03].

## 2.3. Scheduling in data stream systems

Data stream systems process great amounts of continuous data that derive from sensor networks, position tracking, fabrication line management, network management, and financial portfolio management, where data come in continuous and asynchronous fashion, in volumes and rates so high that it is not possible to store them in a



traditional DBMS. Because of their idiomorphic nature, data stream systems must perform basic operations such as selections (filters in ETL and data streaming terminology) and joins without the service of a DBMS system.

The sequence of the applied operators in one or more input streams defines a data flow diagram (Figure 2.2). One issue that rises is how these operators will be executed. Two basic patterns can be proposed for the design of an execution engine for data stream systems. One is to have one thread per operator, and all operations are executed simultaneously. The second pattern is to execute one operator at a time, so using one single thread is sufficient. In either case having a scheduler that will coordinate the execution of the query, even a naive scheduler that will apply a FIFO or a Round Robin scheduling policy, is necessary. More advanced scheduling policies are essential because in most cases some extra requirements must be met. These requirements typically involve the (a) minimization of memory usage, (b) response time and (c) execution time. The related work that is presented throughout this chapter concerns of scheduling the execution of streams in some well known stream systems. We specifically focus on the design of their scheduler, and how the requirements mentioned above are accomplished.

### *2.3.1. Aurora Data Stream Manager*

The Aurora stream manager [CCR+03] has three techniques for scheduling operators in streams, for minimizing execution time (MC), latency time (ML) and memory (MM). The Aurora system can execute more than one query (continuous queries) for the same input stream(s). Every stream is modeled as a graph with operators. Scheduling each operator separately is not very efficient, so the notion of a superbox is introduced. A superbox is a sequence of boxes that is scheduled and executed as an atomic group. A superbox is not necessarily a whole query.

There is a two-level scheduling algorithm for the Aurora stream manager. The first level is to decide which superbox to execute, while the second level is to schedule the operators inside the selected superbox. There are two ways to deal with this problem.

Specifically, at the first level, the scheduler chooses dynamically or statically the next superbox. The static approach is rather simple, a single superbox is pre-defined for every query, and a scheduling policy can be applied (e.g., round robin) for selecting every time which superbox to execute. The dynamic approach defines at run time which will be the next superbox to execute. In [CCR+03] the static approach is used. Three strategies are proposed for the second level, to minimize the execution time (MC), the latency time (ML) and the memory consumption (MM).

The minimum cost (MC) strategy serves the basic idea of minimizing the number of box calls per output tuple. This means that every operator will be executed only if the preceded operators are already scheduled. Every operator is scheduled only once.

The minimum latency (ML) strategy uses a metric called output cost whose value is an estimate of the latency incurred in producing some output data and processing them to all following operators of the stream, until they reach the streams final output. Each time, the operator with the smallest output cost is selected.

The minimum memory (MM) strategy tries to maximize the data consumption per time unit. In other words it yields the maximum increase of the available memory. The formula that is used estimates the memory reduction rate per operator. The operator with the largest value is selected.

### 2.3.2. Chain Scheduling Policy

The Chain [BBDM03] scheduler reduces the required memory when executing a query in a data stream system. [BBDM03] 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 selects a group of operators instead of one. The authors use a progress to explain the functionality of their scheduler. The horizontal axis of the progress chart represents time and the vertical axis represents tuple size. The chart contains operator points. The operators that participate in the execution create an operator path, which is the flow of data

during the execution of the query. Every time the scheduler runs, a part of the operator path is selected dynamically. To accomplish this, the scheduler must get a snapshot of the system. The progress chart is refreshed and demonstrates the current state of the system; the selectivity of every operator and its input. Based on a mathematical formula some adjacent points are grouped. The first and the last point of the group are connected with a dashed line. Every such group is called as a lower envelope. The steepness of every line indicates how effective each group would be if it is set for execution.

The scheduling strategy is rather simple, every time they select the steepest lower envelope. The system makes sure that there are no tuples in the middle of any operator group. This makes possible to treat all lower envelopes (operator groups) as single units of processing. In other systems the basic idea to decrease the required memory is to select one operator that has the biggest data consumption. In this work this idea is expanded a little by selecting a group of operators instead.

### *2.3.3. Pipeline Scheduling*

[UrFr01] presents two scheduling algorithms when pipelining is employed in query execution. Both algorithms aim to improve the system's response time; therefore it is necessary that all operators are non-blocking. The scheduler needs to compute the output rate of every operator in the stream and select the one with the highest rate.

At first three non-blocking join operators SHJ, DPHJ and XJOIN are discussed. These operators have one, two and three stages of execution. In every stage, the XJOIN operator has a different behavior, and this is something that affects the scheduler. In every stage, the scheduler must use different formulas to estimate correctly the operators output rates.

The authors propose a rate based algorithm, which schedules streams, rather than operators. A stream is considered as an execution unit which consumes tuples and produces output data. In every execution of the scheduler the stream with the biggest

output rate is selected. The scheduler runs every one second. If the selected stream finishes in less than one second, the next stream that is selected is based the previous estimates of the scheduler.

A second approach on this problem is also discussed; here the authors consider that some data are more important than other, based on the preference of the user (an ORDER BY clause in the query). The second algorithm presented can schedule streams in a way that important data will be preceded than others in the execution, and reach the final output first. Every tuple is assigned a rank which shows how important this tuple is. The important data are favored at the join operators, will others are put aside for some time. There are two formulas to estimate the importance of data, CM and AM. Two variants of the algorithm are presented, called SIP and SJP. The above formulas can be applied in both cases. In SIP when a tuple arrives there is a check on its rank and it is compared with a random value. If the tuple's value is greater it is processed. SJP works in a similar manner with the difference that it decides to process the tuple, not when the tuple arrives but when it is about to be joined.

#### *2.3.4. Summaries of the studied algorithms*

In this summary we present all the scheduling algorithms we studied in related work papers. All algorithms have some common properties, which are presented in Table 2.2. Our classification is performed through the following axes:

- **Who is next:** This dimension presents the parameter or parameters each scheduler uses to select the next operator.
- **For how long:** This dimension tells us whether each algorithm is dependent on the use of time slot or not. In the latter case, an operator typically becomes idle if all its input is consumed or its output queue is full. Some algorithms can incorporate both criteria in the calculation of the duration of the execution of an operator. Concerning the preemptiveness property, one could possibly argue that time-slot based algorithms are preemptive in a sense, since their execution is stalled whenever they reach their designated deadline. Still, since the most clear case of non- preemptiveness is the case where each operator

consumes all its input, possibly stores it, and then passes the execution to the next operator, it is clear that several degrees of preemptiveness can be considered.

- **Criterion:** In this dimension we can see the criterion each algorithm tries to favor.
- **Decision:** Some algorithms base their decision on each operator's condition only, while others need to consider more than one operator to make a decision. For example **MC** checks the input size of every operator, while **ML** for every operator needs to know the output rates of its successors in the stream.
- **Parameters:** This dimension presents the parameters that every algorithm requires for its decision.

For an ETL engine the criteria *fairness*, *execution time* and *memory consumption* are important and we provide a scheduling algorithm for each criterion. Our scheduling policies are explained in detail in chapter 4. The criterion *response time* is not appropriate for an ETL engine because response time mostly concerns interactive query processes where an end user is involved, while the ETL setting we are interested in, involves off-line refreshment of the warehouse. Moreover, the presence of blocking activities, such as aggregator and join, eliminates any chances to improve the system's response time.

Table 2.2 Summary table of all stream scheduling algorithms

<b>Name</b>	<b>Source</b>	<b>Who Is Next</b>	<b>For How Long</b>	<b>Criterion</b>	<b>Decision</b>	<b>Parameters</b>
FIFO	[BBDM03], [UrFr01]	next token	until idle / time slot	Fairness	Local	operator ID
Round Robin	[BBDM03], [UrFr01]	next ready token	until idle / time slot	Fairness	Local	operator ID
Equal Time	[UrFr01]	least executed time	until idle / time slot	Fairness	Global	execution time
Cheapest First	[UrFr01]	least processing cost	until idle	response time	Local	processing cost
Greedy Scheduling	[BBDM03]	least selectivity	time slot	memory consumption	Local	selectivity
Min Latency	[CCR+03]	largest output size	until idle	response time	Global	selectivity, cost
Rate Based	[UrFr01]	largest output size	until idle	response time	Global	selectivity, cost
Min Cost	[CCR+03]	largest input size	until idle	execution time	Local	input size
Min Memory	[CCR+03]	largest data consumption	until idle	memory consumption	Local	input size, selectivity, cost
Chain Scheduling	[BBDM03]	largest data consumption	time slot	memory consumption	Global	input size, selectivity, cost

## CHAPTER 3. SYSTEM ARCHITECTURE

---

- 3.1. General Idea
  - 3.2. Description and logical representation of an ETL Scenario
  - 3.3. Logical and physical perspective of an ETL scenario
  - 3.4. Execution model and requirements for the ETL engine
  - 3.5. Scheduler
  - 3.6. Implementation of the logical level
  - 3.7. Implementation of the physical level
- 

This thesis focuses on the design and the implementation of a parametric ETL system, in which simple and complex ETL scenarios can be defined and executed. The user of this ETL execution engine is able to define the scenarios easily. In this thesis we centre our efforts to implement the execution model of this system. Another basic goal is to design a scheduler for this system, able to tune the execution of the data cleaning and transformations, based on an operating policy the user has selected.

### **3.1. General Idea**

When starting the design of an ETL engine, we must consider a few basic issues. At first, we should provide some functionality to the user, so that he will be able to define a scenario, and all of the components that compose a complete ETL scenario. The definition of the scenario will keep a level of abstraction, so that some of the implementation details will not be a part of it. Then, we need to design the logical representation of these components. We also need to design a model for the physical representation of the same objects, in which implementation and execution details will

be important, and an execution engine which will be responsible for the correct execution the ETL scenario.

### 3.2. Description and logical representation of an ETL Scenario

As mentioned before, the definition of an ETL scenario is a composition of the definitions of the elements that form a scenario, as well with its respective parameters. At this point we will describe its basic components, and also the logical model on which our definition is based.

The execution of a scenario can be divided into three basic steps. At first, data are extracted from several data sources (text files, databases, etc), then certain transformation, cleaning or integration operations are applied on the input data, and finally the processed data are put into a data warehouse.


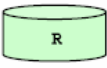



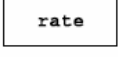



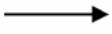
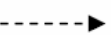
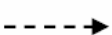

Data Types	Black ellipsoid		RecordSets	Cylinders	
Function Types	Black rectangles		Functions	Gray rectangles	
Constants	Black circles		Parameters	White rectangles	
Attributes	Unshaded ellipsoid		Activities	Triangles	
Part-Of Relationships	Simple lines with diamond edges*		Provider Relationships	Bold solid arrows (from provider to consumer)	
Instance-Of Relationships	Dotted arrows (from instance towards the type)		Derived Provider Relationships	Bold dotted arrows (from provider to consumer)	
Regulator Relationships	Dotted lines				

Figure 3.1 Notation of the architecture graph [VSG+05].



The basic structure of an ETL scenario consists of data sources and targets, which we will refer to them as recordsets, and a set of operations that are performed on the data which we will refer to them as activities. The activities are the transformation, cleaning or integration operations, while the recordsets are the places where data is either extracted or loaded by the system. An activity can be a filter, a join or an aggregation operator. In an ETL scenario the activities that are applied on some data have a specific execution order. Therefore it is important to define the order the activities are executed and we can treat an ETL scenario as a composite workflow. The full layout of an ETL scenario, involving activities and recordsets can be modeled by a graph, which we call the architecture graph [VSG+05], in which all the details relating to the ETL scenario are enclosed.

The architecture graph is directed and acyclic. The direction of the graph represents the flow that the input data will follow inside the ETL scenario. The nodes of the architecture graph will be the activities and the recordsets, while the edges will provide information for the flow of the processed data, and which node (activity or recordset) will work as a data provider for another node (the data consumer). For every activity or recordset, we need to set some properties. An activity can be defined as an entity with possibly more than one input schemata, an output schema and a list of parameters that specify the current activity. A recordset, can be defined as an entity with one input (or output) schema, and a parameter list that identify the data source or target. The edges of the graph describe the relationships between the nodes. There is more than one type of edges in the architecture graph. The basic relationship is the provider relationship, which illustrates the provider-consumer relationships between the activities and recordsets of the scenario. The schemata of the data are also shown in the architecture graph. The part-of relationship associates each schema with one activity or recordset. The regulator relationship shows the relation between attributes of the input and output schemata of an activity. The complete notation for the architecture graph is shown with detail in Figure 3.1 [VSG+05]. A complete definition of a complex scenario might give us a heavy and overloaded representation of the graph. It is not expected for the user to fully design the graph, but the graph is used mainly to give the user a graphic perspective of the scenario.

In Figure 3.2, a simple ETL scenario is shown with the use of the notation described. In this figure some details are omitted, so that the reader can understand the actual scenario, as well as the basic structure of the architecture graph.

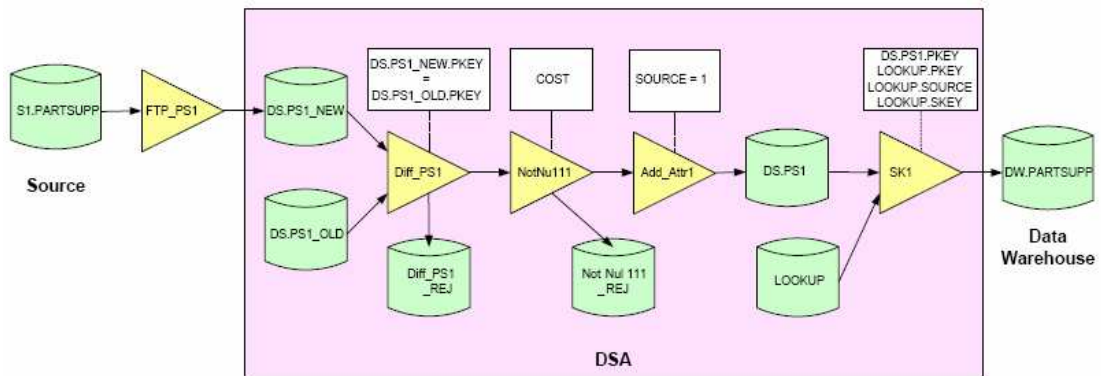


Figure 3.2 Representation of an ETL scenario with the architecture graph.

The scenario that is presented in Figure 3.2 is mainly composed by four activities. We will now briefly describe this scenario. The big rectangle on the figure is the data staging area (DSA).

The first thing to do is to start extracting data from our data source. Depending on the scenario, it is not necessary to always wait for all input data to be loaded. Since input data are loaded, or have started loading in the DSA, the first activity will separate the new tuples from the old ones. It is supposed that we have extracted data from the same source at some point in the past, and now we want to process only the tuples that have been created since. So, the first activity will reject all the tuples that have been already loaded in the system. The remaining tuples are (a) persistently stored for checkpointing reasons and (b) passed to the next activity for further processing.

The next activity will perform a null-check on the new tuples, on the attribute "cost". All tuples that have the NULL value at this field will be rejected.

When extracting data from more than one source, it is very useful to add an attribute to each tuple indicating the data source. The third activity does that operation, adding one attribute to each tuple, in this case the integer value 1.

The last activity of the scenario adds a surrogate key to each tuple. The need for a global key for all the tuples is pretty much clear. All tuples have a primary key from their source. Usually each source has a different data type for a primary key, or if it is of the same type, it is most likely that the same value (e.g. id = 5) is already assigned to more than one tuple. Since all tuples will be placed into the same table their existing primary key can not be used. The activity has a lookup table, and also uses the attribute added from the previous activity to provide each tuple with a unique value, which usually is an integer for performance issues.

We have now defined the logical model to represent an ETL scenario. We must provide the user with some functionality so that he can be able to easily create his own scenarios. We could use a graphic environment in which the user actually sketches the architecture graph [VSG+05]. Another way to do so is to use a declarative language for the definition of the ETL scenario. In [VVS+01] the SADL language is proposed. A variant of this language can be used, in which the user will specifically define the nodes and the edges of the graph, in terms of activities, recordsets, schemata etc.

### **3.3. Logical and physical perspective of an ETL scenario**

The model we just described briefly has a certain level of abstraction for all the elements of the ETL scenario. As mentioned before, we follow a traditional approach and group the design elements into logical and physical, with each category comprising its own perspective. At the logical perspective, we classify the design elements that provide an abstract description of the workflow environment, where as in the physical perspective all the design elements enclose the details and parameters required for their execution. In other words, the activities defined at the logical layer (in an abstract way) are materialized and executed through the specific software modules of the physical perspective.

Since we have decided that the logical and physical level to be independent, we need a mechanism that given as input the logical representation of an ETL scenario, it will provide us the respective physical representation. This mechanism is responsible for the correct and efficient mapping of logical objects to the respective physical objects, which are the appropriate software modules that exist inside the system. For example, this mechanism is in charge of to decide which join operator should be selected (Nested Loops Join, Merge Join, etc) when an activity in the logical level is defined as join. Also, inside this mechanism we could integrate an optimizer in order to achieve a different representation of the objects at the physical level, which will possibly lead to a faster execution of the scenario. Designing an optimizer for this system is not a part of this thesis. In Figure 3.3 there is a simple sketch illustrating this general mechanism and how it interacts with other parts of the system.

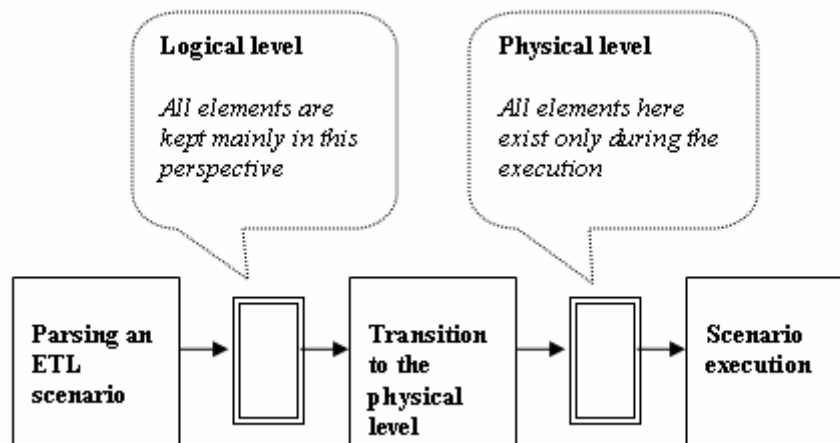


Figure 3.3 Logical and physical level for the scenario elements.

In order to correctly depict the design elements to the physical level, a set of template classes can be used [VSG+05]. The objects that exist in the physical perspective of the ETL scenario are instances of these template classes. In Figure 3.4 the mapping between the logical level and the physical level appears through the template classes. When the mapping process is completed, the execution of the ETL scenario can be initiated.

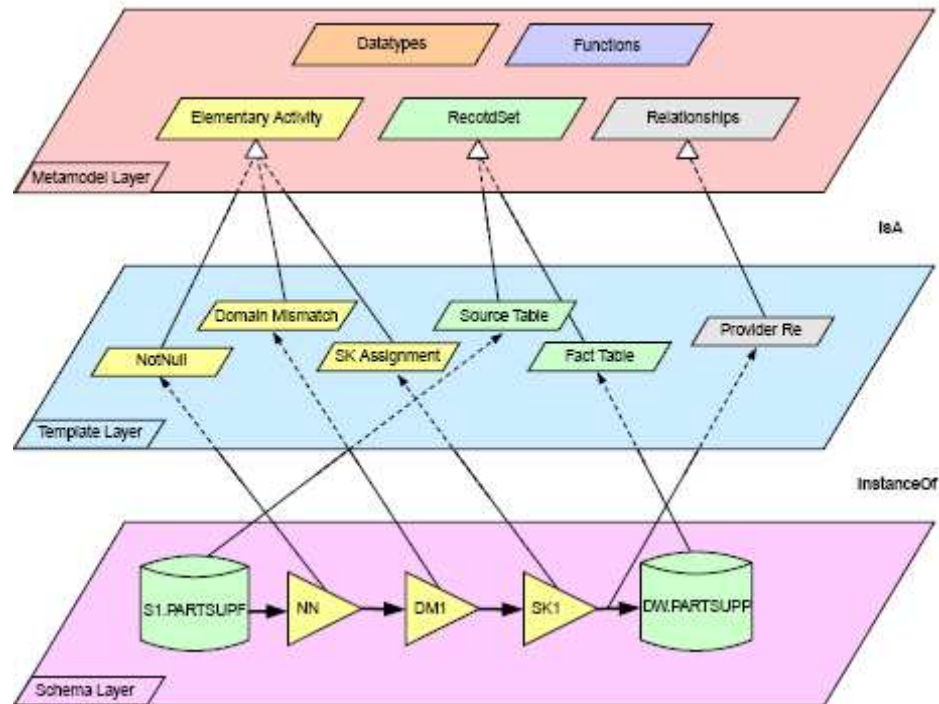


Figure 3.4 Association of the logical and physical level [VSG+05].

### 3.4. Execution model and requirements for the ETL engine

Having defined the logical and the physical level, we need to design an execution model for the activities of the physical level. Generally, each activity receives tuples and processes them, and puts the result tuples into the input of another activity, according to the edge in the architecture graph. On the other hand recordsets do not always have both producers and consumers. Recordsets are entitled to feed the workflow with source data from an external source (e.g., text file, database) or write output data to an external target (e.g., a data warehouse).

There are three fundamental issues that we need to resolve in the design of an ETL execution engine, and a scheduler for it:

- The management of intermediate data.
- The strict requirement for zero data losses.
- The avoidance of deadlocks.

**Management of intermediate data:** One basic issue that rises is how to manage the intermediate data that are produced. One idea is to execute each activity separately, and store its output to a file. When this activity is done, its consumer activity can be started by reading the providers output file. This approach is simple, but it has two main disadvantages: (a) the need for disk space, which might not always be available and (b) the overhead of temporarily staging intermediate results and subsequently retrieving them again for the next activity. An alternative solution is to keep input and intermediate tuples into main memory and the activities will process them without the need to store intermediate data. Parts of the workflow that do not contain blocking operators can take advantage of the pipelining method. With this approach all activities need to be executed simultaneously, since we can not load all the input data into main memory. Every activity will read and write tuples from the appropriate shared data structures, such as queues (Figure 3.5). In the case of blocking operators (e.g., aggregator, sort-merge join) the intermediate data need to be stored temporarily.

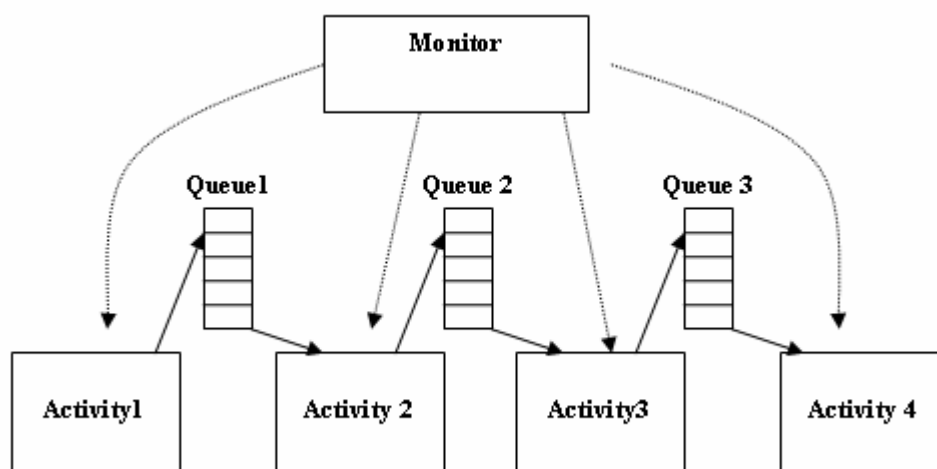


Figure 3.5 Pipelined execution of an ETL scenario.

