Pedro Miguel de Oliveira Martins

ELASTIC ETL+Q FOR ANY DATA-WAREHOUSE USING TIME BOUNDS

Tese de Doutoramento em Programa de Doutoramento em Ciências e Tecnologia da Informação da Faculdade de Ciências e Tecnologia, orientada por Pedro Nuno San-Bento Furtado e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Julho, 2015

· U C ·

UNIVERSIDADE DE COIMBRA

Pedro Miguel de Oliveira Martins

ELASTIC ETL+Q FOR ANY DATA-WAREHOUSE USING TIME BOUNDS

Tese de Doutoramento em Programa de Doutoramento em Ciências e Tecnologia da Informação da Faculdade de Ciências e Tecnologia, orientada por Pedro Nuno San-Bento Furtado e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Julho, 2015

· U C ·

UNIVERSIDADE DE COIMBRA

Resumo

O problema abordado nesta tese é: Como fornecer escalabilidade horizontal totalmente automatizada para qualquer parte do processamento do ETL e da data-warehouse, de modo a que o projectista da data-warehouse apenas tenha de se preocupar com a parte lógica do sistema e fornecer/configurar limites de tempo para todas as partes envolvidas na execução do ETL e de pesquisas (ETL+Q). Em simultâneo, propõe-se uma forma de obter resultados actualizados em qualquer momento. A abordagem deve garantir os limites de tempo desejados e adaptar o sistema a qualquer momento para assegurar esses limites, escalando para cima ou para baixo cada parte do ETL e de pesquisas que tenham necessidade de mais eficiência.

Embora algumas aplicações tenham um grande volume de dados, requisitos apertados de tempo de processamento, elevados ritmos de dados e necessidade de respostas rápidas, a maioria das implementações de data-warehouse atuais não estão preparadas para escalar automaticamente. A solução passa pela utilização de arquitecturas e mecanismos paralelos para acelerar a integração de dados e para processar os dados mais recentes de forma eficiente. Estas abordagens paralelas devem escalar automaticamente. Desejavelmente, o projectista das data-warehouses deve concentrar-se unicamente no modelo lógico (por exemplo, requisitos de negócio, esquemas lógicos de armazenamento de dados), enquanto que os detalhes físicos, incluindo mecanismos de escalabilidade, actualização de dados e integração de dados a elevado ritmo de chegada, podem ser deixados para ferramentas automaticas.
Nesta tese investigamos como fornecer escalabilidade automatica para o pro-

cesso de ETL e para processamento de pesquisas (ETL+Q), bem como a forma de disponibilizar resultados que necessitam de dados mais recentes do que os já integrados na data-warehouse. A proposta desta tese lida com a paralelização e escalabilidade da data-warehouse quando necessário. Não se limita a escalar para cima (scale-out), para aumentar a capacidade de processamento, mas também se adapta quando os recursos deixam de ser necessários (scale-in). Em geral, a actualização instantânea dos dados para se refletirem nos resultados de pesquisas também não é garantida nestes contextos, uma vez que o carregamento de dados, transformação e integração são tarefas computacionalmente pesadas que são feitas apenas periodicamente, durante periodos em que o sistema não tem movimento (offline). Mas a nossa proposta é desenhada para garantir que os dados extraídos mais recentemente possam ser integrados nas pesquisas, mesmo sem que estes estejam na data-warehouse.
A proposta é uma solução universal de escalabilidade de data-warehouses que apelidamos Auto-Scale. Isto significa que a escalabilidade e a actualização de dados é automática para qualquer data-warehouse e processo de ETL, desde que o projectista inclua um conjunto de interfaces que permita ligar os seus diversos módulos à solução Auto-Scale (AScale) proposta.

No **Capítulo 1** introduzimos os problemas que a tese propõe resolver no âmbito de escalabilidade automática de processos de ETL e processamento de pesquisas. São ainda introduzidos os objectivos da tese, mecanismos propostos e contribuições. Cada etapa do ETL é separada de modo a que possa ser escalado/replicado de modo horizontal, conforme as necessidades.

O **Capitulo 2** aborda o estado-da-arte em optimização do processamento de ETL, escalabilidade e actualização das data-warehouses para fornecer resultados actualizados, e processamento contínuo.

O **Capítulo 3** resume cada um dos mecanismos propostos no resto da tese.

O **Capítulo 4** explica como é que um projectista de data-warehouses consegue integrar os módulos que desenvolve para o seu projecto, tendo em conta o desenho conceptual da data-warehouse. O AutoScale fornece interfaces no formato de API para esse efeito.

Os **Capítulos 5**, **6** e **Capítulo 7**, descrevem em mais detalhe como e gerida automaticamente a escalabilidade do ETL e das pesquisas, como são assegurados os dados mais recentes nos resultados das pesquisas, e como é feita a integração no processamento de dados que chegam continuamente.

O **Capítulo 8** é experimental. Nesse capítulo são feitos testes às propostas com a finalidade de provar que os mecanismos propostos permitem escalar quando necessário, de modo a assegurar os limites de tempo definidos para processar cada etapa do pipeline ETL+Q. Nos resultados experimentais compara-se o impacto, sem e com a solução proposta. Criámos cenários experimentais nos quais sem o AScale, o processo de ETL e as pesquisas não cumprem tempos definidos. Usando o AScale mostramos que a data-warehouse escala automaticamente e resolve os problemas de escalabilidade inerentes.

O **Capítulo 9** apresenta um resumo das principais contribuições desta tese, e aponta algumas questões interessantes, em aberto, que requerem investigação adicional.

Declaration

I, Pedro Miguel de Oliveira Martins, declare that this thesis titled, 'Elastic ETL+Q for any data-warehouse using time bounds' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

Acknowledgments

These lines I want to dedicate for giving my thanks to all people without who this thesis would have not been possible.

I want to thank my thesis adviser, professor Pedro Furtado, first for believing in me and for giving me the opportunity to work with him on this novel approach, and then for being a great help during my work, with his constant patience, useful advises and suggestions. I am especially grateful to him for encouraging me to think wider about the problems and to delve deeper to find the right and quality solutions.

I would like to thank the Department of Informatics at University of Coimbra, especially those members of my doctoral committee for their input, valuable discussions and accessibility.

Thanks for the financial support by the Portuguese Foundation for Science and Technology through the PhD grant SFRH/BD/86274/2012.

Special thanks to my friend José Cecilio, João Pedro Costa and my dear wife Maryam Abbasi, for helping me reviewing my thesis and all feedback. My thanks also go to my friends who, during the rough moments, cheered me up and gave me the strength to go on.

I owe hugely to my dear parents. Their permanent love and confidence in me have encouraged me to go ahead in my study and career, who gave me the chance to even be at this place and who supported me the most through all my life and education. I hope I will give them the reason to be proud of me. Specifically I want to appreciate them for taking care of my daughter for the last year when I was studying.

Finally, and most importantly, I express my gratitude to my dear wife Maryam who I met at the start point of my research, married and very recently had a beautiful baby girl. Her support, encouragement, quiet patience and unwavering love were undeniably the bedrock upon which the past 4 years of my life have been built. Her tolerance of my occasional vulgar moods is a testament in itself of her unyielding devotion and love. I dedicate all my efforts to her and my little daughter, Nina.

Publications

Pedro Martins, Maryam Abbasi, Pedro Furtado. AScale: Auto-Scale In and Out ETL+Q Framework. Beyond Databases Architectures and Structures (BDAS) - 12th international conference, Ustron, Krakow, Poland. 31 May - 03 Jun, 2016.

Pedro Martins, Maryam Abbasi, Pedro Furtado. AScale: Big/Small data ETL and real-time data freshness. Beyond Databases Architectures and Structures (BDAS) - 12th international conference, Ustron, Krakow, Poland. 31 May - 03 Jun, 2016.

Pedro Martins, Maryam Abbasi, Pedro Furtado. AScale: Simple and fast ETL+Q scaling for small and big data. The Second International Conference on Big Data, Small Data, Linked Data and Open Data (ALLDATA 2016), February 21 - 25, 2016 - Lisbon, Portugal.

Pedro Martins, Maryam Abbasi, Pedro Furtado. Near-real-time parallel ETL+Q for automatic scalability in BigData. Fourth International Conference on Software Engineering and Applications (SEAS-2015), January 2-3, Zurich, Switzerland.

Pedro Martins, Maryam Abbasi, Pedro Furtado. Efficient ETL+Q for Automatic Scalability in Big or Small Data Scenarios. The Tenth International Conference on Software Engineering Advances, ICSEA 2015, November 15 - 20, 2015 - Barcelona, Spain.

Pedro Martins, Maryam Abbasi, Pedro Furtado. Data-warehouse ETL+Q auto-scale framework. International Journal of Business Intelligence and Systems Engineering, 2015.

Pedro Martins, Maryam Abbasi, Pedro Furtado. AutoScale: Automatic ETL scale process. 19th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events (ADBIS 2015), Futuroscope, Poitier - France, September 8-11, 2015.

Pedro Martins, Maryam Abbasi, Pedro Furtado. Preparing a full auto-scale framework for data-warehouse ETL+Q. (Accepted on) IEEE BigData Congress 2015, 4th International Congress on Big Data, June 27 - July 2, 2015, New York, USA.

Pedro Martins, Maryam Abbasi, Pedro Furtado. AScale: Automatically scaling the ETL+Q process for performance. Special Issue on BigData, International Journal of Business Process Integration and Management, 2014.

Pedro Martins, Maryam Abbasi, Pedro Furtado. AuDy: Automatic Dynamic Least-Weight Balancing for Stream Workloads Scalability. BigData 2014, 3rd International Congress on Big Data, June 27 - July 2, 2014, Anchorage, Alaska, USA.

Pedro Martins, Pedro Furtado. DynLW: Balancing and Scalability for Heavy Dynamic Stream-DB Workloads. International Journal of Business Intelligence and Data Mining, Volume 9, 2014.

Nickerson Fonseca, **Pedro Martins**, Pedro Furtado. Near Real-Time with Traditional data-warehouse Architectures: Factors and How-to. 17th International Database Engineering & Applications Symposium, IDEAS 2013, October 9-11, 2013, Barcelona, Spain.

Pedro Martins, Maryam Abbasi, Pedro Furtado. Cloudy: Heterogeneous middleware for in time queries processing. 17th International Database Engineering & Applications Symposium, IDEAS 2013, October 9-11, 2013, Barcelona, Spain.

Abstract

Most data-warehouse deployments are not prepared to scale automatically, although some applications have large or increasing requirements concerning data volume, processing times, data rates, freshness and need for fast responses. The solution is to use parallel architectures and mechanisms to speed-up data integration and to handle fresh data efficiently. Those parallel approaches should scale automatically.

Desirably, data-warehouse developers should concentrate solely on the conceptual and logic design (e.g. business driven requirements, logical warehouse schemas, workload analysis and ETL process), while physical details, including mechanisms for scalability, freshness and integration of high-rate data, could be left to automated tools.

In this thesis we investigate how to provide scalability and data freshness automatically, and how to manage high-rate data efficiently in very large data-warehouses. The framework proposed in this thesis handles paralelization and scales of the data-warehouse when necessary. It does not only scale-out to increase the processing capacity, but it also scales-in when resources are under used. In general, data freshness is also not guaranteed in those contexts, because data loading, transformation and integration are heavy tasks that are done only periodically, instead of row-by-row. The framework we propose is designed to provide data freshness as well.

The proposal is a universal data-warehouse scalability solution. This means that scalability and freshness become automatic for any data-warehouse and ETL, as long as the developer includes a set of interfaces that are necessary to plug and take advantages of scaling mechanisms of the proposed framework.

Contents

Abstract **xv**

List of figures **xxiii**

Abbreviations **xxv**

1 Introduction **1**

1.1 Scenarios 2

1.2 Problem statement 3

1.3 Objectives of the thesis 4

1.4 The ETL+Q pipeline 4

1.5 Investigated mechanisms 8

1.6 Contributions 11

1.7 Structure of the document 12

2 State-of-the-Art **13**

2.1 ETL optimization and scalability 13

2.2 Data-warehouse scalability 17

2.3 Freshness in data-warehouses 22

2.4 High-rate data-warehouse with stream processing scalability . 26

3 Overview of automatic scalability and freshness **29**

3.1 Adding ETL+Q scalability to a data-warehouse design 30

3.2 Architecture for automatic scalability 32

3.3 Scalability mechanisms 33

3.4 Scaling total freshness 35

3.5 Scaling high-rate continuous results 37

4 Integrating any data-warehouse with AScale 41

4.1 AScale modules and connections 41
4.2 Data sources 43
4.3 Extraction 45
4.4 Transformation 47
4.5 Data buffer and Data Switch modules 50
4.6 Data-warehouse 52

5 Auto-scalability of ETL and processing 57

5.1 Configuring AScale for automatic scalability 57
5.2 Decision algorithms for scalability 61
5.2.1 Extraction & data distributors 61
5.2.2 Transform 64
5.2.3 Data buffer 65
5.2.4 Data switch 66
5.2.5 Data-warehouse 66
5.2.6 Global ETL scalability 70
5.3 Data distribution policies 71
5.3.1 From the data sources to extraction & data distributor nodes 71
5.3.2 Extraction & data distributor nodes to transform nodes 73
5.3.3 From transform nodes to data buffer nodes 73
5.3.4 Data switch extraction from the data buffers 74
5.3.5 Load data into the data-warehouse nodes 75

6 Total freshness 77

6.1 Mechanisms to achieve total freshness, D-DW 77
6.2 Scaling out/in 81
6.3 Mechanisms to achieve 24/7 availability 83

7 High-rate scalability over continuous results 87

7.1 CEP Overview 88
7.2 Query types 89
7.3 Continuous results architecture 90
7.4 Overload detection and provisioning 92

7.5 Queries distribution and relocation 92
7.6 Load shedding and admin alerts 94
7.7 Overloading handling algorithm 95
7.7.1 CEP queries scheduler 95
7.7.2 Load-Shedding and admin alert 97
7.7.3 Resource de-provisioning 98

8 Experimental evaluation 99
8.1 Experimental setup 99
8.1.1 Hardware & Software 102
8.2 Objectives and organization 102
8.3 Performance limitations without automatic scalability 103
8.4 Typical data-warehouse scenario 107
8.5 ETL with huge data rates and sizes 109
8.6 Near-Real-time and minimal downtime scenarios 113
8.6.1 Scalability of data extraction 115
8.6.2 Transformation scalability 116
8.6.3 Data Buffer nodes 118
8.6.4 Data-warehouse load and query scalability 120
8.6.5 Freshness and dynamic-data-warehouse scalability . . 124
8.7 Continuous results 127
8.7.1 Gold run 129
8.7.2 Degradation run 129
8.7.3 Workload experiments 131
8.8 Conclusions 135

9 Conclusions and future work 137
9.1 Future Work 138

A Background work 141
A.1 High Availability 142
A.2 Database Replication 143
A.3 Scale-out and scale-up 144

A.4 Elastic scalability and data warehouse parallelization 145
A.5 Common distributed architectures 148
A.6 Parallel Database Systems 149
A.7 ETL parallelism 150
A.8 Virtualization 151
A.9 Map-Reduce paradigm 152
A.10 Map-Reduce limitations 153
A.11 No-Sql paradigm 154
A.12 DBMSs Vs. Map-Reduce 156
A.13 Current related industry and academic technology focus . . . 158

B Continuous results queries and data schema 163

B.1 Data set 163
B.2 CEP queries 164

C Data warehouse queries 173

D Data warehouse indexes and views 179

E API specification 185

E.1 Data Sources 185
E.2 Extraction nodes 186
E.3 Transformation configuration 188
E.4 Data buffer 190
E.5 Data Switch 191
E.6 Data warehouse 192
E.7 Queries 194
E.8 Freshness 194

F XML Configuration 197

List of Figures

1.1 Basic data-warehouse architecture 5
1.2 AScale pipeline 8
1.3 AScale for freshness 9
1.4 AScale continuous results 10

3.1 Single server phases 30
3.2 AScale architecture for automatic scalability 32
3.3 AScale, ETL+Q scalability 34
3.4 Achieving total freshness using in-line approach 36
3.5 Achieving total freshness using parallel approach 37
3.6 High-rate over continuous results 38
3.7 High-rate scalability over continuous results 39

4.1 AScale pipeline modules and connections 42
4.2 Data Sources 43
4.3 Extraction from (1)(a) into (2) 45
4.4 Extraction method 46
4.5 Transformation 48
4.6 Sequence diagram, transformation 49
4.7 AScale data buffer and data switch nodes 51
4.8 Data-warehouse nodes 52
4.9 AScale Querying 53
4.10 Query execution sequence diagram 55

5.1 AScale framework for scalability 58
5.2 Extraction algorithm - scale-out 62
5.3 Extraction algorithm - scale-in 63

5.4 Transformation - scale-out 64
5.5 Transformation - scale-in 65
5.6 Data Buffers - scale-out 66
5.7 Data switch - scale 67
5.8 Data warehouse - scale 68
5.9 Data warehouse - scale based on query time 69
5.10 Example, estimating the scaling proportion of the ET and L . 71
5.11 Data and load distribution, sources to extraction 72
5.12 Loaders extraction from the Data Buffers 74

6.1 AScale without fast integration of fresh results 77
6.2 Dynamic-data-warehouse for fast integration and fresh results 78
6.3 Query submission 79
6.4 Executing a query for freshness 80
6.5 Scale-out based on the queue size 82
6.6 Scale-out based on query execution time 82
6.7 Scale-in based on queue size and query execution time 83
6.8 Support for 24/7 query answering 84
6.9 Loading data for 24/7 availability 84
6.10 Query execution for 24/7 availability, loading (6a) 85
6.11 Query execution for 24/7 availability, loading (6b) 86

7.1 CEP node 88
7.2 Architecture for continuous queries results 90
7.3 CEP query scheduling 96
7.4 Load Shedding mechanism 98

8.1 TPC-H and Star Schema Benchmark (extracted from [1]) . . 100
8.2 Extract and transform without automatic scalability 104
8.3 Loading data, one server vs two servers 105
8.4 Average query time for different data sizes and number of sessions 106
8.5 AScale for simple scenarios 108
8.6 AScale, 9 hours limit for ETL 109

8.7 Extraction (60 minutes frequency and 30 minutes maximum extraction time) 110
8.8 Transformation scale-out 111
8.9 Data warehouse load and re-balance scale-out time 112
8.10 Near-real-time, full ETL system scale-out 114
8.11 Near-real-time, full ETL system scale-in 114
8.12 Extraction scalability 116
8.13 Automatic transformation scalability 117
8.14 Data buffer swap into disk and scaling 119
8.15 Data warehouse load scalability 121
8.16 Data warehouse scalability, workload 1, 50GB data set 122
8.17 Data warehouse scalability, workload 2, 50GB data set 123
8.18 Automatic freshness scalability 125
8.19 Automatic freshness scalability, workload 1, 5GB 126
8.20 Automatic freshness scalability, workload 2 127
8.21 Base line comparison of performance architecture. 129
8.22 Overall performance degradation due to heavy queries (Qdb-2).130
8.23 Query overlapping and estimated overheads in time. 131
8.24 AScale admission control vs no admission control, with light queries, Ql. 132
8.25 AScale admission control vs no admission control, with heavy queries, Qh-m. 133
8.26 Workloads 1, 2 and 3 134

A.1 Example of how to distribute data in a start schema. 147

Abbreviations

24/7	24 hours a day, 7 days a week
API	Application Programming Interface
AScale	Automatic ETL+Q scaling framework
AVG	Average
CDR	Call Detail Records
CEP	Complex Event Processing
CPU	Central Processing Unit
D-DW	Dynamic-data-warehouse
DB	Data Base
DBMS	Data Base Management System
DFS	Distributed File System
DSA	Data Staging Area
DW	data-warehouse
DW + D-DW	data-warehouse and Dynamic-data-warehouse
EPS	Events Per Second
ETL	Extract Transform Load
ETL+Q	Extract Transform Load and Query
GAT	Guaranteed Achievable Throughput
GB	Gigabyte
HDFS	Hadoop Distributed File System
HS	Hash
I/O	Input/Output
IP	Internet Protocol
JDBC	Java Database Connectivity
JVM	Java Virtual Machine

LW	Least Weight
LWR	Least Work Remaining
LWRn	Least Work Remaining based on number of queries
MAS	Multi-Agent-System
MB	Megabyte
Mem	Memory RAM
min	Minutes
MR	Map-Reduce
Node	Processing unit (e.g. a processor core or a physical computer)
OLAP	On-Line Analytic Processing
OLTP	On-Line Transaction Processing
P/C	Producer/Consumer
PDI	Pentaho Data Integration
Q	Query
Qdb	Queries that use database data
Qdb-2	Stream Queries With Data Base that makes nodes unresponsive
Qdb-stream	Stream Queries With Data Base
Qh	Heavy queries
Qh-db	Heavy Queries with access to data base
Qh-m	Heavy in-memory queries
Qh-mlu	Heavy Queries with in-memory lookup and updates
Ql	Light Query
RDD	Resilient Distributed Data
Ready-Node	Represents one or more nodes that are on stand-by ready to work
RG	Range
RR	Round-Robin
RT	Real Time
S-DW	Stream data-warehouse
SDC	Data Sources
sec	Seconds
SQL	Structured Query Language

Sream-DB	Stream queries that require Data Base access
SSB	Star Schema Benchmark
TPC-H	Transaction Processing Council Ad-hoc/decision support benchmark
UDF	User Defined Function
VM	Virtual Machine

Chapter 1

Introduction

Data Warehouses (DW) and decision support systems in general are important infrastructures in many organizations, allowing them to analyse and learn from historical business data. They are most frequently organized for ease of reporting and analysis. In some scenarios, such as telecommunications, banking, energy and stock market, data warehouses are stressed with heavy Extraction, Transformation and Load (ETL) operations [2], to keep decision support and other front end systems updated with data that is as recent as possible. ETL processes suffer performance problems when data from the sources that needs to be integrated into the decision support systems has high volume and/or is produced at high-rates, especially when fast data integration is desired so that new data is reflected into analysis results quickly. In many scenarios, systems become unavailable to users to perform the ETL operations at specific time instants. Moreover, traditional data-warehouses do not incorporate the most recent data in query results, since recent data was not yet integrated. We refer to this problem as data freshness issue.

Query performance is also important. For instance, every time a user analyses some data, the system should reply and display the results as fast as possible. This issue raises many performance and scalability problems that ultimately also affect the whole ETL and Querying (ETL+Q) pipeline.

Another need that was raised more recently concerning data warehouses and data analysis is the need to also support continuous and realtime monitoring and operation in scenarios with high-rate streaming incoming data. In such scenarios the traditional (one query submission-one answer) paradigm is replaced by a (register query once-continuous evolving answer) alternative. This paradigm can be considered and used in complement to the more traditional data warehouse analysis mechanisms, and is materialized in the form of Complex Event Processing (CEP) systems [3]. Instead of storing and then analyzing data, these systems shorten the analysis cycle for some operators, using fast in-memory analysis of short windowed streams of incoming data.

Future generation systems should be able to handle all these time-related requirements by automatically scaling out when needed, for instance, during high-data-rates moments/days. When resources are no longer necessary, they should also be able to scale-in automatically, and at the same time have the information inside the data-warehouse available and updated (i.e. fresh).

1.1 Scenarios

Consider developers designing data warehouse solutions oriented for a single physical machine, without any provisioning regarding scalability. The ETL process consists of collecting items from operational systems and storing them into a data-warehouse for analysis. As the business grows and expands, the system becomes slower, until processing times of some ETL or query module, are no longer acceptable. There are two solutions, vertical scalability by upgrading the server hardware, or horizontal scalability, by adding more machines.

One frequent scenario where automated scalability is very useful is when it is necessary to ensure that the entire extract-transform-load (ETL) process takes more than a predefined limit of the data-warehouse offline period. In particular, the load can take a lot of time due to the need to refresh many redundant structures and to rebuild huge indexes. This case happens both when the ETL should take at most for instance 4 hours during the night and is actually taking 9 hours, or when the designers want to minimize downtime by specifying that ETL should take less than, for instance, 5 minutes.

Another scenario is when designers want to ensure near-real-time integration of new data into the analysis, which we call data freshness. By minimizing the amount of time from when some relevant business-related event is recorded in an operational system (e.g. some fraudulent bank transaction) to when it is incorporated into the data-warehouse analysis (fraud detected), we can design pro-active systems that are able to act quickly on changing situations. In full data freshness, data that is not yet integrated into the data-warehouse can be queried and inferred into the data-warehouse analysis.

1.2 Problem statement

The problem statement of this thesis is therefore:

How to provide totally automated horizontal scalability with data freshness to any part of any ETL plus data-warehouse analysis system, such that developers design only a logical view of the system and provide time bounds for all intended parts? And how to provide a solution that can be used by any data-warehouse developer?

From a logical view of the system, the approach must adapt at any time to meet the stated time bounds, scaling-out or -in each part that may need to adapt (ETL+Q). As a complement, it also has to support stream pro-

cessing, in order to optimize continuous querying and monitoring using the data-warehouse.

1.3 Objectives of the thesis

The objective of this thesis is to propose a new concept, named AScale, for automatic scaling of ETL+Q processes, allowing the data-warehouse developer to focus in the logical models instead of concerning with ETL+Q scalability details (e.g. how to scale, detection mechanisms, load balancing, data distribution and replication, in-time query processing). The developer only programs transformations, data-warehouse schema and queries (using any language or tool), as well as interfacing with AScale. Based on configured performance indicators, the proposed framework monitors performance of the ETL+Q pipeline, deciding when to add more resources (scale-out) for more performance, or when to decommission them (scale-in).
Besides providing automatic scalability to ETL+Q processes, the proposed framework also supports data freshness. This means that the most recent data that is not yet integrated in the data-warehouse can be included in query results without performance impact for the ETL+Q process.
As an extension to the typical data-warehouse modules, we also propose automatic scalability for time-bounded continuous query processing (CEP).

1.4 The ETL+Q pipeline

Data-warehouses (DW) are composed of the front stage, concerning end-users who access the data-warehouse with decision support tools and automated data analysis systems, where automated processes populate the data-warehouse with data obtained from several sources. The architecture of a data-warehouse exhibits various layers in which data from one layer is

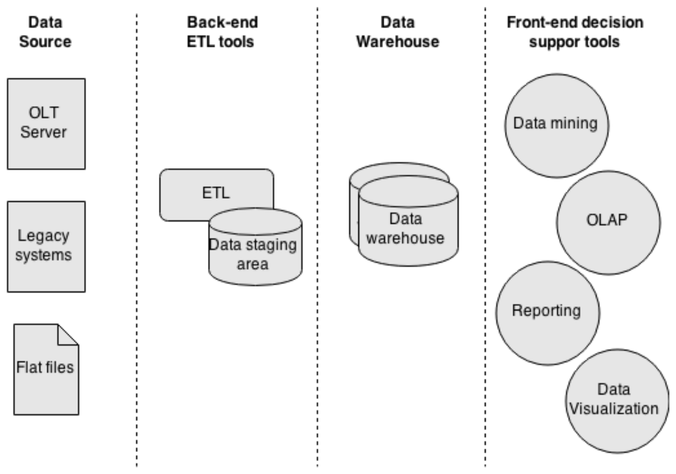


Figure 1.1: Basic data-warehouse architecture

derived from data of the previous layer, as represented in Figure 1.1.

Data Sources build the first layer. They consist of data stored in On-Line Transaction Processing (OLTP) databases, other legacy systems, or files. The next layer comprises the back-stage part of a data-warehouse, where the collection, integration, cleaning and transformation of data takes place, in order to populate the warehouse. An auxiliary area of volatile data, called Data Staging Area (DSA), is employed for the purpose of data transformation and cleaning. The central layer of the architecture is data-warehouse (DW). The data-warehouse keeps a historical record of data that results from the transformation, integration, and aggregation of data found in the data sources. Moreover, this layer involves client warehouses, which contain highly aggregated data, directly derived from the data warehouse. There are various kinds of client warehouses, such as Data Marts or On-Line Analytical Processing (OLAP) tools, which may use relational database systems or specific multidimensional data structures.

The data-warehouse front-end level consists of applications and techniques that analysers and automated tools can use to interact with data stored in the data-warehouse. Data mining techniques are used for discovery of data patterns, for prediction analysis and for classification. OLAP techniques are used for a specialized style of querying that explores the data-warehouse. Reporting and data visualization tools are used to formulate and show data.

Extraction-Transformation-Loading (ETL) tools are pieces of software responsible for the data extraction from several sources, cleansing, customization, transformation to fit analytic models, to be finally loaded into the data-warehouse. All ETL processes (extraction and transportation, transformation, cleaning and loading) present specific issues, making data-warehouse refreshing a troublesome task. In order to clarify the complexity of ETL processes, next we briefly discuss issues, problems, and constraints in each phase.

Problems and constraints, [4] mentions that 90% of the problems in data-warehouses arise from the nightly batch cycles that load the data. During this period, tools have to deal with problems such as (a) efficient data loading, (b) concurrent job mixture and dependencies. Moreover, ETL processes have global time constraints, including the initiation time and completion deadlines. In fact, in most cases, there is a tight "time window" in the night that can be explored for the refreshment of the data-warehouse, since the source system is off-line or not heavily used during this period. This problem is highly magnified when near-real-time data integration and/or minimal downtimes are required.

Extraction & transportation. During the ETL process, the first tasks that must be performed is the extraction of the relevant information that has to be further propagated to the warehouse [5].

Transformation & cleaning. Typical tasks that take place during the transformation and cleaning phase of an ETL process are: (a) look up and validate the data from tables, files or memory; (b) Sorting; (c) Joining; (d) Aggregation, (e) Generation, (f) Splitting.

Loading. The loading of the data-warehouse poses its own challenges. The load process can be performed using several techniques such as row-by-row, batch inserts and batch file loading. Loading is not just uploading data from files into database tables. It is also concerned with refreshing multiple tables indexes, materialized views and other auxiliary structures.

Feasible solutions for ETL performance problems should rely on modeling and tracking each ETL part separately. This allows us to identify performance failure points and individually scale each part of the ETL pipeline as necessary. However, scaling raises other issues related with performance monitoring, data extraction and distribution algorithms, staging area replication, and load methods for distributed shared-nothing models.

Querying and analysis. Querying and analysis is a very demanding part of the ETL+Q pipeline. It concerns providing analysis results to users and automated tools as fast as possible, possibly within predefined or required time-bounds. Whether it is SQL-like query processing or heavy data mining algorithms, analysis tools scan possible huge amounts of data in the data-warehouse repeatedly, while users and automated pro-active systems require short analysis processing times.

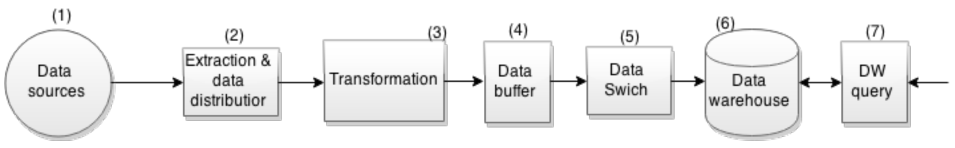


Figure 1.2: AScale pipeline

1.5 Investigated mechanisms

In this thesis we propose a set of mechanisms and a framework to automatically scale the ETL process and query execution (ETL+Q) according to configured performance parameters. The proposed framework is called "AScale". We also propose an interface mechanism for data-warehouse developers to use AScale with their designs.
According to the specification of AScale, the developer should only worry with the logical implementation of transformations, data-warehouse schema and queries. AScale provides a runtime environment that scales automatically to reach the required performance, based on configured parameters.

Figure 1.2 shows the AScale pipeline modules that independently and automatically scale each part of the pipeline. It allows to scale any part from (2) to (6), according to requirements and configuration parameters. In AScale, each independent part is monitored using some mechanisms that we will discuss later. Based on thresholds, the monitoring mechanism triggers scaling out, based on the addition of nodes for the part that needs scaling, and scaling in, when resources are underutilized. The addition of nodes enables parallelism of individual parts.
The **Data Sources (1)** represent relational data-models and other kinds of sources from where data is extracted. This part, being external to the proposed framework, requires an extra module to be executed at the same location to make data extraction possible. **Extraction (2)** extracts data from sources. **Transformations (3)** are programmed by the developer, for instance, to perform data cleansing and transform into a uniform schema

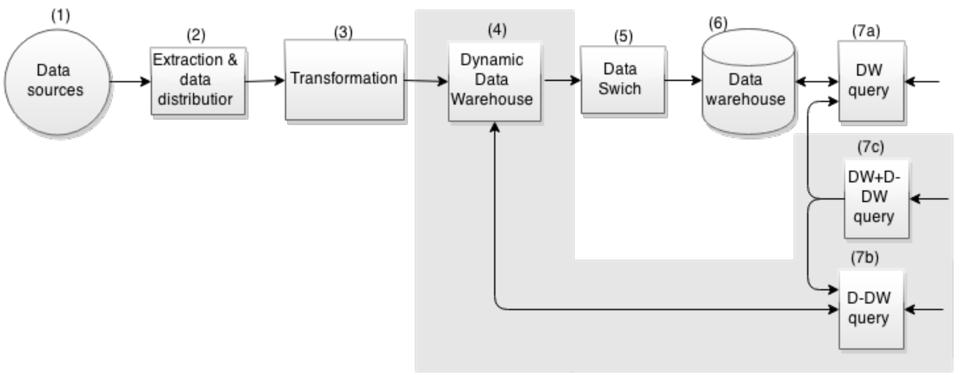


Figure 1.3: AScale for freshness

for later loading and querying. The **Data Buffer (4)** holds transformed data until the next data-warehouse load instant. **A Data Switch (5)** replicates and distributes data across multiple data-warehouse nodes (6), when the data warehouse is scaled to multiple nodes. Queries are submitted to the data-warehouse query processor (7). When there are multiple nodes, the query processor re-writes queries to execute in parallel over those data-warehouse nodes.
Detecting performance bottlenecks and scaling each part of the AScale framework is performed by monitoring maximum configured execution times and data queues sizes at each stage of the proposed pipeline.

Freshness is achieved in our proposal by transforming the Data Buffer that holds the transformed data prior to loading (4) into a fully query-capable repository called dynamic-data-warehouse (D-DW), Figure 1.3. The D-DW consists of a small data-warehouse, also scalable as necessary, that only contains the most recently transformed data. Because it has a small size (it only contains the most recent data) querying over the dynamic-data-warehouse is fast. Queries can be submitted to the DW (7a), or to the D-DW (7b) or to both (7c). Results are merged after collecting the parts from (7a) and (7b).

We also study processing scalability and balancing for continuous re-

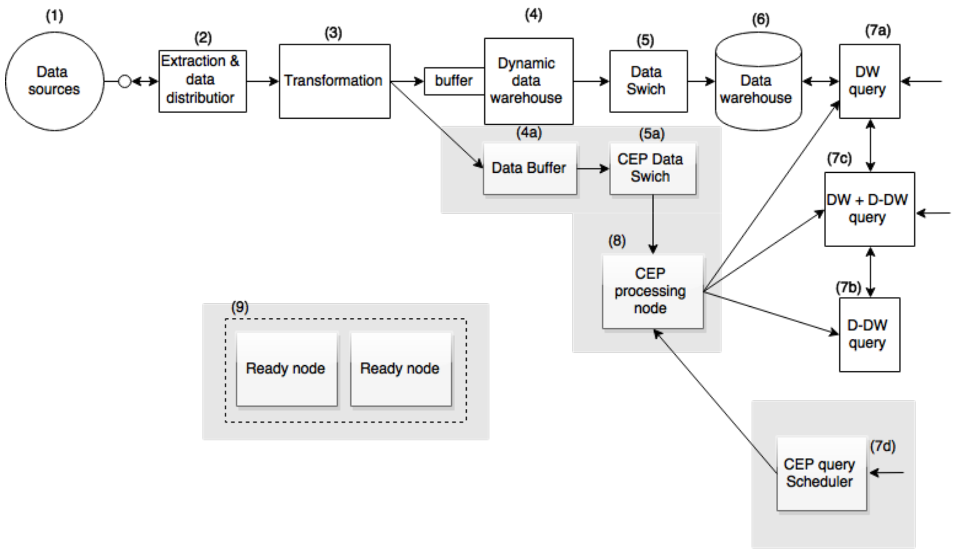


Figure 1.4: AScale continuous results

sults, working as a complement to the existing ETL+Q pipeline. In order to integrate a CEP engine into the previously defined ETL+Q pipeline, after transformation, Figure 1.4 (3), besides being loaded later into the data warehouse, transformed data must also be loaded into a data buffer (4a) that is connected to a CEP Data Switch (5a). The CEP data Switch (5a) extracts data from the Data Buffer (4a) and sends the tuples to the CEP nodes (8). Continuous queries can be registered and un-registered using a CEP query scheduler (7d). This scheduler uses a set of performance rules to select the best node to register each query. Additionally, each CEP node (8) can also access the DW (6) or the D-DW (4) by submitting queries to (7a, 7b or 7c).

In Figure 1.4, we also introduce Ready-Nodes (9). Ready-nodes represent the extra resources available to scale any part of the AScale framework. When additional resources are required to scale some part of the framework, they are retrieved from the Ready-nodes area, and after a scale-in is triggered, they are returned to the Ready-Nodes pool. A node can be either a computer, a core or a virtual machine.

1.6 Contributions

This thesis proposes AScale, a framework for automatically scaling and guaranteeing freshness for the ETL+Q process, allowing the developer to focus only in the conceptual ETL+Q model.

The following contributions are made, as parts of the AScale framework:

1. An approach to automatically parallelize ETL and Query execution (ETL+Q), able to modify individual components when they need to scale out or in - starting from a logical single-server ETL and data-warehouse, we define an approach for automatically parallelizing the system with any data warehouse system and to offer freshness;
2. An approach and interfaces (API) for data-warehouse designers to integrate the logic with AScale so that AScale can be applied automatically with any data-warehouse.
3. Mechanisms to monitor each part of the ETL and query execution, to detect where and when scaling is necessary, and mechanisms to dynamically and automatically adapt the system by scaling;
4. Dynamic-data-warehouse (D-DW). We propose an in-memory dynamic store and processing approach which, when added to AScale, provides total freshness and real-time. In order to handle very high data ingress rates and large query workloads, the (D-DW) can be parallelized (parallel dynamic-data-warehouse), providing total freshness scalability. Querying over the DW is integrated with querying the D-DW, returning both fresh and complete results (i.e. result including the most recent data, older data and both);
5. Scalable stream data-warehouse - some current applications require processing of continuous results as well as freshness. We propose an

approach to continuously integrate data and provide results to registered queries while also balancing and scaling workloads that may require access to both stream and database-stored data. The stream data-warehouse also prevents overloading of the streams using admission control techniques;

6. Experimental evaluation of the proposals, showing that AScale is able to scale-out when performance bottlenecks are detected, and that it is also able to scale-in when resources are not needed. We implemented an AScale prototype to test the approaches proposed in this thesis.

1.7 Structure of the document

Chapter 2 discuses the related work. Chapter 3, provides and overview of the approaches proposed in the remaining chapters of the thesis. Chapter 4, describes the interfaces (API) that AScale provides and how data-warehouse developers interface their systems with AScale. Chapter 5 proposes the mechanisms for ETL and query processing scalability. Chapter 6 proposes the mechanisms for total freshness. Chapter 7 describes how high-rate scalability over continuous results is integrated in the ETL+Q pipeline. Chapter 8 provides an experimental evaluation of the entire auto-scale ETL+Q framework, showing how it scales-out and -in. Chapter 9, concludes the thesis and discusses future work.

Chapter 2

State-of-the-Art

In this section we review related work on scalability and freshness. Section 2.1 reviews related work on ETL optimization and scalability. Section 2.2 reviews works on scalability of the data-warehouse. Section 2.3 reviews related work on dealing with freshness of data/results. Finally, in Section 2.4 we review related work in the field of complex event processing (CEP), since we also propose integrating CEP processing into AScale.

2.1 ETL optimization and scalability

AScale optimizes ETL by automatically scaling each part of the processing pipeline. Next we review previous work related to optimizing and scaling ETL.

To increase the efficiency of the ETL process, [6], [7] propose searching methods based on heuristic algorithms that minimize the ETL execution cost, by modeling the problem as a space search graph to decide which execution is more efficient. Graphs are created by the decomposition of relational algebra operators. Heuristics are created based on the number of times each state is visited.

Work [8] studies how to manage large ETL processes by implementing a set of basic management operators, such as "MATCH", "MERGE", "INVERT", "SEARCH", "DEPLOY". The framework is web-based. The user creates the ETL flow using drag-and-drop with the available filters, then the framework determines the best execution order for the ETL using a set of optimization algorithms.

Work [9] discusses the problem of scheduling the execution of ETL activities (a.k.a. transformations, tasks, operations), with the goal of minimizing ETL execution time and allocated memory. The paper investigates the effects of four scheduling policies (Round-Robin, Minimum Cost Prediction, Minimum Memory Prediction, Mixed Policies) on different flow structures and configurations. It shows that the use of different scheduling policies may improve ETL performance in terms of memory consumption and execution time.

In [10] the authors propose a distributed ETL engine architecture based on Multi-Agent Systems (MAS) data partitioning technology. They also investigate methods of partitioning the massive data streams in both horizontal and vertical ways. The system partitions workflows into multiple sub-workflows for parallel execution in agents, also adding a spliter node to distribute work. Each sub-workflow is executed by an agent, so that multiple agents could work together to complete the collaborative work. At the end, another extra node, the merger, will merge all results.

In [11] the authors describe an Extract-Transform-Load programming framework using Map-Reduce to achieve scalability. Data sources and target dimensions need to be configured and deployed. The framework has built-in support for star schemas and snowflakes. Users have to implement the parallel ETL programs using the framework constructors. They use pygrametl [12], a Python based framework for easy ETL programming. The flow consists of two phases, dimension processing and fact processing. Data is read from sources (files) on a Distributed File System (DFS), transformed and processed into dimension values and facts by the framework instances, which materialize the data into the DW. The framework requires users to declare (code) target tables and transformation functions. Then, it uses a master/worker architecture (one master, many workers), each worker running jobs in parallel. The master distributes data, schedules tasks, and monitors the workers.

ETLMR [13] proposes a tool to build the ETL processes on top of Map-Reduce to parallelize the ETL operation on commodity computers. ETLMR contains a number of novel contributions. It supports high-level ETL-specific dimensional constructs for processing both star-schemas and snowflake-schemas, and data-intensive dimensions. Due to its use of Map-Reduce, it can automatically scale to more nodes (without modifications to the ETL flow) while at the same time also providing automatic data synchronization across nodes (even for complex dimension structures like snowflakes). Apart from scalability, Map-Reduce also gives ETLMR a high degree of fault-tolerance. ETLMR does not have its own data storage (note that the offline dimension store is only for speedup purposes), but is an ETL tool suitable for processing large scale data in parallel. ETLMR provides a configuration file to declare dimensions, facts, User Defined Functions (UDFs), and other run-time parameters.

In [14] the authors consider the problem of data flow partitioning for achieving real-time ETL. The approach makes choices based on a variety of trade-offs, such as freshness, recoverability and fault-tolerance, by considering various techniques. In this approach partitioning can be based on round-robin (RR), hash (HS), range (RG), random, modulus, copy, and others [15].

Pentaho Data Integration (PDI) [16], is an ETL (graphical) design tool and provides Hadoop support. It consists of a data integration (ETL) engine, and GUI applications that allow the user to define data integration jobs and transformations. It supports deployment on single node computers as well as on a cloud, or cluster. Internally, it allows connection/integration with other systems using for instance sockets, web-services (e.g. SOAP, XML).

Analysis:
The mentioned works focus optimization of each individual ETL process by optimizing data access, reordering operators execution and managing the available computational resources as well as possible. Some do not use parallelism, which limits capability to scale. Such approaches can easily be used together our proposed framework (AScale).
To guarantee adequate ETL and query processing services in demanding environments, it is essential for the systems to scale automatically. A direct scalability approach would be to use Map-Reduce to implement the entire ETL process as same of the related works propose. However, the Map-Reduce model does not offer:

- Automatic real-time performance monitoring and scaling mechanisms, new nodes must be added manually;
- There is more network traffic in consequence of using a distributed file system and Map-Reduce paradigm (Appendix A, Section A.10);

- The entire ETL process must be coded in Map-Reduce programming model, adding more complexity and potential performance limitations;
- Efficiency depends on implementation method of used operators, specially when data exchange is required.

The proposed AScale framework offers better usability and performance by:

- Providing an automatic scaling mechanism based on monitoring;
- Allowing to scale each part of the ETL+Q process independently;
- Allowing to define the ETL using any programming language as long as a connector for the AScale framework is provided;
- Data (dimension tables or all tables) can be replicated across the data-warehouse nodes, avoiding high amounts of data exchanges over the network to merge results.

2.2 Data-warehouse scalability

There is a vast literature on query processing and load balancing in parallel databases systems [17], [18] and distributed systems [19], using many different technologies. In this section we focus on data-warehouse scalability, resorting to different storage and processing approaches and paradigms, with the main purpose of optimizing query execution.

The work [20] studies performance and availability of parallel data-warehouses. In this work the author provides a series of solutions to improve, performance models. This is done using hash indexes for efficient load-balancing, replication in different nodes for availability, and distribution of data using small data blocks and their indexation. The model uses indexation to place data blocks and respective replicas efficiently. The author concludes that by using this approach to distribute data and replicas, it is possible to improve flexibility, fault tolerance and data placement efficiency.

In [21] the author proposes an architecture for data distribution and speeding up query processing. Data is placed (and replicated) into the nodes in such way that it can be related (join) without the need to transfer information between nodes, and merging it. With this approach the author is able to minimize random data access and network usage to transfer intermediate results, making queries execute faster.

In the works [22] [23] [24] the authors present an incremental selection of horizontal data partitioning techniques based on adapting the current fragmentation schema of the data-warehouse in order to deal with the workload. To optimize data-warehouse performance they propose, the combination of three major optimization techniques: optimization using indexes, views, materialized views; horizontal partitioning applied to dimension tables; decomposing a table based on the fragmentation schema of another table.

In [25], [26] the authors also propose an approach for distributing data efficiently in distributed data-warehouses. Dimension tables are replicated by each node of the cluster and fact tables are distributed over available nodes.

Parallel DBMS first appeared about three decades ago [27]. The main difference between parallel DBMSs and Map-Reduce is that parallel DBMSs are designed to run long pipeline queries over nodes that hold partitioned data, instead of small independent tasks over a distributed file system as in Map-Reduce. Parallel DBMS were recently compared with Map-Reduce when processing analytic queries. Pavlo et al. [28] and Stonebraker et al. [29] both conducted experiments to compare the open source Hadoop Map-Reduce implementation with two parallel DBMSs (a row-based and a column-based) in large scale data analysis. The results demonstrated that parallel DBMSs are significantly faster than Hadoop when running analytic tasks. But they diverge in the effort needed to tune the two classes of systems. Dean et. al. [30] argue that there are mistaken assumptions about Map-Reduce in the comparison papers and that Map-Reduce is highly effective and efficient for large-scale fault-tolerant data analysis. They agree that Map-Reduce allows some complex data analysis to be programmed that would not be able to be written in SQL, while parallel DBMSs excel at efficient querying on large data sets [29]. However, the Map-Reduce framework has so far been optimized for one-pass batch processing of on-disk data, which makes it slow for interactive data exploration [31].

Systems based on Map-Reduce [32] architectures (e.g. Hadoop, Hive, Hbase) claim to solve all problems related to scalability and availability. Despite the huge amount of research around this paradigm to improve performance, scripting language, processing methods, or the addition of more functionality, the fact that a distributed file system is used and Map Reduce architecture is the basis of processing, compromises the overall performance and flexibility in terms of performance and configuration of ETL processes. Data warehousing tools using Map-Reduce include, Hive [33], [34], Pig [35] and, among others Apache Spark [36], [31]:

- In Map-Reduce survey [32] the authors describe several research and commercial projects. However, they do not fit into the performance and flexibility necessary for high-rate ETL processing. The architecture itself introduces too much overhead, for instance to activate the

Map and Reduce functions for a large parallel data-warehouse with very frequent data integration cycles. Many limitations related with scripting languages and their speed are mentioned. High amounts of data materialization and disk usage is also a problem, which influences performance. Though Map-Reduce is a framework well suited for large-scale data processing on clustered computers, it has been criticized for being too low-level, rigid, hard to maintain and reuse [35], [33];

- Pig and Hive, both offer scripting languages for data stored on the Hadoop Distributed File System (HDFS). Pig and Hive share large similarities, such as using Hadoop Map-Reduce, HDFS as their data storage, integrating a command line user interface, having a scripting language, being able to do some ETL data analysis, and others. Hive provides an SQL-like language (HiveQL) and a shell, Pig provides a scripting language Pig Latin and a shell; both Hive and Pig require users to write data processing scripts explicitly. Secondly, in Hive and Pig, an external function or user customized code for a specific task can be implemented as a user defined function, and integrated into their own language, e.g., functions for data serialization/deserialization. Hive and Pig achieve the functionality of ETL constructs through a sequence of user-written statements, which are later translated into execution plans, and executed on Hadoop. Thirdly, although Hive and Pig are both able to process star and snowflake schemas, technically, implementing an ETL, even the simplest star schema, is not a trivial task, as users have to dissect ETL, write the processing statements for each ETL step, implement user defined functions, and do numerous testing to make them correct. Moreover, Hive and Pig do not support UPDATE operations.

  Instead of requiring the person to code using these dialects and most probably to have to code some parts using the Map Reduce model, we opt to give the programmer the freedom to use the programming languages and frameworks he is used to, then using connecting APIs for AScale. Then AScale parallelizes and scales automatically.

- Spark [36], [31] started from discussions with Hadoop users, where they wanted to run rich applications that the single-pass, batch processing model of Map-Reduce does not support efficiently. Spark comes to provide:

  – More complex, multi-pass algorithms, such as the iterative algorithms that are common in machine learning and graph processing.

  – More interactive ad-hoc queries to explore the data.

  Although these applications may at first appear quite different, the core problem is that both multi-pass and interactive applications need to share data across multiple Map-Reduce steps (e.g. multiple queries from the user, or multiple steps of and iterative computation). The only way to share data between parallel operations in Map-Reduce is to write it to a distributed file system, which adds substantial overhead due to data replication and disk I/O. Spark overcomes this problem by providing a new storage primitive called Resilient Distributed Data sets (RDDs). RDDs let users store data in memory across queries, and provide fault tolerance without requiring replication, by tracking how to recompute lost data starting from base data on disk. This lets RDDs be read and written up to 40x faster than typical distributed file systems, which translates directly into faster applications [31]. Apart from making cluster applications faster, Spark also seeks to make them easier to write, through a concise language-integrated programming interface in Scala, a popular functional language for the JVM;

Analysis:
We have reviewed a set of systems that were designed to scale and improve the performance of other tools and paradigms.
Our proposal, AScale, provides techniques to distribute and replicate data, and automatic data-warehouse scaling-in/out when necessary. The auto-scale is obtained by automatic scale need detection based on resources monitoring (e.g. disk), ingress data load monitoring (i.e. using buffer queues to monitor load speed) and query execution time monitoring. This auto-scale capability of AScale framework can also be applied in other environments such as Map-Reduce architectures, a possibility that we propose as future work.

2.3 Freshness in data-warehouses

Typically, in large data-warehouses the load process is not done immediately as data appears in sources, for performance reasons, instead it may be done for instance every night or weekend. In this section we discuss data freshness (a.k.a. near real-time) in data-warehouses. The idea is to add approaches to the data-warehouse architecture that will allow it to be real-time.

The work [37] is focused on methods for real-time data-warehouse support. The work considers two main combinations of approaches: incremental batch refresh (based on time-stamp); and continuous refresh, where the data-warehouse is always updated. The framework considers three aspects to ascertain the level of update importance: impact from record measure/-precision in results; number of affected records; requests frequency. Based on these aspects the framework decides if it should or not update the data-warehouse in real-time.

To achieve near-real time ETL in [15] the authors suggest an architecture with mechanisms to ensure constant data refresh inside the data-warehouse. This architecture has five layers: extract data from the sources; data processing area, for data extraction and synchronization with previews data; transformation area; synchronization between the data processing area and the data-warehouse; storage layer. All the components together allow to isolate and identify bottlenecks to scale-out.

In [15] the authors include a discussion on approaches for refreshing the data-warehouse. They take the approach of designing an architecture with multiple levels for data extraction, data processing, loading and query processing. The authors discuss how those can be scaled and synchronized to allow constant refreshing of the data-warehouse. Two important issues are not discussed there: the existence of indexes and materialized views that slowdown loading and refreshing, and the simultaneity between online querying and continuous data loading is not considered or evaluated.

The authors of [38] and [39] propose a solution based on a Service Oriented Architecture. The solution relies on mechanisms to collect data from web-services which are hosted in other subsystems. The framework extracts the data and stores it in memory caches organized hierarchically. These caches have periodic update times. After the data passes though all caches, it is stored inside the data-warehouse.

In [40] the authors use an extraction strategy different from the usual. Instead of Extract Transform and Load, they use Extract Load and Transform. Data is extracted from the sources and immediately loaded into the staging area. The architecture also has many Materialized Views which are used to answer user queries faster. These views are updated from the base data on demand when receiving new queries.

The work [41] shows an architecture that uses multiple partitions to store information regarding a certain period of time, for instance an hour or a day. Then, depending on the queries necessities, different partitions are used. With this work not only queries are optimized but also indexing and data integration speed.