This approach has a few more benefits. In the case where the input data are not stored locally, but the system receives them from a remote computer there is an extra communication cost for receiving the data from the remote computer. During this time, we can start processing the tuples that have arrived. The method of pipelining is also efficient in the case where the produced data must be sent to a remote computer.

The output data which are already produced can be sent to the remote target without having to wait for the execution of the scenario to finish. In both cases we can reduce the execution time since it overlaps with the time spent for communications.

**Zero data losses:** An essential principle for this system is that there is no data loss. All the tuples that are present in the input must be appropriately processed and propagated as the scenario dictates. The execution model must guarantee that no tuple will be lost during execution, due to the fact that some output buffer has been filled and its producer continues to output data to this buffer. Moreover, we need to come up with an implementation in the absence of the luxury of load shedding. When a DSMS (Data Stream Management System) experiences data overload, the load shedding technique is applied and some of the input data are ignored; then, each query is executed with the remaining data. Load shedding is useful in such cases, so that possible time constraints are satisfied. On the contrary, in our setting, all data are important, so we must ensure that we have zero data losses.

**Deadlocks:** One vital issue in the case of pipelined execution is that it is possible to have deadlocks during execution. The method of pipelining is commonly used when an SQL query is executed by a DBMS. There are cases where in a pipelined execution a deadlock might appear [DSRS01]. In a similar manner we may experience deadlocks when executing an ETL scenario. Our system should avoid the appearance of deadlocks.

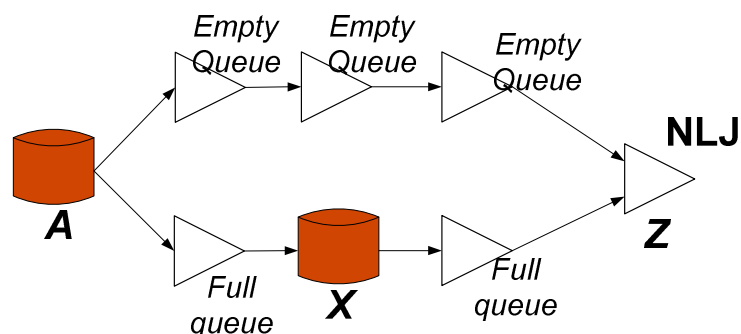


Figure 3.6 Deadline example (preliminary implementation).

In our preliminary implementations, when the scheduling was left to the operating system, the following kind of deadlock was observed: an original producer  $A$  would feed two parallel "lines" of activities, that would ultimately converge to a binary activity  $Z$ . Assuming that a blocking operator  $X$  participates in one of the two "lines", then  $Z$ 's input queues could possibly come to the state where one was completely empty and the other full. At the same time, all the queues between  $X$  and  $Z$  are empty and the queues in the other "line" of activities between  $A$  and  $Z$  were full. Then,  $Z$  cannot execute since one of its input queues is empty and  $A$  cannot execute since one of its output queues is full.

### **3.5. Scheduler**

Designing the engine's scheduler is one of the main tasks for the construction of an ETL engine. There are many possibilities for tuning a scheduling protocol. A first, simple to implement opportunity (without the existence of a scheduler) involves having the activities of the scenario running concurrently in random (as threads). The lack of a user-level scheduler means that we rely on the scheduling provided by the underlying operating system; we cannot get any guarantee that this is the best way (e.g., fastest, memory efficient) to execute the scenario. Designing a user-level scheduler gives us the ability to schedule the running activities with our own standards; therefore we can achieve a more efficient execution of the ETL scenario. Based on a user selected policy, the scheduler can tune the execution of the running activities. The user can pick from a palette of fundamental goals, e.g., (a) select to tune the scenario so that the total execution time is minimized or (b) to minimize the memory requirements, average and maximum.

### **3.6. Implementation of the logical level**

As mentioned before, the definition of an ETL scenario is a composition of the definitions of the elements that form a scenario, as well with its respective parameters. A scenario is a graph, so is composed of nodes and edges. A node could be either an



activity or a recordset. Every node has some input schemata, and one output schema. Every schema is a finite list of attributes. Finally, attributes are characterized by their name and data type.

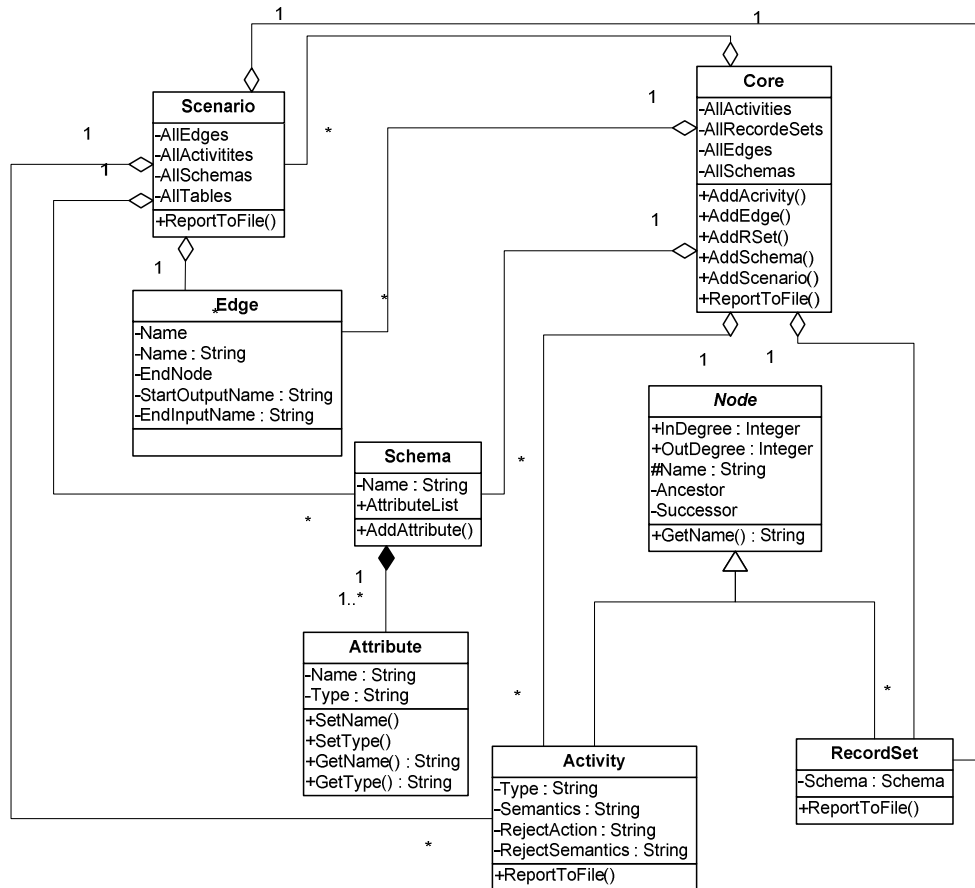


Figure 3.7 The class diagram of the logical level.

Therefore a scenario is a set of activities, recordsets, edges and schemata. In Figure 3.7 a class diagram of the logical level is depicted. Since every scenario is a graph, node and edge classes are defined in the class diagram. A node can be an activity or a recordset, so the *Node* class is extended to an *Activity* class and a *Recordset* class. The *Node* class is abstract, since it does not represent a specific element of an ETL scenario. The *Schema* class and the *Attribute* class represent the schemata and attributes of a scenario. There is also a *Scenario* class which holds all the elements that define it in collections. A simple declarative language is used, in which scenarios and all of its elements can be declared. Given a declaratively specified scenario the

engine's parser transforms it into objects of the aforementioned classes. We used the AntLR [AntLR07] parser, a simple and efficient tool that generates the code for a parser. With the parser a *Loader* class was created that reads the information the parser provides and creates the instances of the classes in Figure 3.7, and ensures that all elements are created and loaded correctly to a *Core* object.

### 3.7. Implementation of the physical level

Considering the above requirements for the ETL engine, its implementation necessitates the use of threads. Every activity and recordset will be executed from a single thread. A monitor thread is also essential for the control of the scenario execution, and will have a supervising role over the executing activities and recordsets. In order for the threads to communicate, a messaging system must be put into operation. This architecture is easy and simple to understand.

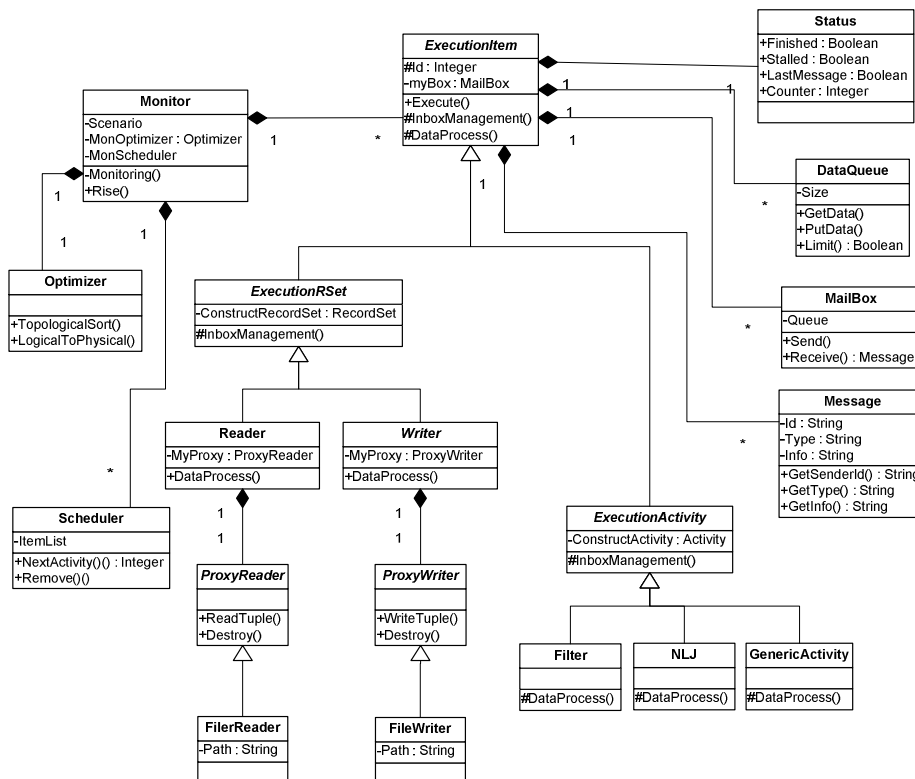


Figure 3.8 The class diagram of the physical level.

The design of the physical level requires dealing with ETL scenarios from a different perspective. Every node of a scenario is a unit that performs a portion of processing, even if that is simply reading or writing data. Therefore we consider every node (activity or recordset) as an execution item or operator. All intermediate data that execution items process must be stored in queues, so that the pipelined execution is accomplishable. These queues are called *data queues* and they contain tuples. As seen in the Aurora Stream Manager [CCR+03] processing every tuple separately is not efficient, so we use *row packs*, a structure which holds a number of tuples. Data queues keep row packs instead of tuples.

Since the graph is directed and acyclic every node can be characterize its neighbors as producers or consumers. Every execution item must have a mailbox, in order to support the messaging system that is required. Every execution item should know the mailbox of its producers and consumers, as well as the monitor's mailbox. The *monitor* is a component of the system that supervises and directs the execution. In Figure 3.8 there is the class diagram of the physical level. The two basic components of the physical level are the *Monitor* and the *Execution Item* classes. The *Execution Item* class is extended to the *Execution Recordset* class and the *Execution Activity* class. These two classes are also abstract. The basic functionality though exists in the *Execution Item* class. The other classes simply provide functionality for assigning producers and consumers to the nodes. A recordset has either only one consumer or only one producer, while an activity can have both many producers and many consumers.

### 3.7.1. Execution Item functionality

When a scenario starts to execute, the *Execute()* function is called for every operator. The execution of the operator is complete when the function returns. At each time point, it is possible that some of the operators will not have any data to process. For performance reasons we need to stall them for a small portion of time (every thread sleeps for a small time fragment).

The *Execute()* function is constructed as a loop (Table 3.1) in which (a) the operator checks its inbox regularly and (b) decides whether to process some data or to stall for a small time fragment. Every operator has a *status* flag that indicates whether it must process data or not and a *finished* flag that indicates whether the operator should exit the *Execute()* function. Thus, it is necessary for the operator to check its inbox frequently, since the monitor or some other operator might have sent an appropriate message. The operator will exit the while loop when the *finished* flag will turn its value from false to true. The *DataProcess()* function is not implemented in the *ExecutionItem* class, but by a concrete sub-class that overrides this function.

Table 3.1 The *Execute()* function of the *ExecutionItem* class

```

Sub Execute()
  InitExecute()
  While (Not OperatorStatus.Finished)
    InboxManagement()
    If (OperatorStatus.Stalled) Then
      Thread.Sleep(EngineStallTime)
    Else
      DataProcess()
    End If
  End While
  EndExecute()
End Sub

```

In any case, though, a critical point in our design has to do with the implementation of the *DataProcess()* function. As we shall see later in this section, the inbox of an operator receives messages from a monitor of the engine, with directives on when the current round of its execution completes and another operator must be activated. If we want an operator to relate to these notifications, the *DataProcess()* function must be constructed in such a way that it processes a small number of data -- small enough, so that their processing will have been completed before the designated deadline arrives. Moreover, the implementation of the *DataProcess()* function must respect the constraint that whenever the output queue is full, the operator must be stalled.

As seen in Figure 3.8 the recordsets of the scenario can be instantiated as *Readers* or *Writers*. These classes are responsible to feed the workflow with input data or store the output data, correspondingly. Every *Reader* or *Writer* uses a proxy inside the *DataProcess()* function. The proxy is simply a wrapper for objects that read from (or write to) text files, databases etc. 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. In Table 3.2 we see the implementation of the *DataProcess()* function of a *Reader* class. The *Status* variable keeps the status of the consumer's data queues. If the queues are full the operator must stop processing data.

Table 3.2 The *DataProcess()* function of a *Reader*

```
Protected Overrides Sub DataProcess()
    Dim Status As Boolean

    For I As Integer = 1 To EnginePackSize
        MyProxy.ReadTuple(CurrentTuple)
        If (CurrentTuple Is Nothing) Then 'Reached End Of Input
            OperatorStatus.Finished = True
            Exit For
        Else
            Status = ForwardToConsumers(CurrentTuple)
            If (Not Status) Then
                StallThread()
            Exit For
        End If
    End If
Next
End Sub
```

The activities of the scenario will be instantiated to a sub-class of the *Execution Activity* class. The *DataProcess()* function will contain the code that reads from its data queues, process the tuples and then forwards them to its producers. In Table 3.3 we see the implementation of the *DataProcess()* function of a *Filter* class. Again the operator must check the status of the consumer's queue, and if they are full the data processing must temporarily stop. In other activities, such as joins the *DataProcess()* function is more complex; yet the logic is the same. In every case the *DataProcess()*

function must process only a small amount of input data, so that the operator can check its inbox frequently.

Every *Execution Item* has a *Status* object. The *Status* class simply holds some flags and values for the status of the operator. This class acts as a grouper of these values, simply to keep the code organized and nice. The values that a *Status* class gathers are:

- *Stalled* (boolean): This value, if true, allows the operator to call the *DataProcess()* function
- *LastMessage* (boolean): This value is set true only when the operator will not receive any more messages from its producers. This will happen only if all of its producers are finished.
- *Finished* (boolean): This value is set true if the execution of the operation is complete.

Table 3.3 The *DataProcess()* function of a *Filter*

```
Protected Overrides Sub DataProcess()
    Dim Status As Boolean = True

    Producer.Queue.GetData(InPack)
    If (InPack Is Nothing) Then
        If (OperatorStatus.LastMessage) Then
            OperatorStatus.Finished = True
        Else
            StallThread()
        End If
    Else
        While (InPack.GetRow(CurrentTuple))
            If (FilterCalculator.Evaluate(CurrentTuple)) Then
                Status = Status And ForwardToConsumers(CurrentTuple)
            End If
        End While
        If (Not Status) Then
            StallThread()
        End If
    End If
End Sub
```

### 3.7.2. Monitor functionality and messages

The *Monitor* thread is responsible for the correct initialization and execution of the scenario. The execution of this thread has three basic steps. The first step is to create the physical object with respect to the logical objects the *Loader* created. In our system this mapping process is done by the *Optimizer* class. For all source recordsets the *Optimizer* returns a *Reader* object, and for target recordsets it returns a *Writer* object. For activities the *Optimizer* returns an object that inherits the *ExecutionActivity()* class. Since designing an *Optimizer* is not part of this thesis, the user defines explicitly e.g., which join activity prefers. Also during this process the *Optimizer* assigns a unique id to every operator. This id is used as an identifier so that threads can communicate to each other. Moreover, the *Optimizer* performs a simple check on the graph. Some errors, such as the incorrect definition of an edge, that were not detected from the parsing process will be found here. The *Optimizer* also makes sure that every thread knows its producers and consumers data queues and mailboxes.

After creating the physical objects, the monitor raises the threads of every operator by calling the *Execute()* function and then the monitoring process starts. This is the basic functionality of the monitor. When the execution starts all threads begin in stalled mode and simply wait for a message from the monitor to begin the execution process. The monitor uses an *ArktosScheduler* object, which selects the next thread to activate. Its interface is simple; on creation it creates a list with all threads. The *NextActivity()* function returns the id of the selected thread and the *Remove(Id As Integer)* function removes a thread from the list. This function is used when a thread is finished, to remove it from the list of the scheduler. Every time the monitor wants to activate a thread and allow it to execute, it must use the *NextActivity()* function to select the best operator according to the scheduler.

The monitoring process is a loop in which the monitor thread checks its mailbox and gathers some statistics. The statistics it gathers concern the required memory during the scenario execution. The monitor checks its messages to see when an operator has stalled or finished its execution. Depending on the message the monitor acts accordingly. Every operator has a mailbox and knows the mailbox of the monitor, as

well as its neighbors. All these objects communicate by sending messages. There are a few message types that the threads use. In Table 3.4 we see a brief description of the message types.

Table 3.4 The message types of the ETL engine

Message type	Description
MsgEndOfData	This message is sent among operators so that an operator will notify its consumers that it has finished producing data.
MsgTerminate	This message forces the thread to terminate even if the data process is not complete. If it is sent to the monitor it notifies that the sender has terminated.
MsgResume	When an operator receives this message it resumes the data processing by switching the flag <i>Stalled</i> to false.
MsgStall	When an operator receives this message it stops temporarily processing data by switching the flag <i>Stalled</i> to true.
MsgDummyResume	This message type is used to force all operators to execute once the <i>DataProcess()</i> function. This is used only when the scheduler could not select the next thread. This will give the chance to the operators to update some flags used internally.

### 3.7.3. Unary activities

These activities are filters and function activities. They have only one input edge and they are rather simple to implement. The function activities simply change a field of the tuple and forward the result tuple to its consumers. The filter activities are also simple and check the tuples they process based on a constraint defined at the semantics of every activity. This could be a domain filter or a null check.



Our implementation supports filters that compare fields with constant values that are of type *integer*, *double* and *string*. Also the filters can perform a null check for a set of fields. Every filter uses an instance of a *SingleTupleEvaluator* class. This class is abstract and it is instantiated to a specific concrete object when the filter is initialized. For example, if the filter performs the check "age > 0" the evaluator object will be an instance of the *IntegerBigger* class, because the field "age" is of type integer and the comparison is of type "bigger". If the filter performs a null check the instance will be the *NotNullCheck* class, which also inherits the *SingleTupleEvaluator* class. Selecting the correct evaluator is simple and should be done at the initialization of the execution. Using the evaluator is straightforward, since all that is necessary is to call the *Evaluate(Tuple As String)* which will return a Boolean value, indicating whether the tuple is to be kept or rejected.

#### 3.7.4. Binary activities

The binary activities our system supports are *Join*, *Surrogate Key*, *Diff* (with sort-merge and nested-loops variants) and *Aggregator*. These activities are blocking (or semi-blocking in the case of nested-loops) since they have to gather all input data to text files and sort them. Every operator handles these files. These operators have two stages of execution.

The first stage simply collects all input data and places them at a file. When all input is put into the file, it is sorted on the join field (except from the case of nested-loops activities, where only one input is blocking and this input is stored to a file).

The second stage is the joining process, where the sorted inputs are read and joined. In the case of the aggregator in this stage the grouping process occurs.

The *SMJoin* class performs a join, based on the join condition provided. If there is no match, no tuple will be produced. The *SMSkey* class adds a surrogate key to the tuple based on a lookup table. The *SMDiff* class implements the diff operation on the two input datasets. The *Aggregator* class groups the tuples based on one or more fields and calculates all aggregate functions (*maximum*, *minimum*, *sum*, *average* and *count*).

### 3.7.5. *Sorter*

The sorter is a useful tool that allowed us to implement the binary operators mentioned above. The file that is used to store the incoming tuples is handled by the *VBSorter* class. When a tuple is stored to the unsorted file, one extra field is put on the start of the tuple. This field is the sorting field. In some cases we need to sort with more than one field. In this case we sort with the concatenation of the sorting fields. In order for this technique to work, each sorting field must be of equal size, so we use padding to achieve equality in the length of the sorting fields. For padding, we use the space character. When a sorting field is a string or date we add padding from the left and when the sorting field is an integer or double we add padding from the right. This trick allows us to treat the sorting field as a string. The concatenation of the fields is done after the padding is added.

When the operator calls the *Sort()* function the sorter runs a batch file in which the input data are sorted. We use an external program to sort our files, borrowed from the cygwin UNIX emulator [Cygwin07]. Before using this sorter, we tried to find the source code of a file sorter, but we didn't find something that would suite us. In all cases the sorting process was very slow. For instance one of the sorters we found required two or three hours to sort a few hundred tuples, while the cygwin sorter managed to finish sorting in a few seconds.

The unsorted data are put in text files. Putting all input in one file creates one big file that the sorter cannot sort; there was no CPU utilization and the function never returned. To override this difficulty we adopted the following approach. The input is spitted into many text files. Every such file has a maximum capacity of 1,000,000 rows. If the unsorted input is more than 1,000,000 rows, it is divided to many such files. Then, every file is sorted separately and then merged, again with the help of the *Sort()* function. When the sorting is complete, the *Sort()* function returns and when the operator asks for a tuple the sorter removes the unnecessary sorting field from the tuple and returns it to the caller.

## CHAPTER 4. SCHEDULING ALGORITHMS

- 
- 4.1. Problem formulation
  - 4.2. Categories of algorithms
  - 4.3. Round Robin
  - 4.4. Minimum Cost
  - 4.5 Minimum Memory
- 

In this section we formalize our problem suggesting a mathematical definition. Then will describe the algorithms we implemented, and finally discuss a few simple examples of how these algorithms work.

### 4.1. Problem formulation

Consider a graph  $G(V,E)$ , and  $V = V_A \cup V_R = V_A \cup \{V_{SOURCE} \cup V_{TARGET} \cup V_{INTERM}\}$ .  $V_A$  denotes the activities of the graph and  $V_R$  the recordsets.  $V_R$  can be further divided into three disjoint sets; for the source, intermediate and target recordsets.

Also the set of all the nodes of the graph can be considered as  $V = \{V_{FINISHED} \cup V_{CANDIDATES}\}$ , where  $V_{CANDIDATES}$  is the set of operators that are active and participate in the execution and  $V_{FINISHED}$  is the set of nodes that have finished their processing.

For each activity node  $v \in V_A$  we define:

- $\mu(v)$ , as the consumption rate of node  $v$ .
- $queue(v)$ , as the sum of all input queue sizes (not capacity) of node  $v$
- $\sigma_v$ , as the selectivity of node  $v$ .