The industry is also investing in this concept, solutions for real-time data-warehouses of large enterprises can be found in the market. Oracle for instance is using Oracle Data Integrator [42] and Oracle GoldenGate [43] which claim to guarantee real-time data integration. More solutions can be seen in Vertica [44] engine, where a hybrid architecture, memory and disk, is suggested. Data in memory is accessed and stored very fast, the migration to disk is done asynchronously. The data is stored in disk using traditional architectures.

In [45], the goal is to exploit database management systems and emerging cloud technologies (e.g., virtualization and distributed cloud storage systems) to improve the deployment and usability of database systems in the cloud. More specifically, it is intended to use cloud technologies and relational database systems to build a highly available and elastic scalable database service in the cloud, while providing strong consistency and supporting a full SQL interface.

In [46] the authors describe Liquid, a data integration stack that provides low latency data access to support near real-time in addition to batch applications. It supports incremental processing, and is cost-efficient and highly available. Liquid has two layers: a processing layer based on a stateful stream processing model, and a messaging layer with a highly-available publish/subscribe system. The processing layer (i) executes ETL-like jobs for different back-end systems according to a stateful stream processing model [47]; (ii) guarantees service levels through resource isolation; (iii) provides low latency results; and (iv) enables incremental data processing. A messaging layer supports the processing layer. It (i) stores high-volume data with high availability; and (ii) offers rewindability, i.e. the ability to access data through meta-data annotations. The two layers communicate by writing and reading data to and from two types of feeds, stored in the messaging layer.

The work in [48] describes solutions for deploying data-warehouse systems for real-time, typically relying on a smaller repository holding the most recent data. Since this repository is small, it allows faster loading and refreshing, without affecting the performance of querying activity [49], [50].

In [51] and [52] the authors propose temporary-tables based approach to deal with Real-Time. For instance, in [51] the authors add time-interval granularity partitions, e.g. one for the last hour and one for the last day. In those solutions there is not adequate decoupling between the data loading and querying servers, since they are done in the same database instance in the same machine, and with offline refreshing. Those limitations and the study of the limitations of data warehousing when dealing with near-real-time data [50] led to the proposal in [53]. In that proposal queries and data loading occur simultaneously with minimum performance degradation, since they can happen in different database instances and in different machines.

In the work [54] using partition techniques, the architecture presented relies on temporary tables to store the most recent data, allowing faster data integration. This is possible by keeping small temporary tables and removing all indexes and auxiliary structures which require constant updating when adding new data.

Analysis:

Data freshness (near real-time results) is normally accomplished by separating the most recent data from the rest of the data of the data-warehouse. This optimizes querying performance over new data which was not yet integrated into the main data-warehouse. Immediate integration of new data can not only damage the data-warehouse current querying performance, as it is a complex process that requires many resources to update in-place data-warehouse structures (e.g. indexes, views).

In AScale, we include a dynamic-data-warehouse using in-memory tables. The dynamic-data-warehouse holds only the most recent data until the next data-warehouse load period. This allows us to integrate data much faster into the dynamic-data-warehouse, because there are no indexes or any kind of views being updated simultaneously, thus, querying most recent data is faster. AScale, our auto scaling framework, adds performance monitoring and automatic scalability (in OR out) to the dynamic-data-warehouse when necessary.

2.4 High-rate data-warehouse with stream processing scalability

Complex Event Processing (CEP) systems [55] are used in various scenarios, processing several thousands of events per second with sub-milliseconds latency [56]. CEP engines like ESPER [57], StreamBase [58], Oracle Complex Event Processing (OCEP) [59], have the aim of efficient stream processing for data arriving at high-rates. In general, these types of applications are dedicated to fields like (but not limited to): stock trading [60], stream

monitoring, information integration workflows [61]; exception management; financial services; health care; IT monitoring; telecommunication; logistics; sensor networks; fraud detection, finding patterns in data. Recently, these topics have received significant attention in the research community [62], [63], [64], [65]. The choice of such applications lies in the need to process events in real-time. The main considerations for such applications are the need for high-throughput, between 1000 and 100k messages per second, low-latency processing for event streams (between milliseconds and seconds) and the logical complexity.
CEP systems are able to process data at high-rates. Simultaneous accesses to data-warehouses are usually not performed, because they tend to be slow. However, it is useful to be able to correlate stream information with data-warehouse information. We investigate how to make CEP automatically scale and load balance the processing while at the same time querying the main data-warehouse.

There are several commercial CEP systems, such as RuleCore CEP server [66], Coral8 Engine [67] and Esper [57] (also with an opens source version), several open research prototypes also exist [68]; Cayuga [69] with two versions, a single CEP engine designed to achieve high performance and a distributed version of the system, that merges events and algebra expressions. Many academic prototypes have been developed to distribute operators, Borealis [70], DCEP [71] and others [72], [73], [74]. One of the most cited systems, Borealis [70], an extension to Aurora [72] and Medusa [75], focuses on operator placement and other strategies across a set of Aurora Engines.

Other research groups started focusing on elastic streaming over Map-reduce [76]. These approaches use distributed file systems, such as HDFS, and large amounts of memory to attain elastic scalability with high-rate data processing, using a set of Map-Reduce nodes, focusing on CEP queries and operator placement optimization.

Analysis:

Some of these systems include the capability to query databases, however they become dependent of the database processing speed and this may compromise the whole context that assumes periodic results. If the data-warehouse process speed is slow, then the processing of events which also query the data-warehouse will be affected.

Neither academy nor industry have investigated algorithms for speeding up the timely execution of queries in CEP and databases together. The investigation work that we did provides monitoring, scaling and admission control guarantees over high-rate CEP processing that includes analysis of DB data as part of the queries. We also investigate the auto-scale mechanisms for integration of data-warehouse and support of CEP queries with automatic load balancing and scaling (out/in) when necessary.

Chapter 3

Overview of automatic scalability and freshness

In this chapter we investigate how to provide (ETL+Q) automated scalability and data freshness in data-warehouses. Moreover we provide an overview of the solutions proposed in the thesis.

The main proposal is a (ETL+Q) auto-scale framework, named AScale. According to the approach, the developer designs a logical view of the ETL+Q and data-warehouse (single server), writing only the transformations, without worrying about scalability details. Additionally, the developer specifies some parameters needed by the auto-scaling mechanisms.

To guarantee total scalability and freshness, AScale has to deal with both ETL scalability, processing scalability and data freshness guarantees.

Section 3.1 describes how ETL+Q scalability is added to a data-warehouse. Section 3.2 overviews the framework of AScale. Section 3.3 describes the proposed scalability mechanisms. Sections 3.4 describes the approaches for data freshness and how to scale for total data freshness. Section 3.5 describes how the system provides high-rate continuous results and scalability when processing continuous results.

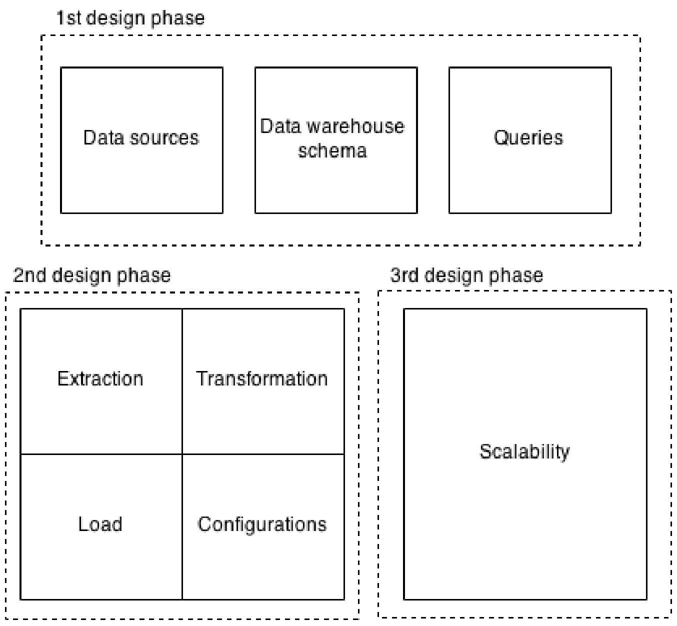


Figure 3.1: Single server phases

3.1 Adding ETL+Q scalability to a data-warehouse design

When building a data-warehouse, the main phases are shown in Figure 3.1. The "1st design phase" represents the parts to be accounted for before any other system development happens. The related items include:

- "Data sources" - includes the configuration of the data sources origin and destination, data extraction format and frequency;
- "data-warehouse schema(s)" - the schema must be defined according to the data and queries to be performed. Generally star-schema models are used for data-warehouses;
- "Queries" - represent relevant processing load of the system. Queries are also directly linked to the data-warehouse schema. If the schema changes, the queries must be rewritten accordingly.

The "2nd design phase" regards the integration, implementation and configuration. This phase affects the global performance (e.g. queries execution time, transformation time, integration time, and so on). The related items include:

- "Extraction" - when extracting data from sources, a set of parameters must be specified, such as: extraction frequency, maximum window extraction size;
- "Transformation" - after extracting the data from sources, it needs to be transformed and cleaned before loading into the data-warehouse. Transformation operations need to be defined and implemented. For processing the transformation, the system also needs to know a set of parameters, among them the input and output data formats;
- "Load" - is the process of loading the data into the data-warehouse. This process is crucial and involves the definition of loading scripts, loading periods and duration, load mechanisms, load buffer size;
- "Configurations" - includes all adjustments to the system to optimize the available resources depending on the desired objectives. For instance, one needs to configure extraction frequency, data-warehouse integration methods, buffer sizes and so on.

The "3rd design phase", concerns the scalability of the system or parts of it (e.g. extraction, transformation, loading, storage nodes). When using the proposed AScale framework, the referred "3rd design phase", scalability, is automatically included and managed. Thus the user only needs to define how the logical system works for a single server architecture (1st and 2nd design phases), without concerns regarding scalability.

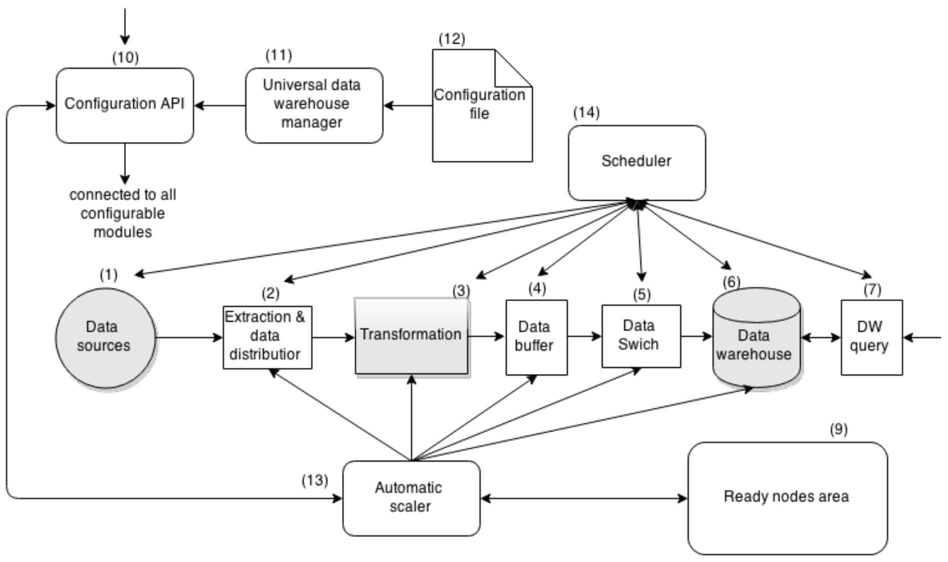


Figure 3.2: AScale architecture for automatic scalability

3.2 Architecture for automatic scalability

In this section we describe the main components of the proposed framework, AScale. Figure 3.2 shows its main components:

- Components (1) to (7), except (5) are the Extract, Transform, Load and Query (ETL+Q) pipeline;
- The "Automatic Scaler" (13), is the component responsible for performance monitoring and scaling the system when necessary;
- The "Configuration file" (12) represents the location where all user configurations are saved;
- The "universal data-warehouse manager" (11), uses the configurations provided by the user and the available "Configuration API" (10) to set the system to perform according with the desired parameters and selected algorithms. The "Universal data-warehouse Manager" (11), also sets the configuration parameters for automatic scalability at (13) and the policies to be applied by the "Scheduler" (14);

- The "Configuration API" (10), is an access interface which allows to configure each part of the proposed data-warehouse framework, automatically by (11) or manually by the user;
- The "Scheduler" (14), is responsible for applying the data transfer policies (least-work-remaining, round-robin, manual) between components.
- The "Ready nodes area" (9) represent nodes that are not being used. These nodes can be added to parts (2) to (6) of the system to scale-out, improving performance where needed, or removed to scale-in, saving resources that can be used in other places.

All these components when set to interact together provide automatic scalability to the ETL+Q and to the data-warehouse processes without the need for the user to concern about its scalability or management.
Instead of programming the entire ETL pipeline, the user can focus only in programming the transformations and data-warehouse schema (Figure 3.2, highlighted in grey color). The other scalability details are handled automatically by AScale. Additionally, the developer can choose any data-warehouse engine to store data (e.g. Relational data-warehouses, column oriented, noSQL, Map-Reduce architectures) by configuring AScale to connect them.

3.3 Scalability mechanisms

In this section we introduce how each part of AScale (ETL+Q auto-scale framework) scales individually to obtain the necessary performance configured by the developer.

Figure 3.3 depicts each part of the ETL scaling, including:

(1) Each data source (1) has an extraction frequency associated with it

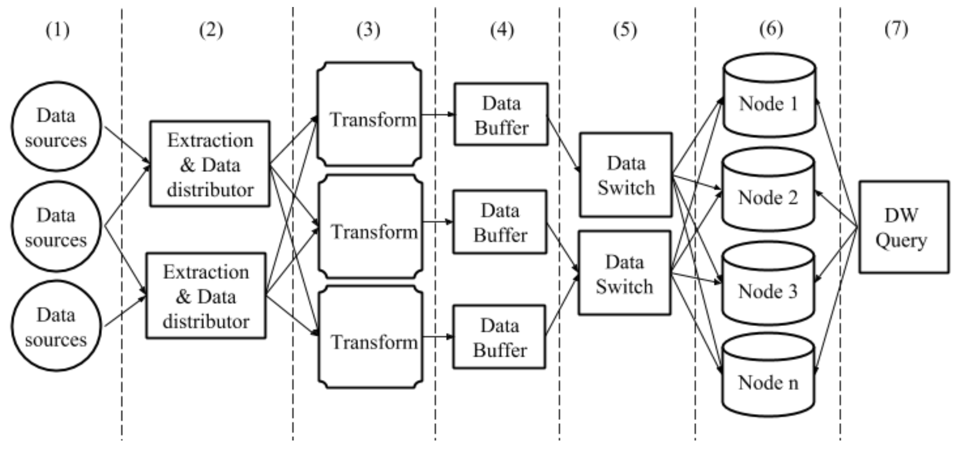


Figure 3.3: AScale, ETL+Q scalability

(e.g. every minute). The increase of data sources nodes (1), and data source rates, implies the increase of data, leading to the need to scale other parts of the proposed framework;

(2) The "Extraction & Data Distributor" nodes forward and/or replicate the extracted (raw data) into the transformer nodes. If the extraction time is larger than a maximum configured limit, or if data extraction is not complete until the next extraction instant (e.g. every minute), more data distributor nodes (2) are required. Scaling needs in (2) are detected by monitoring the extraction time;

(3) Transformation nodes process the data transformations programmed by the user. These nodes include a buffer queue to monitor data ingress. If the queue increases its size above a certain limit, the transformation node is scaled by replicating the transformation code to another node;

(4) The data buffer holds the transformed data, it can be in-memory or/and disk. These nodes are scaled based on memory monitoring parameters. If at any time the used memory reaches the maximum configured usable memory, a new node must be added;

(5) Data switches are responsible for data distribution (pop/extract) from the "Data Buffers" and placing it in the correct nodes for loading into the data-warehouse. Each data switch is configured to support a maximum data-rate (e.g. 10000 rows/sec). When that limit is passed or reached during a defined time window, more data switch nodes are added;

(6) The data-warehouse can be in a single node, or parallelized over multiple nodes. Scalability of the data-warehouse is based in two parameters: the loading time and query response time. If the data-warehouse nodes take more time to load data than the maximum configured time, more nodes are added and data is re-distributed [21]. If the average execution time of queries is more than the desired response time, data-warehouse nodes must also be added to guarantee more performance;

Finally, the last scalability mechanism introduced in AScale is the global desired ETL processing time. A global time for the entire ETL process can be defined. If that global time is exceeded, then the AScale pipeline component that is nearest to its scaling limit is scaled-out.

3.4 Scaling total freshness

Data-warehouses typically only load data at specific time instants, with a specific periodicity. This means that between loading periods fresh data may not be available. By contrast, total freshness refers to query answers incorporating the most recent data. We define that a query result is totally fresh (freshness) if it incorporates all the data produced until submission time.

Figure 3.4 depicts an in-line (non distributed) processing approach of the designed system to support freshness. In Figure 3.4 we introduce a dynamic-data-warehouse (4) (D-DW). This consists of a temporary data-warehouse

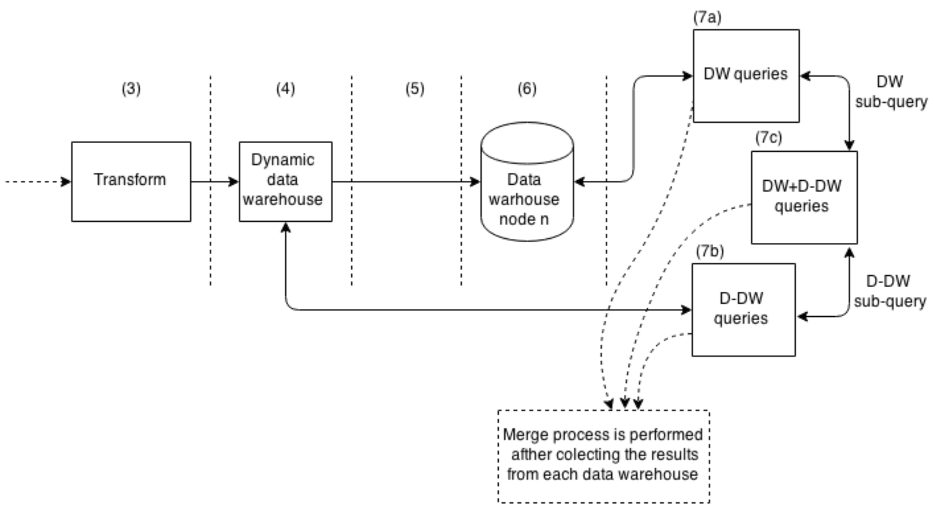


Figure 3.4: Achieving total freshness using in-line approach

storage, mainly in in-memory, for the most recent data that was not yet integrated into the data-warehouse (6) nodes. The purpose of this new component is to include the most recent data into query results, even though that data has not been loaded into the main data-warehouse yet. When executing queries they can be submitted to the dynamic-data-warehouse (4) or to the main data-warehouse (6). Modules, (7a), (7b) and (7c) manage the queries submission and execution into each or both data-warehouse types, (4) or (6).

Figure 3.5 shows the parallelization of the Dynamic-data-warehouse (4). Data is transformed (3) and sent to the dynamic-data-warehouse (4). Data Switches (5) load the data already transformed and stored in the Dynamic-data-warehouses (4) into the data-warehouse (6) nodes according to a configured period of time.

The Dynamic-data-warehouse (4) scaling decision is based on monitoring data ingress queues sizes. If the queue size increases above a certain limit, data is not being integrated into the D-DW (4) fast enough, thus there is need to scale-out. Additionally, if D-DW queries response times take longer than the desired response time, more nodes must be added.

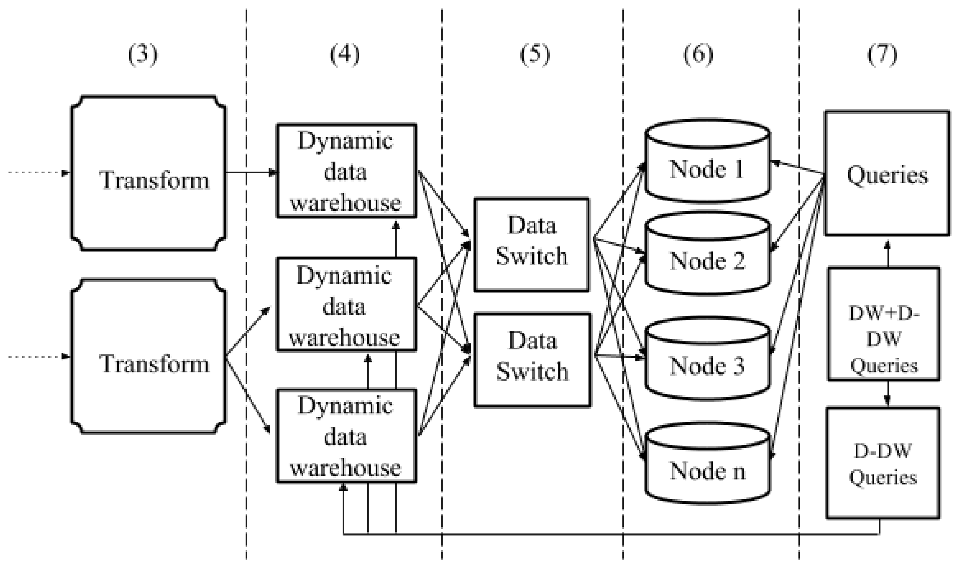


Figure 3.5: Achieving total freshness using parallel approach

Queries (7) can be executed against the data-warehouse (6) or against the Dynamic-data-warehouse (4) or against both the Dynamic-data-warehouse (4) and the data-warehouse (6) to include the most recent data. Assume a query that includes the most recent data, and calculates a simple average. It would be decomposed into three queries, one to run in each DW and D-DW, other to merge the results of each data-warehouse node, and, finally one to merge the results of the DW and D-DW.

3.5 Scaling high-rate continuous results

We introduce the model to support CEP processing together with the data-warehouse.

In Figure 3.6, in grey color we highlight the relevant modules. Just after Transformation, a new data path is added to a CEP-related Data Buffer that holds transformed data for integration into the CEP module. Besides also using the data that is output fro Transformation, the added CEP module does not change anything in the normal ETL+Q processing pipeline.

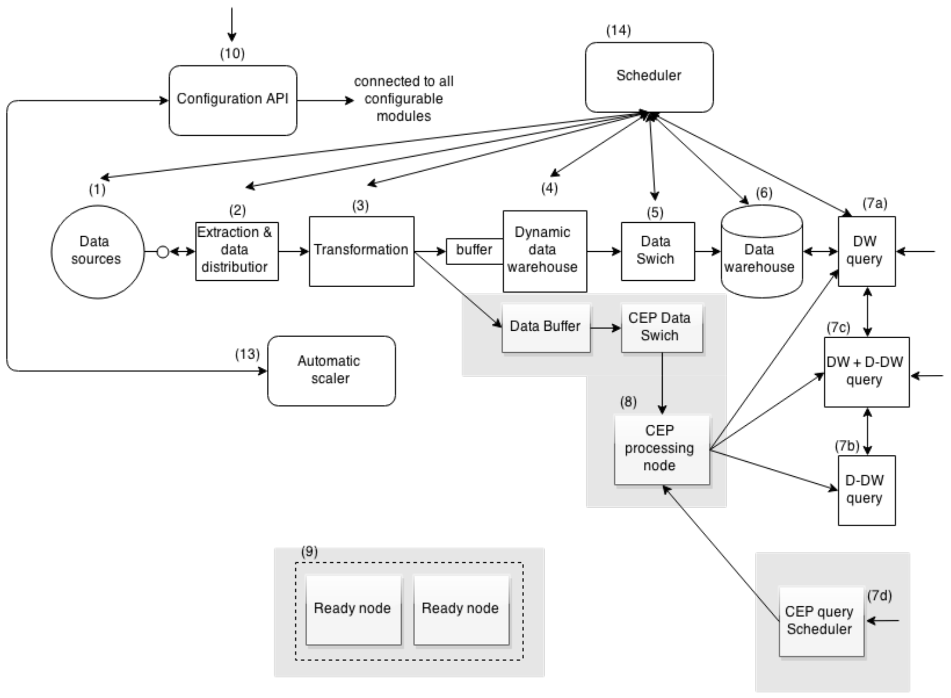


Figure 3.6: High-rate over continuous results

Note that some arrow connections are omitted for simplicity.
At the same level of a Data Switch (5) that distributes and replicates data into the data-warehouse (6), we introduce a CEP Data Switch. It works in the same fashion as the normal Data Switch, but replicates all data across all CEP processing nodes (8).
The CEP Query Scheduler (7d), decides where to register CEP queries (which CEP processing node (8) runs each query). CEP queries can also interact with the data-warehouse (6) or dynamic-data-warehouse (4) or both.

Figure 3.7 depicts the main components that are added to provide scalability over continuous results.