For each recordset node  $v \in V_R$  we define:

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

Furthermore, for each source recordset node  $v \in V_S$  we define:

- $volume(v)$  as the size of the recordsets input of node  $v$ .

We consider  $T$  as an infinite countable set of timestamps and a scheduler with policy  $P$ . The scheduler divides  $T$  into disjoint and adjacent intervals  $T = T_1 \cup T_2 \cup \dots$  with:

- $T_i = [T_i.first, T_i.last]$
- $T_i.last = T_{i+1}.first - 1$

Whenever a new interval  $T_i$  begins, (at timestamp  $T_i.first$ ) the scheduler decides one or some of the following actions; Option (1) is mandatory.

1.  $active(T_i)$ , the next activity to run.
2.  $T_i.last$ . This value is the timestamp that the operator  $active(T_i)$  will stop executing. In other words it is  $T_i.length()$ , the scheduler's time slot.
3. Status of all queues at  $T_i.last$ .

The operator  $active(T_i)$  will stop its processing if one of the following occurs:

1.  $clock = T_i.last$ . That means that the time slot has exhausted.
2.  $queue(active(T_i)) = 0$ . This means that the active operator has no more input data to process.
3.  $\exists v, v \in consumer(active(T_i))$  such that  $queue(v) = M(v)_{max}$ . This means that one of the consumers of the active activity  $active(T_i)$  has a full input queue.

At this point we must check if  $active(T_i)$  should be moved to  $V_{FINISHED}$ . In order for an operator  $v$  to be moved to  $V_{FINISHED}$ , both of the following must be valid.

- $\forall v \in producer(active(T_i)), v \in V_{FINISHED}$ , and
- $queue(active(T_i)) = 0$  or  $volume(v) = 0$ , if  $v \in V_{SOURCE}$ .

A workflow  $G(V, E)$  ends when  $V = V_{FINISHED}$ . The interval during which this event takes place is denoted as  $T.last$ .

Based on the previous we can implement a scheduling policy  $P$  for a scenario  $G(V, E)$  such that:

- $P$  creates an appropriate division of  $T$  into intervals  $T_1 \cup T_2 \cup \dots T_{last}$
- $\forall t \in T, v \in V \text{ queue}(v) \leq \text{Max}(\text{queue}(v))$  (i.e., all data are properly processed).
- One of the following holds:
  - $T_{last}$  is minimized,  $T_{last}$  is the interval where  $G$  stops
  - $\max \sum \text{queue}_{T_i}(v)$  is minimized,  $t \in T, v \in V$ .

Table 4.1 Categories of scheduling algorithms

Category	Description
Token Based	Every operator has a token, and based on that the scheduler assigns the CPU
Execution Time	This category contains scheduling policies that target to optimize the system's execution time.
Response Time	Such scheduling policies try to improve the systems response time
Memory	In this category the scheduling policies aim to minimize the required memory during the execution

## 4.2. Categories of algorithms

While studying the related work we discerned four basic categories of algorithms. The first category includes the token based algorithms, such as Round Robin. These algorithms are used mostly as a baseline to compare other algorithms. The second category includes the algorithms that aim to reduce the total execution time. In the third category reside the algorithms that aim to improve the response time and the last category includes the algorithms that target to reduce the required memory during the execution. In Table 4.1 we see these categories with a brief description. As mentioned in chapter 2, the improvement of response time is npt a requirement, thus we choose to design one algorithm from every category apart from the third one. Therefore, we

concentrate on the three other categories, and propose one policy for each of them (Table 4.2).

Table 4.2 Scheduling steps of the studied scheduling policies

	RR	MC	MM
Pick Next	Operator ID	Max Input Queue Size	Max tuple consumption
Reschedule when	Input queue is exhausted	Input queue is exhausted	Time slot

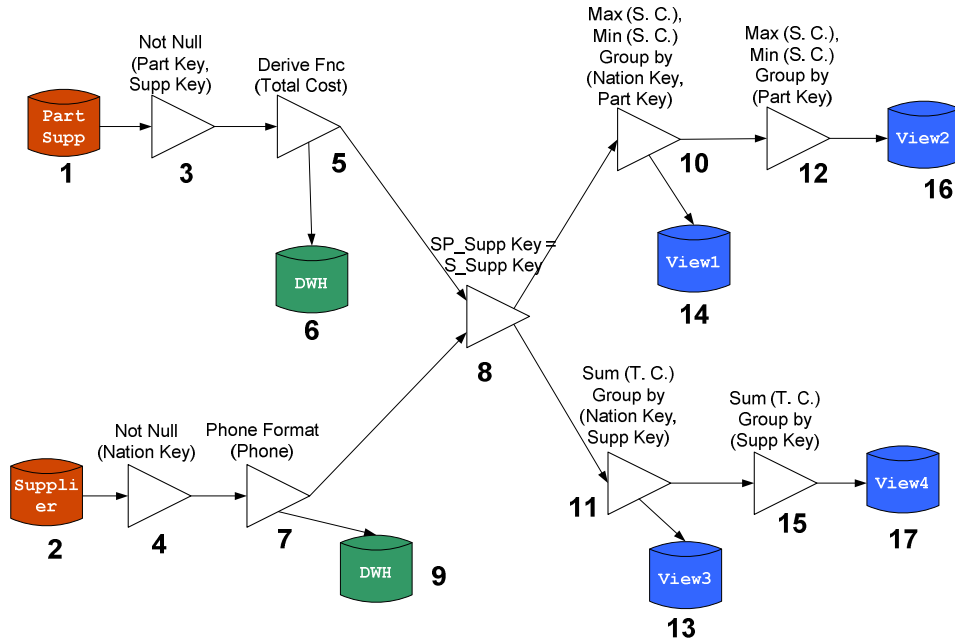


Figure 4.1 An example butterfly scenario.

### 4.3. Round Robin

The *Round Robin* (RR) scheduling algorithm is very simple to implement since its only requirement is to assign a unique identifier to every operator and order them with this identifier. Then, based on this order the scheduler sets the operators to run. This simple algorithm has some very good properties; every operator gets the same

chances to run (fairness) and it will not lead the system to starvation. In Figure 4.1 there is an example scenario with their operators (activities and recordsets) numbered. The numbering is random. This scenario will be used as an example so that we can demonstrate how the scheduling algorithm works.

The *Round Robin* algorithm selects the operators based on their identifier. At start up only operators **1** and **2** can do some data process. The algorithm selects at first operator **1**. The scheduling that *Round Robin* will apply is presented in Table 4.3. In step 10 the operator **10** is set to run, but at this point **10** has no data to process, since **9** is blocking.

Table 4.3 Scheduling steps of *Round Robin*

Step	Can Select	Selects
1	1	1
2	Next( <b>1</b> ) = <b>2</b>	2
3	Next( <b>2</b> ) = <b>3</b>	3
4	Next( <b>3</b> ) = <b>4</b>	4
5	Next( <b>4</b> ) = <b>5</b>	5
6	Next( <b>5</b> ) = <b>6</b>	6
7	Next( <b>6</b> ) = <b>7</b>	7
8	Next( <b>7</b> ) = <b>8</b>	8
9	Next( <b>8</b> ) = <b>9</b>	9
10	Next( <b>9</b> ) = <b>10</b>	10

#### 4.4. Minimum Cost

The *Minimum Cost* (MC) scheduling algorithm minimizes the scenario's execution time. This is achieved by minimizing any overhead that occurs from the scheduler and mainly from the communications between the threads. The operator that is selected must have data to process, and preferably, this will be the operator with the most input data. In addition, no time slots are used, so that the selected operator can process all its input data with no interrupts from the monitor thread. We consider that all operators that read data from an external source are always available for execution. In

order to demonstrate the scheduling of the *Minimum Cost* algorithm we will use the example in Figure 4.1.

Table 4.4 Scheduling steps of *Minimum Cost*

Step	Can Select	Selects
1	<b>1</b> (R), <b>2</b> (R)	1
2	<b>1</b> (R), <b>2</b> (R), <b>3</b> (100)	3
3	<b>1</b> (R), <b>2</b> (R), <b>5</b> (90)	5
4	<b>1</b> (R), <b>2</b> (R), <b>6</b> (80), <b>8</b> (80)	6
5	<b>1</b> (R), <b>2</b> (R), <b>8</b> (80)	8
6	<b>1</b> (R), <b>2</b> (R),	2
7	<b>1</b> (R), <b>2</b> (R), <b>4</b> (100)	4
8	<b>1</b> (R), <b>2</b> (R), <b>7</b> (50)	7
9	<b>1</b> (R), <b>2</b> (R), <b>8</b> (30), <b>9</b> (30)	8
10	<b>1</b> (R), <b>2</b> (R), <b>9</b> (30)	9

In Table 4.4 we see how the *Minimum Cost* algorithm works. The second column shows the id's of the operators that are candidates for execution accompanied with the size of their input queue. Since some operators are *Readers* (i.e., proxies for source recordsets and as such they continuously retrieve the next available tuple from their recordset, which they add to their input queue), we use the symbolism (R). The third column has the choice of the scheduler. In every step, the scheduler selects the operator with the biggest input size. Between operators with equal input size, we can select either, without affecting the performance of the execution.

#### 4.5. Minimum Memory

The *Minimum Memory* (MM) scheduling algorithm tries to schedule the operators in a way that the maximum and average memory that the system requires during the execution of a scenario is minimized. The scheduler must select the operator that will *consume* the biggest amount of data. The amount of data 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. In order to achieve this scheduling we



need small selectivity as well as large processing rate and input size from the preferred operator. The input size should be large so that the operator can process and possibly reduce many data. The selectivity needs to be small enough so that the operator can actually consume its input tuples. Finally the processing rate should be large in order for the data consumption to occur as fast as possible.

Alternatively we could compute the consumption rate directly, considering the number of tuples consumed (input data - output data) divided by the processing time of the input data. The overall memory benefit is the input size of an operator multiplied by its input size, as seen in equation (4.1).

$$\text{MemB}(p) = ((\text{In}(p) - \text{Out}(p)) / \text{ExecTime}(p)) * \text{Queue}(p) \quad \text{Eq. 4.1}$$

In this equation  $p$  is the operator.  $\text{In}(p)$  and  $\text{Out}(p)$  denote the number of tuples that  $p$  has as input and as output respectively.  $\text{ExecTime}(p)$  is the time the operator  $p$  needed to process the  $\text{In}(p)$  tuples.  $\text{Queue}(p)$  is the number of tuples that are in  $p$ 's input queues. The MM scheduler selects the operator with the biggest  $\text{MemB}()$  value at every scheduling step.

Table 4.5 Scheduling steps of *Minimum Memory*

Step	Can Select	Input Size	Selects
1		<b>1</b> (R), <b>2</b> (R)	1
2	<b>1</b> (-0.16)	<b>3</b> (11)	3
3	<b>1</b> (-0.16)	<b>5</b> (11)	5
4	<b>1</b> (-0.16)	<b>8</b> (11)	8
5	<b>1</b> (-0.16), <b>8</b> (0.27)	<b>8</b> (6), <b>6</b> (5)	8
6	<b>1</b> (-0.16)	<b>6</b> (11)	6
7	<b>1</b> (-0.16)	<b>1</b> (R), <b>2</b> (R)	1
8	<b>1</b> (-0.26), <b>3</b> (5.75)	<b>3</b> (23)	3
9	<b>1</b> (-0.26), <b>5</b> (1.3)	<b>5</b> (23)	5
10	<b>1</b> (-0.26), <b>6</b> (1.03), <b>8</b> (1.13)	<b>6</b> (23), <b>8</b> (23)	8
11	<b>1</b> (-0.26)	<b>6</b> (23)	6

When the scenario starts to execute, no operator has processed any data, so the above formula cannot apply. In this case we use the logic of *Minimum Cost* algorithm, so the

operator with the biggest input size is selected. In Table 4.5 we see how the MM algorithm behaves. The second column contains the calculations of  $MemB(p)$  for every scheduling step. In every step, some operators are omitted because the  $MemB()$  value is equal to 0. In such a case, or when all operators have a negative  $MemB()$  value, we select an operator based on its input size. The third column contains the input size for every operator in every scheduling step.

## CHAPTER 5. EXPERIMENTS

---

- 5.1. Measures and Parameters
  - 5.2. Datasets
  - 5.3. Scenarios and data sources
  - 5.4. Tuning scheduling policies
  - 5.5. Line workflow
  - 5.6. Wishbone workflow
  - 5.7. Primary flow workflow
  - 5.8. Butterfly workflow
  - 5.9. Tree workflow
  - 5.10. Fork workflow
  - 5.11. Observations deduced from experiments
- 

This section provides the details for the experiments performed in order to test the efficiency of the Arktos scheduler. The first section of this chapter describes the metrics and the measures that are of interest, the second section has a brief description of the datasets used during the experiments and the remaining sections present and comment the experimental results. In Table 5.1 we see the hardware and software specifications of the computer we conducted the following experiments.

Table 5.1 Development environment

<b>Hardware</b>	
CPU	Dual Core 2 @ 2.13 Ghz
M/M	1 GB
Hard Disk	230 GB
<b>Software</b>	
Operating System	Windows XP Professional SP2
Development Software	Visual Studio 2005, SP1
Programming Language	VB 2005, C# 2005 .NET 2.0 framework.

### 5.1. Measures and Parameters

The measures that concern us in this thesis are following:

- **Execution Time**
  - Execution time is a basic measure to quantify each scheduling policy's efficiency.
- **Memory consumption**
  - Memory consumption measures the memory requirements of every scheduling policy during execution. We are concerned for average, as well as, maximum memory requirements. 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 gives eventually the average memory.

The input parameters that will be used to quantify the above measures will be:

- **Workflow size**
  - The number of activities in an ETL scenario will have an effect on the above measures. The kind of activities (blocking or non-blocking) is also a considerable parameter.
- **Workflow selectivity**

- If in an existing scenario most of the input data are dirty, the execution time and the memory requirements can be affected. As workflow selectivity we consider the selectivity a workflow has from its sources to its body (Figure 5.1).
- **Time Slot**
  - This parameter defines the time interval that each operator runs. At the end of each time interval the scheduler selects the next operator.
- **Row Pack Size**
  - The selected size of the row pack defines the granularity of the *DataProcess()* function.
- **Queue size**
  - Using various queue sizes we can see whether the execution time will increase linearly with the input size, and if there are any changes in the memory minimization algorithm.
- **Workflow Structure**
  - We handle the complexity of workflow characteristics with a set of characteristic scenarios instead of employing large and randomly generated workflows. To this end, a broad category of workflows is used, called *Butterflies* [Tzio06]. A *butterfly* is an ETL workflow that consists of *three* distinct components: (a) the *left wing*, (b) the *body* and (c) the *right wing* of the butterfly. The left and right wings are two non-overlapping groups of nodes which are attached to the body of the butterfly. In Figure 5.1 there is the basic structure of a butterfly workflow. Different variations of this structure are used in the experiments, which are discussed in section 5.3.2.

We tune row pack, time slot size and queue size parameter for every scheduling policy. We determine best possible values with micro-benchmarks so that we can proceed to the experiments. In section 5.4 we present the experiments we conducted to determine these values.

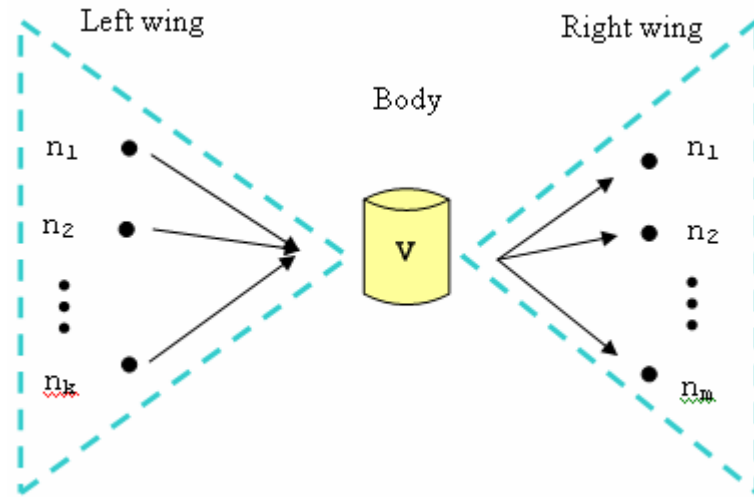


Figure 5.1 The basic structure of a butterfly workflow.

## 5.2. Datasets

One popular benchmark for evaluating database systems is the TPC-H benchmark. Recently the TPC-DS benchmark was released, as a follower of the TPC-H. A draft version of the TPC-DS benchmark is also available.

### 5.2.1. TPC-H

The TPC Benchmark™ H (TPC-H) [TPCH07] is described as a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. Also, this benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

TPC-H evaluates the performance of various decision support systems by the execution of sets of queries against a standard database under controlled conditions. The queries that this benchmark provides give answers to real-world business questions and simulate generated ad-hoc queries. They are far more complex than

most OLTP transactions and they include a rich breadth of operators and selectivity constraints. Also, they generate intensive activity on the part of the database server component of the system under test. The relational schema of the data that TPC-H provides consists of eight separate tables, as illustrated here in Figure 5.2. It describes a sales system, keeping information for the parts and the suppliers, and data about orders and the supplier's customers. The dataset can be generated in variety of sizes up to 100 TB. Update datasets are also provided but in this benchmark there are no update functions.

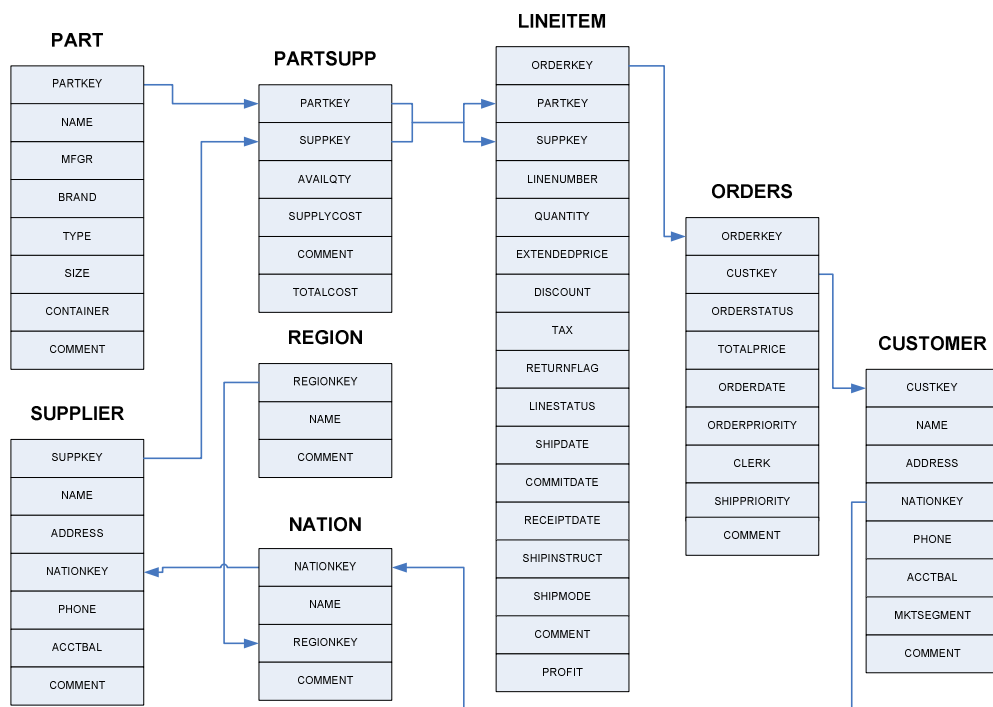


Figure 5.2 The TPC-H relational schema.

### 5.2.2. TPC-DS

The TPC Benchmark™ DS (TPC-DS) [TPCDS07], [OtPo06] is a new Decision Support (DS) workload being developed by the TPC. This benchmark models the decision support system of a retail product supplier, including queries and data maintenance. Although the underlying business model of TPC-DS is a retail product supplier, the database schema, data population, queries, data maintenance model and

implementation rules have been designed to be broadly representative of modern decision support systems.

The relational schema of this benchmark is more complex than the schema presented in TPC-H. There are three sales channels, store, catalog and the web. There are two fact tables in each channel, sales and returns, and a total of seven fact tables. In this dataset the row counts for tables scale realistically. Specifically in fact tables the row count grow linearly, while in dimension tables grow sub-linearly.

This benchmark also provides update data. Moreover it has a set for update functions. All these functions are primary flows, in which surrogate and global keys are assigned to all tuples.

### 5.3. Scenarios and data sources

This section contains all the experimental scenarios we have designed in order to test our system. As a source for the experiments the dataset from the TPC-H benchmark was used, in various sizes. The dataset is about a sales system. The information kept is for the parts and its suppliers. Also detailed information is kept about the orders that the suppliers have, and some demographic data for the customers. The scenarios that are used in the experiments clean and transform the source data into the desired warehouse schema. The schema of the data warehouse consists of the table **PART** (s\_partkey, name, mfgr, brand, type, size, container, comment), the table **SUPPLIER** (s\_suppkey, name, address, nationkey, phone, acctbal, comment, totalcost), the table **PARTSUPP** (s\_partkey, s\_suppkey, availqty, supplycost, comment), the table **CUSTOMER** (s\_custkey, name, address, nationkey, phone, acctball, mktsegment, comment), the table **ORDER** (s\_orderkey, custkey, orderstatus, totalprice, orderdate, orderpriority, clerk, shippriority, comment) and table **LINEITEM** (s\_orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate, receiptdate, shipinstruct, shipmode, comment, profit). The relational schema of each source is similar to the TPC-H schema in Figure 5.2.



### 5.3.1. Data Sources

The sources for our experiments are of two groups, the storage houses and the sales points. Every storage house keeps the data for the suppliers and the parts, while every sale point keeps the data for the customers and the orders. The storage house schema consists of the table **PART** (partkey, name, mfgr, brand, type, size, container, comment), the table **SUPPLIER** (suppkey, name, address, nationkey, phone, acctbal, comment) and the table **PARTSUPP** (partkey, suppkey, availqty, supplycost, comment) who relates the previous two. The storage house is in Figure 5.3.

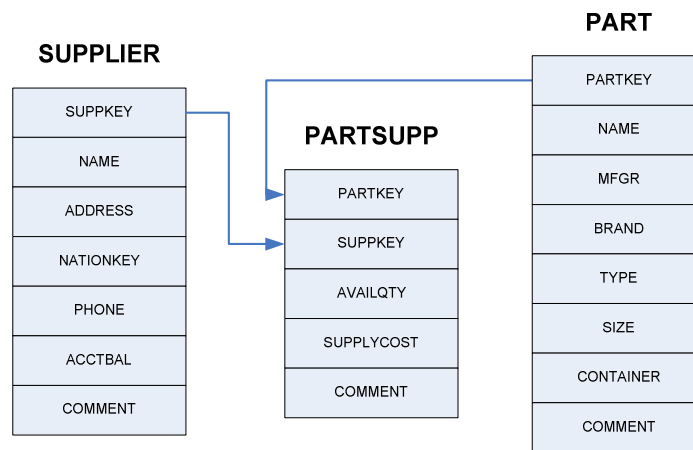


Figure 5.3 The storage house relational schema.

The sales point schema consists of the table **CUSTOMER** (custkey, name, address, nationkey, phone, acctball, mktsegment, comment), the table **ORDER** (orderkey, custkey, orderstatus, totalprice, orderdate, orderpriority, clerk, shippriority, comment) and table **LINEITEM** (orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate, receiptdate, shipinstruct, shipmode, comment), The schema of the sales points is in Figure 5.4

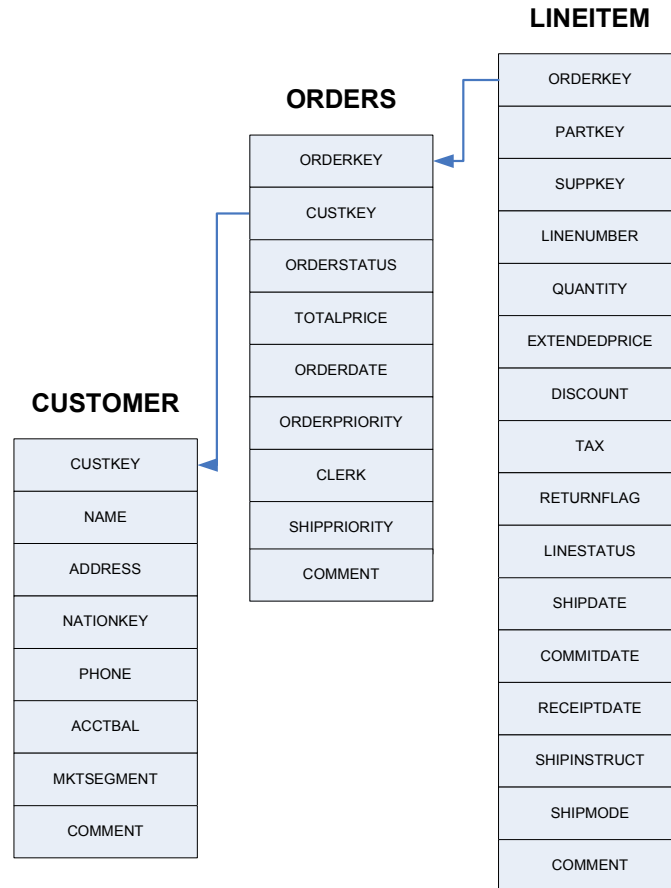


Figure 5.4 The sales point relational schema.

### 5.3.2. ETL Scenarios

The experiments for the cleaning of the data sources include many workflow types, which are explained in detail in [Tzio06]. These workflow types are: (a) **line**, (b) **wishbone**, (c) **primary flow**, (d) **butterfly**, (e) **tree** and (f) **fork**. The scenarios that appear in this section will be used to evaluate our system.

An example of a **line** workflow is in Figure 5.5. This scenario type is used to filter a source table and make sure that the data meet the logical constraints of the data warehouse. In the example in Figure 5.5 the applied operations are:

1. Checking the fields "partkey", "orderkey" and "suppkey" if they have NULL values.

2. Converting the dates in the "shipdate" and "receiptdate" fields into a date id, a unique identifier for every date.
3. This activity is a calculation of a value "profit". This value derives from other fields in every tuple; is the amount of "extendedprice" subtracted by the values of the "tax" and "discount" fields.
4. This activity changes the fields "extendedprice", "tax", "discount" and "profit" to a different currency. The results of this operation are loaded into the data warehouse.
5. The workflow is not is not stopped since we would like to create some materialized views. This operation keeps only the data that its return status is "False".
6. This is an aggregation, calculating the sum of "profit" and "extendedprice" fields grouped by "partkey" and "linestatus".
7. This activity keeps the tuples that the "linestatus" field has the value "delivered".
8. This final aggregation calculates the sum of "profit" and "extendedprice" fields grouped by "partkey".

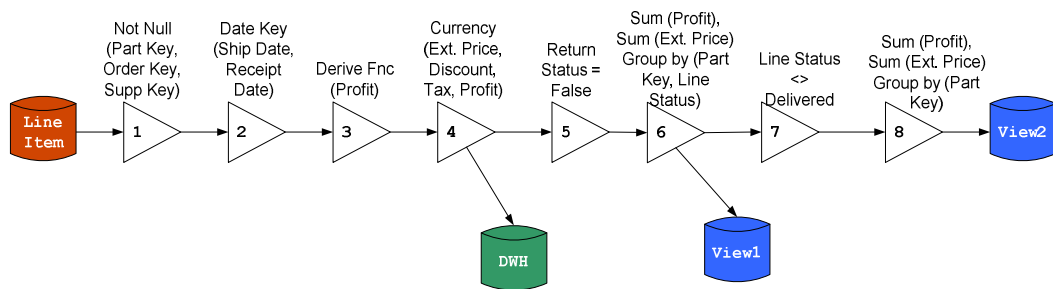


Figure 5.5 A Line Scenario

A **wishbone** workflow joins two tables into one, as appears in Figure 5.6. This scenario is preferred when two tables in the source database must be joined in order to be loaded to the data warehouse. The example scenario in Figure 5.6 has as input data the tables "customer" and "orders".

1. This activity checks for NULL values in the "nationkey" field.

2. This activity converts the phone numbers in a numeric format, removing dashes and replacing the '+' character with the "00" equivalent.
3. This activity checks the "custkey" fields for NULL values.
4. The Date-Key activity is applied on the "orderdate" field.
5. This activity applies the currency operation on the "totalprice" field.
6. On this activity the source tables are joined. The Sort-Merge Join activity will be used at the experiments.
7. On the joined result an aggregation is applied calculating the sum and the maximum of the "totalprice" field, grouped by the "nationkey" and "orderdate" field.
8. This aggregator calculates the sum and the maximum of the "totalprice" field, as in the previous activity, but grouped only by the "nationkey".

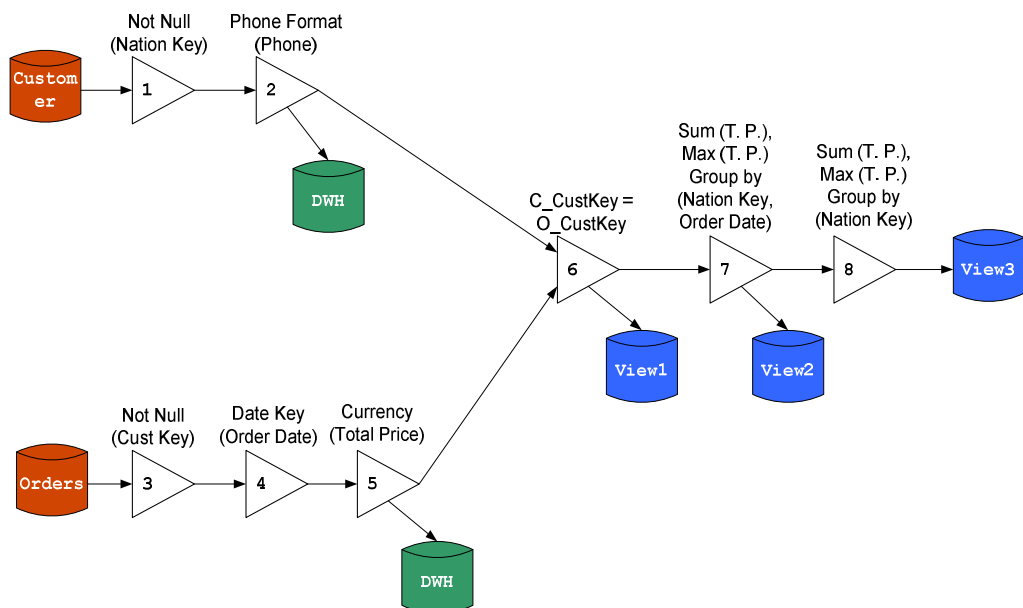


Figure 5.6 A Wishbone Scenario

The **primary flow** scenario is a common scenario in cases where the source table must be enriched with surrogate and global keys. It is not a line scenario because the operator that adds surrogate keys to every tuple is a join variant. In general, a primary flow could easily have a join operator. An example of a primary flow scenario is in Figure 5.7. This primary flow scenario has as input the "lineitem" table

(1-4).The first four activities are the same four of the line scenario.

(5-7).The other three activities assign to each tuple a surrogate key for the "partkey", "suppkey" and "orderkey" fields which are business keys.

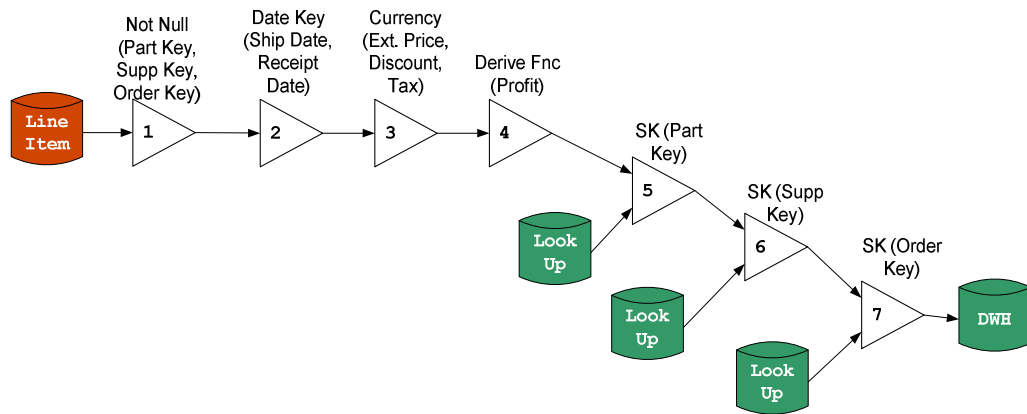


Figure 5.7 A Primary Flow Scenario

The most common scenario type is a **balanced butterfly** scenario. It joins two or more source tables into one, then a set of aggregations are performed on the result of the join. The left wing of the butterfly joins the source tables, while the right wing performs the desired aggregations producing materialized views. An example of a butterfly scenario is in Figure 5.8. For this scenario the "partsupp" and "supplier" tables are used as input.

1. Checking for NULL values on the "partkey" and "suppkey" fields.
2. Calculating and adding to each tuple the "totalcost" field.
3. Checking the "nationkey" field for NULL values.
4. This activity transforms the "phone" field.
5. This activity joins results from activities 2 and 4 on the "ps\_suppkey" and "s\_suppkey" fields.
6. This aggregation calculates the maximum and the minimum value of the "supplycost" field grouped by the "nationkey" and "partkey" fields.
7. This aggregation calculates the maximum and the minimum of the "supplycost" field grouped by the "partkey" fields.

8. This activity calculates the sum of the "totalcost" field grouped by the "nationkey" and "suppkey" fields.
9. This activity calculates the sum of the "totalcost" field grouped by the "suppkey" field.

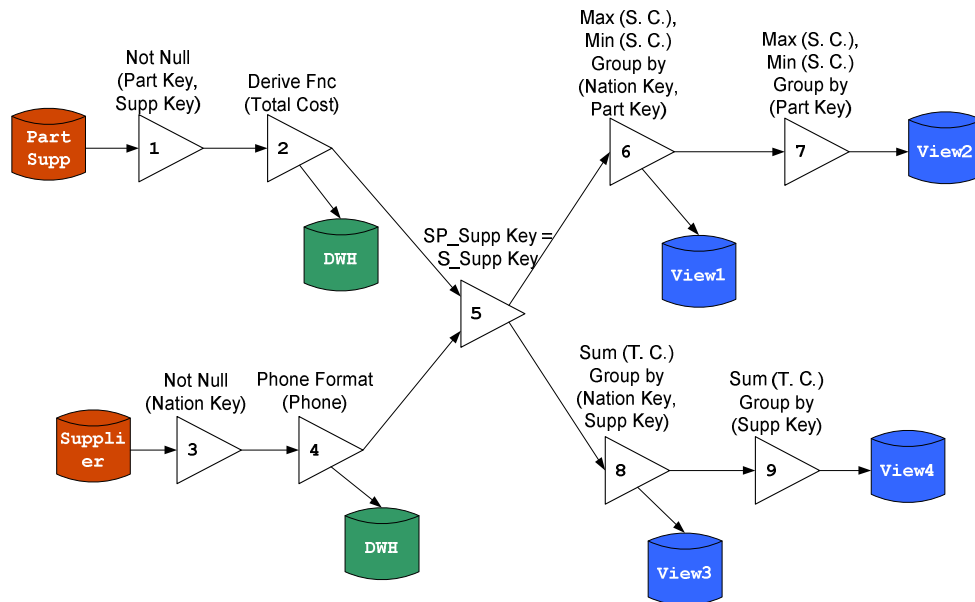


Figure 5.8 A Balanced Butterfly Scenario

The **tree** scenario in Figure 5.9 joins two or more source tables and applies aggregations on the result recordset. This tree scenario uses as input the "partsupp", "part" and "supp" tables.

1. This activity checks for NULL values the "suppkey" and "partkey" fields of the "partsupp" table.
2. This activity calculates the "totalcost" field for the tuples of the "partsupp" table.
3. This activity checks for NULL values the "partkey" field of the "part" table.
4. This activity joins the transformed "part" and "partsupp" tables on the "partkey" field of every table.
5. This activity checks for NULL values the "suppkey" fields of the "supplier" table.
6. This activity transforms the "phone" field.

7. This activity joins the "supplier" table with the result of the activity (4) on the "suppkey" field of every input.
8. The last activity aggregates the result of the activity (8), calculating the maximum and the minimum value of the "totalcost" field, groupd by the "suppkey" and "partkey" fields.

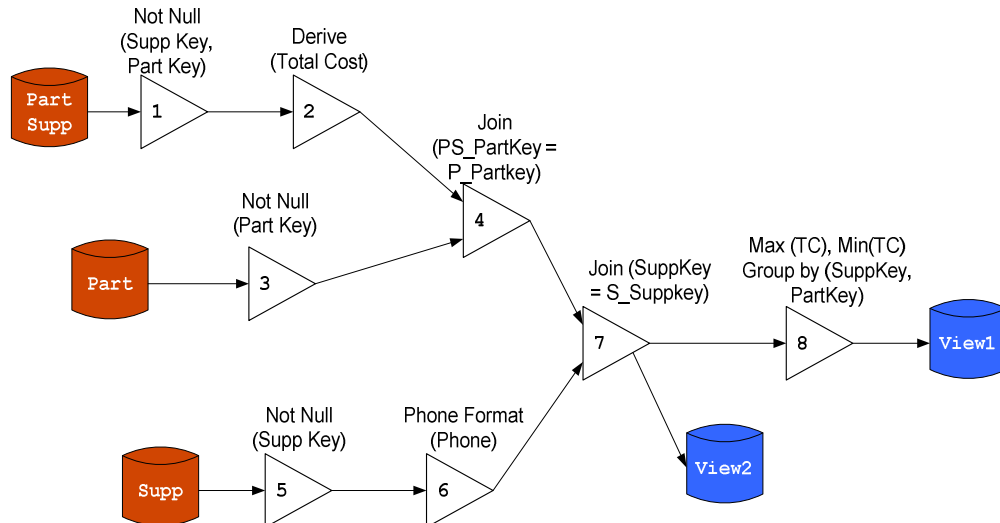


Figure 5.9 A tree scenario

Finally the **fork** scenario applies a set of aggregations on a single source table. First the source table is cleaned, just like in a line scenario and the result table is used to create a set of materialized views. The tree scenario in Figure 5.10 uses as input the "lineitem" table.

1. Checking the fields "partkey", "orderkey" and "suppkey" if they have NULL values.
2. Converting the dates in the "shipdate" and "receiptdate" fields into a date id, a unique identifier for every date.
3. This activity is a calculation of a value "profit". This value derives from other fields in every tuple; is the amount of "extendedprice" subtracted by the values of the "tax" and "discount" fields.

4. This activity changes the fields "extendedprice", "tax", "discount" and "profit" to a different currency. The result of this scenario will be forwarded so that a number of aggregations can be performed.
5. This filter activity keeps the tuples where the "returnstatus" field has the value "true".
6. This aggregator calculates the sum of the "profit" and "extendedprice" fields grouped by the "partkey" and "linestatus" fields.
7. This aggregator calculates the sum of the "profit" and "extendedprice" fields grouped by the "linestatus" fields.
8. This aggregator calculates the sum of the "profit" field and the average of the "discount" field grouped by the "partkey" and "suppkey" fields.
9. This filter activity keeps the tuples where the "discount" field has the value "0".
10. This aggregator calculates the average of the "profit" and "extendedprice" fields grouped by the "partkey" and "linestatus" fields.

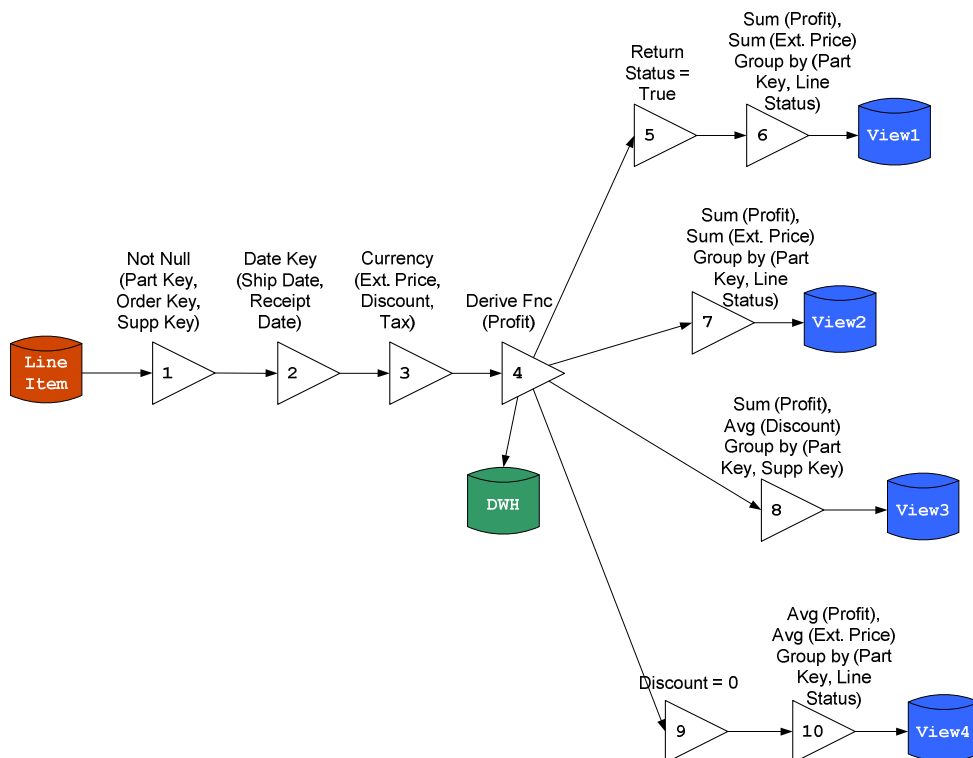


Figure 5.10 A fork scenario



#### 5.4. Tuning scheduling policies

Our ETL engine has a few parameters that can affect the execution. These parameters are:

- **Time Slot:** This value determines the size of the time slot the scheduler will use. The time slot is determined in milliseconds.
- **Stall Time:** This value sets the duration each thread will remain stalled; is measured in milliseconds.
- **Data Queue Size (DQS):** This value sets the maximum size of the systems data queues. In data queues row packs are inserted.
- **Row Pack Size (RPS):** This value sets the size (number of tuples) of every row pack.

The *Stall Time* value is used as parameter for the system command *Thread.Sleep(EngineStallTime)*. This command is not very reliable, since there is no guarantee that the thread will continue its execution after sleeping for *EngineStallTime* milliseconds. So we need to keep it very small; big values lead the system to an idle state for some time. This occurs because the use of big values would make the operators to be idle for a long period of time and also they would read their messages long after it was sent. In all algorithms we used the value of 4 milliseconds.

We need to determine which set of values optimizes the execution of every scheduler. For the RR and MC algorithms we will try to optimize the execution time, while in the MM we will try to find a set of values that give smaller memory demands and relatively good execution time. Using time slots in this RR and MC scheduling policies would lead to more communication and scheduling overhead and finally to a bigger execution time. In MC for example, consider 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. The 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. So in these two policies we will conduct experiments to find an area of good values for the *DQS* and *RPS* parameters. For the MM algorithm we will use the values for the *DQS* and *RPS*

parameters from the previous experiments and try to find a good value for the  $TmSl$  parameter.

To conduct our experiments after tuning our scheduling policies we created variations at the input size and the selectivity of the workflows.

- Concerning the input size, we used the data generator the TPC-H [TPCH07] provided. The data generator has a scale factor (SF) that defines the size of the data to be generated. When SF is set to 1, the data generator produces one GB of data. For our experiments we created three datasets with scale factors 0.1, 0.5 and 1.0.
- Concerning the selectivity of the workflows, we changed the semantics of one or more filter activities so that the desired selectivity occurred. The selectivity values we used are 0.5, 0.8 and 1.0. This broad range will give us a good perspective of how selectivity affects the execution of every scenario.

#### 5.4.1. Tuning Round Robin

To determine the values of  $DQS$  and  $RPS$  for this scheduling algorithm we have conducted two set of experiments. The first set aims to find a good area of values for the  $DQS$  parameter, while the second set aims to find a good area of values for the  $RPS$  parameter. For each set we have used two different scenarios, the butterfly in Figure 5.8 and a small line scenario, (a variation of the line scenario in Figure 5.5, keeping only the first four activities).

For the first set we have used four different values {100, 150, 200, 250} for the  $RPS$  parameter. In Figure 5.11 we can see how RR behaves in the line scenario. For any value of  $RPS$ , we observe that any value of  $DQS$  above 30 performs equally for any value of  $RPS$ . For this range of values the execution time is very close to the best execution time on this chart. When  $DQS$  has small values ( $< 30$ ) the execution time is bigger since small  $DQS$  values require much more scheduling steps; this means that we have more scheduling and communication overhead. When there are many

unnecessary communications the system stays idle and there is not a good utilization of the CPU.

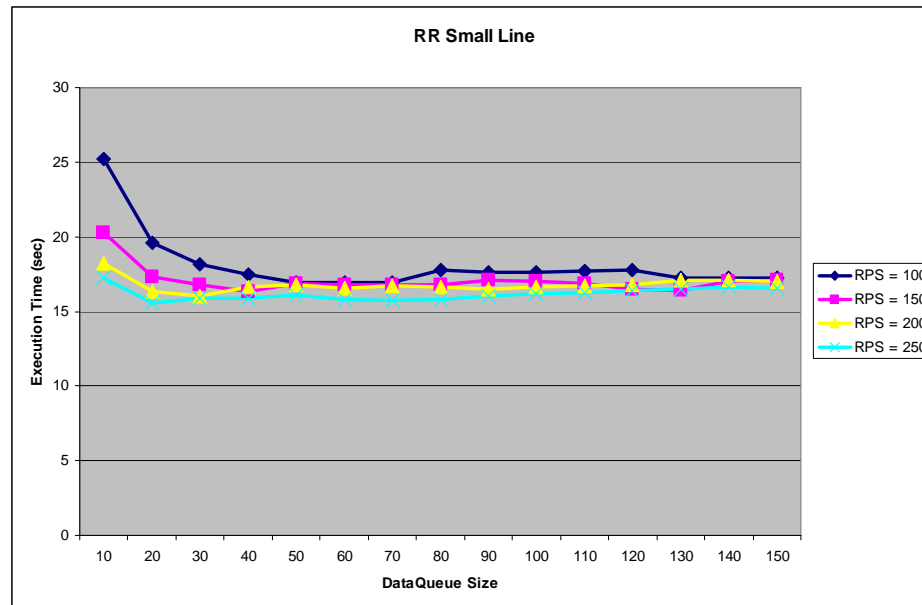


Figure 5.11 Tuning DQS in the small line scenario (RR)

In Figure 5.12 there we can see the results of the RR algorithm with the butterfly scenario. In this chart, the RR scheduler optimizes its execution time when the  $DQS$  parameter has values bigger than 45. Greater values of  $DQS$  do not affect the execution time of the scenario.