(5a) CEP Data Switch distributes and replicates the data by all CEP processing nodes (8). Associated to (5a) there is a Data Buffer (4) to collect and hold data from all transform (3) nodes;

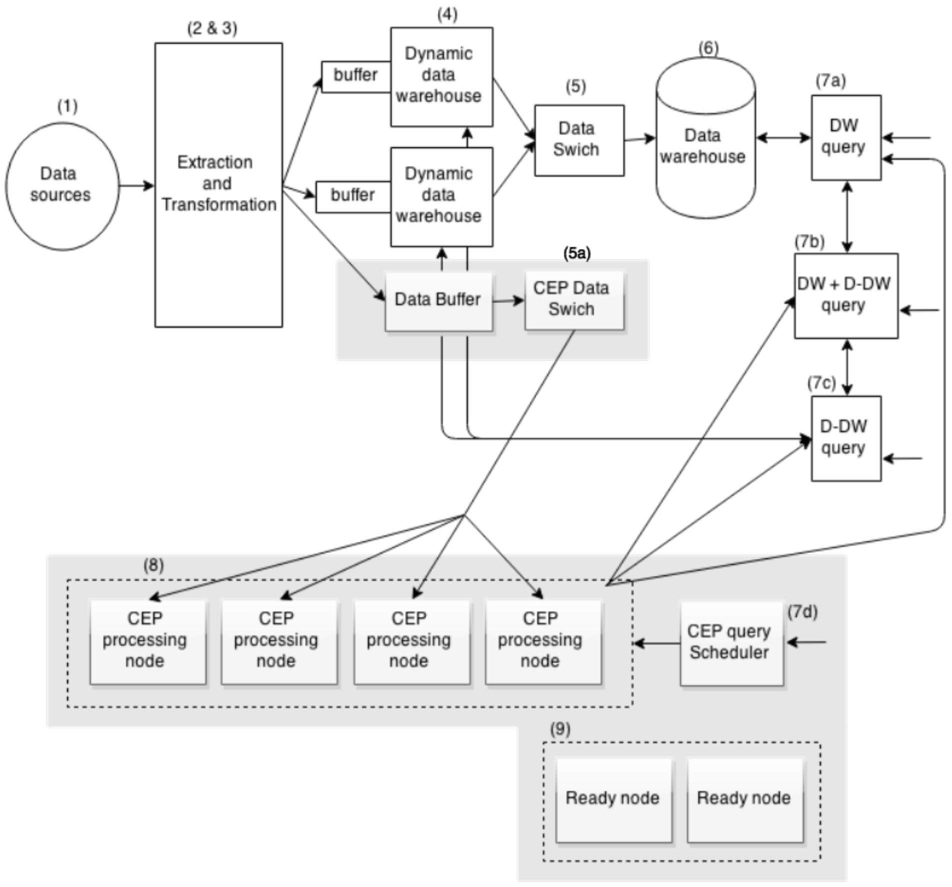


Figure 3.7: High-rate scalability over continuous results

(7) Queries can be submitted to the data-warehouse (7a) and dynamic-data-warehouse (7b). Continuous queries, on the other hand, need to be registered/de-registered from the CEP nodes (7d). CEP Queries can also include database sub-queries that will query the data-warehouse (7a, 7b);

(8) CEP processing nodes are also managed by AScale scheduler module. The nodes can be added and removed dynamically, to provide more efficiency, or to save resources, based on a ingress data rate queue that is monitored for overload detection. Once an overload situation is detected, CEP queries are re-scheduled or a ready-node is added;

(9) The ready-nodes are stand-by nodes that can be added to the processing nodes if necessary (e.g. high-rate data spikes) or when nodes are not being used to their full capacity they can be set on stand-by and used for other processing tasks.

There is also an "Automatic Scaler" (Figure 3.2) which is responsible for maintaining the system working efficiently; balancing CEP queries; managing the addition and removal of nodes; collecting performance information of all parts of the system for better assessment of where more performance is required; making decisions related with query re-balance and relocation.

Chapter 4

Integrating any data-warehouse with AScale

In this chapter we describe how a data-warehouse developer can integrate automatic scalability in a data-warehouse design and deployment process, to avoid having to deal explicitly with scalability and real-time. The approach we propose is embodied in the AScale framework. Therefore, our description is based on that framework. AScale only requires the data-warehouse developer to design the data-warehouse schema and to code the transformations for a single node, using any language or tool, then the setup is connected to AScale via APIs. For each part of the AScale architecture, we present the connection interfaces, to be used and configured by the developer.
All APIs mentioned in this chapter can be found in Appendix E in more detail, and Appendix F shows an XML possible configuration file for AScale.

4.1 AScale modules and connections

In this section we describe AScale ETL+Q pipeline design.
In Figure 4.1, highlighted in grey are the modules which the developer needs to develop himself, the connection modules that need to be configured are

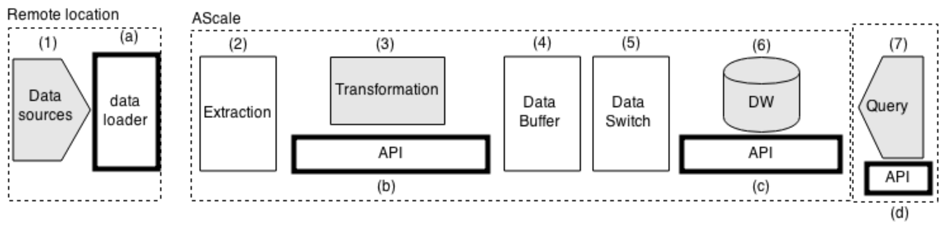


Figure 4.1: AScale pipeline modules and connections

represented with thick black lines, and the rectangles with white background represent the ETL+Q remaining modules of AScale.

(1) and (a) Each data source can be in a remote location. For each data source, the developer must configure a module named ”data loader”. The interface between the data sources and the AScale extraction module (2) can be seen as a producer/consumer pattern. The data sources are producers, the ”data loader” is a the shared buffer, and the extraction nodes are the consumers. The API of (a) needs to be configured to load/read data from (1) and make it available for extraction (2). (a) reads into memory large amounts of data from a set of sequential files. Extraction files are identified by a sequential prefix used to extract in correct sequence.

(2) Performs the extraction from (1)(a) according to configured time bounds, frequency (e.g. every day at 1am extract data for a maximum time of 5 hours) and data extraction sizes.

(3) and (b) the transformation code must be provided by the data-warehouse developer or some tool that generates the transformation process. There are two possible ways to program the connections with AScale: Importing an AScale library implementing the interface API to get data, transform it and submit it back (b); Other option is to connect to a web service and use the API to get the data, perform the data transformations and submit the data

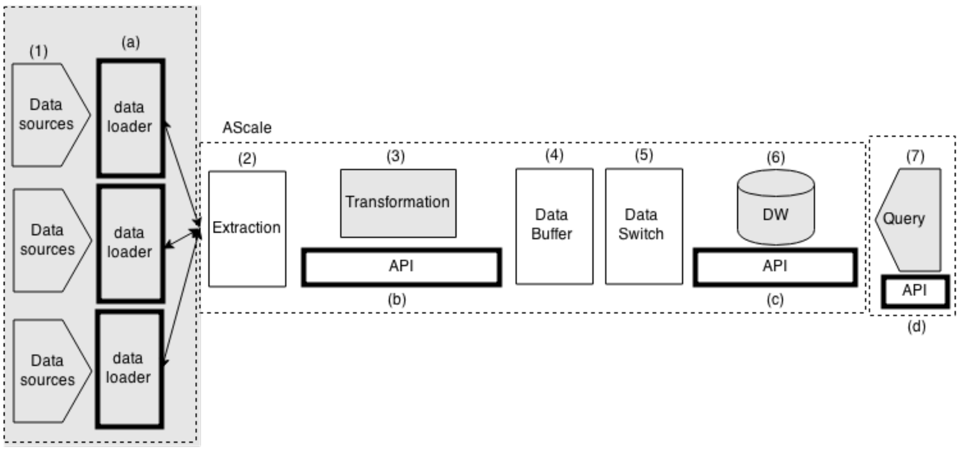


Figure 4.2: Data Sources

back.
Data-warehouse designing tools can integrate with AScale by importing and applying AScale API.

(6) and (c) The developer can choose to use any database engine. The API (c) must be configured to connect to the database.

(7) and (d) Queries can be submitted though a web service or library API. For testing purposes, it is also possible to submit queries to an AScale console.

In the next sections we detail how to connect to each AScale module, and how to configure it for automatic scalability.

4.2 Data sources

In this section we explain how Data Sources interact with AScale Data Loader. The Data Loader bridges Data Sources to AScale.

Figure 4.2 highlights the data sources (1) and the AScale "data loader" (a). Each data source is expected to supply log files with the data to be extracted. In order to avoid simultaneous reading and writing from log files, the data source is expected to provide the data as consecutive log files (numerated as 0, 1, ...). The "data loader" (a), reads into memory data located at the data sources (1) in log files, and makes that data available for extraction by (2). AScale requires configuration of log files name prefixes. After module (a) is configured, data extraction (2) connects directly to the "data loader" (a) (i.e. using sockets) for extraction.
Data extraction is done by following the file sequence (e.g. stock-0, stock-1, stock-2). A file n is only loaded into memory for extraction when n+1 is created. After data extraction is complete, the file is deleted.

Listing 4.1: Data loader configuration

```
1  dataLoaderSetLog(
2  "stock",          //log ID
3  "../home/,        //log file location
4  "stock",          //log file base name
5  "-");             //log file name sequential separator character
```

Listing 4.1 details the data loader configuration parameters. This configuration consists of: the parameter in line 2 (log ID) identifies the data, this ID is concatenated as a prefix to each row of data; The input parameter in line 3 (log file location) identifies the storage path of the respective log file; The input parameter in line 4 (log file base name) identifies the file name; Finally the input parameter in line 5, identifies in the log file name separator that appears before the sequential numbering (e.g. stock-0, stock-1).

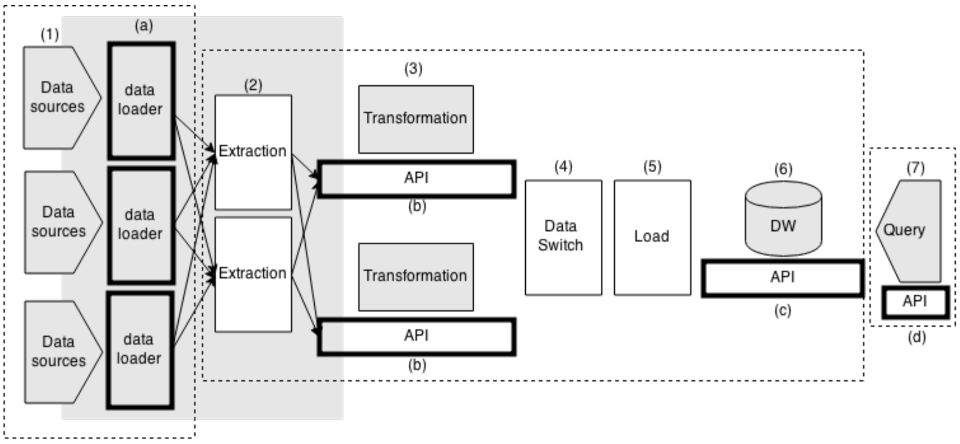


Figure 4.3: Extraction from (1)(a) into (2)

4.3 Extraction

The developer needs to configure extraction parameters such as: extraction frequency, maximum extraction time, the format of extracted data, and data distribution policy, which defines the strategy used to distribute extracted rows into transformation nodes, when there are multiple transformation nodes.

Figure 4.3, highlights the extraction module(s) (2) used to extract data from data sources (1)(a) and distribute it across transformation nodes (3)(b). Data extraction size is configured by the developer as the maximum data chunk size. This size is used to transfer data between (a) and (2), (2) and (3), (4) and (5), (5) and (6).

The developer configures, for each log file, an extraction frequency and the maximum load time window. Data extraction is controlled by a scheduler process (mentioned before in Chapter 3 Figure 3.2(14)), and data is extracted using fixed data chunk sizes, configured by the developer. A data chunk is a set of data rows that are extracted together.

At all moments extraction nodes (2) are connected to all Data Loaders (1)(a). If the connection fails, it is automatically re-established. If re-

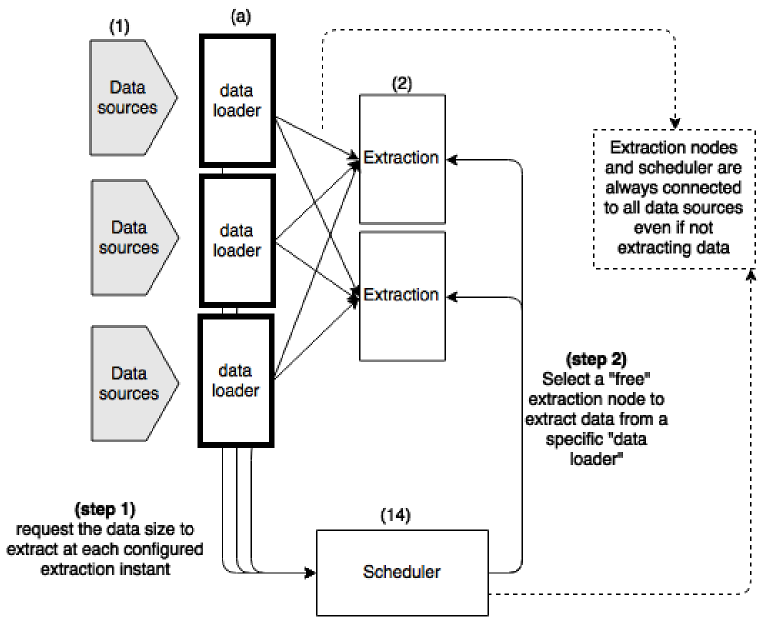

(1)
Data sources
Data sources
Data sources
(a)
data loader
data loader
data loader
(2)
Extraction
Extraction
Extraction nodes and scheduler are always connected to all data sources even if not extracting data
(step 2) Select a "free" extraction node to extract data from a specific "data loader"
(step 1) request the data size to extract at each configured extraction instant
(14)
Scheduler

Figure 4.4: Extraction method

connection is not possible, a warning is issued.

Figure 4.4 describes the default extraction method to get data from (1)(a) into (2). At every data extraction instance (configured) of a log file, the scheduler (14) asks each data source (a) the amount of data each log file has to be extracted, Figure 4.4(step 1).

By default, for each data loader (1)(a) and log file, Figure 4.4(step 2), the scheduler (14) chooses randomly an available Extraction node (2) to perform the data extraction (using the default maximum data chunk size).

After data is extracted, it needs to be sent to the transformation nodes data queues, Figure 4.3(3). By default, data is distributed using Least-Work-Remaining (LWR) to load balance the amount of data to transform in each transformation node. Transformation node data queues are used to detect overload situations and to scale when necessary.

Listing 4.2: Extraction data source format

```
extractSetDataSourceFormat(
2  "source1",        //source ID
3  "stock",          //log ID
4  "|",              //column separate character
5  "\n");            //row separate character
```

Listing 4.2 shows the configuration of the data sources data format specification. Line 2 identifies the data source ID. Line 3 identifies the log ID. Line 4 and 5, identify column and row delimiter character, respectively.
The extraction process adds the log ID as prefix. This extra information is meant to help the developer identifying the transformations to be applied, since they depend on which data is to be transformed (e.g. which fact or which dimension).

4.4 Transformation

Transformation code must be provided either directly by the developer, or by a data warehouse design tool that generates the executable code or the transformation operations. Either the developer or the tool must call functions of the AScale API to interface with the automatic scalability framework.
In this section we explain how the developer can connect a transformation module to AScale and include a sequence diagram to explain how AScale modules interact with data transformation.

Figure 4.5 highlights the transformation nodes (3). Data from the extraction process (2) is placed in data queues of transformation modules. The queues detect scalability needs based on a configured maximum size. Remember that each extracted row is concatenated with a prefix containing identification data log, with the purpose of allowing the transformation to identify which data must be transformed.
In order to apply transformations, the transformation code can be connected using two alternative methods. First, by importing an AScale library API to interface with the input transformation queues (b) and with the output

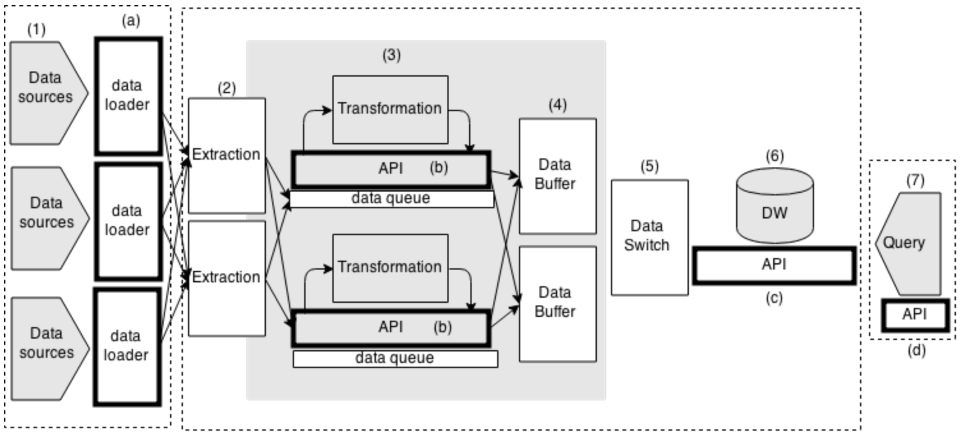


Figure 4.5: Transformation

data buffer, second by connecting to a web Service API (b). Data retrieval from the "data queue" is done using the default data chunk size.

Listing 4.3: Transformation API get data

```
array[] = transformGetExtractData();
```

The transformation module, which is implemented directly by the developer or generated automatically by some data warehouse ETL design tool, is expected to call the function "transformGetExtractData" of Listing 4.3 to read input data from the queues. This is a blocking function, which means that the transformation module will call this function repeatedly, each time waiting until data is available to be read.
In the listing, the output parameter "array" is a data chunk made of a set of rows with extracted data in text format. Note that the header of the data contains the identification of the source and log file.

Listing 4.4: Transformation API submit data

```
transformSetOutput(
"112|Pedro|30|Portugal\n",        \\transformed data
"|",                              \\column separate character
```

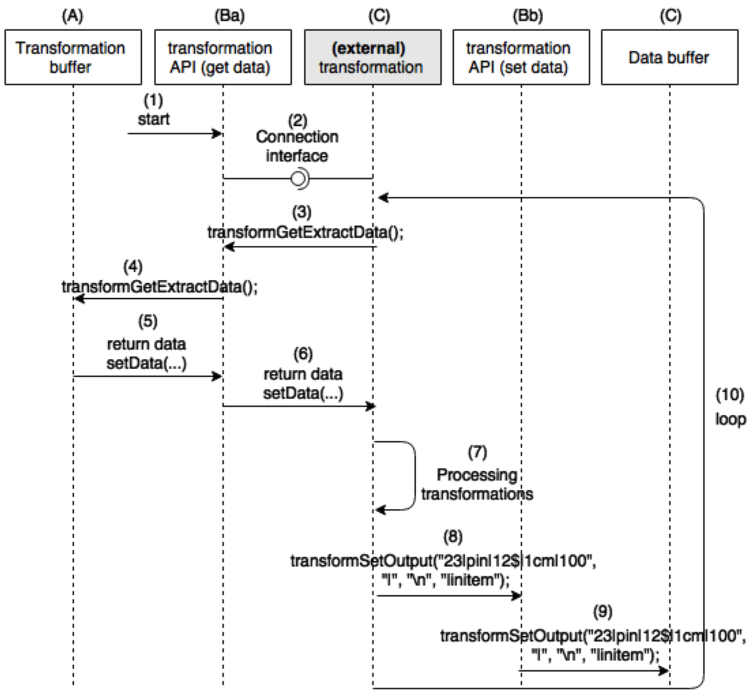

(A)
Transformation buffer
(Ba)
transformation API (get data)
(C)
(external) transformation
(Bb)
transformation API (set data)
(C)
Data buffer
(1) start
(2) Connection interface
(3) transformGetExtractData();
(4) transformGetExtractData();
(5) return data setData(...)
(6) return data setData(...)
(7) Processing transformations
(8) transformSetOutput("23|pin|12$|1cm|100", "|", "\n", "linitem");
(9) transformSetOutput("23|pin|12$|1cm|100", "|", "\n", "linitem");
(10) loop

Figure 4.6: Sequence diagram, transformation

```
4  "\n",                              \\line separate character
5  "users",                           \\data warehouse schema table name
6  );
```

The transformation module is also expected to call the function "transformSetOutput" of Listing 4.4 to submit output data. The output data (line 2) can contain any number of transformed rows. Besides the rows themselves, the call must specify field and row delimiter characters (line 3 and 4) and the data warehouse schema table name (line 5) which the data corresponds to.

After transformation, data is placed inside the data buffer (4) nodes using the LWR distribution policy, by default. It is also possible to configure manually the data distribution policy, Appendix E.

Figure 4.6 shows the sequence diagram to get data from AScale to be transformed, and to submit it back to AScale pipeline.
On top we depict:
(A) The transformation queues, which hold data to be transformed;
(B) The transformation API library or Web Service API;
(C) The transformation code, created by the developer/tool;
(D) The data buffer, which holds transformed data to be loaded into the data warehouse;

The sequence of events to extract and transform data occurs as follows:
(1) The API library or web service is up and ready;
(2) The transformation code (C) connects to the API;
(3) Transformation code (3) requests data;
(4) The API (Ba) gets data from the transformation buffer (A); (5) The transformation buffer returns and forwards the data to (Ba);
(6) The (Ba) forwards the data to (C); (7) Transformations are applied;
(8) Transformation code (C), submits the transformed data back to AScale connection API (Bb);
(9) The connection API (Bb) sends the transformed data to be stored at the data buffer (C) until the next load instant;
(10) represents the loop, which the developer transformation code must perform to extract and transform data.

4.5 Data buffer and Data Switch modules

Figure 4.7 shows the data buffers (4) used to store the transformed data temporarily until the next data-warehouse load period.
As explained before transformed data is submitted into the data buffer nodes using an LWR policy (by default). That data is later retrieved by the data switches (5) using a scheduler-based method similar to the one used to ex-

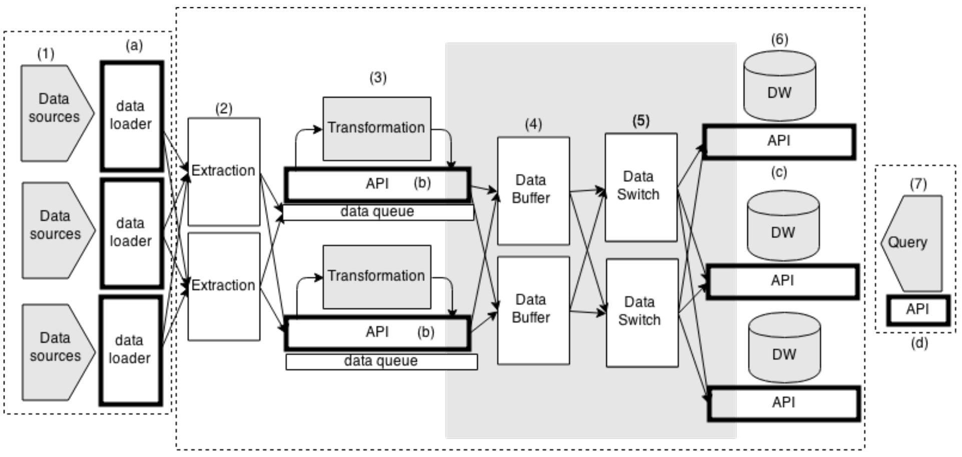


Figure 4.7: AScale data buffer and data switch nodes

tract data from the sources (1)(a).

The data buffer has four storage parameters: memory size to use; maximum memory size before data starts being swapped into disk; disk size; and maximum disk size.

Data loading from the data buffers (4) is managed by a scheduler. When data switch nodes are free (5), the AScale scheduler selects the data buffer holding more data to be extracted. The data switch node (5) retrieves a data chunk size, processes the data according with distribution and replication configurations, and places it in data-warehouse nodes for loading.

The data buffer nodes (4) scale based on their I/O capacity to prevent the memory from reaching the maximum configured size, or before the disk space reaches the maximum size limit.

Data switch nodes (5) scale based on a configured maximum data-rate. If that maximum data-rate is reached for a certain time window, then AScale adds more data switch nodes.

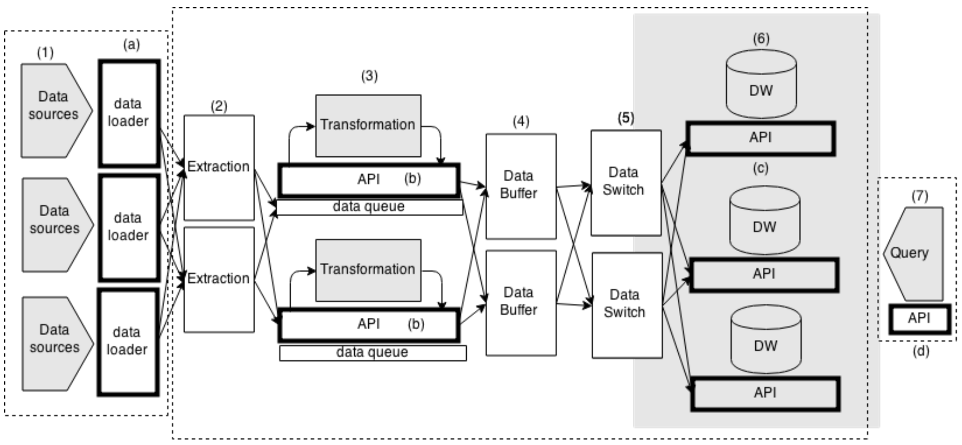


Figure 4.8: Data-warehouse nodes

4.6 Data-warehouse

This section describes how to interface the data-warehouse schema with AScale, and the different ways data-warehouse nodes monitoring can trigger scale-out and scale-in.

Figure 4.8 highlights the data warehouse (6) and the connection API (c). To set the data-warehouse nodes schema and operations to perform before and after the load instant, the developer needs to submit three files to AScale framework: first, the data-warehouse schema creation script, including SQL commands to create all tables, indexes and views; second and third, the pre-load and post-load scripts, in SQL format. The pre-load and post-load scripts are executed prior to the load instant and at the end of the load instant respectively. Pre-load tasks include, for instance, drop of all indexes and views. The post-load script includes, rebuilding of dropped structures, such as indexes and views.

Listing 4.5: Data warehouse schema configuration

```
dataWarehouseSetSchema(
"/.../script");
```

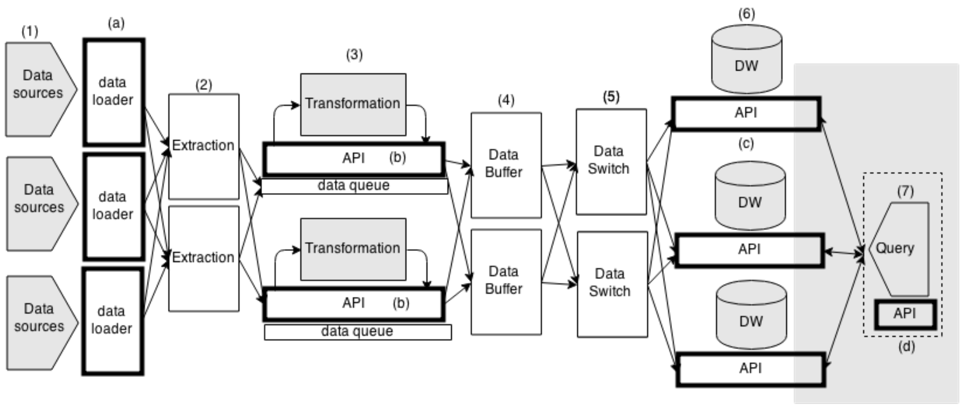


Figure 4.9: AScale Querying

Listing 4.5 shows the call to the API to upload the data-warehouse schema. The schema must be submitted in standard SQL format. The indexes, views and key identifiers must be defined. All data-warehouse nodes will have the same schema.

Listing 4.6: Data warehouse pre-load and pos-load configurations

```
dataWarehouseSetPreLoadTasks(
"../*/preLoad.sql");

dataWarehouseSetPosLoadTasks(
"../*/postLoad.sql");
```

Listing 4.6 shows the call to the API to upload the pre-load and post-load configurations.