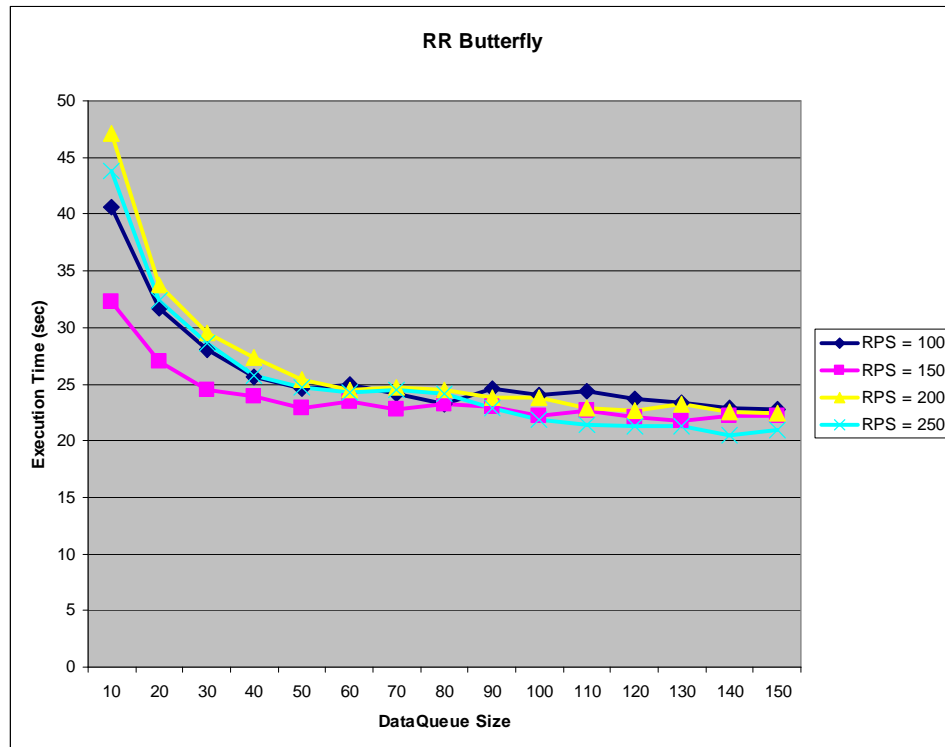


Figure 5.12 Tuning DQS in the butterfly scenario (RR)

In Figure 5.13 and in Figure 5.14 we can observe the results for the second set of experiments. We used for the *DQS* parameter the values {80, 100, 120}. The *RPS* parameter has a range from 100 to 550 tuples. In both cases (small line and butterfly scenarios) the execution time remains at the same levels with slightly a better performance between 200 and 500.

Table 5.2 Configuration of RR

	Good Areas	Configuration
<b>TmSl</b>	0	0
<b>DQS</b>	30-150	100
<b>RPS</b>	200-500	400

Based on the above observations we end up with a good configuration for RR, which is presented in Table 5.2. This configuration is used in the subsequent experiments.

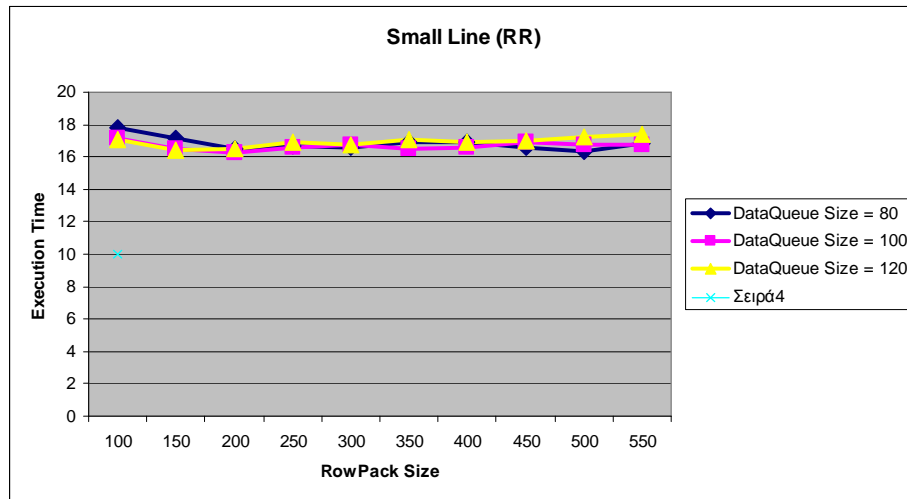


Figure 5.13 Tuning RPS in the line scenario (RR)

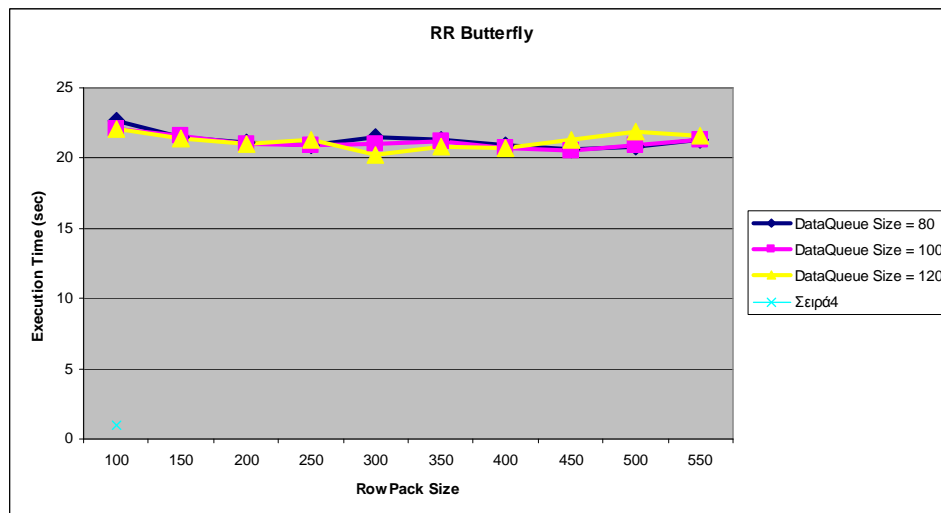


Figure 5.14 Tuning RPS in the butterfly scenario (RR)

#### 5.4.2. Tuning Minimum Cost

We conduct the same set of experiments with RR for the MC scheduling policy. The scheduling of RR and MC in the small line scenario is identical. We present results of the MC scheduler only for the butterfly scenario. In Figure 5.15 the schedule behaves in a similar manner with RR. While the value of  $DQS$  increases the execution time decreases and when  $DQS$  is over 80 the execution time remains steady.

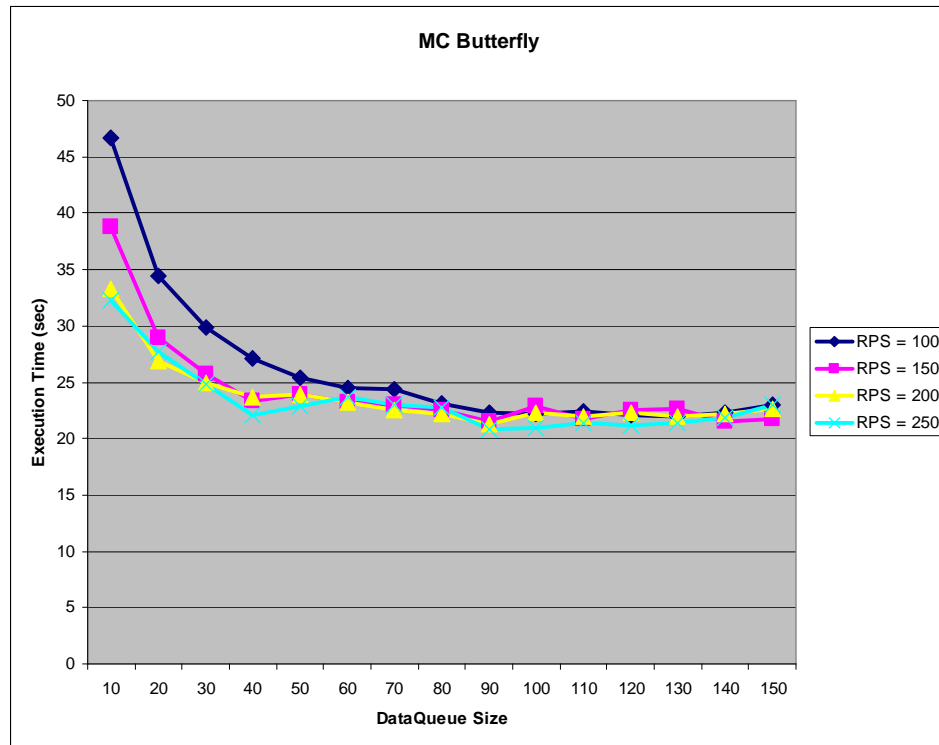


Figure 5.15 Tuning DQS in the butterfly scenario (MC)

In Figure 5.16 we can observe the behavior of MC at the second set of experiments that tunes *RPS*. The execution time is not affected at all from the different values of *RPS* we see on the chart. In Table 5.3 we can see the configuration we used for the subsequent experiments.

Table 5.3 Configuration of MC

	Good Areas	Configuration
<b>TmSl</b>	0	0
<b>DQS</b>	80-150	100
<b>RPS</b>	200-450	400

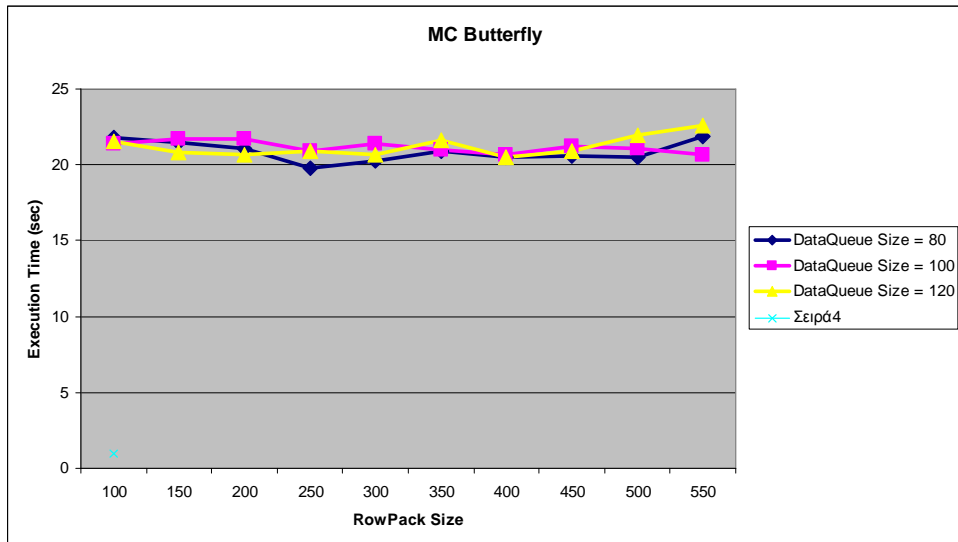


Figure 5.16 Tuning RPS in the butterfly scenario (MC)

#### 5.4.3. Tuning Minimum Memory

For the MM scheduling policy we will use the same values for  $DQS$  and  $RPS$  that we selected for RR and MC. Using the same values we can get an objective perspective of how good MM is. In Figure 5.17 and in Figure 5.18 we can see the execution time for MM at the small line and butterfly scenario for different values in the time slot parameter. As the  $TmSl$  increases the execution time decreases. The reason for that is that a communication overhead occurs since more scheduling steps are required. For the small line scenario MM seems to remain unaffected for  $TmSl$  values. The workflow size is a parameter for that behavior.

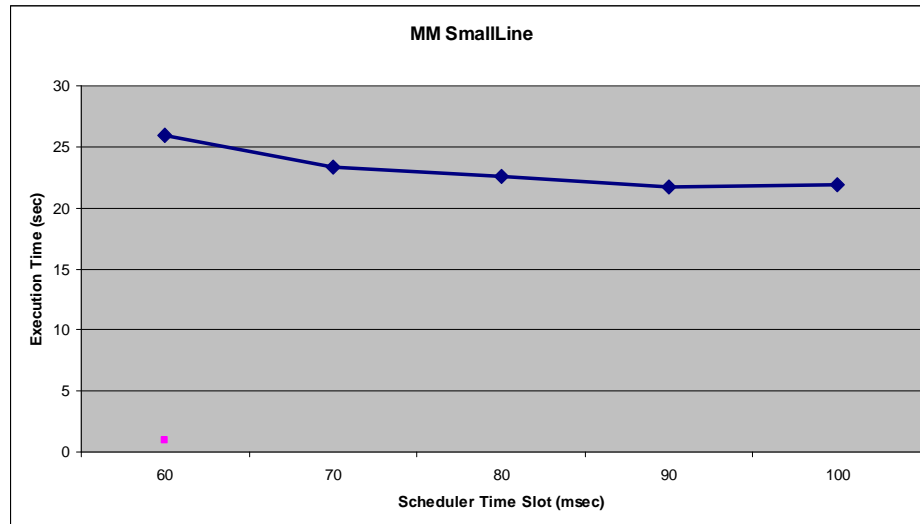


Figure 5.17 Execution time and  $TmSl$  in the small line scenario (MM)

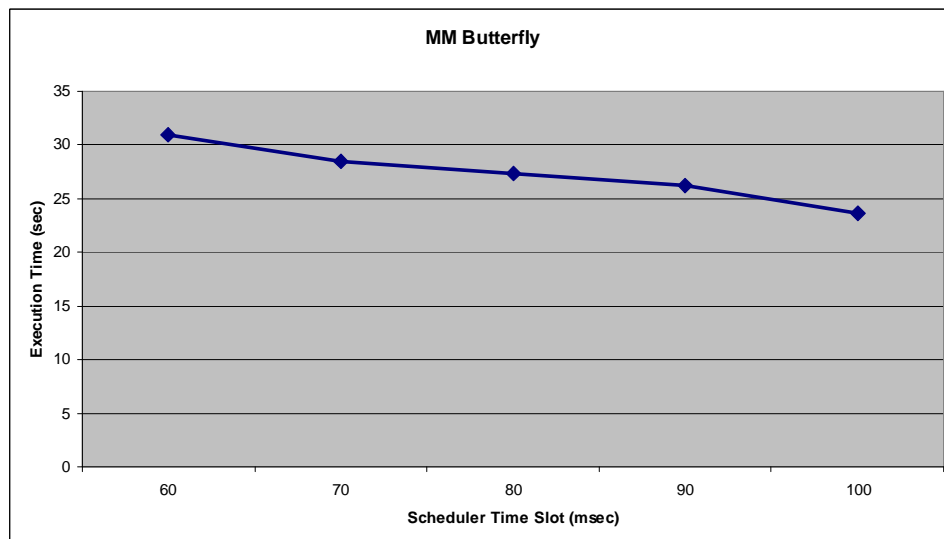


Figure 5.18 Execution time and  $TmSl$  in the butterfly scenario (MM)



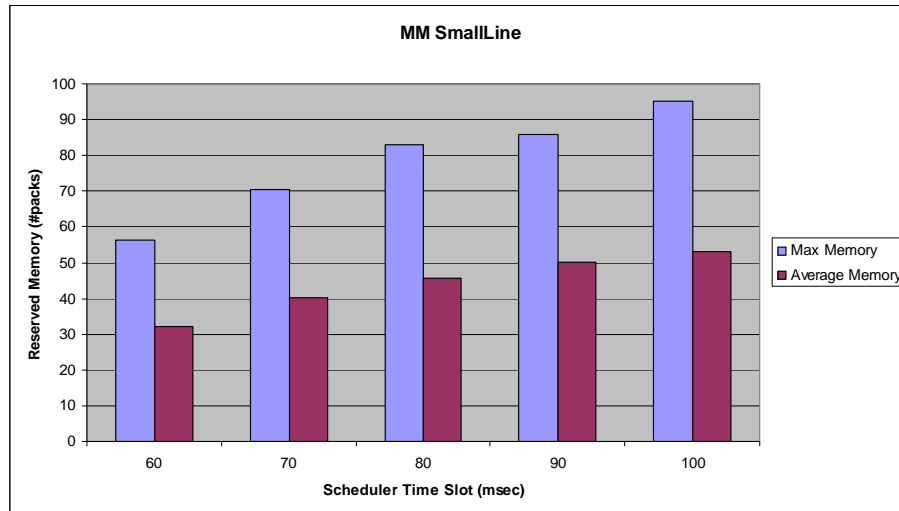


Figure 5.19 Max and avg memory and  $TmSl$  in the small line scenario (MM)

In Figure 5.19 and in Figure 5.20 we observe how the memory requirements change for different values of  $TmSl$ . Using smaller values in  $TmSl$  we achieve smaller requirements in maximum and average memory. Considering the increase of execution time for this range of  $TmSl$ , we choose to use for  $TmSl$  the value 70. In Table 5.4 we see the configuration of MM.

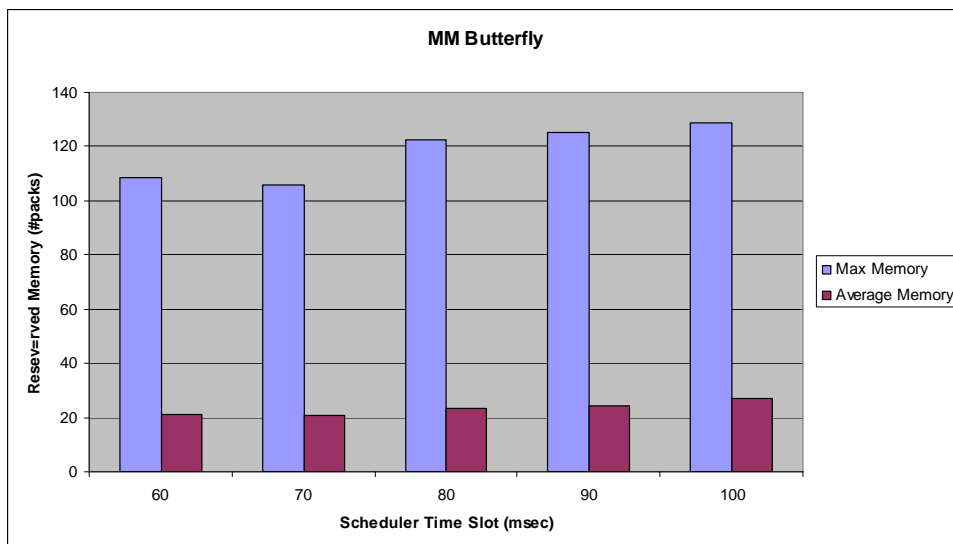


Figure 5.20 Max and avg memory and  $TmSl$  in the butterfly scenario (MM)

Table 5.4 Configuration of MM

	<b>Good Areas</b>	<b>Configuration</b>
<b>TmSI</b>	60-70	70
<b>DQS</b>	-	100
<b>RPS</b>	-	400

### 5.5. Line workflow

The experiments we present in this section show the behavior of a line scenario (Figure 5.5) with various input sizes as well with the workflow's total selectivity.

#### 5.5.1. Effect of input size

The chart in Figure 5.21 shows how execution time changes in various input sizes. MC performs better than RR especially in the case of 1GB of data input, while MC is more time consuming than MC and RR. In all three scheduling policies the increment in the y-axis is practically linear, as one would typically expect from a linear workflow.

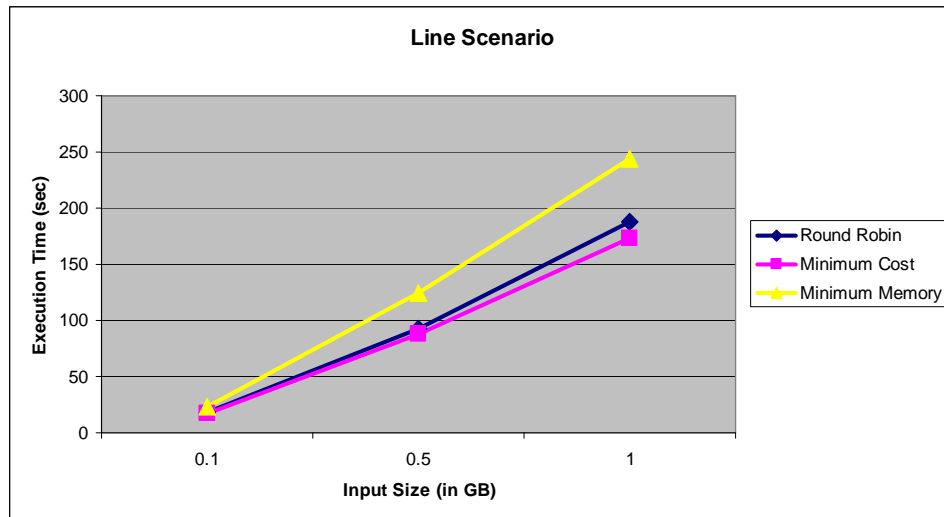


Figure 5.21 Execution time for a line scenario (Sel = 0.5)

In Figure 5.22 and Figure 5.23 we can see how the average and maximum memory requirements of the three scheduling policies. RR has the greatest requirements in average memory. MC is a bit better than RR, while MM achieves a 50% smaller memory consumption compared to MC and RR. All policies have the similar maximum requirements in memory. Since MM has much lower average values, we come to the conclusion that the result are not so often peaks during the scenario execution.

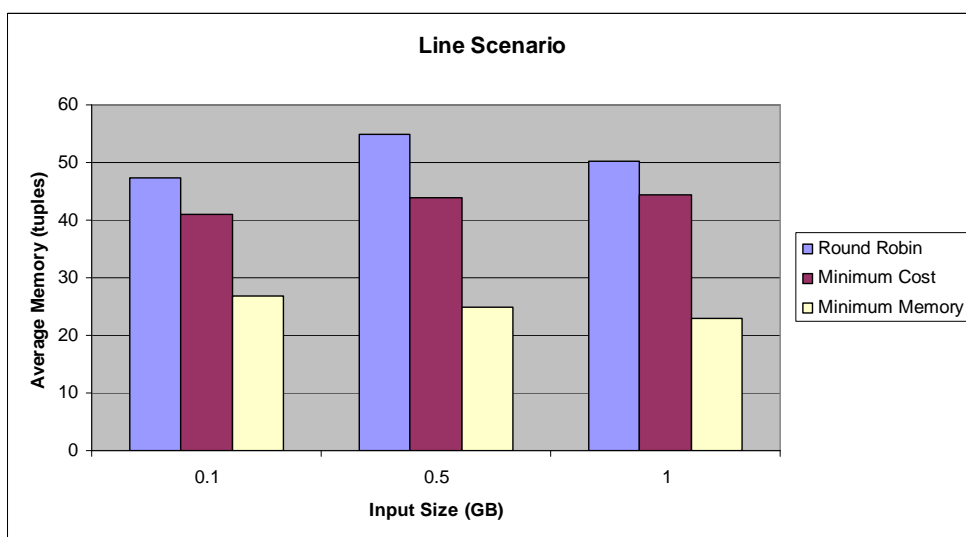


Figure 5.22 Average memory for a line scenario (Sel = 0.5)

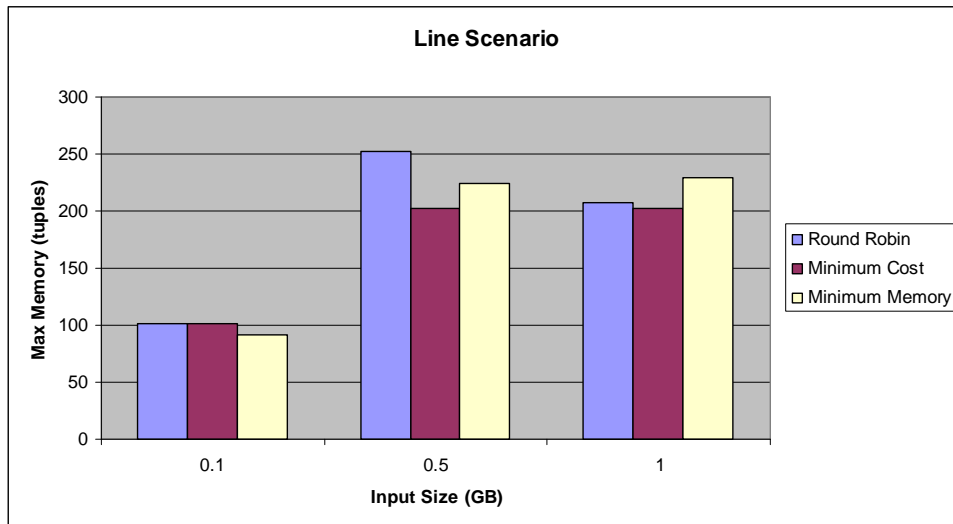


Figure 5.23 Maximum memory for a line scenario (Sel = 0.5)

### 5.5.2. Effect of workflow selectivity

In Figure 5.24 we see how our execution time changes for different selectivity values, from 0.5 to 1.0. RR and MC are close but MC performs a little better. MM needs more time to complete the execution.

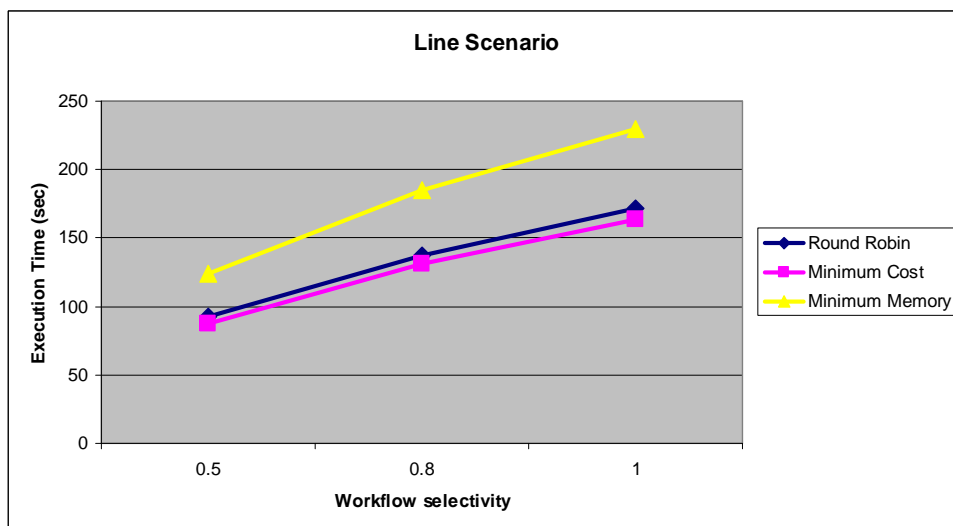


Figure 5.24 Execution time for a line scenario (SF = 0.5)

In Figure 5.25 and in Figure 5.26 we depict the memory requirements of each scheduling policy. RR has the biggest requirements in average and maximum memory. MC performs better and MM has smaller memory requirements than MC and has similar maximum requirements.

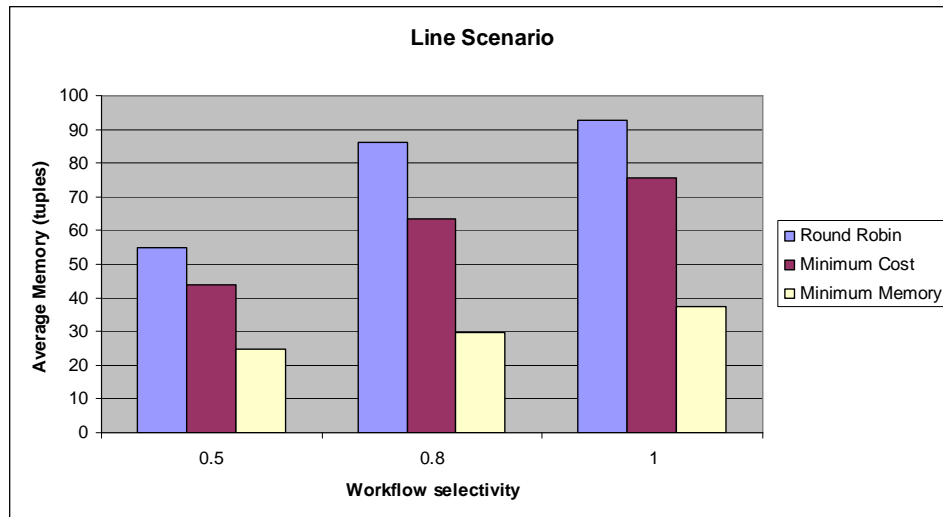


Figure 5.25 Average memory for a line scenario (SF = 0.5)

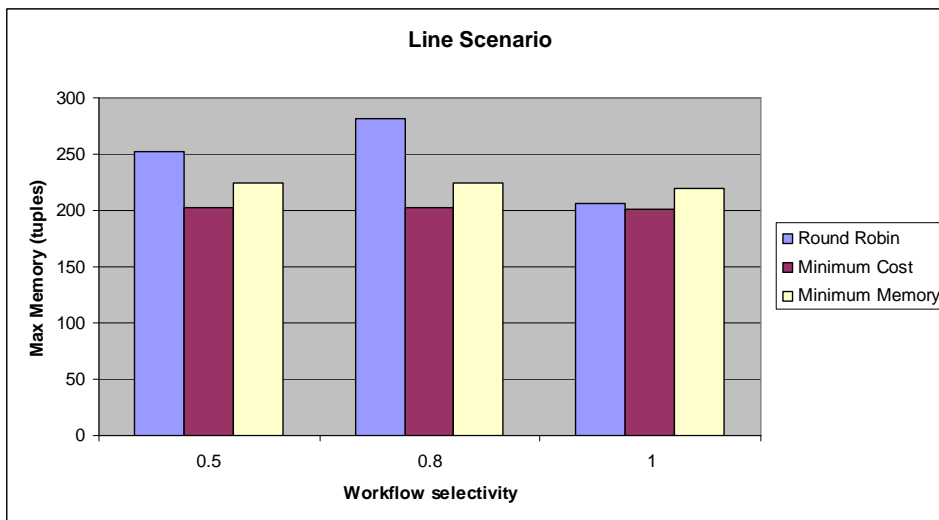


Figure 5.26 Maximum memory for a line scenario (SF = 0.5)

## 5.6. Wishbone workflow

The experiments presented in this section show the behavior of a wishbone scenario (Figure 5.6) with various input sizes as well with the workflow's total selectivity.

### 5.6.1. Effect of input size

The chart in Figure 5.27 shows how execution time changes in various input sizes. MC performs better than RR when the input size is 0.5 GB or more, while MC is more time consuming than MC and RR. In all three scheduling policies the increment in the y-axis is practically linear.



Figure 5.27 Execution time for a wishbone scenario (Sel = 0.5)

In Figure 5.28 and in Figure 5.29 we can see how the average and maximum memory requirements of the three scheduling policies. RR and MC have the greatest requirements in average memory. MM achieves a 60% memory consumption compared to MC and RR. RR and MM have the similar maximum requirements in memory. MC performs better for big input sizes. Again since MM has much lower average values, it is safe to come to the conclusion that there are not so often peaks during the scenario execution.

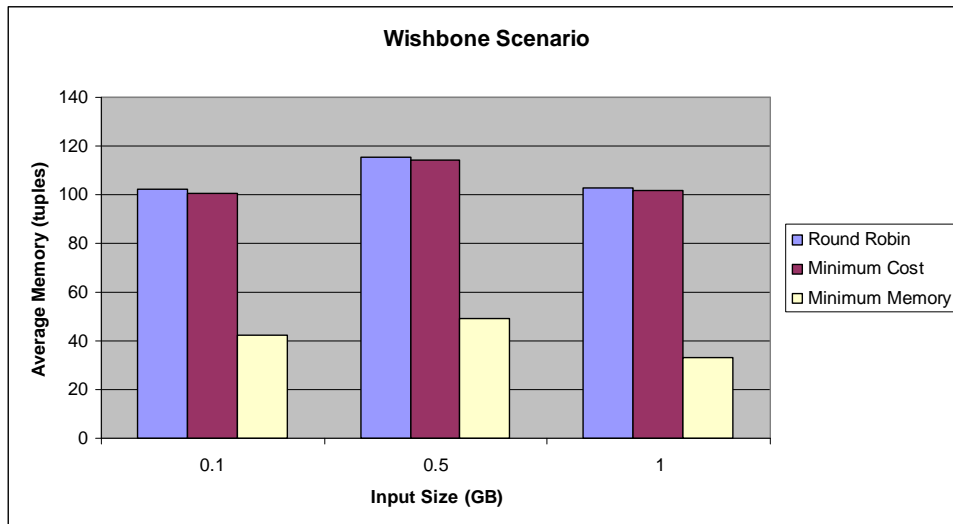


Figure 5.28 Average memory for a wishbone scenario (Sel = 0.5)



Figure 5.29 Maximum memory for a wishbone scenario (Sel = 0.5)

### 5.6.2. Effect of workflow selectivity

In Figure 5.30 we see the performance of our engine in a wishbone scenario. MC is clearly better than RR, but again MC is more time consuming than the others. It is interesting though that all algorithms behave the same when the selectivity is above 0.8 the execution time does not increment as expected but practically remains the same. This workflow has only one join operation; this operation is costly, mainly

because of the sorting actions this operator performs. The filter we used to achieve the different selectivity values is applied on the small recordset. So the big recordset in all cases is the same and its sorting process is the one that defines the sorting cost.

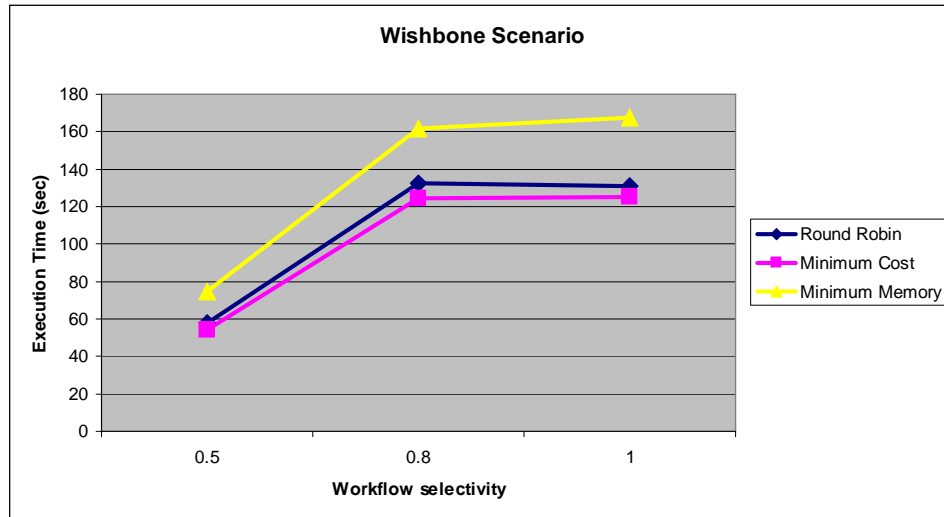


Figure 5.30 Execution time for a wishbone scenario (SF = 0.5)

In Figure 5.31 and in Figure 5.32 we see our scheduling policies memory requirements. For average memory, RR AND MC perform worse than MM who has a very low average here. For maximum memory MC is performing better than RR and MM, which have similar values.



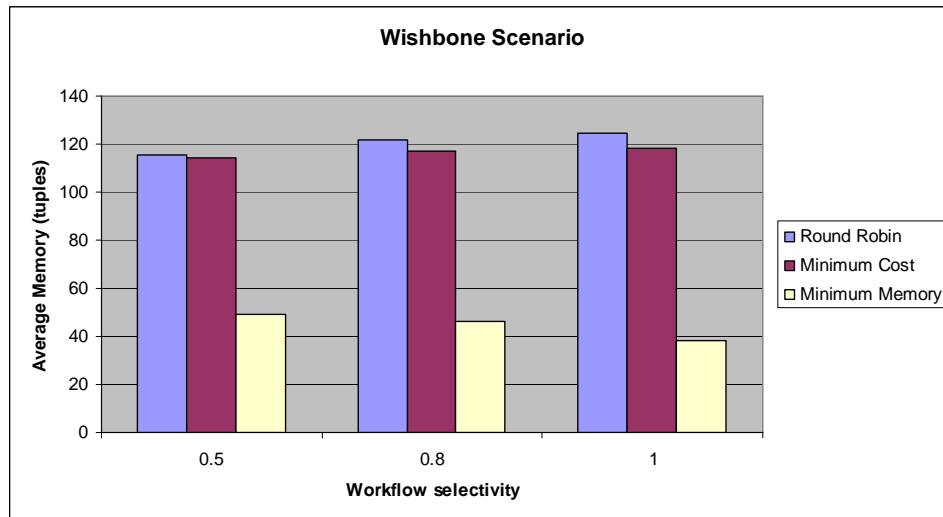


Figure 5.31 Average memory for a wishbone scenario (SF = 0.5)

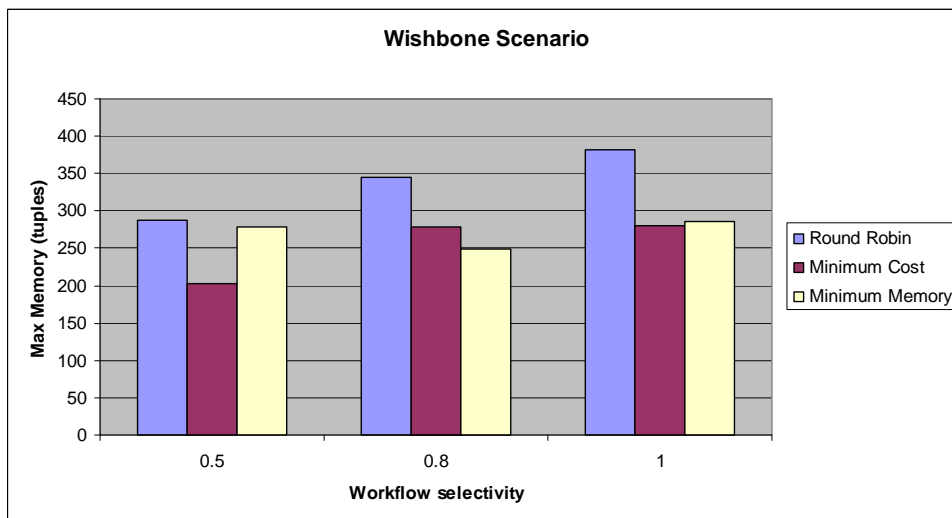


Figure 5.32 Maximum memory for a wishbone scenario (SF = 0.5)

## 5.7. Primary flow workflow

The experiments we present in this section show the behavior of a primary flow scenario (Figure 5.7) with various input sizes as well with the workflow's total selectivity.

### 5.7.1. Effect of input size

The chart in Figure 5.33 shows how execution time changes in various input sizes. MC performs slightly better than RR. Again MM is more time consuming than MC and RR. In all three scheduling policies the increment in the y-axis is practically linear. MC and RR are very close because all operators have data to process. There is no operator that all its producers are blocking activities. Even so, MC is slightly better.

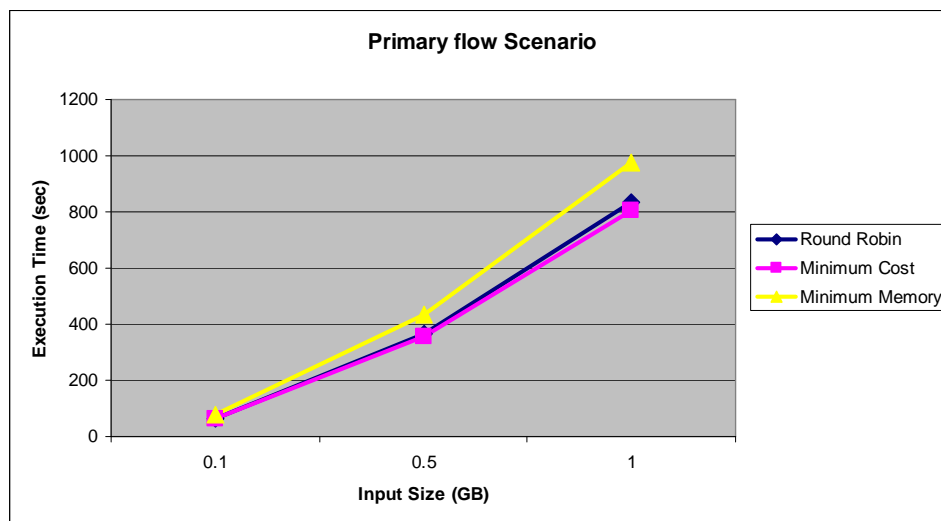


Figure 5.33 Execution time for a primary flow scenario (Sel = 0.5)

In Figure 5.34 and in Figure 5.35 we can observe the average and maximum memory requirements of our scheduling policies for the case of the primary flow scenario. RR performs much worse than the other two. The reason for this is that RR will schedule many recordsets before it schedules an activity that might consume data. Remember that in a primary flow there are many input source recordsets because of many look up tables. Concerning average memory MC has lesser requirements than RR and MM is better than MC and RR. Concerning maximum memory RR has the biggest maximum requirements in memory. MM is better than RR but MC is doing much better.

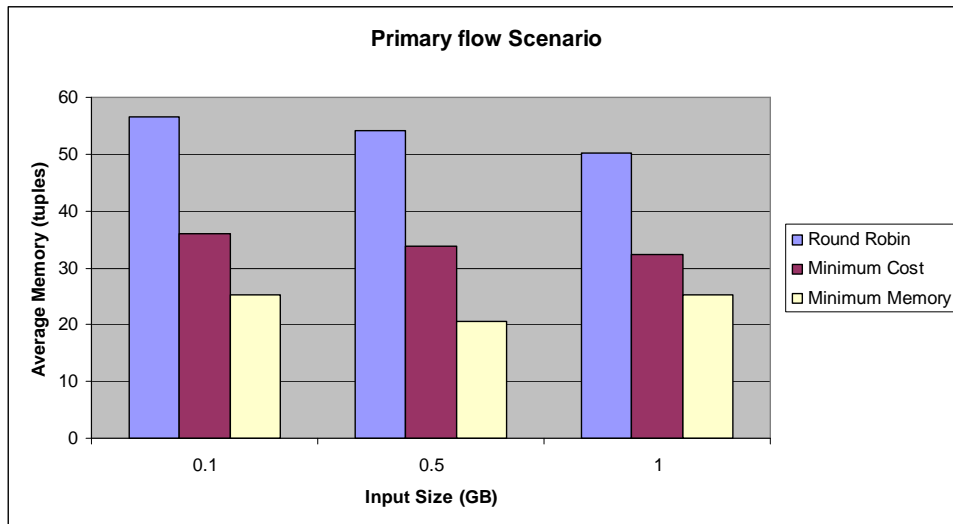


Figure 5.34 Average memory for a primary flow scenario (Sel = 0.5)

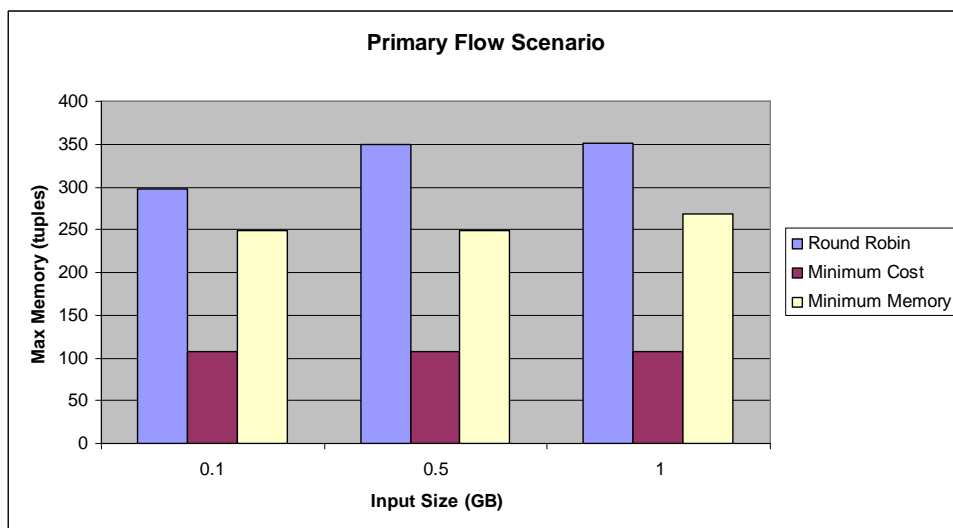


Figure 5.35 Maximum memory for a primary flow scenario (Sel = 0.5)

### 5.7.2. Effect of workflow selectivity

In Figure 5.36 we see the execution time of our scheduling policies for different workflow selectivity values. RR and MC are close, with MC having slightly better times. MM consumes more time to complete the execution of the scenario.

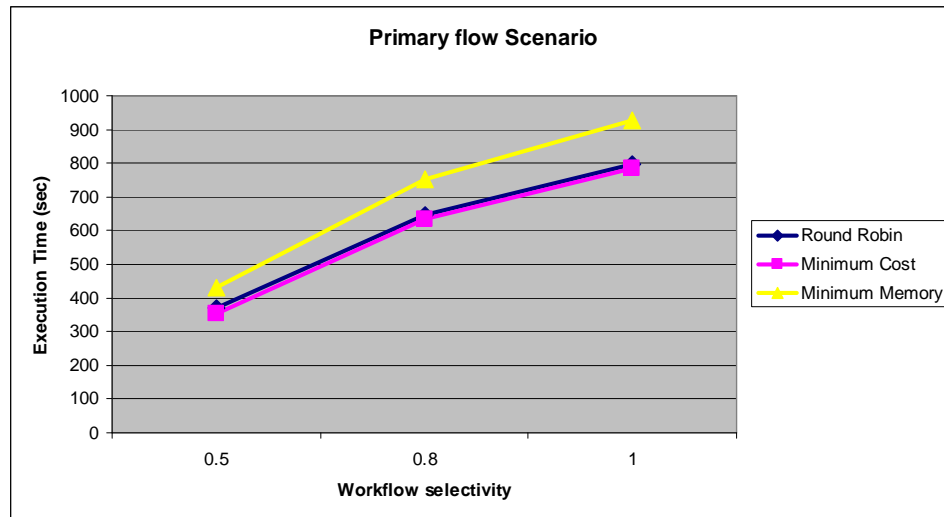


Figure 5.36 Execution for a primary flow scenario (SF = 0.5)

In Figure 5.37 and in Figure 5.38 we see the average and maximum memory requirements of our scheduling policies for the primary flow scenario. RR performs much worse than the other two. The reason for this is that RR will schedule many recordsets before it schedules an activity that might consume data, because of the presence of many input source recordsets (many look up tables). MC has less average memory requirements than RR and MM is better than MC and RR. RR has the biggest maximum requirements in memory. MM is better than RR but MC is doing better.