Data-warehouse scalability needs can be triggered by the load time limit being overpassed. The developer configures the period or instant, and configures the maximum load time (maximum duration of a load). If the maximum load time is reached, then the data-warehouse nodes are scaled-out and data is re-balanced across all nodes.

Figure 4.9 highlights the query execution module (7)(d), including the API to connect and submit queries.

As mentioned before, queries can be submitted using the AScale terminal, an API library or a Web Service. In all three modes, the interfaces for querying are the same.
Queries can be executed against the main data-warehouse, and/or the dynamic-data-warehouse. In all cases, after query execution, results from all nodes are merged to obtain the final result.

Listing 4.7: Query execution

```
array[] = querieSetRun(
"SELECT sum(price), product, city FROM sales, product, customer
 WHERE sales.prodid=product.prodid and sales.custid=customer.custid
 GROUP BY product, city",        //submitted query
"DW");                           //data storage system to query
```

Listing 4.7 shows the API to perform querying to the data-warehouse. Input parameters: the SQL query; the data-warehouse to run the query against (DW = data-warehouse, D-DW = dynamic-data-warehouse, D-DW+DW = both). Output parameters: an array with the query results.

The data-warehouse nodes also scale based on the query execution time. The developer configures a maximum desired query execution time, if queries take more time than the maximum, the data-warehouse and/or dynamic-data-warehouse nodes are set to scale and data is re-balanced.

The sequence diagram of Figure 4.10 shows the interaction between the developer application (A), AScale query API (B), Query manager (C) (Figure 4.9 (7)) and the data-warehouse (D) (Figure 4.9 (6)(c)).

When a query is submitted by the developer application (A), the next sequence of events occurs:
(1) The application (A) connects to the AScale query API (B);
(2) The application (A) submits a query to the data-warehouse via query

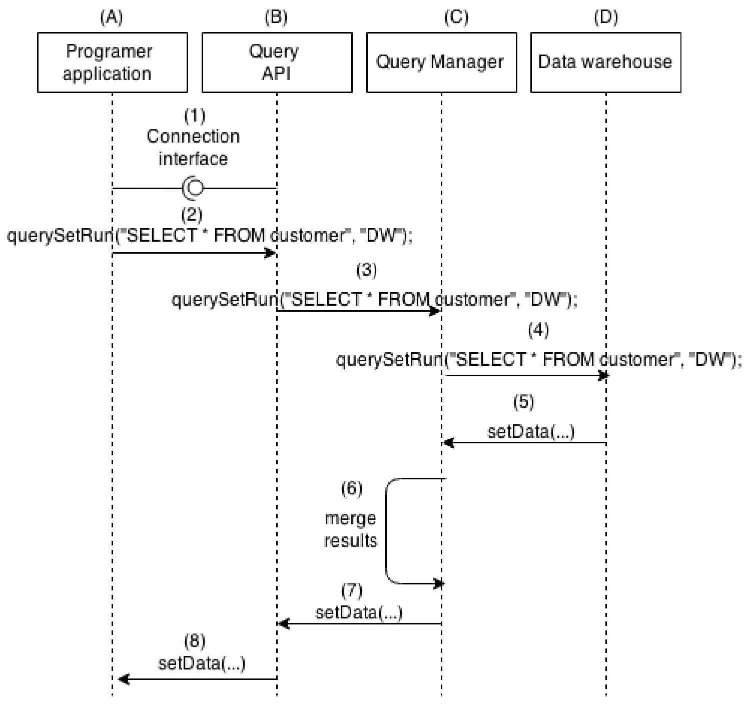


Figure 4.10: Query execution sequence diagram

API (B);

(3) The query API forwards the request to the query manager (C);

(4) The query manager (C) re-writes the query to be executed in the distributed data-warehouse using the approach in [77], and submits it to the data-warehouse nodes (D);

(5) Results from the data-warehouse (D) are returned;

(6) Query manager (C) merges all results;

(7) query manager (C) return the final result to the query API (B) which forwards it (8) to the developer application (A).

Chapter 5

Auto-scalability of ETL and processing

In this chapter we explain the mechanisms and configuration parameters that AScale uses to monitor, detect and scale each processing module. After explaining some of the most relevant configuration parameters, we detail how scaling decisions are made. In order to do that, we explain the implemented algorithms and illustrate how they work. Finally, we specify, for each part of AScale, the data distribution policies used to share and transfer data across different AScale modules.
For more details regarding API configuration, we redirect to Appendix E.

5.1 Configuring AScale for automatic scalability

Each part of the ETL+Q process must scale in order to overcome performance limitations. For instance if we have many Data Sources supplying data, and at each stage of the processing a single computing node may not be able to handle all data extraction, transformation or any other part of the AScale pipeline.
In this section we describe the scalability configuration parameters used by

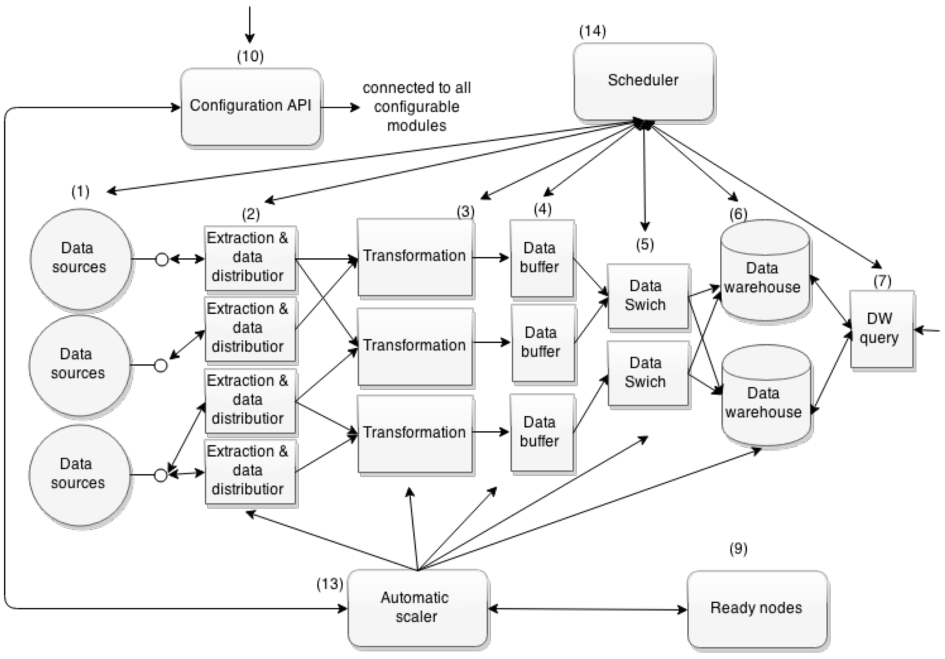


Figure 5.1: AScale framework for scalability

AScale to scale each module, independently and as necessary.

Figure 5.1 shows each AScale module that may need to scale to offer desired performance.

Listing 5.1: Extraction, scalability configuration

```
1  extractSetDataSourceLog(
2  "source1",                                          //source ID
3  {"logA", "logB"},                                   //log ID
4  {"*/10␣*␣*␣*␣*␣*", "*/10␣*␣*␣*␣*␣*"},               //extraction frequency
5  {"5s", "5s"});                                      //maximum extraction time
```

Extraction nodes (2) are monitored to determine scaling needs based on extraction frequency, Listing 5.1 line 4 (Using Unix cronjob format), and the maximum extraction time, line 5.
If the maximum extraction time is exceeded, then more extraction nodes are added. If the maximum extraction time is not defined, then, if the extraction

takes longer than the frequency cycle duration, more nodes (2) are added from the ready-nodes area (9).

Listing 5.2: Transformation, scalability configuration

```
1  transformSetMaxSize(
2  "16GB",          //maximum queue size
3  "5GB");          //maximum limit size for scaling detection
```

Transformation nodes (3) include a data queue with a maximum load size. Listing 5.2 line 2, specifies the maximum queue size, and line 3 specifies the maximum limit size for scaling detection.
Ingress data goes inside the queues, then the transformation nodes, (with the transformation operations programmed by the data-warehouse developer), extract and transform data. If at any point the queue starts filing up above a certain configured limit, it indicates that the ingress data-rate is more than the output transformation data-rate. Thus, more transformation nodes must be added.
When scaling-up, a new node is added and the entire transformation process, present in other nodes, is replicated to the new node.

Listing 5.3: Data buffer, scalability configuration

```
dataBufferSetSize(
"dataBuffer1",
"5GB",          //maximum buffer memory size
"10GB",         //limit buffer memory size
"500GB",        //maximum buffer disk size
"250GB",        //limit buffer disk size
"D:");          //data buffer disk location
```

The **Data Buffer nodes (4)** hold transformed data until the next data-warehouse load instant.
Scaling decisions are made based on a number of parameters: maximum

allowed in-memory buffer size; maximum allowed data write speed; and maximum allowed disk size. Listing 5.3 illustrates these parameters.
If the memory usage reaches the maximum configured data buffer memory size, then data is swapped into disk. If even so the memory becomes completely full, reaching the maximum memory size, more data buffer nodes must be added. Also, if the disk space reaches a configured limit, more data buffer nodes must be added.

Listing 5.4: Data switch, scalability configuration

```
1  dataSwitchSetDataRate(
2  "dataSwitch1",  //data switch ID name
3  "80000l/s",     //maximum supported data-rate
4  "2m");          //maximum time delay to trigger scale-out
```

Data Switch nodes (5) distribute and replicate data across the data-warehouse nodes. These nodes extract data from the data buffers (4) using a scheduler based extraction policy and load it into the data-warehouse nodes (6). However, there are limitations regarding the amount of data each data switch node can handle. The command line in Listing 5.4 line 3, is used to specify the maximum supported data-rate in lines per second. Line 4, represents the maximum time delay before triggering scale-out mechanisms. If, for the configured time duration, the data switch is always working at the maximum configured data-rate, that means that these nodes are working at their maximum capacity (according to configuration) and must be scaled.

Listing 5.5: Data warehouse, scalability configuration

```
dataWarehouseSetLoad(
"* 30 1 * * *", //load frequency
"5h",           //maximum load time
"100MB");       //maximum batch file size
```

The **data-warehouse nodes (6)** load data during fixed instants for a certain time window.
Listing 5.5, line 2, represents the load frequency using the Unix cronjob time format, and Line 3, represents the maximum allowed load time. If the maximum allowed load time is exceeded, then more data-warehouse nodes need to be added.
Another data-warehouse scale scenario regards queries execution time. If queries take more time than a maximum configured limit to output the results, data-warehouse nodes (6) must scale to offer more performance.

Listing 5.6: Maximum query execution time configuration

```
querySetMaxDWQueryExecutionTime(
value); //max execution time for DW queries

querySetMaxD−DWQueryExecutionTime(
value); //max execution time for D−DW queries
```

Listing 5.6 shows the API to configure the maximum query execution time. The input parameters include the maximum desired execution time in seconds (s) or minutes (m).

5.2 Decision algorithms for scalability

This section defines scalability decision methods as well as algorithms which allow AScale to automatically scale-out and scale-in each part of the proposed pipeline.

5.2.1 Extraction & data distributors

Flowchart 5.2, describes the algorithm used to scale-out. Depending on the number of existing sources and increasing data generation rate, eventually extraction nodes have to scale.

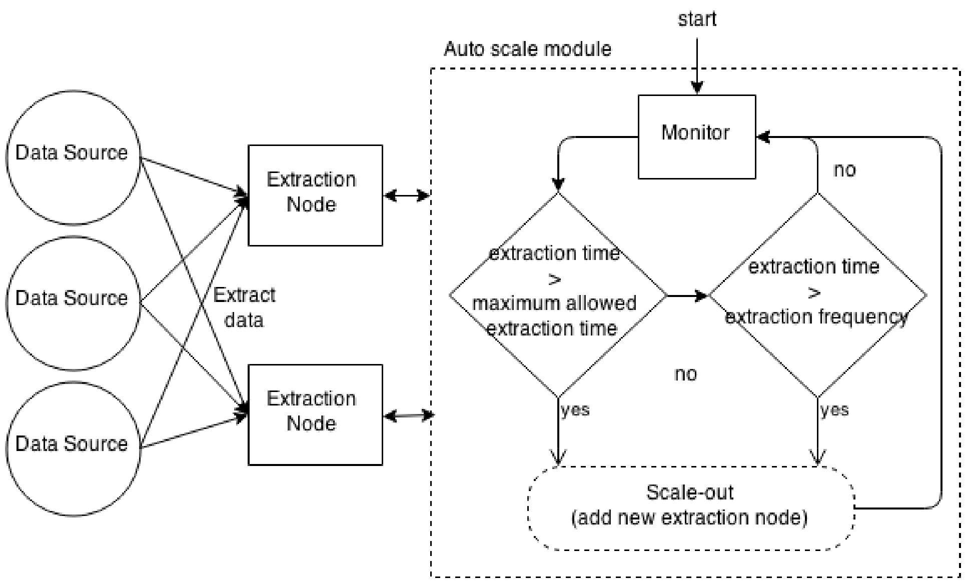

start
Auto scale module
Data Source
Data Source
Data Source
Extraction Node
Extraction Node
Extract data
Monitor
extraction time > maximum allowed extraction time
extraction time > extraction frequency
no
no
yes
yes
Scale-out (add new extraction node)

Figure 5.2: Extraction algorithm - scale-out

The addition of more "extraction & data distributors" nodes (2) depends on whether the current number of nodes is able to extract and process data with the correct frequency, within the configured maximum extraction time bound. For instance, if the extraction frequency is specified as every 5 minutes and extraction duration 10 seconds, then every 5 minutes, the "Extraction & Data distributor" nodes cannot spend more than 10 seconds extracting all data. Otherwise a scale-out is needed. If the maximum extraction duration is not configured, then the extraction process must finish before the next extraction instant, as specified by the extraction frequency parameter.

Flowchart 5.3, describes the algorithm used to scale-in.
To save resources and reuse them, data extraction nodes can scale-in. The decision is made based on last execution times. If previous execution times of at least two or more nodes are less than half of the maximum extraction time, minus a configured variation parameter (X), one of the nodes is set on standby (as ready-node) or removed and the other ones takes over.

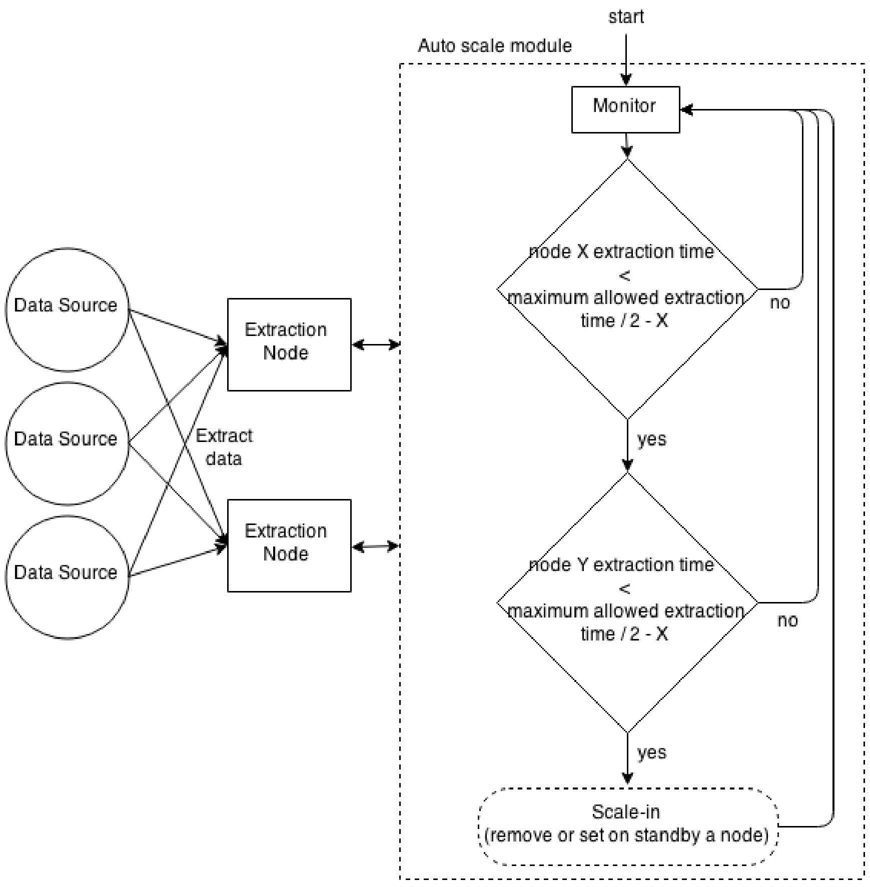

start
Auto scale module
Monitor
node X extraction time < maximum allowed extraction time / 2 - X
no
yes
node Y extraction time < maximum allowed extraction time / 2 - X
no
yes
Scale-in (remove or set on standby a node)
Data Source
Data Source
Data Source
Extraction Node
Extraction Node
Extract data

Figure 5.3: Extraction algorithm - scale-in

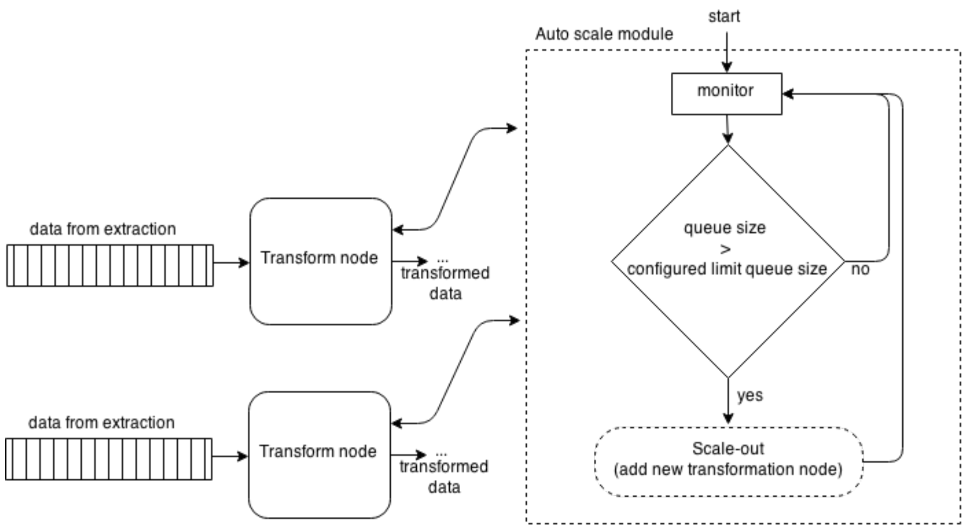


Figure 5.4: Transformation - scale-out

5.2.2 Transform

If the transformation is running slow, data extraction at the current data-rate may not be possible, therefore information will not be available for loading and querying when necessary.

Transformation nodes have an input queue, as shown in Figure 5.4. In the figure we show the transform queue, used to determine when to scale the transformation phase. If this queue reaches a limit size (configured by the developer) because the actual transformer node(s) is not being able to process all data that is arriving (i.e. current ingress data-rate is larger than transformation output data-rate), then it is necessary to scale-out. Flowchart 5.4, describes the algorithm used to scale-out.

Flowchart 5.5, describes the algorithm used to scale-in. If queues size at a specific moment are less than half of the limit size for at least two nodes, then one of those nodes is set on standby (as ready node) or removed.

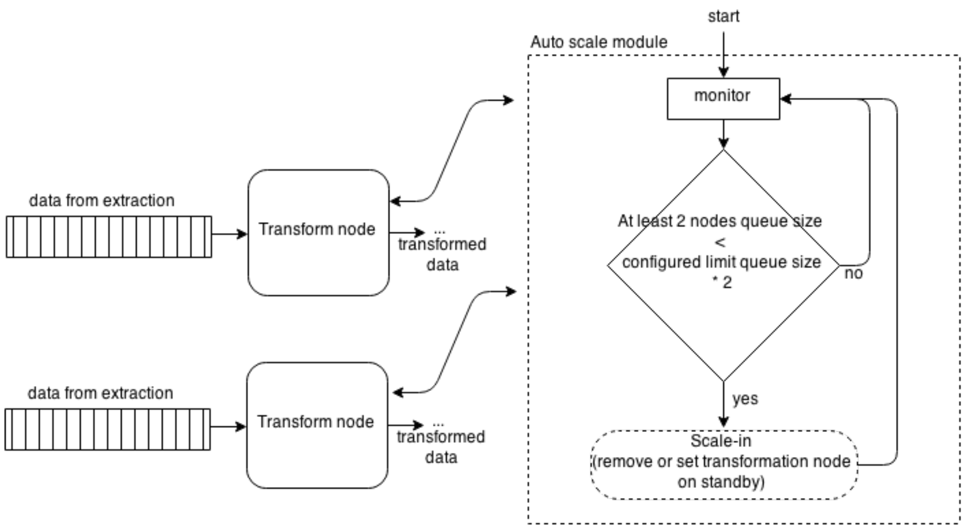


Figure 5.5: Transformation - scale-in

5.2.3 Data buffer

The data buffer nodes scale-out based on the memory size, the data swap capacity from memory into disk and the available storage space to hold data. When the available memory becomes above a certain limit, data starts being swapped into disk to reduce memory use under the limit size. If even so the data buffer memory reaches the maximum memory limit size, then the data buffer scales-out. This means that the incoming data-rate (going into memory storage) is not being swapped to the disk storage fast enough, therefore more nodes are necessary.
When the used disk space in a node is full above a configured limit, the data buffers are also set to scale-out.
Flowchart 5.6, describes the algorithm used to scale-out the data buffer nodes.
Data buffers can also scale-in. In this case the system will do so if the data from any data buffer can fit inside the data buffer of any other node.

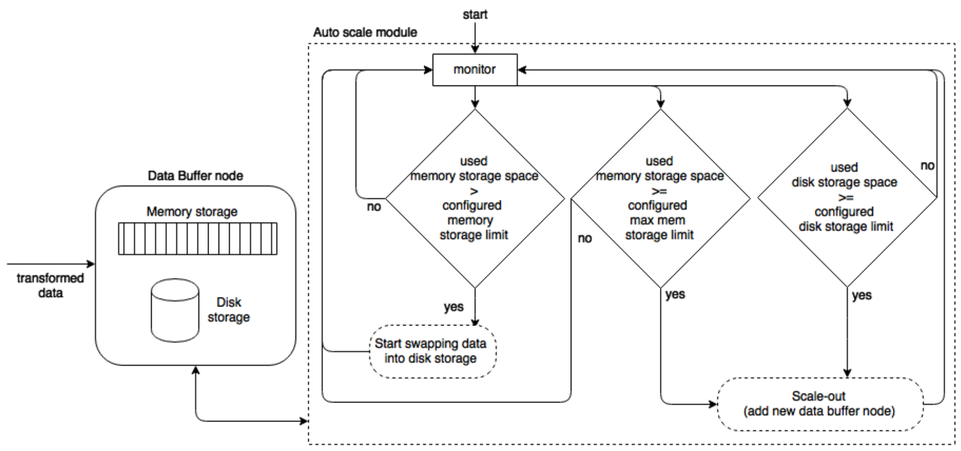

start
Auto scale module
monitor
used memory storage space > configured memory storage limit
used memory storage space >= configured max mem storage limit
used disk storage space >= configured disk storage limit
no
no
no
yes
yes
yes
Data Buffer node
Memory storage
Disk storage
transformed data
Start swapping data into disk storage
Scale-out (add new data buffer node)

Figure 5.6: Data Buffers - scale-out

5.2.4 Data switch

The Data Switch nodes scale based on a configured maximum supported data-rate. That data-rate cannot be reached or passed for more then a configured time window. If the average data-rate rises above the configured limit for a certain time window, data switch nodes are set to scale-out. Flowchart 5.7, describes the algorithm used to scale the data switch nodes. The data switches can also scale-in. In this case the system will allow it if the data-rate for at least two nodes is half of the configured maximum, minus a (Z) configured variation parameter.

5.2.5 Data-warehouse

Data-warehouse scalability needs are detected after each load process or after any query execution. The data-warehouse load process has a configured limit time duration to be executed every time it starts. If that time is exceeded, then the data-warehouse must scale-out. Likewise, queries also have a maximum execution time. If query execution time is exceeded, the data-warehouse must scale-out. The number of nodes to scale-out is determined assuming linear scalabilty based on previous number of nodes and execution

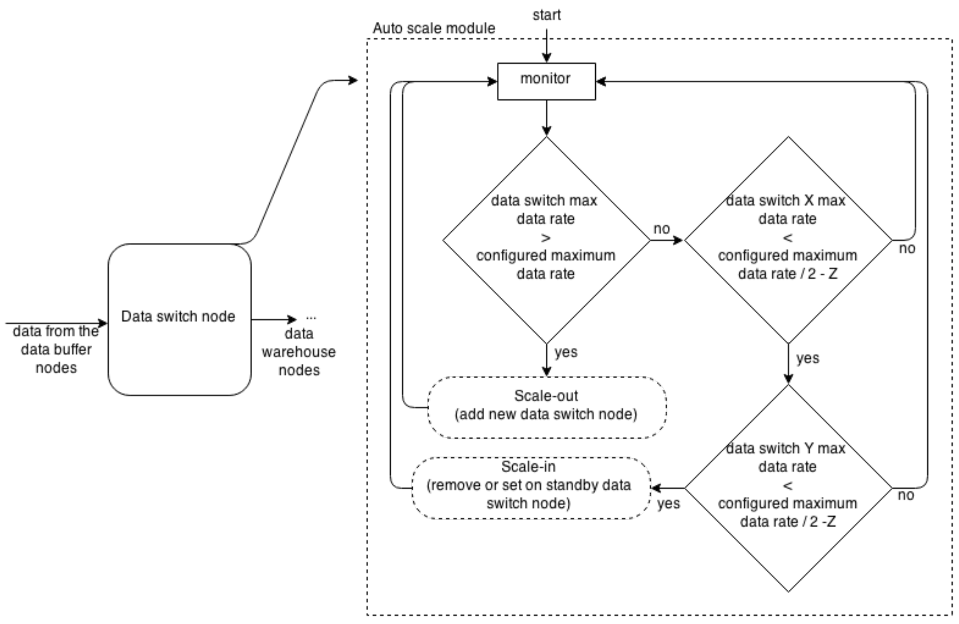

start
Auto scale module
monitor
data switch max data rate > configured maximum data rate
no
yes
data switch X max data rate < configured maximum data rate / 2 - Z
no
yes
Scale-out (add new data switch node)
Scale-in (remove or set on standby data switch node)
yes
data switch Y max data rate < configured maximum data rate / 2 -Z
no
data from the data buffer nodes
Data switch node
...
data warehouse nodes

Figure 5.7: Data switch - scale

time ($\frac{loadTime}{targetTime} \times n$). In this equation the "loadtime" represents the last recorded load time, "targetTime" represents the desired target load time and "n" represents the current number of nodes. Flowchart 5.8, describes the algorithm used to scale the data-warehouse when the maximum load time is exceeded.