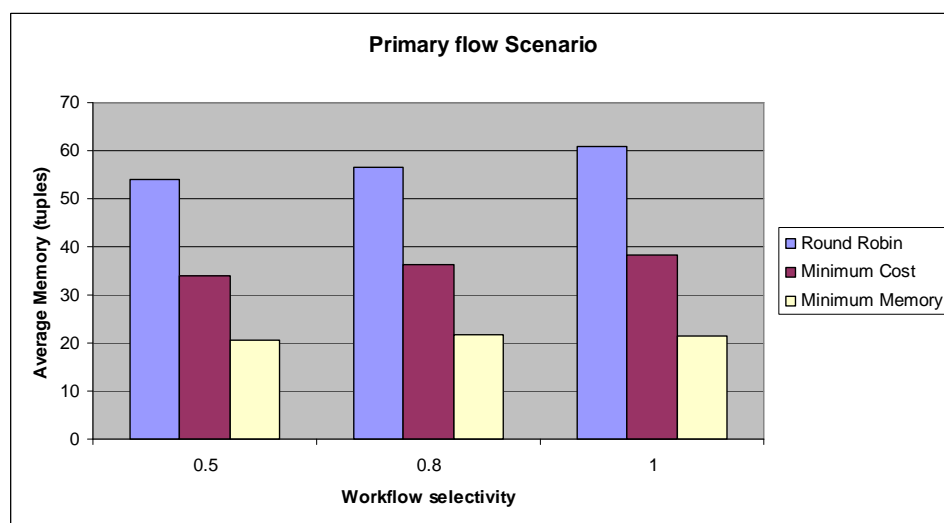


Figure 5.37 Average memory for a primary flow scenario (SF = 0.5)

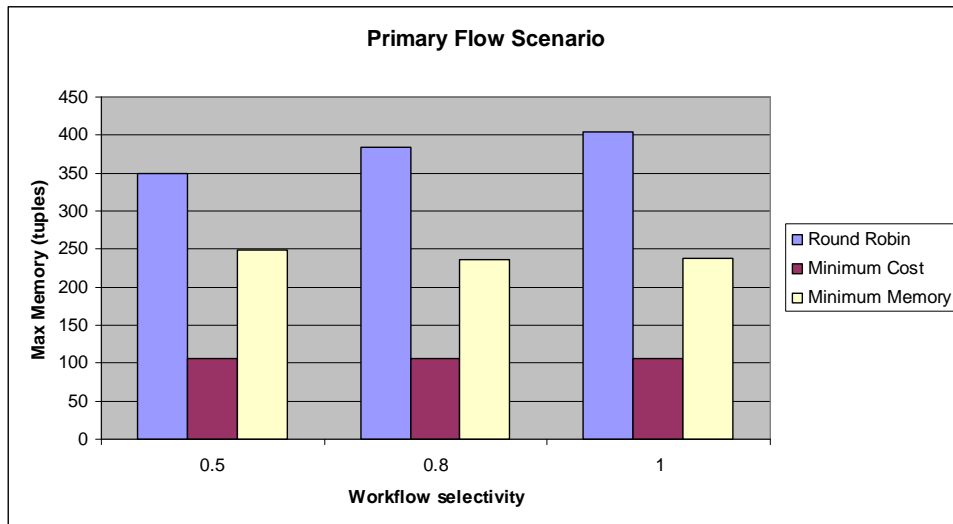


Figure 5.38 Maximum memory for a primary flow scenario (SF = 0.5)

## 5.8. Balanced butterfly workflow

The experiments we present in this section show the behavior of a balanced butterfly scenario (Figure 5.8) with various input sizes as well with the workflow's total selectivity.

### 5.8.1. Effect of input size

The chart in Figure 5.39 shows how execution time changes in various input sizes in a balanced butterfly scenario. MC performs better than RR. MM performs worse than the other two. In all three scheduling policies the increment in the y-axis is practically linear.

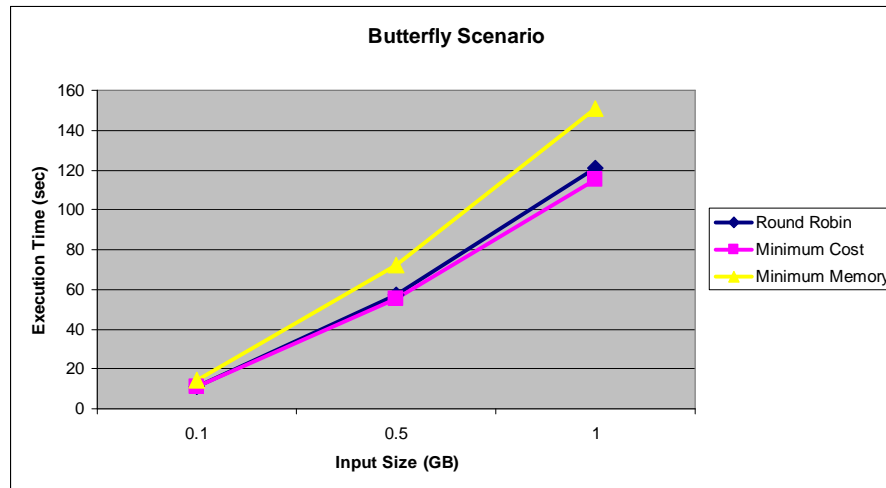


Figure 5.39 Execution time for a balanced butterfly scenario (Sel = 0.5)

In Figure 5.40 and in Figure 5.41 we see the memory demands of our scheduling policies for a balanced butterfly scenario. RR has the greatest values in average and maximum memory requirements, except when the input size is 1 GB, where RR and MC have very close values. MM is doing very well since it manages to achieve very low average memory requirements, about 15% and 20% of the demands of RR and MC. For this scenario MM has the lowest value in for maximum memory, especially when the input size is 0.1 GB; the maximum value is very small comparing to RR and MC. In a balanced butterfly scenario we have small non-blocking parts (sequence of non-blocking operators) and many blocking operators. This forces the system to gather all its input data temporarily many times. This state helps MM to avoid high memory peaks.

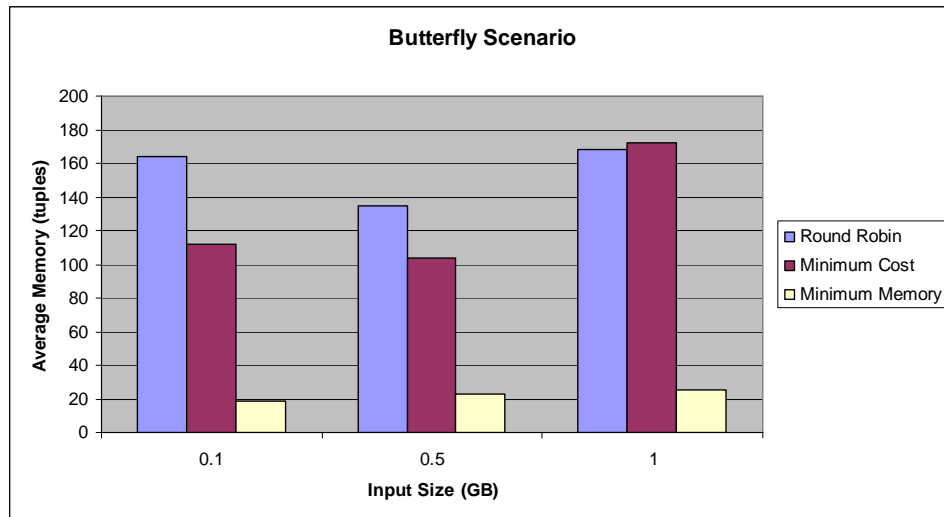


Figure 5.40 Average memory for a balanced butterfly scenario (Sel = 0.5)

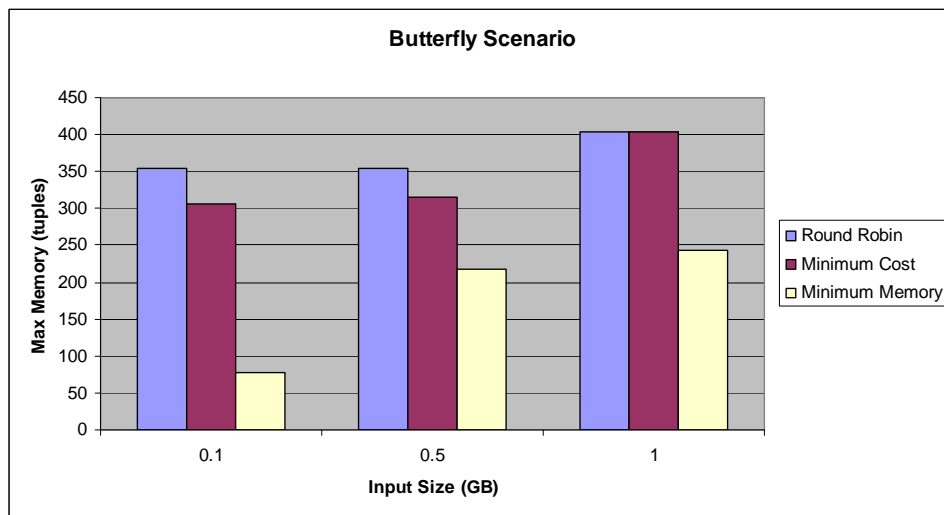


Figure 5.41 Maximum memory for a balanced butterfly scenario (Sel = 0.5)

### 5.8.2. Effect of workflow selectivity

In the case of the balanced butterfly workflow the execution time all of our scheduling policies' increases linearly (Figure 5.42) as the workflow selectivity increases. Again MC is a little better than RR, while MM is much more time consuming.

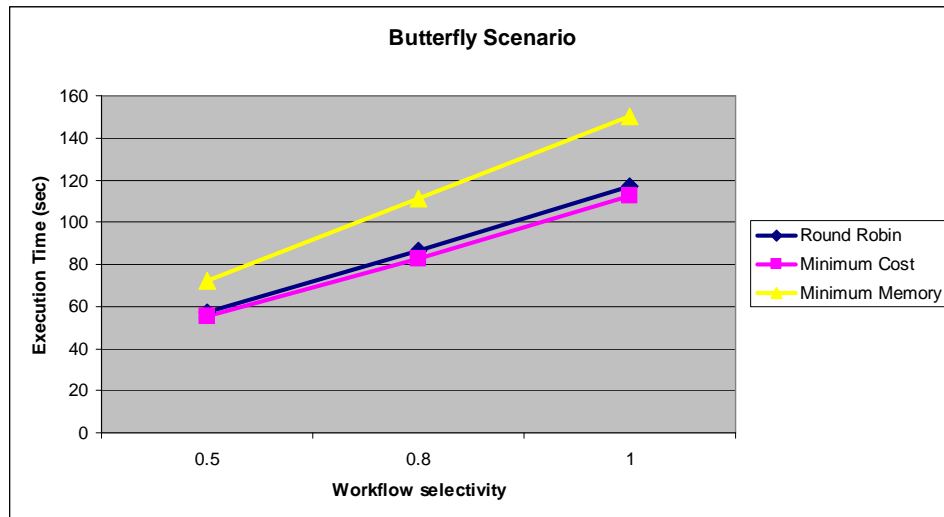


Figure 5.42 Execution time for a balanced butterfly scenario (SF = 0.5)

In Figure 5.43 and in Figure 5.44 we depict the average and maximum memory requirements for our scheduling policies. RR and MC are close, but MC outperforms RR when the selectivity is below 1.0. MM behaves very well since it requires only the 20% of average memory of MC and RR. Also, for the balanced butterfly workflow MM has the best maximum memory requirements.

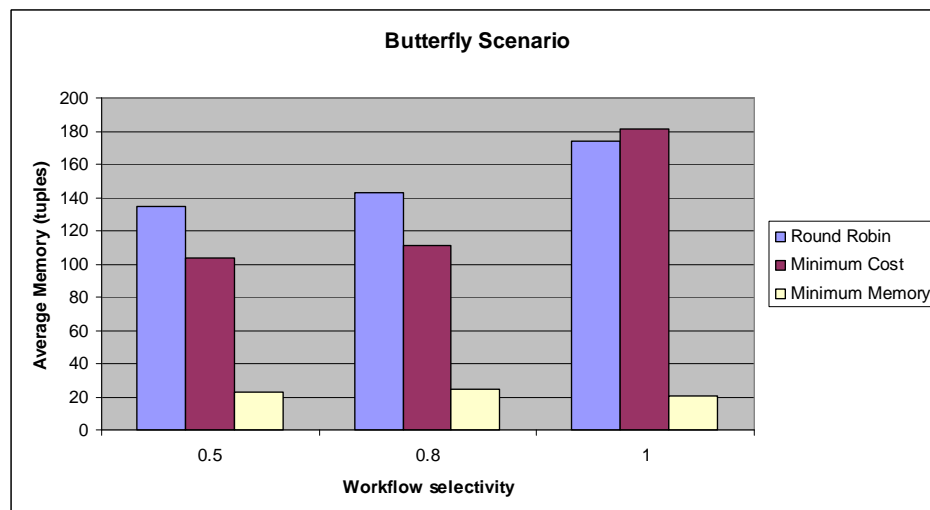


Figure 5.43 Average memory for a balanced butterfly scenario (SF = 0.5)



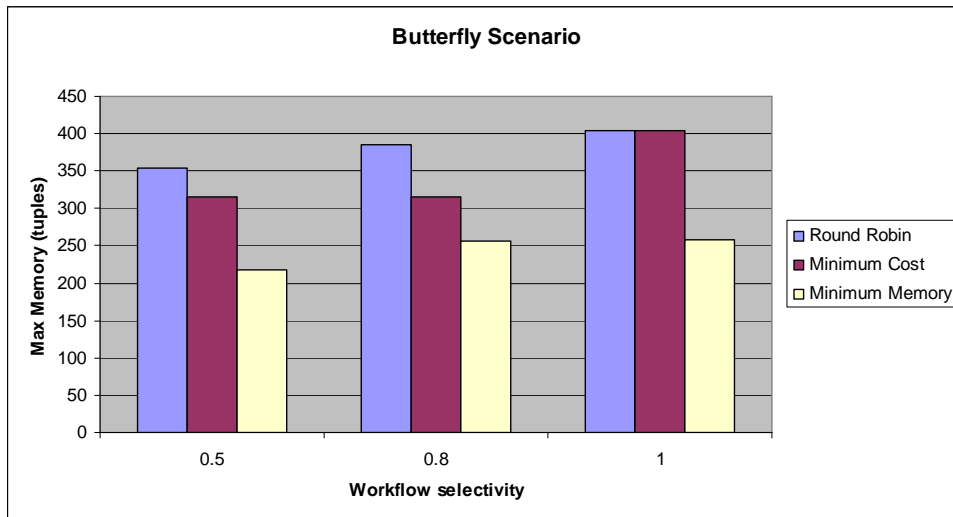


Figure 5.44 Maximum memory for a balanced butterfly scenario (SF = 0.5)

## 5.9. Tree workflow

The experiments we present in this section show the behavior of a tree scenario (Figure 5.9) with various input sizes as well with the workflow's total selectivity.

### 5.9.1. Effect of input size

In Figure 5.45 we see the time performance of the three scheduling policies as we vary the input size. Again RR and MC are very close, but MC is slightly better. MM needs more time to complete the execution of the scenario.

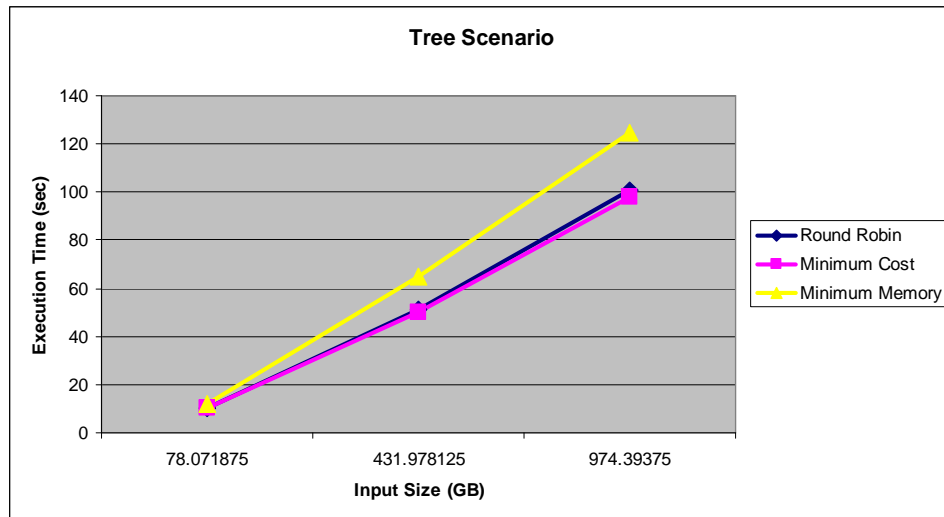


Figure 5.45 Execution time for a tree scenario (Sel = 0.5)

In Figure 5.46 and in Figure 5.47 we see the memory requirements for the tree scenario. RR and MC are close, but MC is performing slightly better. MM has about the 20-25% of RR's and MC's average memory requirements. In the case of the maximum memory metric all policies are close except for MC, where in the case of 0.1GB has a small maximum value, since the input is small (therefore the execution time was also small), there were no peaks during the execution.

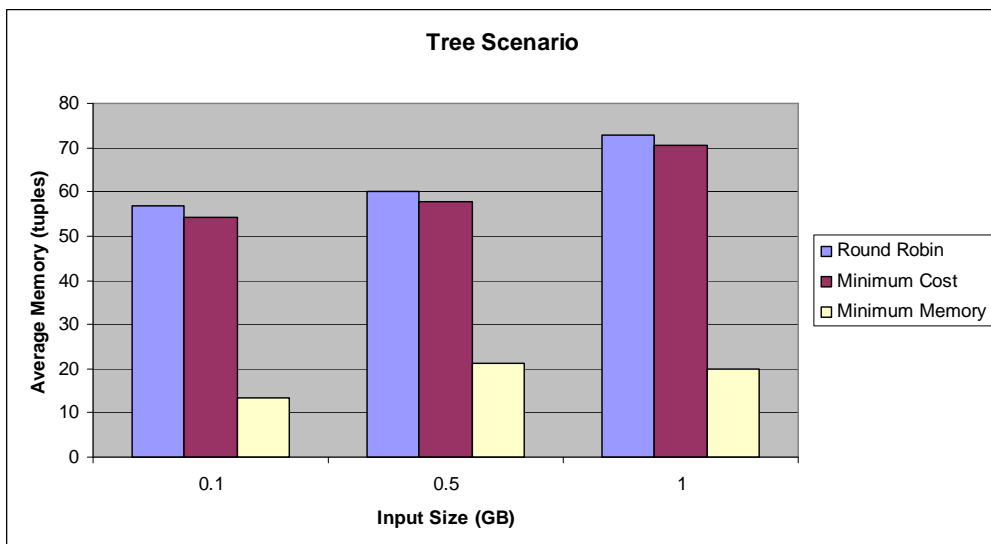


Figure 5.46 Average memory for a tree scenario (Sel = 0.5)

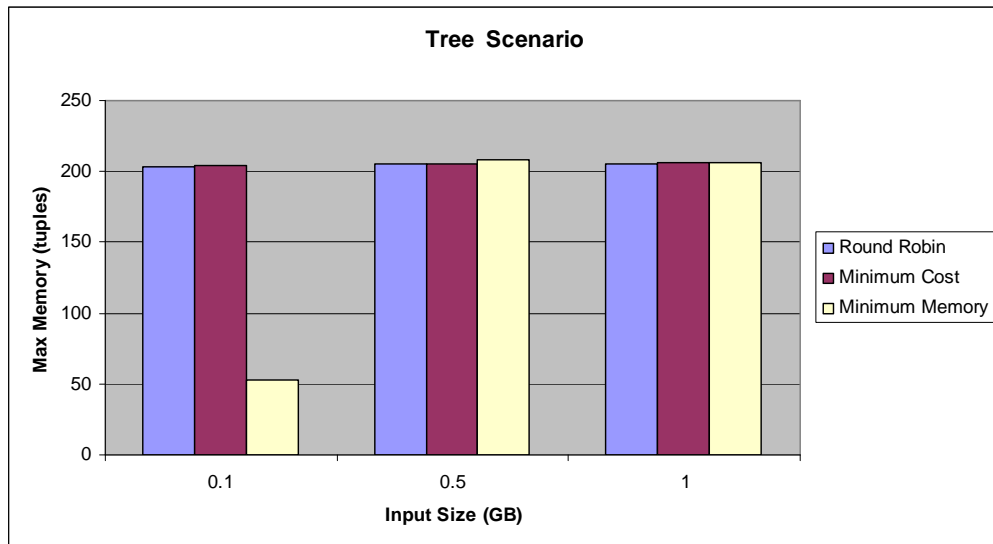


Figure 5.47 Maximum memory for a tree scenario (Sel = 0.5)

### 5.9.2. Effect of workflow selectivity

In Figure 5.48 we can observe the time each scheduling policy needs to complete the execution of a tree scenario. All three scheduling policies behave as expected. The execution time increases slowly and RR is slightly worse than MC. Finally MM needs more time to finish.

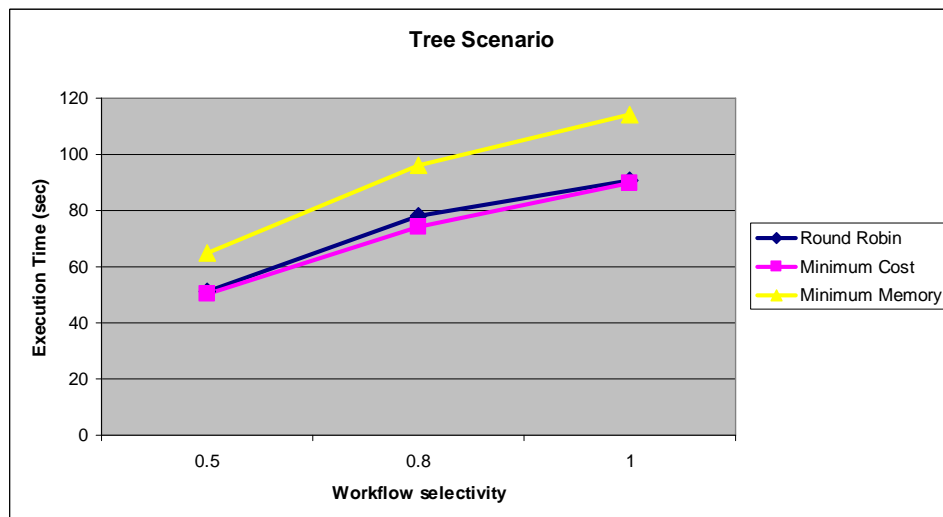


Figure 5.48 Execution time for a tree scenario (SF = 0.5)

In Figure 5.49 and in Figure 5.50 we can observe the memory requirements of each scheduling policy. RR and MC have the biggest requirements in average memory. MM has smaller average memory requirements. All scheduling policies have similar maximum memory requirements except for the case of (sel=1.0) where MM has a distinguishably bigger maximum value.

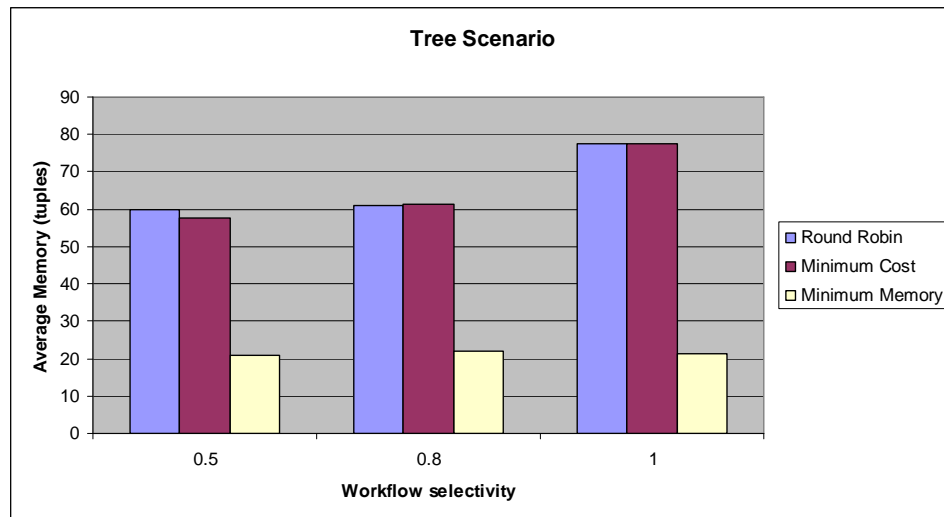


Figure 5.49 Average memory for a tree scenario (SF = 0.5)



Figure 5.50 Maximum memory for a tree scenario (SF = 0.5)

## 5.10. Fork workflow

The experiments we present in this section show the behavior of a **fork** scenario (Figure 5.10) with various input sizes as well with the workflow's total selectivity.

### 5.10.1. Effect of input size

In Figure 5.51 we see the time performance of the three scheduling policies as we vary the input size. Again RR and MC are very close, but MC is a little better. MM needs much more time to complete the execution of the scenario.

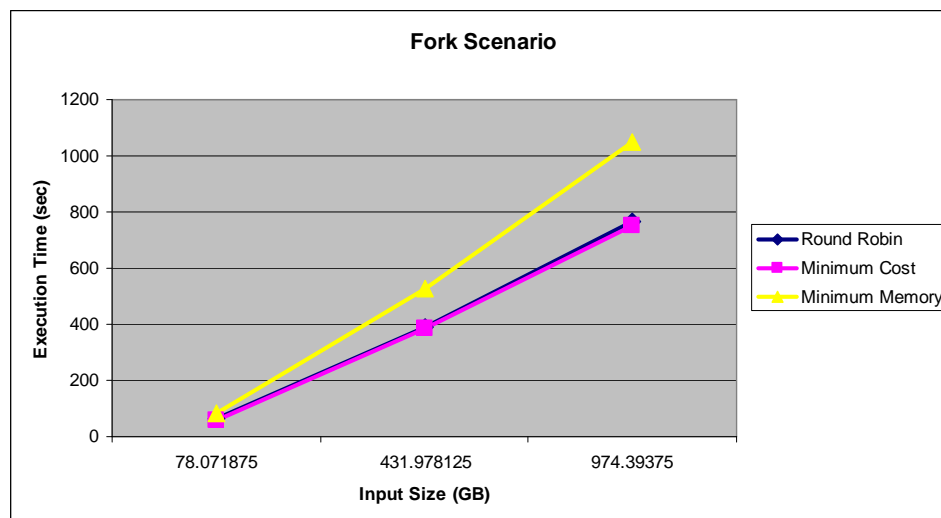


Figure 5.51 Execution time for a fork scenario (Sel = 0.5)

In Figure 5.52 and in Figure 5.53 we see our scheduling policies' memory requirements. Concerning average memory, RR performs worse than the other two, while MC is a little better than RR. MM has a very low average here. For maximum memory MC is performing much better than RR and MM, which have similar values. Again when the input is small MM has the smallest maximum memory.

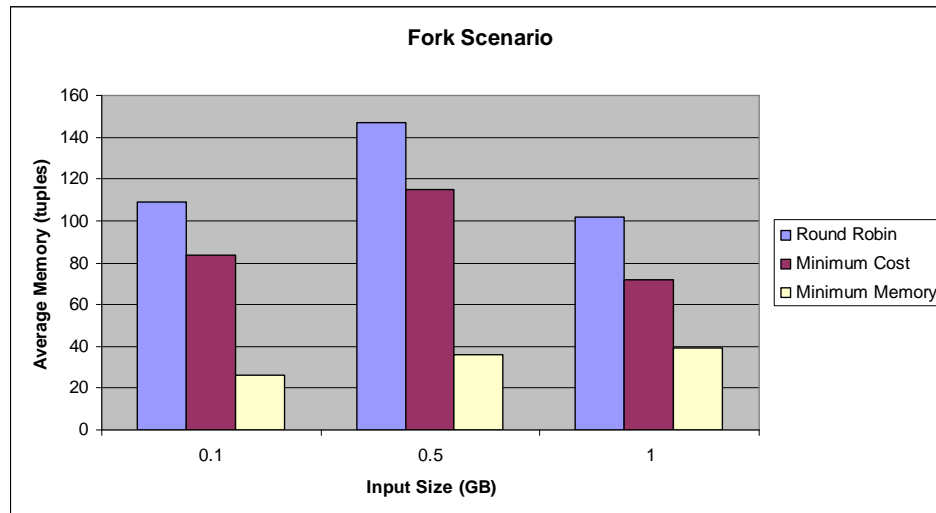


Figure 5.52 Average memory for a fork scenario (Sel = 0.5)

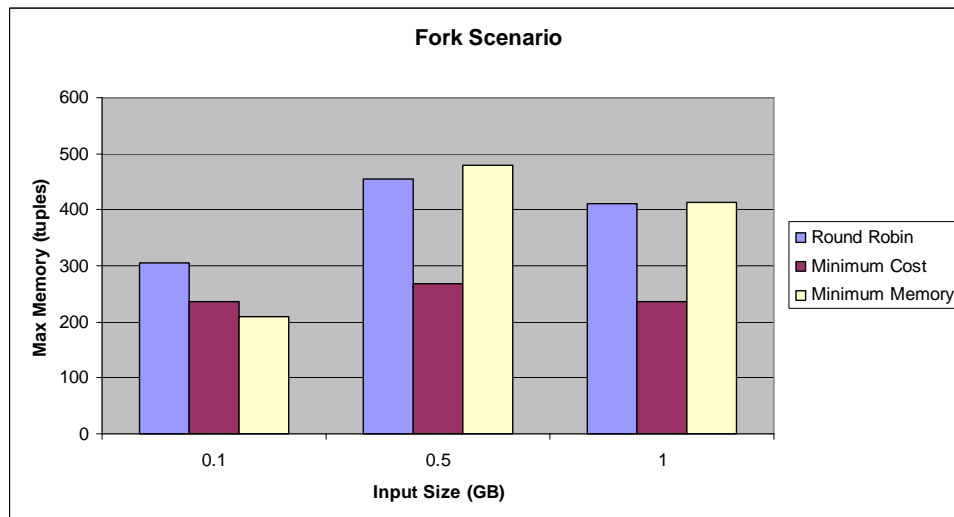


Figure 5.53 Maximum memory for a fork scenario (Sel = 0.5)

### 5.10.2. Effect of workflow selectivity

In Figure 5.54 we see how our scheduling policies perform in the case of a fork scenario. For all scheduling policies the execution time increases linearly.

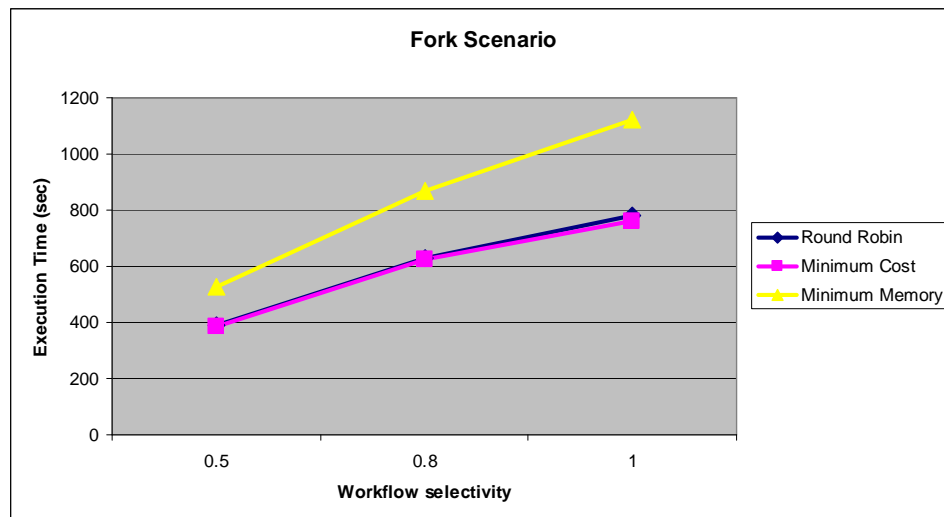


Figure 5.54 Execution time for a fork scenario (SF = 0.5)

In Figure 5.55 and in Figure 5.56 we see the average and maximum memory requirements for a **fork** scenario. RR has the worst average memory requirements and MC is doing a little better than RR. MM though outperforms RR and MC, having very low average values. All three scheduling policies have similar values for maximum memory.

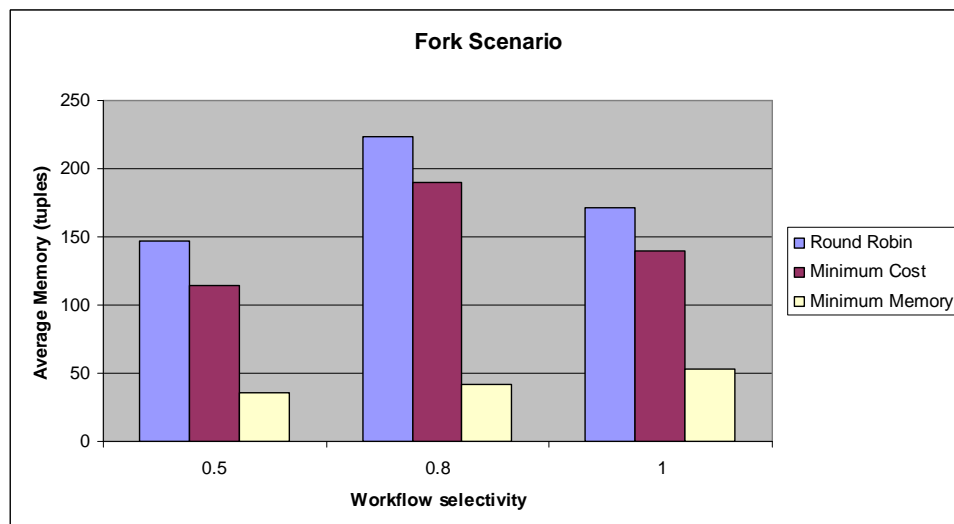


Figure 5.55 Average memory for a fork scenario (SF = 0.5)

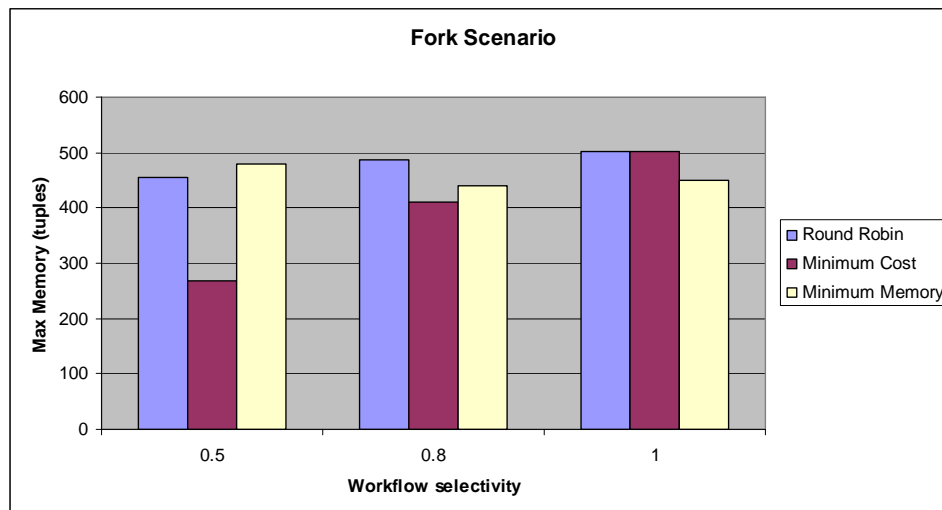


Figure 5.56 Maximum memory for a fork scenario (SF = 0.5)

### 5.11. Observations deduced from experiments

At first we conducted some preface experiments so that we can tune and optimize each scheduling policy. For RR and MC we found a good set of values that optimize the executions' time. For MM we chose asset of values that could give a good execution time as well as distinctly smaller memory requirements.

From our experiments we come to the following conclusions:

- **RR**: This simple scheduling policy does not perform well; in all cases was worse than MC, in terms of execution time and memory requirements, both average and maximum.
- **MC**: This scheduling policy outperforms the other two for the execution time metric. Also, in most cases it has better maximum and average memory requirements.
- **MM**: This scheduling policy manages to outperform the other two, when it comes to average memory requirements. MM could be used in an environment where more than one concurrent operations run, and being memory efficient is important, but memory can be available at peak times.



In general, our three scheduling algorithms increment their execution time linearly as the selectivity of a workflow or the input size increases. As the selectivity or the input size increase, MC outperforms RR. Also, regardless of the input size or the selectivity the average memory requirements are not affected. Also, when the execution time of a scenario is relatively small, MM might not have any peaks at all.



## CHAPTER 6. CONCLUSIONS AND FUTURE WORK

---

### 6.1 Conclusions

### 6.2 Future Work

---

#### 6.1. Conclusions

In this thesis we designed an ETL engine, powerful enough to support all possible data operations. The architecture of our engine is simple. Every logical-level activity that participates in an ETL scenario is implemented in more than one physical-level operators. Every operator participating in the scenario's execution is performed by using a single thread. The threads communicate and exchange data, through the data structures they share. Disk usage is necessary only by blocking operators for saving data temporarily, e.g., when they need to sort their input and the size is too big to fit in the system's main memory. The progress of the execution is controlled by a monitor thread. The monitor thread performs the execution's scheduling. At every scheduling step the monitor activates the operator the scheduler suggests.

In our system we have implemented three scheduling policies. Round Robin (RR), Minimum Cost (MC) and Minimum Memory (MM). RR is a simple and fair scheduling policy, since it schedules the operators according to a pre-defined order. MC schedules the operator that has many data to process, achieving better execution times. Finally MM is a time slot-based scheduling policy, and at every scheduling step it selects the operator that consumes many data. We consider that an operator consumes data when it process and rejects data.

Finally, a set of fiducial ETL workflows is proposed as an experimental methodology, lacking related methodology in the research area of ETL tools. This well organized set contains a broad variety of workflows covering many cases of ETL scenarios.

## 6.2. Future Work

There are many issues that are of interest for future research. The execution engine, though well designed, can be expanded to a more mature architecture. Also, there are a few issues concerning our scheduling policies.

- A set of well designed software modules (page-based database algorithms) could be embedded so that the common operations supported by the engine can function in a more efficient manner. For example, our external sorter is one issue, since the engine has no control over it and any unexpected behavior cannot be handled (e.g., a possible crash would require the sorting to start over). The adaptation of optimizing techniques is also a good opportunity for future research.
- A more specialized design for the physical-level object would offer the ability to embed easily more activity types. The design of binary and unary templates is a first step towards this direction.
- One important issue in this engine is the communication cost we experienced while conducting our experiments. A lighter and faster messaging system could benefit all scheduling policies.
- A failure handling system could also be designed, so that in case of a system failure (e.g., process termination), the engine could recover and continue the scenario's execution.
- MM can be improved so that we will not experience any peaks in maximum memory requirements.
- Also a different approach could be used, based on the idea that some operators need more memory to keep their input tuples, while the sum of all queues capacity will remain fixed.
- Finally, adapting our scheduling policies in order to schedule not only one operator at each scheduling step is of interest, since multi-core computers are

very common in our days. This could be achieved by having the scheduler to propose two operators instead of one, the one that seems most appropriate and the operator that is the second most appropriate.



## BIBLIOGRAPHY

---

[Asc03] Data Warehousing Technology.

Available at: <http://www.ascentialssoftware.com/products/datastage.html/>

[AntLR07] *A parser generator tool*. Available at: [http://www.antlr.org/](http://wwwantlr.org/)

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, *Chain: Operator Scheduling for Memory Minimization in Data Stream Systems*, SIGMOD 2003, June 912, 2003, San Diego, CA.

[CCR+03] Don Carney, Ugur Cetintemel, Alex Rasin, Stan Zdorik, Mitch Cherniack, Mike Stonebraker, *Operator Scheduling in a Data Stream Manager*, Proceedings of the 29th VLDB Conference, Berlin, Germany 2003.

[Cygwin07] A UNIX emulator for Windows.

Available at: <http://www.cygwin.com/>

[DSRS01] Nilesh N. Dalvi, Sumit K Sanghai, Prasan Roy, S. Sudarshan. *Pipelining in MultiQuery Optimization*, PODS '01 Santa Barbara USA.

[Gart03] *ETL Magic Quadrant Update: Market Pressure Increases*. Gartner's Strategic Data Management Research Note, M-19-1108.

[GFSS00] Galhardas, H., Florescu, D., Shasha, D., and Simon, E. (2000). *Ajax: An Extensible Data Cleaning Tool*. In Proceedings ACM SIGMOD International Conference on the Management of Data, page 590, Dallas, Texas.

[IBM07] IBM Data Warehouse Manager.

Available at <http://www-306.ibm.com/software/data/integration/datastage/>

[Infrm07] Informatica (2007). PowerCenter 8. Available at:

<http://www.informatica.com/products/powercenter/>

[Inmo02] W. Inmon, *Building the Data Warehouse*, John Wiley & Sons, Inc. 2002.

[Oracle07]. *Oracle Warehouse Builder 10g*. Retrieved, 2007.

Available at <http://www.oracle.com/technology/products/warehouse/>

[OtPo06] Raghunath Othayoth, Meikel Poes. *The Making of TPC-DS*, VLDB '06, September 12–15, 2006, Seoul, Korea.

[RaHe01] Raman, V., and Hellerstein, J. (2001). *Potter's Wheel: An Interactive Data Cleaning System*. In Proceedings of 27th International Conference on Very Large Data Bases (VLDB), pages 381-390, Roma, Italy.

[Sched06] Description of scheduling algorithms in Wikipedia, 2006.

<http://en.wikipedia.org/wiki/Scheduling%28computing>

[SiVS05] A. Simitsis, P. Vassiliadis, T. Sellis. *Extraction-Transformation-Loading (ETL) Processes*. In L.C.Rivero, J.H.Doom, V.E.Ferragine (eds.), *Encyclopedia of Database Technologies and Applications*, Idea Group, August 2005, ISBN 1-59140-560-2.

[SSIS07] Microsoft. SQL Server 2005 Integration Services (SSIS).

Available at: <http://technet.microsoft.com/en-us/sqlserver/bb331782.aspx/>

[SVSS03] Simitsis, A., Vassiliadis, P., Skiadopoulos, S., and Sellis, T. *Modeling of ETL Processes Using Graphs*. In the Proceedings of 2nd Hellenic Data Management Symposium (HDMS03), Athens, Greece.



[SVSS07] A. Simitsis, P. Vassiliadis, S. Skiadopoulos, T. Sellis. *Data Warehouse Refreshment*. In R. Wrembel, C. Konsilia (eds.), *Data Warehouses and OLAP: Concepts, Architecture and Solutions*, IGI Global, ISBN-10: 1599043645, ISBN-13: 978-1599043647, 2007.

[TPCH07] *The TPC-H benchmark*. Available at: <http://www.tpc.org/tpch/>

[TPCDS07] *The TPC-DS benchmark*. Available at:

<http://www.tpc.org/tpcds/tpcds.asp>

[Tzio06] Vasiliki Tziouara, *Order aware workflows*. MSc Thesis, Computer Science, University of Ioannina, Hellas, 2006.

[UnSched07] Jinzhong Niu *Uniprocessor Scheduling*, Available at:

[www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/1201-](http://www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/1201-)

[UniprocessorScheduling.pdf/](http://www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/1201-UniprocessorScheduling.pdf/)

[UrFr01] Tolga Urhan, Michael J. Franklin, *Dynamic Pipeline Scheduling for Improving Interactive Query Performance*, Proceedings of the 27th VLDP Conference, pp 1-10, Roma, Italy, 2001.

[VaSS02] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Modeling ETL Activities as Graphs. In Proceedings of the 4<sup>th</sup> International Workshop on the Design and Management of Data Warehouses (DMDW'2002) in conjunction with CAiSE'02, pp. 52-61, Toronto, Canada, May 27, 2002.

[VaSS02] Vassiliadis, P., Simitsis, A., and Skiadopoulos, S. *Conceptual Modeling for ETL Processes*. In the Proceedings of the 5th Data Warehousing and OLAP (DOLAP '02 ), McLean, VA, USA.

[VSG+05] Panos Vassiliadis, Alkis Simitsis, Panos Georgantas, Manolis Terrovitis, Spiros Skiadopoulos. *A generic and customizable framework for the design of ETL*

*scenarios*. Information Systems, vol. 30, no. 7, pp. 492-525, November 2005, Elsevier Science Ltd.

[VVS+01] Panos Vassiliadis, Zografoula Vagenas, Spiros Skiadopoulos, Nikos Karayannidis, Timos Sellis. *Arktos: Towards the modeling, design, control and execution of ETL processes*. Information Systems, vol. 26, no. 8, pp. 537-561, December 2001, Elsevier Science Ltd.

## APPENDIX

---

Table A.1 Experiments from the Aurora Scheduler [CCR+03]

Description	Y-axis	X-axis
Comparison between the thread-per-box and the Aurora architecture. The thread-per-box is not scalable.	Average Latency (seconds)	Number of Boxes
The second experiment shows that two level scheduling (application at a time) is more efficient than simple scheduling (box at a time), specifically using the MC strategy.	Average Latency (seconds)	System load (input queue capacity)
Comparison between MC and ML strategies on average latency for different processing costs in each operator box.	Average Latency (seconds)	Cost per box (msec)
Comparison of MC, ML and MM strategies for memory consumption during the run of a superbox.	Memory required (normalized on MM)	Time (sec)
This experiment shows how tuple batching can reduce overhead in bursty inputs. There are measures for three burst sizes, in each case there is less overhead when the train size is bigger	Average overhead (tuples / sec)	Train size
A graph that shows the distribution of execution workloads with different scheduling tactics. The tactics compared are "tuple at a time", "tuple train" and "superbox"	Relative overhead (percentage values from 0 to 100)	The three scheduling tactics

Table A.2 Experiments from the Chain Scheduler [BBDM03]

Description	Y-axis	X-axis
A simple comparison of FIFO and greedy scheduling. The greedy algorithm performs much better.		
Comparison of all scheduling algorithm in a single stream with two operators and a real data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with two operators and a synthetic data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with four operators and a real data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with four operators and a synthetic data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with two operators and a synthetic data set and $s > 1$ .	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with sliding-window join, three selections and a real data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm in a single stream with sliding-window join, three selections and a synthetic data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm with multiple queries and a real data set	total queue size (Kbytes)	Time (msec)
Comparison of all scheduling algorithm with multiple queries and a synthetic data set	total queue size (Kbytes)	Time (msec)

Table A.3 Experiments from the X-Join Scheduler [UrFr01]

Description	Y-axis	X-axis
Shows how the scheduling algorithms behave when we schedule 4 streams with 16 input relations (lower is better)	Time (sec)	Size of final output (#tuples)
Shows how the scheduling algorithms behave when we schedule 2 streams with 4 input relations (lower is better)	Time (sec)	Size of final output (#tuples)
These results show the selective input and join processing behave, with each algorithm. Also they measure the simple case of ordered and unordered input data. Joined relations are of equal size. Here are the results after 5 seconds of execution.	Percentage of final output	Methods that are compared
Here are the results after 25 seconds of execution. Joined relations are of equal size	Percentage of final output	Methods that are compared
Here are the results after 5 seconds of execution. Joined relations are not of equal size.	Percentage of final output	Methods that are compared
Here are the results after 25 seconds of execution. Joined relations are not of equal size.	Percentage of final output	Methods that are compared



## **SHORT CV**

---

Anastastios Karagiannis was born in Thessaloniki in 1982 and finished high school in 1999. He obtained his B.Sc. in Computer Science in 2004 from the Computer Science Department of the University of Ioannina. Anastastios Karagiannis has enrolled in the Graduate Program of the Computer Science Department of the University of Ioannina as an M.Sc. candidate in the academic year 2003 - 2004. His research interests are Databases, Data Warehouses and ETL tools.