Data-warehouse scalability is not only based on the load & integration speed requirements, but also on the maximum execution queries time. After a query is executed, if the query time is more than the configured maximum, then the data-warehouse is set to scale-out. Flowchart 5.9, describes the algorithm used to scale-out the data-warehouse based on the query execution time.

Data-warehouse nodes scale-in is performed iff the average query execution time and the average load time respect conditions 5.1 and 5.2 (where n represents the number of nodes):

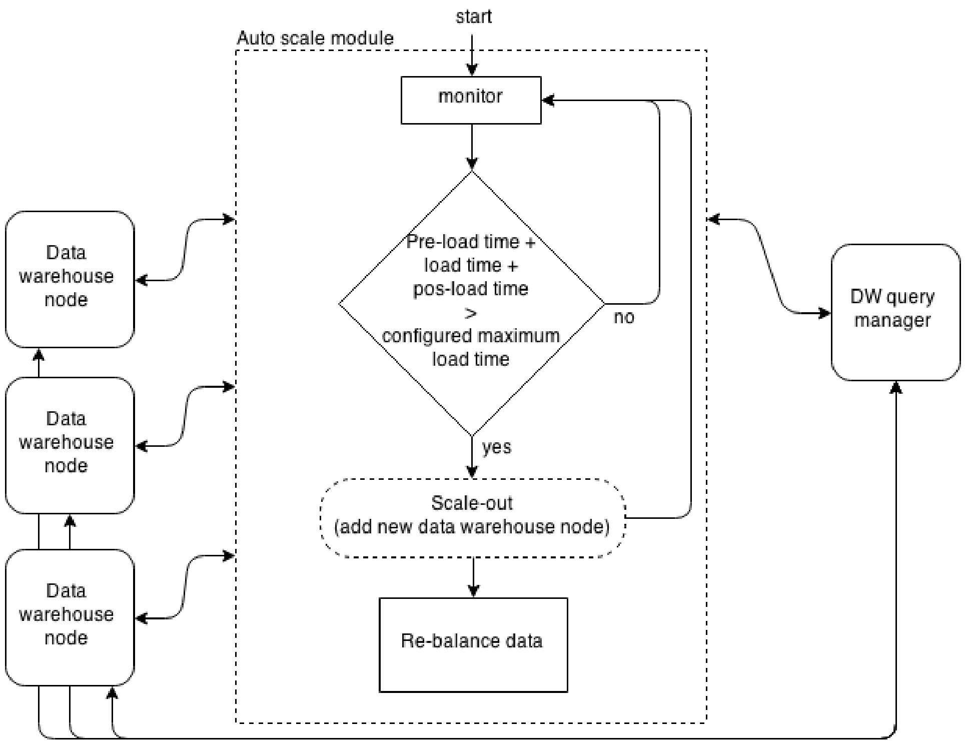


Figure 5.8: Data warehouse - scale

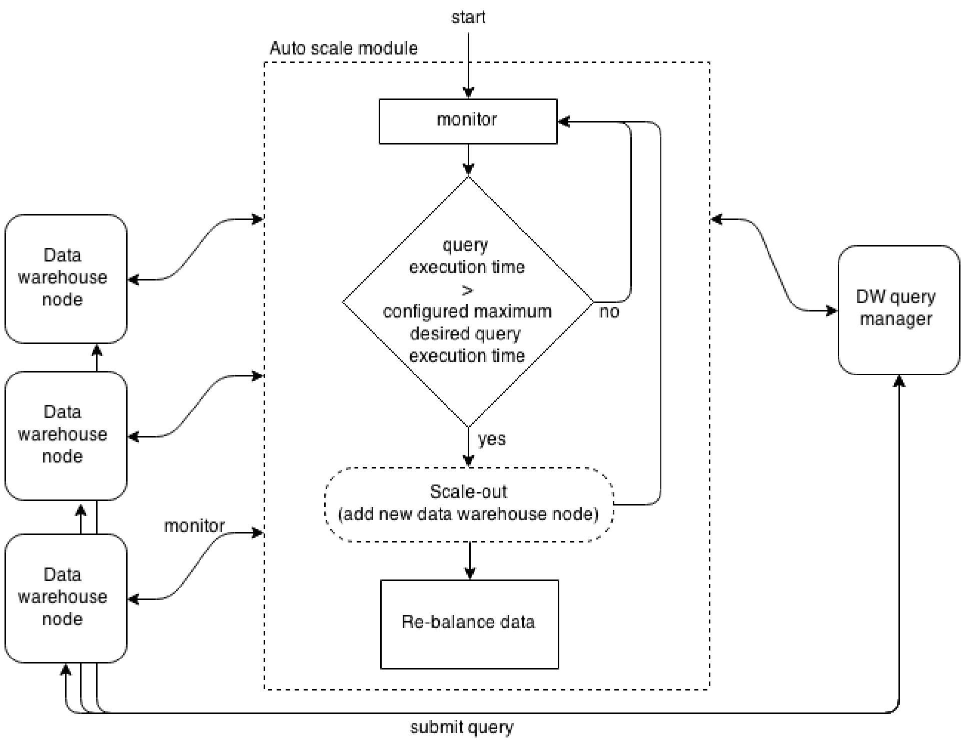


Figure 5.9: Data warehouse - scale based on query time

$$\frac{(n-1) \times avgQueryTime}{n} \leq desiredQueryTime \tag{5.1}$$

and

$$\frac{(n-1) \times avgLoadTime}{n} \leq maxLoadTime \tag{5.2}$$

Every time the data-warehouse scales-out or scales-in, the data inside the nodes needs to be re-balanced. The default re-balance process to scale-out is based on the phases: extract and replicate data from data-warehouse nodes; load the extracted information into the new nodes (data is extracted and loaded across the available nodes as if it is new data).

5.2.6 Global ETL scalability

Besides defining partial limits for each part of the ETL+Q pipeline, it is also useful to configure only a desired global ETL processing time. In this case, AScale will choose to scale-out the part of the pipeline that is performing slower.

Figure 5.10 explains the scaling of the ET and L based on an example. Assume that the current execution time of ET and L are respectively 2 and 10 hours, and the desired execution time is 5 hours. Based on these times, the target time for the ET is 0.83 hours and for the L is 4,17 hours. Then by applying the following formula the necessary number of nodes is estimated linearly, $\frac{currentTime}{targetTime} \times n$, where "currentTime" is the current execution time, "targetTime" is the desired execution time and "n" represent the current number of nodes. For the given example this results in: for the ET we would need $\frac{2}{0.83} \times 1 = 2,4 nodes$ which corresponds to 3 nodes, and for the L, $\frac{10}{4.17} \times 1 = 2,39 nodes$, which corresponds to 3 nodes. Note that, in the ET the extra nodes are added to the E process only, then the T process scales based on ingress data queues monitoring to keep up with the extraction rate.

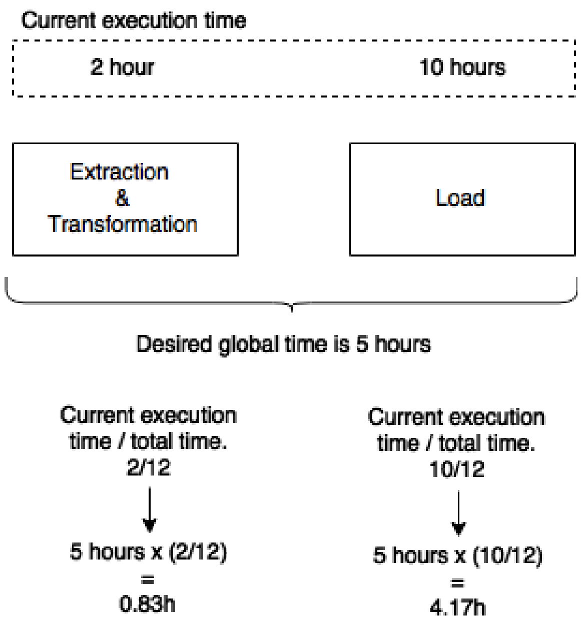

Current execution time
2 hour
10 hours
Extraction & Transformation
Load
Desired global time is 5 hours
Current execution time / total time. 2/12
Current execution time / total time. 10/12
5 hours x (2/12) = 0.83h
5 hours x (10/12) = 4.17h

Figure 5.10: Example, estimating the scaling proportion of the ET and L

Another option is to configure both types of time bound limits: a global time bound for the entire ETL process and at the same time local bounds for the parts. In this case AScale can use the local bounds to decide where to scale.

5.3 Data distribution policies

This section discusses data distribution policies used at each AScale pipeline stages. The next data distribution policies are considered: manual, round-robin, least-work-remaining, and based on AScale scheduler decisions.

5.3.1 From the data sources to extraction & data distributor nodes

Figure 5.11 represents the extraction process using the AScale scheduler module to command the data extraction based on source (1) data size, and

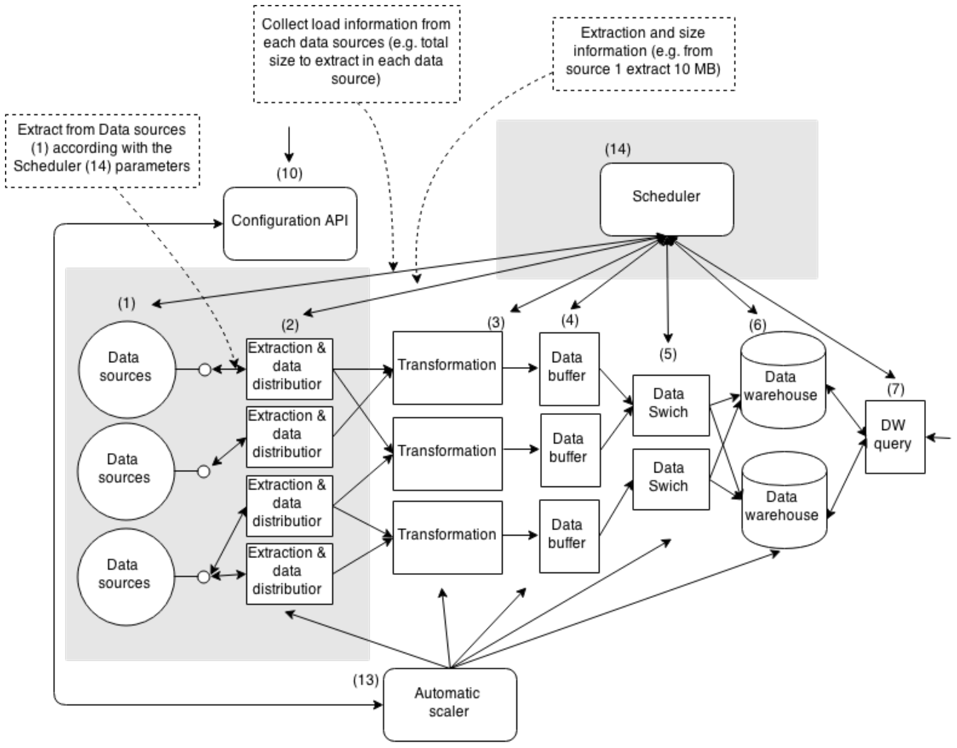

Collect load information from each data sources (e.g. total size to extract in each data source)
Extraction and size information (e.g. from source 1 extract 10 MB)
Extract from Data sources (1) according with the Scheduler (14) parameters
(10)
Configuration API
(14)
Scheduler
(1)
(2)
(3)
(4)
(5)
(6)
(7)
Data sources
Extraction & data distributior
Transformation
Data buffer
Data Swich
Data warehouse
DW query
(13)
Automatic scaler

Figure 5.11: Data and load distribution, sources to extraction

extraction node (2) availability. At each extraction instant, the scheduler (14) requests to all data sources to report the corresponding data size available for extraction. As soon as an extraction node notifies the scheduler that it is free for more work, the scheduler assigns to it the data source holding more data and the log file to extract data from. The extraction size is based on the chunk maximum size, configured by the developer for the entire AScale.
AScale also supports manual data extraction configuration, in this case the developer specifies, for each Extraction node (2), the Data Source (1) and log file to extract data from.

5.3.2 Extraction & data distributor nodes to transform nodes

Regarding the data distribution for the "Extraction & Data distributor" nodes (2) to the "Transform" nodes (3) , Figure 5.11, the default approach to place data from the extraction nodes (2) into the transformation nodes (3) is based on least-work-remaining (LWR) data distribution. Data is first placed on the transformation nodes with smaller data queues. The default maximum data chunk size is used to transfer the data.
Other possible distribution policies are: manual data distribution, where the developer specifies, for each Extraction (2) node, the corresponding Transformation (3) node; and also round-robin based.

5.3.3 From transform nodes to data buffer nodes

The modules involved in the temporary data storage from the Transformation nodes (3) into the Data Buffers (4) follow by default a LWR based data distribution policy. In this case we decided to not consider the use of round-robin, because if all transformation nodes become synchronized, the data buffers would overload, and another data buffer would need to be added (scale) without being needed.

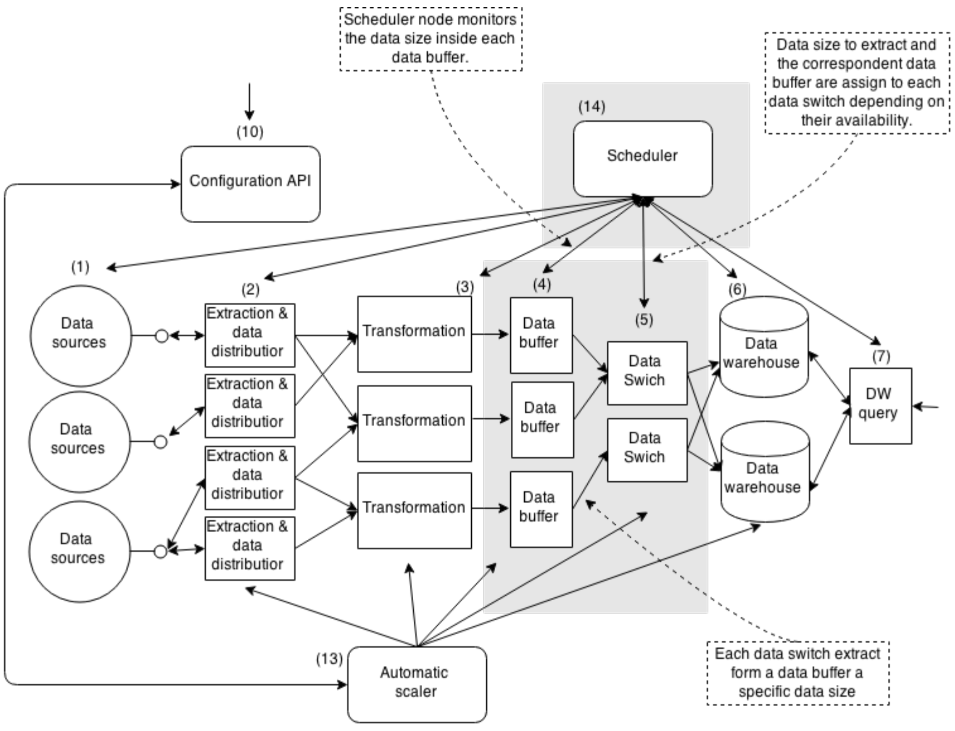

Scheduler node monitors the data size inside each data buffer.
Data size to extract and the correspondent data buffer are assign to each data switch depending on their availability.
(10)
Configuration API
(14)
Scheduler
(1)
(2)
(3)
(4)
(5)
(6)
(7)
Data sources
Extraction & data distributior
Transformation
Data buffer
Data Swich
Data warehouse
DW query
(13)
Automatic scaler
Each data switch extract form a data buffer a specific data size

Figure 5.12: Loaders extraction from the Data Buffers

Other possible distribution policies are: manual data distribution, where the developer specifies, for each Transformation node (3), the corresponding data buffer (4) node.

5.3.4 Data switch extraction from the data buffers

Figure 5.12, highlights the "Data Switch", "Data Buffers" nodes and the "Scheduler". Data extraction from the "Buffer nodes" (4) is done using a scheduler based approach managed by the "Scheduler" (14). This approach is similar to the one used during data extraction from the data sources, (1) to (2).

When the data-warehouse load instant starts, the Scheduler (14) asks to all data buffer nodes (4) the storage data size for each stored transformed data. Following the same extraction policy as in the extraction nodes, (1) to (2), the Data Switches (5) are set to extract data from the Data Buffers

(4) using the configured maximum data chunk size.
Other possible extraction policy to extract data from (4) to (5) is manual data extraction. To each Data Buffer (4), one or more data Data Switch (5) nodes can be assigned to extract data.

5.3.5 Load data into the data-warehouse nodes

The developer configures replication and partitioning parameters for each data-warehouse table (defined in the data-warehouse schema script). Data switches (5) extract chunks of data from specific stored logs of transformed data. Data is then distributed or replicated, acording with the developer configurations for each data-warehouse table) across the data-warehouse nodes (6) for loading (using configured size batch files) during defined periods.

Listing 5.7: Data Switch replication configuration

```
dataSwitchSetSchemaReplication(
"nation",          //table name
true);             //true or false for replication
```

Listing 5.7 shows the schema replication configuration command line. The developer must specify which tables, present in the data-warehouse schema, should be replicated. Data that is not replicated is distributed by the data-warehouse nodes using (by default) a round-robin algorithm.

Manual data distribution can also be applied to the data-warehouse nodes, the developer can select manually which tables go to each data-warehouse nodes.

Listing 5.8: Data Switch manual load policy configuration by table referencing

```
dataSwitchSetManualLoadPolicyByTable(
"nation",                 \\table name
"DWnode1, DWnode2");      \\destination data-warehouse nodes ID
```

Listing 5.8 shows the data load policy configuration when sending data into a specific data-warehouse nodes.

Chapter 6

Total freshness

In this chapter we concentrate on data freshness to allow querying the most recent data, which is not yet integrated in the data-warehouse and we also concentrate on freshness scalability.
Section 6.1 describes the mechanisms used for total freshness. Section 6.2 describes the algorithms used to scale-in and -out each part of the ETL process. Section 6.3 describes how to achieve 24/7 availability.

6.1 Mechanisms to achieve total freshness, D-DW

Fast data integration for fresh results require additional storage mechanisms.

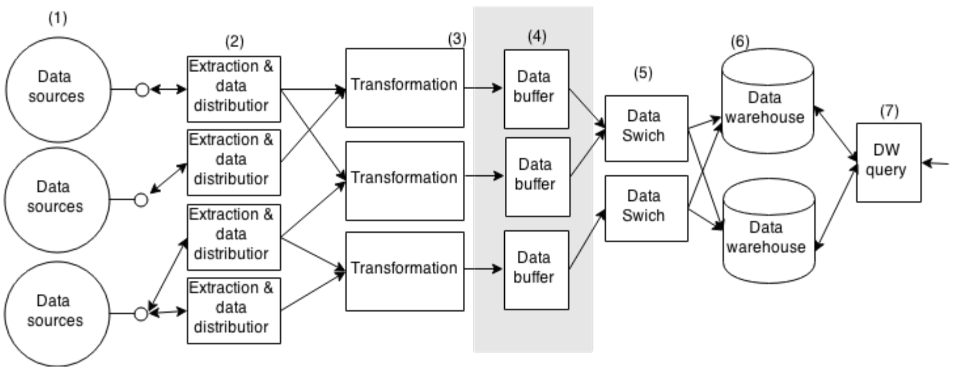


Figure 6.1: AScale without fast integration of fresh results

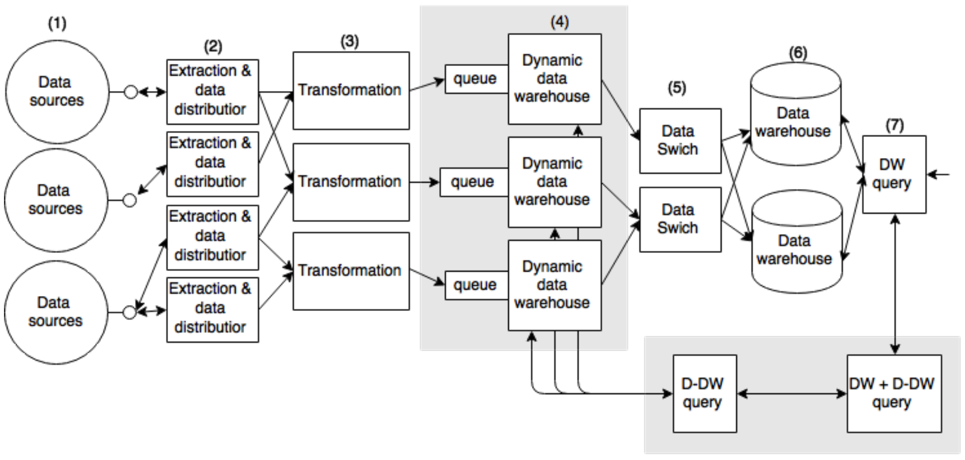


Figure 6.2: Dynamic-data-warehouse for fast integration and fresh results

Considering the AScale model in Figure 6.1, it is not possible to query (7) data that was just transformed (3) and stored in (4) until the next load instant that integrates new data into the data-warehouse nodes (6) (e.g. every night). This way, query results cannot include the most recent data, from both the data-warehouse (6) and the data buffer nodes (6), and it is not possible to query the most recent data (e.g. the data of a day or hour). The proposed solution is to transform the Data Buffers (4) into a fast storage system, preferentially in-memory, that is scalable for performance and at the same time allows to query the data that was not yet loaded into the main data-warehouse.

In Figure 6.2 we show the necessary architectural changes to support fast data integration and fresh results (highlighted with gray color).
A new module in column (4) is introduced. After the "Data Buffers", we add a "Dynamic-data-warehouse" (D-DW), which loads the data from a buffer. Note that, for performance monitoring, each "Dynamic-data-warehouse" (D-DW) requires a "queue" to be associated with it.
The "Dynamic-data-warehouse" is a fast storage mechanism, preferably in-memory, dedicated to hold only the most recent data that has not been integrated into the data-warehouse yet. Because the "Dynamic-data-warehouse"

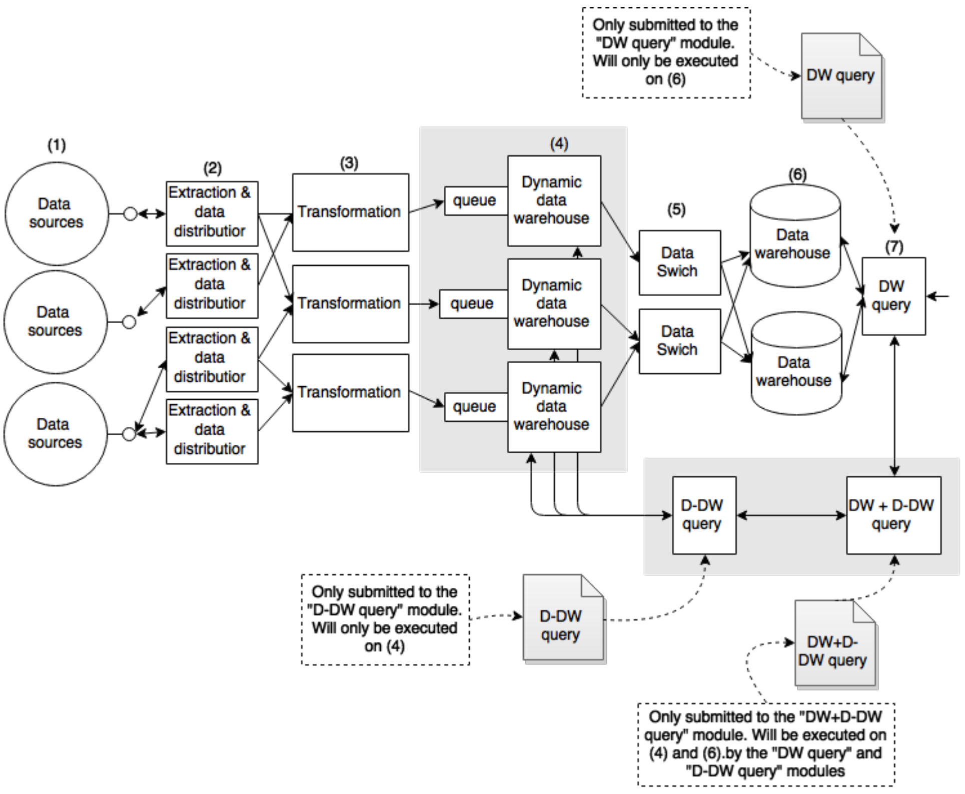
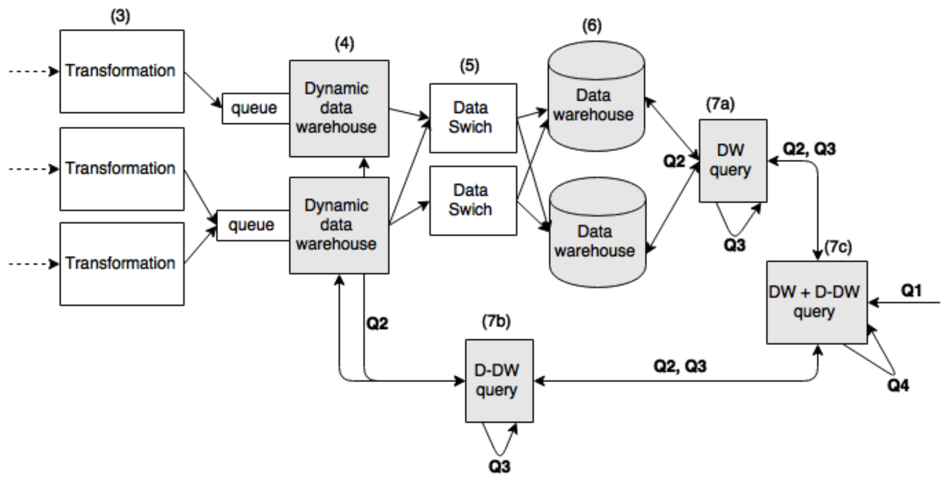

Only submitted to the "DW query" module. Will only be executed on (6)
DW query
(1)
Data sources
Data sources
Data sources
(2)
Extraction & data distributior
Extraction & data distributior
Extraction & data distributior
Extraction & data distributior
(3)
Transformation
Transformation
Transformation
queue
queue
queue
(4)
Dynamic data warehouse
Dynamic data warehouse
Dynamic data warehouse
(5)
Data Swich
Data Swich
(6)
Data warehouse
Data warehouse
(7)
DW query
D-DW query
DW + D-DW query
Only submitted to the "D-DW query" module. Will only be executed on (4)
D-DW query
DW+D-DW query
Only submitted to the "DW+D-DW query" module. Will only be executed on (4) and (6).by the "DW query" and "D-DW query" modules

Figure 6.3: Query submission

will hold small amounts of data and the storage mechanisms used are fast, indexes and/or materialized views are removed or reduced to a minimum.

Figure 6.3 shows how queries are submitted to each module. Query execution depends on the type of desired results. If results do not require the most recent information, then queries are executed only against the "data-warehouse" (6), and final results are merged. If results require only the most recent information, then queries are executed only against the "Dynamic-data-warehouse" (4), and final results are merged. When query results need past data and the most recent on as well, then queries are ran against the "DW + D-DW", which executes the query in both "data-warehouse nodes" and "dynamic-data-warehouse" nodes, merging the results from both. This way, results will include not only "data-warehouse" information, but also the most recently transformed data that has not yet been integrated into



Q1

```
SELECT Name, AccountID, AVG(balance)
FROM clients, account, movements
WHERE clientID.clients = clientID.account
AND accountID.account = accountID.movements
GROUP BY Name, AccountID
ORDER BY Name
```

Q2

```
SELECT Name, AccountID, sum(balance) as s, count(*) as c
FROM clients, account, movements
WHERE clientID.clients = clientID.account
AND accountID.account = accountID.movements
GROUP BY Name, AccountID
```

Q3

```
SELECT Name, AccountID, sum(s) as s, sum(c) as c
FROM temp
GROUP BY Name, AccountID
```

Q4

```
SELECT Name, AccountID, ( sum(s) / sum(c) ) as average
FROM temp
GROUP BY Name, AccountID
ORDER BY Name
```

Figure 6.4: Executing a query for freshness

the main "data-warehouse" nodes.

Figure 6.4, shows the distributed data-warehouse modules (6) and the, dynamic-data-warehouse (4). Queries Q1 to Q4 represent an example of query decomposition to execute against the distributed schema. Q1 is the query we want to execute, this query is being decomposed into: Q2 to execute against the DW (7a) and D-DW (7b); Q3 to merge results from each data-warehouse; Q4 to merge the merged results.

First Q1, is submitted to the module "DW+D-DW queries" (7c). This node decomposes the query and submits Q2 and Q3 to DW (7a) and D-DW (7b). Each data-warehouse will execute Q2 and return the results from each respective DW (7a) and D-DW (7b) to be merged (i.e. execute Q3). The DW (7a) and D-DW (7b) return the merged results to the DW+D-DW

module (7c) for the final result merging (i.e. to execute Q4).
For instance, if instead of an average we want a sum, the merge operation will consist of the sum of sums.

6.2 Scaling out/in

In this section we describe how the new architecture for freshness detects that it needs to scale-out/in. If the dynamic-data-warehouse is not able to integrate the most recent data efficiently, then a scale-out is required.

There are two cases that may require the dynamic-data-warehouse to scale-out. The scale-out based on the queue size is performed as shown in Figure 6.5, and the scale-out based on query execution time, as show in Figure 6.6.
First, if the queues from where the dynamic-data-warehous loads data increase their size above a certain configured threshold, this indicates that the dynamic-data-warehouse is unable to load and integrate data fast enough and more nodes are required. Second, if the queries submitted to the dynamic-data-warehouse cannot execute within the desired maximum execution time, more nodes are necessary, therefore data is split further into more nodes.

To save resources, scale-in mechanisms are also in place. To automatically scale-in the dynamic-data-warehouse, both data "queue" sizes and queries execution time must be considered, Figure 6.7. The system scales-in if at least two nodes have less than the configured minimum size and the average query execution time is half of the configured minimum time. When those conditions are met, one node is removed, data is partitioned across other available nodes, and the node is set on stand-by, as a ready node.

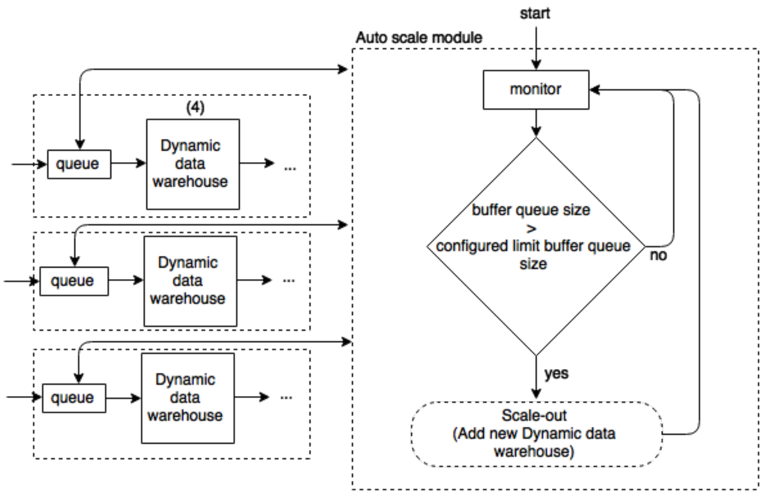

start
Auto scale module
monitor
(4)
queue
Dynamic data warehouse
...
buffer queue size > configured limit buffer queue size
no
yes
Scale-out (Add new Dynamic data warehouse)

Figure 6.5: Scale-out based on the queue size

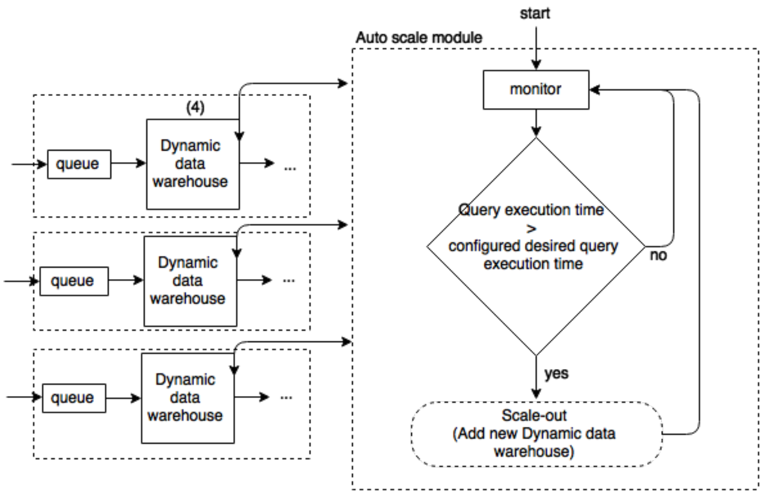

start
Auto scale module
monitor
(4)
queue
Dynamic data warehouse
...
Query execution time > configured desired query execution time
no
yes
Scale-out (Add new Dynamic data warehouse)

Figure 6.6: Scale-out based on query execution time

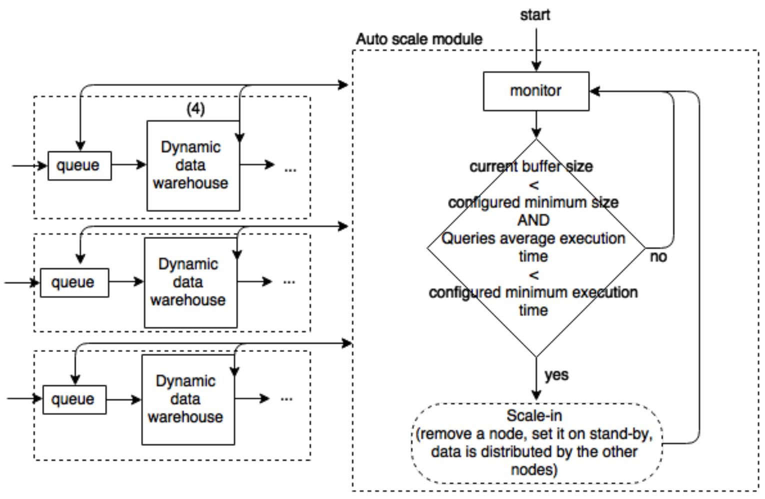

start
Auto scale module
monitor
(4)
queue
Dynamic data warehouse
...
current buffer size < configured minimum size AND Queries average execution time < configured minimum execution time
no
yes
Scale-in (remove a node, set it on stand-by, data is distributed by the other nodes)

Figure 6.7: Scale-in based on queue size and query execution time

6.3 Mechanisms to achieve 24/7 availability

When full query answer availability is required for 24/7 operation, new mechanisms need to be introduced to the "data-warehouse" nodes (6), "loader(s)" (5) and to the "query" execution (7) (Figure 6.8).

Figure 6.8 shows the process used to guarantee 24/7 availability. The data-warehouse nodes (6a) are all replicated (6b). When one data-warehouse is being used to load data, the other is used to answer queries.
When using a 24/7 availability mechanism, scalability is also affected. In this situation, when scaling-out the data-warehouse (6a) the data-warehouse clone (6b) must scale at the same time and in the same way. Previous monitoring algorithms to scale-out the data-warehouse also apply to the 24/7 model. Scale-in implies to scale-in both data-warehouses in the exact same way, so that one data-warehouse is always the replica of the other.

Figure 6.9 shows how data is loaded and replicated into the data-warehouse nodes. In the dynamic-data-warehouse (4), data that is being loaded (D1) is separated from new arriving data (D2). Once D1 is loaded into (6a) then

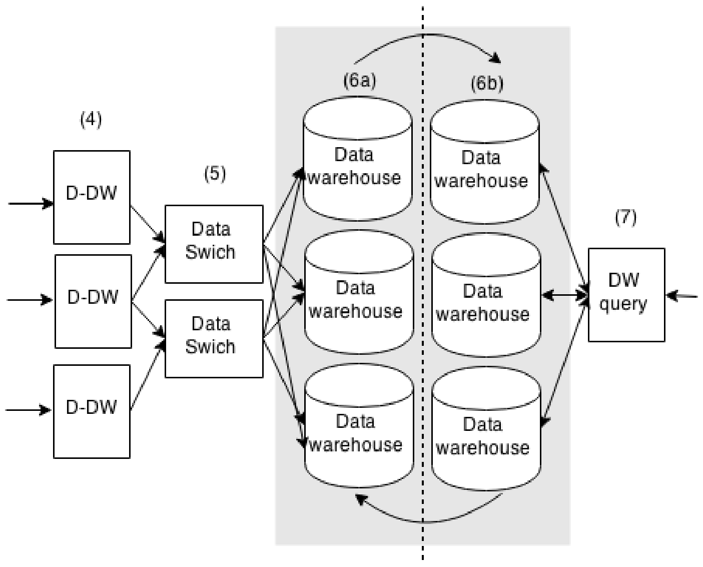


Figure 6.8: Support for 24/7 query answering

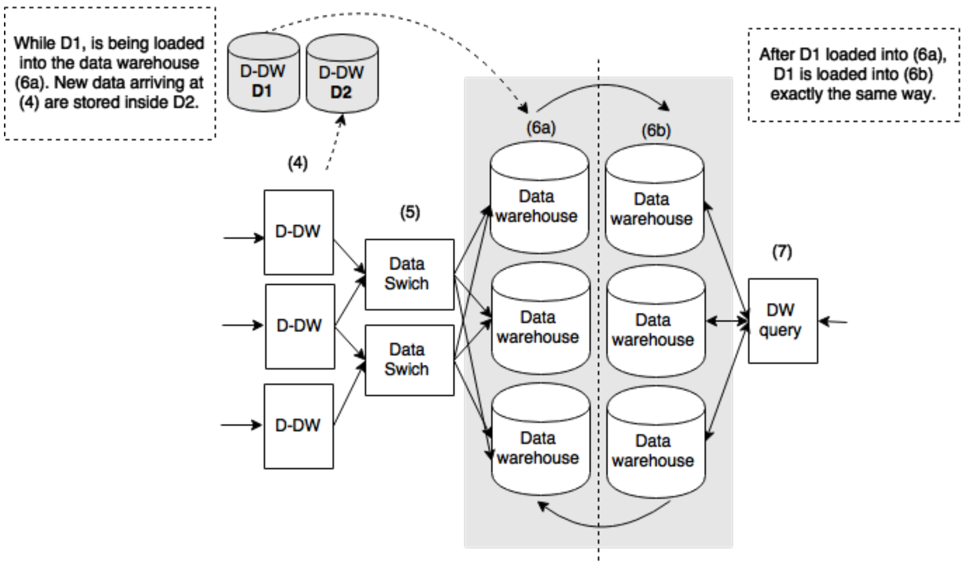

While D1, is being loaded into the data warehouse (6a). New data arriving at (4) are stored inside D2.
D-DW D1
D-DW D2
After D1 loaded into (6a), D1 is loaded into (6b) exactly the same way.
(4)
D-DW
D-DW
D-DW
(5)
Data Swich
Data Swich
(6a)
(6b)
Data warehouse
Data warehouse
Data warehouse
Data warehouse
Data warehouse
Data warehouse
(7)
DW query

Figure 6.9: Loading data for 24/7 availability

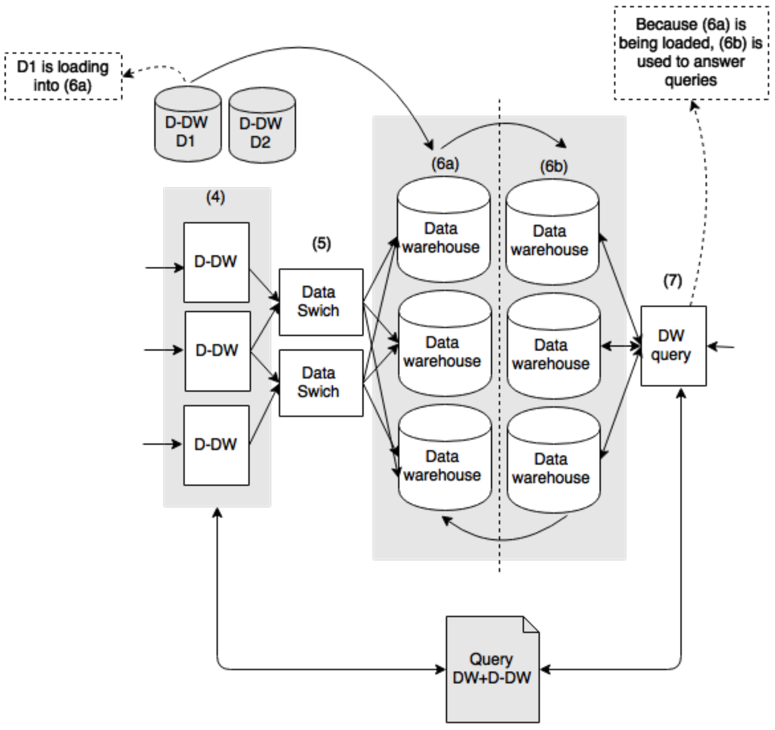


Figure 6.10: Query execution for 24/7 availability, loading (6a)

the same process is repeated for (6b). Query execution is always performed in the data-warehouse (6a) or (6b) that is not being used to load data.

When loading data and performing queries at the same time, synchronization issues regarding data replication in query results must be accounted for. First case, Figure 6.10, we show when both data-warehouses are not updated with the most recent data. Once (6a) starts loading D1, queries run only against the data inside data-warehouse (6b). Then, to incorporate the most recent data, queries must also execute against the D1 and D2 dynamic-data-warehouse data.

Second case, Figure 6.11 shows the case when (6a) is updated with D1 but not (6b). For this case, queries perform against the data-warehouse (6a), already updated with D1, and on top of of the dynamic-data-warehouse D2 (4) to include the most recent data.

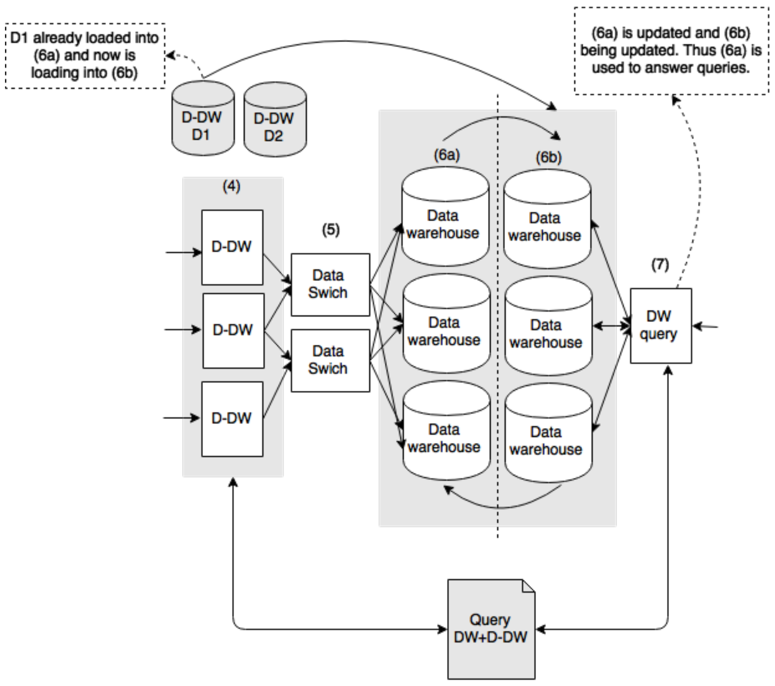


Figure 6.11: Query execution for 24/7 availability, loading (6b)

Chapter 7

High-rate scalability over continuous results

Complex Event Processing (CEP) or stream processing is a different model from traditional database processing. In stream processing, queries are registered and run continuously over the most recent window of data, analyzing it and producing alerts and other periodic results.

CEP systems can also require access to additional information that is present in the data-warehouse (e.g. data analysis of telecommunications call detail records; ATM card fraud detection; stock market). Some mixed queries may need to query big data-warehouses, which need to be optimized for near-real-time answering as well, raising performance and scalability problems.

We investigate the inclusion of a CEP engine in the proposed architecture, providing support for efficient continuous queries, load-(re)balancing, admission control and automated elasticity.

The proposed approach (re)schedules queries that are running on overloaded nodes. If this is not enough, and there are stand-by ready nodes, it adds them automatically and (re)schedules execution. Otherwise, it looks at data shedding specifications to try to accommodate the data rate iff the user agrees to some specified level of shedding. Finally, it alerts administrators

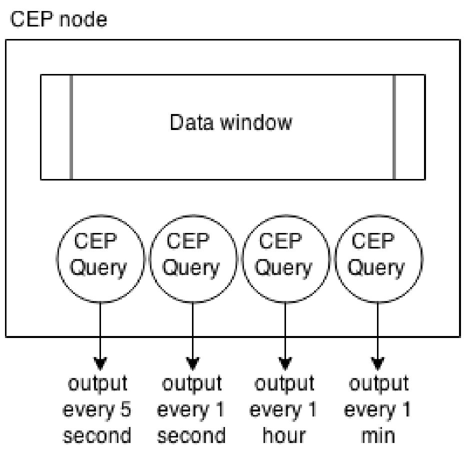


Figure 7.1: CEP node

if no automated adaptation is possible.

Section 7.1 overviews CEP and its use. Section 7.2, describes the different types of queries which our framework supports. Section 7.3, describes the proposed architecture to support continuous results processing. Section 7.4, introduces methods to detect overload situations. Section 7.5, explains how queries are distributed and relocated when overload situations are detected. Section 7.6, discusses how we apply load shedding to data and queries. Section 7.7, explains in more detail the algorithms to distribute data and queries.

7.1 CEP Overview

Complex event processing systems can process high-rate ingress data efficiently.

Typically CEP systems do not store data. They rely on a window of data to keep results updated. Some systems register hundreds of queries which periodically output results (e.g. every 15 seconds or n events). Queries are registered and de-registered dynamically as they are needed.

Figure 7.1 shows the basic structure of a CEP node. There is a window (e.g. size 1 hour or 1 minute) which holds the data temporarily. Query results are updated based on the data window and can have different outputs (e.g. every second, every minute) and processing requirements (e.g. memory, storage).

7.2 Query types

Queries are registered statements outputting results periodically, but may simultaneously require heavy processing over the data-warehouse (we call those stream-DB queries), or other heavy duty operations.
We consider two base query types:

- In-memory queries are queries that do not required any IO from disk, and use mainly the CPU;
- Stream-DB queries are queries that require access to the data-warehouse for querying, inserting, updating or deleting data;
- "Killer" queries, are stream-DB queries that make the CEP node become unresponsive. Consider for instance a CEP query that must output results every minute but which accesses the data-warehouse using a query that takes 5 minutes to run. Since every minute a new 5 minutes database query is launched, this will result in the node becoming unresponsive.

Queries are submitted to a scheduler, Figure 7.2 (7d), which balances the query workload, by assigning submitted queries to nodes according to a load balancing algorithm, and by preventing overload conditions. This balancing is preferably based on a least-weight algorithm, which places the new query in the node that is currently less loaded, as assessed by monitoring performance variables (e.g. CPU, memory and disk utilization).

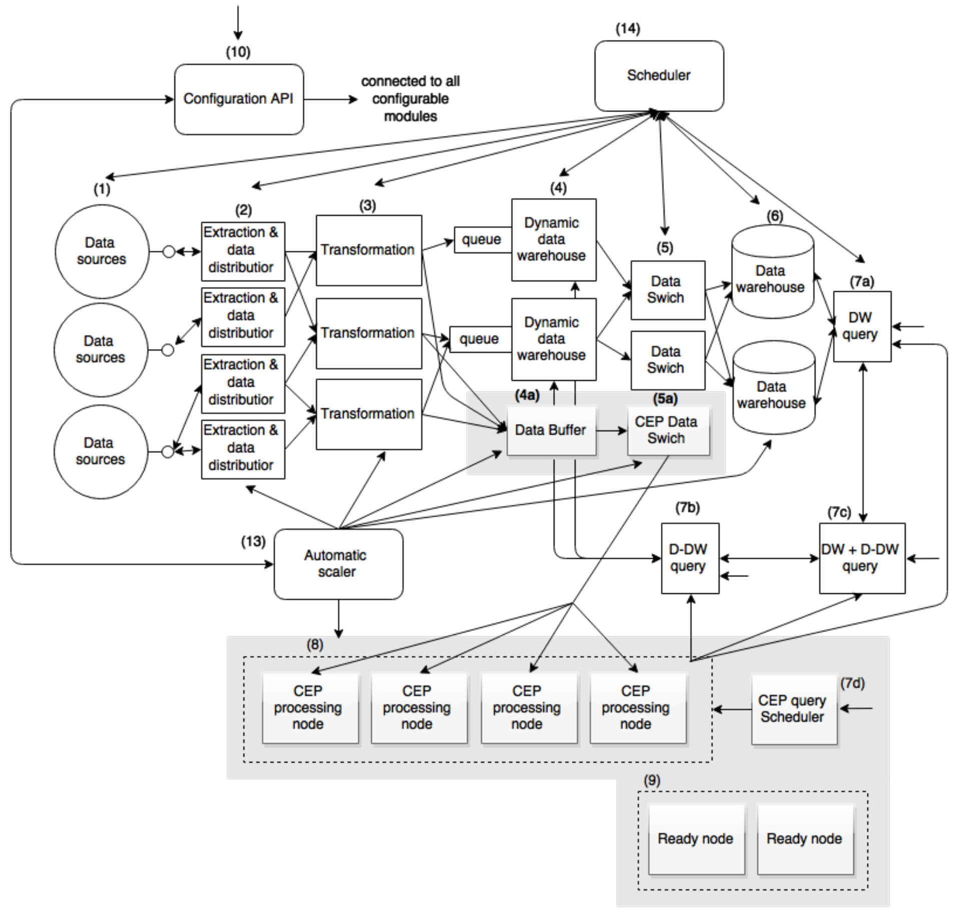


Figure 7.2: Architecture for continuous queries results

7.3 Continuous results architecture

In this section we describe the modifications of the auto-scale framework to support CEP and stream-DB scalable processing.
Figure 7.2 shows the architecture for continuous results. The architecture is designed to process efficiently streams of data and queries as well as stream-DB workloads, with any underlying hardware and stream-processing software.

The architecture shown in Figure 7.2, is a generic parallel processing architecture that handles overloads even with very dynamic workloads by adding processing nodes and by (re)scheduling queries for load balancing and load shedding. The new system components, highlighted in Figure in 7.2 are:

- "CEP data switch" (5a), this node requires a "Data Buffer" (4a) to hold data coming from the transformation nodes (3). The "CEP Data Switch" (5a) replicates the incoming data by all processing nodes.
- Scheduler node "CEP query scheduler" (7d), for distributing and (re)scheduling queries by the nodes according to their load. The load assessment is based on either round-robin, least-work-remaining based on the number of queries or least-weight based on collecting and analyzing memory, CPU, disk I/O and the queues sizes of all processing nodes.
- "CEP processing nodes" (8), that run queries submitted by (7d). Each node contains a data P/C queue for overload detection. When a node queue reaches a configurable limit size, it is assumed that the node is getting overloaded and queries should be relocated. Then the system tries to (re)balance the queries.
- Ready-Nodes (9), represent one or more nodes that are in standby mode, ready to accommodate query relocation. Those nodes are free and without any processing load. When a new node is commissioned from the pool of ready nodes and a query is registered into it, the query is also left running in the node that was getting overloaded until results are produced in the new node (fill window). Using this strategy, scalability can be obtained without loss of results.
- Nodes that are lightly loaded can automatically become ready nodes.
- The data-warehouse (6) and the Dynamic-data-warehouse (4) are also accessed by the stream-DB queries to compute mixed CEP-DB query results.

7.4 Overload detection and provisioning

Figure 7.2 shows the CEP data switch (5a) that replicates all data by all nodes. Each node (8) has a certain number of CEP queries registered and running (load), while the ready-nodes are free to be added when overload is detected.
Overload detection is based on a data queue at place in each node. If the queue reaches a limit configurable size, the system is getting overloaded, and additional measures must be taken.
Each node queue is monitored by the auto scaler (13). When it reaches a limit size, configurable by the admin, the node sends to the scheduler the last submitted query for a new registration (this is an attempt to relocate the query to another node), to attempt to free resources. Note that the query is not immediately unregistered from the initial node. Instead, the scheduler only un-registers it after the new node is already providing results.
Another mechanism is triggered when the queue reaches a maximum size. This second mechanism tries to apply load shedding and alerts the administrator if it cannot solve the problem.
Elasticity and scalability is achieved by adding new nodes to the set of ready-nodes (9). When new resources are necessary, they can be fetched and set as processing-nodes (8).

7.5 Queries distribution and relocation

Query assignment decisions are based on a load balancing algorithm. The simplest load balancing algorithm is round-robin (RR), whereby the scheduler assigns queries to nodes in a round-robin fashion, without looking at the load of each node. The advantage of this alternative is that no load information is required.
An improvement over Round-Robin is Least-Work based on the number

of queries running (LWRn). This algorithm requires knowledge about the number of queries running at each node, and chooses the node with fewer queries at the assignment instant.
Finally, the Least-Weight (LW) algorithm needs to measure current load in terms of parameters such as CPU, memory and IO in order to determine the less loaded node, then it assigns the query to the less-loaded node.

When a new query arrives at the "CEP query scheduler", Figure 7.2 (7d), it is set running into the node (8) with less load (if we assume least-weight balancing). If the queue of the processing node increases and reaches a limit size, then the query is removed from it, and put to run in the ready-node. The ready-node becomes a processing-node. If the queue of the ready-node reaches a limit size as well, then two actions can be taken: If allowed, data load-shedding mechanisms are applied; otherwise the query is not admitted, since it is assumed that it is a "killer" query.
Another situation is when queries need to be relocated because the input data rate increases so much that many P/C queues become overloaded. This condition is detected by observing a rise in the input data rates, and if more than a predefined fraction of P/C queues become overloaded in a predefined interval of time.
Following actions are taken (to last inserted queries in node, by order of arrival):

1. The query is sent back to the "CEP queries scheduler" (7d) to be relocated into another nodes according to balancing algorithm;

2. If no overload is detected in the new node, then, as soon as results start being provided by the new node, the query is removed from the overloaded node;

3. If overload is detected in the new node, activate a ready-node if avail-

able;

4. If overload is detected in the second relocation, then stop the query in the processing node and keep it running in the ready-node. This overload is no longer damaging the other nodes, since the query is running only in the ready node;

5. Since it is still causing overload, a load shedding or admin alert mechanism deals with it.

7.6 Load shedding and admin alerts

Load shedding is a mechanism that selects data to be discarded [78], to relieve processing load. In this case results are an approximation.
Every time a P/C queue reaches the maximum size (configurable parameter), queries removal or load shedding decisions need to be made. Based on the limit of load shedding specified upon the submission of each query, the node will start discarding data gradually (load-shedding) as long as it still complies with all shedding limits for all queries in that node.
If this action does not result, then the node will choose randomly a query to be removed, from a set of removable queries (queries have a parameter telling whether they can be removed). Every time a query is removed, then the mechanism of load shedding is reset to allow the system to re-evaluate and (re)balance itself.
If all the previous options are exhausted and the system is still overloaded, it will alert the administrator, indicating the node and queries in overload condition. The administrator can decide to add more ready-nodes, remove more queries, change something in the data rate, or alter queries parameters.
Two additional clauses are added to queries:

`ALLOW LOAD SHEDDING` <value%>

`CAN BE DROPPED` <true / false>

The first clause stipulates the percentage of load shedding allowed for the query, the default being no load shedding. The second clause indicates whether the query can be removed from the system.

7.7 Overloading handling algorithm

In this section we describe how the most relevant parts of the overload-handling algorithm that make the continuous results processing work.

7.7.1 CEP queries scheduler

Figure 7.3 describes how the (re)scheduling algorithm works.

- If the query was (re)scheduled zero times (meaning it is a new query), then the scheduler will find the best node to register it, and the "number of scheduled times" is set to 2 (2 because the query was placed in the best fitting node, if it becomes overloaded it will go directly into a ready-node);
- ... Then the "number of scheduled times" of the previous last registered query is set to one. Because the "CEP query scheduler" already chooses the best node to register the query (i.e. the one with least load) and it did not become overloaded;
- Now, if the auto-scale (13), Figure 7.2, detects an overload situation in a CEP node, it sends the last submitted query in that node (by order of registration) to the scheduler to be resubmitted into a better node, and the parameter "number of scheduled times" is increased to 2 (it was set to one after a new query was inserted, or already to 2);

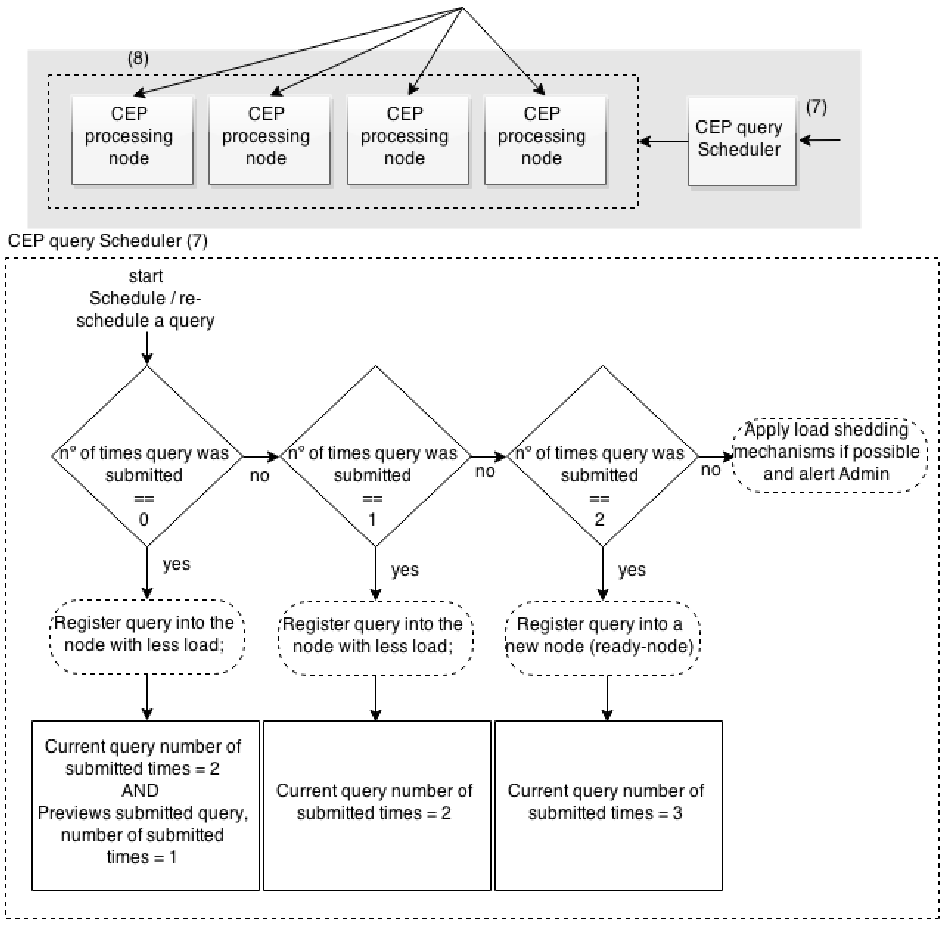

(8)
CEP processing node
CEP processing node
CEP processing node
CEP processing node
CEP query Scheduler
(7)
CEP query Scheduler (7)
start
Schedule / re-schedule a query
n° of times query was submitted == 0
no
n° of times query was submitted == 1
no
n° of times query was submitted == 2
no
Apply load shedding mechanisms if possible and alert Admin
yes
yes
yes
Register query into the node with less load;
Register query into the node with less load;
Register query into a new node (ready-node)
Current query number of submitted times = 2 AND Previews submitted query, number of submitted times = 1
Current query number of submitted times = 2
Current query number of submitted times = 3

Figure 7.3: CEP query scheduling

- At the third ("number of scheduled times" = 2) (re)schedule of a query, it is put into a ready-node if available. If the ready node is not available or it gets overloaded, then the load shedding and admin alert algorithm will deal with the problem.

Every time a query is relocated because of an overload situation, that query is not removed from the overloaded node until the new selected node (the one with most available resources) is providing results from an equal data window. Upon results provided, the scheduler decides in which node to leave the query running. For that, the system scheduler analyses the throughput of both nodes and removes the query from the node with less throughput. This process avoids work loss due to relocation.

7.7.2 Load-Shedding and admin alert

Next we describe the algorithm for "disaster" handling, when overload is detected in many nodes (This condition is detected by observing a rise in the input data rates, when more than a predefined fraction of P/C queues become overloaded in a predefined interval of time), or it is detected in one node but query relocation was unable to solve the problem.
First, the node assesses what is the maximum load-shedding allowed (it corresponds to the minimum load shedding percentage allowed in the node). Based on that value, it will increase the amount of discarded data gradually. Data load shedding is performed by removing gradually x% of the input at equally spaced positions. If this does not solve the problem, and there is still too much load, then the node selects randomly a query that is marked as "can be dropped" (CAN BE DROPPED parameter = true) to be removed. Each time a query is removed, load shedding is set to 0%, so the system can assess again what to do. Figure 7.4.

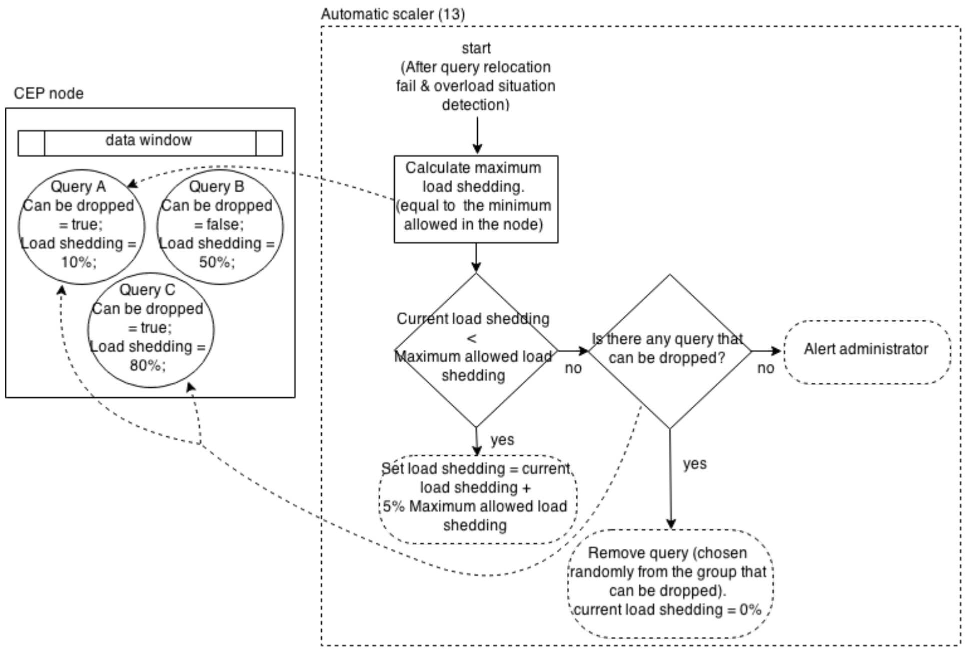

Automatic scaler (13)
CEP node
data window
Query A Can be dropped = true; Load shedding = 10%;
Query B Can be dropped = false; Load shedding = 50%;
Query C Can be dropped = true; Load shedding = 80%;
start (After query relocation fail & overload situation detection)
Calculate maximum load shedding. (equal to the minimum allowed in the node)
Current load shedding < Maximum allowed load shedding
no
Is there any query that can be dropped?
no
Alert administrator
yes
Set load shedding = current load shedding + 5% Maximum allowed load shedding
yes
Remove query (chosen randomly from the group that can be dropped). current load shedding = 0%

Figure 7.4: Load Shedding mechanism

7.7.3 Resource de-provisioning

When a node has a small number of queries and load bellow pre-specified bounds, the following resource-de-provisioning algorithm is run: the node tries to free resources by submitting the queries to the scheduler. The scheduler resubmits the queries to other nodes. If the node becomes free, without any node getting overloaded, it is set on standby as a ready-node.

Chapter 8

Experimental evaluation

In this section we describe the experiments made to evaluate the proposed system, AScale.
Section 8.1 introduces the experimental setup. Section 8.2 explains the objective of each proposed test. Section 8.3 shows the performance limitation problems of a single server approach, by stressing it with heavy data and query workloads. Sections 8.4, 8.5, 8.6 and 8.7 test AScale in different scenarios, respectively considering offline ETL with limited hardware resources, offline ETL using extreme high-data-rates, near-real-time without and with data freshness and finally AScale with continuous queries (CEP). Section 8.8 shows conclusions from the experimental results.

8.1 Experimental setup

In this section we describe the testbed, hardware, software and ETL operations used in the setup. An experimental setup was built to simulate not only the data-warehouse, but also, all ETL processes. The decision of using TPC-H [79] data as source data logs, and SSB [80] as the data-warehouse schema and queries, was taken to reuse related work from the research group, for instance [53], which had already some parts of the framework pipeline de-

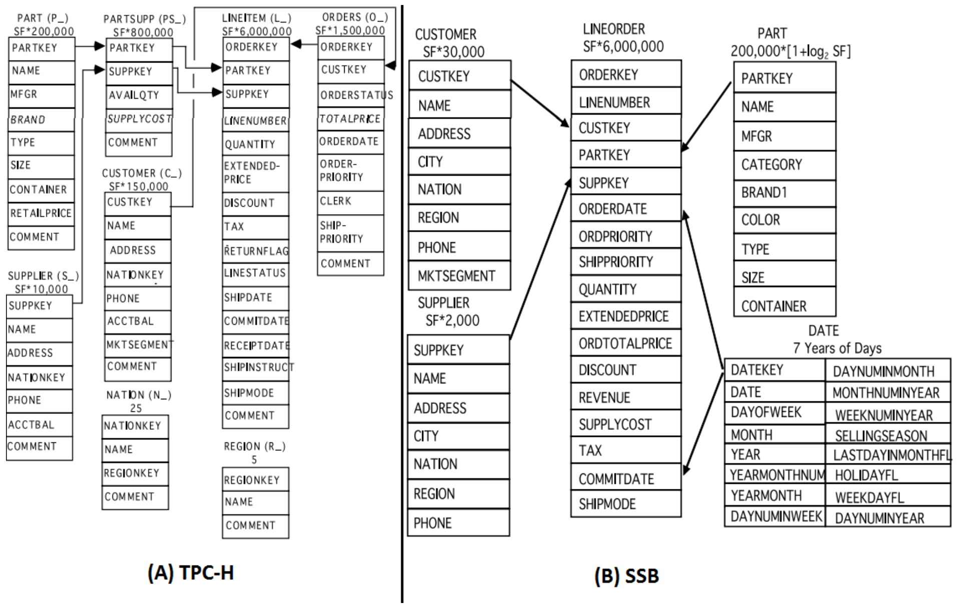


Figure 8.1: TPC-H and Star Schema Benchmark (extracted from [1])

veloped. This option also allowed us to better control data transformations and corresponding staging area volume and data synchronization, allowing us to build it with less complexity, thus easier to test the proposed AScale concepts.

Data sources logs for extraction: The structure of the simulated logs is the same as the TPC-H generated data logs structure (Figure 8.1 (A)), consisting of logs representing each of the tables: part, supplier, nation, region, partsupp. Regarding the tables "lineitem" and "orders", they were merged into a single log with the following structure: the log is a set of "order" rows and for each order the log contains the respective related lineitem rows as subsequent rows of the order.
Data extraction is made considering the start and end of each order (including the respective items), in order to keep data together and consistent.

Data transformation: After data is extracted from TPC-H log files, it

is set to be transformed. The TPC-H tables, part, supplier, nation, region, partsupp are also keep in the transformation nodes (staging area) using a Postgre SQL database. The stored data is transformed in order to recreate the SSB schema (Figure 8.1 (B)).
Table "Date" (Figure 8.1 (B)) was created with the SSB structure using a Java programmed generator.
Additional transformations were applied:

- Names were split and concatenated into last name and given names. The first letter of each name was set to upper-case and remaining letters were set to lower case;
- Addresses were cleaned and transformed by converting keywords (e.g. street) into abbreviated words, using a translation lookup table in memory. Moreover, the first letter of each name was set to upper-case and remaining letters to lower case. The postal code was added (concatenated), using a translation table in memory to correlate each city with a postal code;
- The phone numbers were converted into groups of three numbers, adding the country and city code, depending on the postal address code;
- Categories were converted/written into full text (no abbreviations);
- Dollar coin numbers were converted into Euros;
- Sizes and weights were converted into the normalized international system;

The data output from these transformation operations is then stored and ready to load.

Data warehouse: The data-warehouse has the same base structure as the SSB benchmark. Additional indexes and views are described in Appendix D.

8.1.1 Hardware & Software

The experimental tests were performed using 12 computers, denoted as nodes, with the following characteristics: Intel Core i5-5300U Processor (3M Cache, up to 3.40 GHz); Memory 16GB DDR3; Disk western digital 1TB 7500rpm; Ethernet connection 1Gbit/sec; Connection switch SMC SMCOST16, 48 Ethernet ports, 1Gbit/sec.
Software installed/used. The 12 nodes were formatted before the experimental evaluation and installed with: Windows 7 enterprise edition 64 bits; Java JDK 8; Netbeans 8.0.2 Oracle Database 11g Release 1 for Microsoft Windows (X64); MySQL 5.6.23 used in the dynamic-data-warehouses; PostgreSQL 9.4 used for lookups during the transformation process; Esper 5.1.0 for Java as CEP process engine; TPC-H benchmark data set; SSB benchmark, representing the data-warehouse schema and queries.
For this experimental evaluation, we assume that a node corresponds to a physical machine. However, due to limited available resources, some virtual nodes were created, and other nodes resources redirected and reused, in such way that AScale pipeline processing was not affected.

8.2 Objectives and organization

To evaluate AScale efficiency and limitations, we created fives scenarios.
The first scenario tests performance of a system without automatic scaling. As the data-rate and volume increases, the system eventually fails to execute parts of ETL+Q process within acceptable time.

The second scenario evaluates AScale for use in a simulation of a typical data-warehouse scenario, where the load process is slow and needs to scale. The third scenario tests a case with huge data rates and huge amounts of data in all parts of the pipeline. In this case automatic scalability is required over all the pipeline.

The fourth case evaluates AScale in a near-real-time scenario, with high-rate data and strict freshness requirements. These tests demonstrate AScale scaling-out and -in when ETL is configured to take only 3 seconds and incoming data-rate is increasing and decreasing. In this scenario we test scalability of the dynamic-data-warehouse, which is the part added to provide real-time.

A fifth scenario evaluates the system in a situation in which downtime should be minimized. This is the case, for instance, of a system that loads big amounts of new data offline, but the offline period must be minimized (e.g. less than 1 minute or 15 minutes).

A sixth scenario evaluates scalability of the continuous result processing (CEP) add-on that was proposed in Chapter 7.

8.3 Performance limitations without automatic scalability

In this Section we test both ETL and data-warehouse scalability needs when the entire ETL process is deployed without automatic scalability options. The system is stressed with increasing data-rates until it is unable to handle the ETL or query processing or both in reasonable time. Automatic scalability, which we evaluate in following sections, is designed to handle this problem.

The following deployment is considered: One machine to extract, transform data and store the data-warehouse; extraction frequency is set to perform

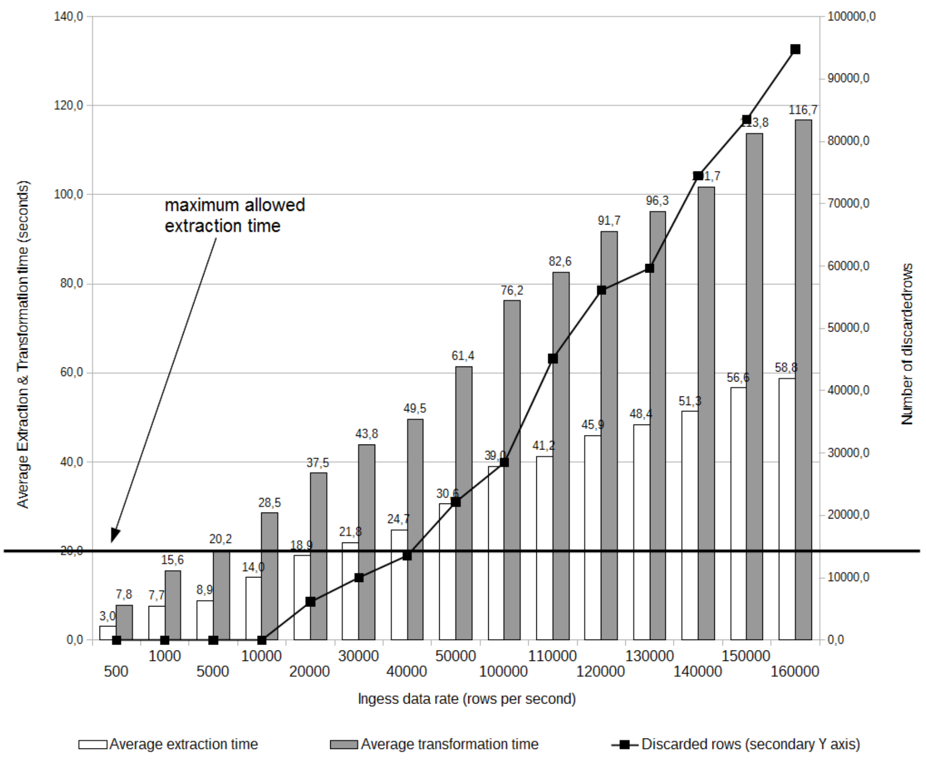


Figure 8.2: Extract and transform without automatic scalability

every 120 seconds; desired maximum allowed extraction time was set to 20 seconds and data load is performed in offline periods.
Based on this scenario, we show the limit situation in which performance degrades significantly, justifying the need to scale the ETL (i.e. parts of it) or/and the data-warehouse itself.

Extraction & transformation: Considering only extraction and transformation, using a single node, Figure 8.2 shows: the left Y axis represents average extraction and transformation time in seconds; the X axis shows data-rate in rows per-second; white bars represent extraction time; grey bars represent transformation time; the line represents the average number of discarded rows (corresponding values in the right axes). For this experi-

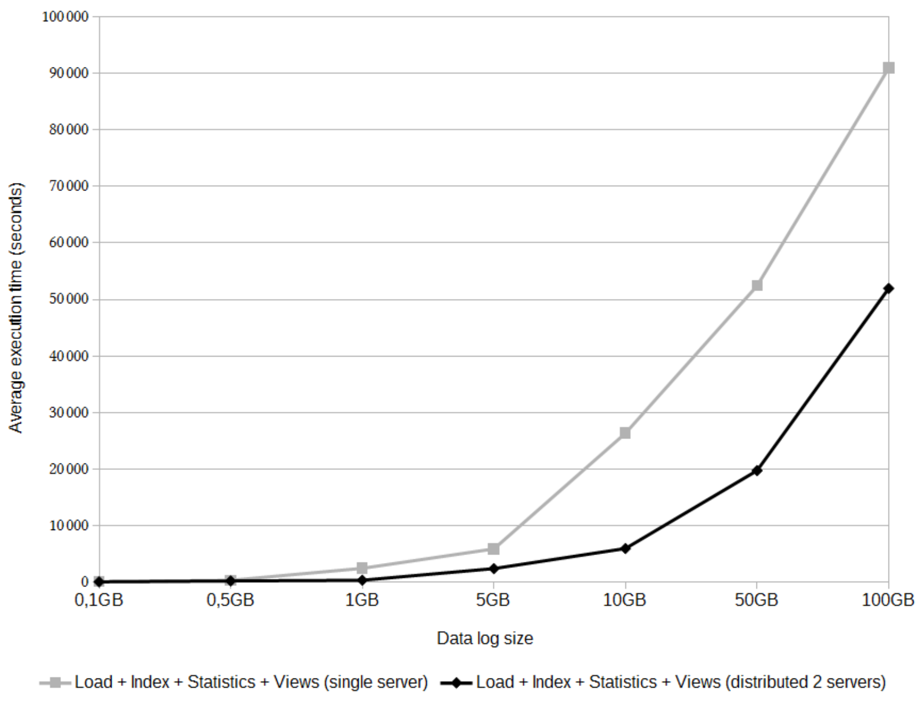


Figure 8.3: Loading data, one server vs two servers

ment we generated log data (data to be extracted) at a rate of λ per second. Increasing values of λ were tested and the results are shown in Figure 8.2. As can be seen in the figure, as the data-rate increases, a single node is unable to handle so much data. At a data-rate of 20.000 rows per second, buffer queues become full and data starts being discarded, because the extraction time is too slow. Additionally, the transformation process is slower than extraction. More resources would be needed for it to perform at the same speed as extraction.

Loading the data-warehouse: Figure 8.3 shows the load time as the size of the logs is increased. It also compares the time taken with single single node versus two nodes.

Figure 8.3 shows: the Y axis represents average load time in seconds; the X axis represents loaded data size in GB; the black line represents two servers;

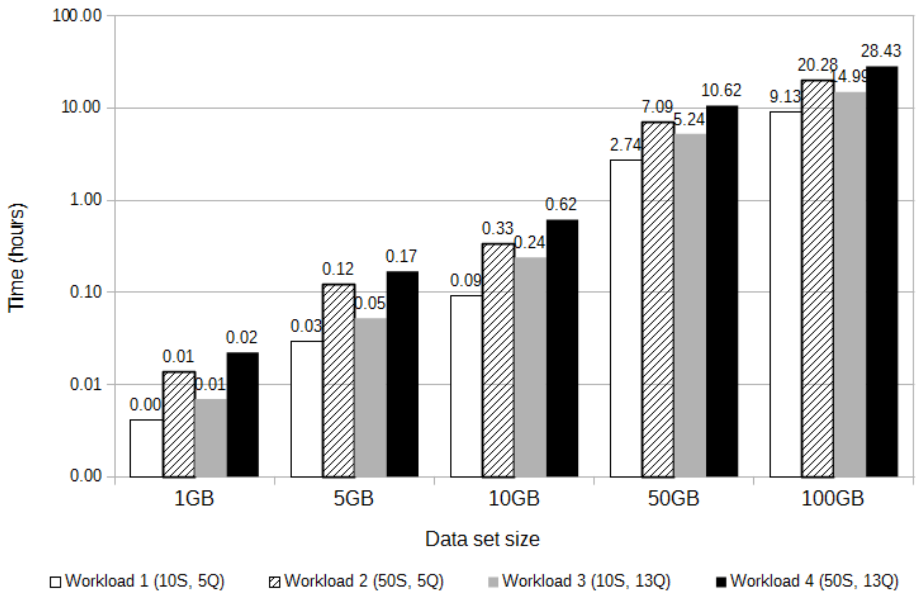


Figure 8.4: Average query time for different data sizes and number of sessions

the grey line represents one server; all times were obtained with the following load method: destroy all indexes and views, load data, create indexes, update statistics and update views; data was distributed by replicating and partitioning the tables.

From Figure 8.3, we can see that load times increase significantly with the load volume. Differences between the case with one node and the one with two nodes are specially noticeable when loading more than 5GB. When adding a second data-warehouse nodes, performance improves and the load time becomes almost less than half.

Query execution: Figure 8.4, shows the average query execution time for a set of tested workloads (using the SSB benchmark queries): workload 1 has, 10 sessions and 5 Queries (Q1.1, Q1.2, Q2.1, Q3.1, Q4.1); workload 2 has 50 sessions and 5 Queries (Q1.1, Q1.2, Q2.1, Q3.1, Q4.1); workload 3 has 10 sessions and 13 Queries (All); workload 4 has 50 sessions and 13

Queries (All); for all workloads, queries were executed in a random order; the desired maximum query execution time was set to 60 seconds.
The Y axis shows the average execution time in seconds. The X axis shows the data size in GB. Each bar represents the average execution time per query from each workload. Note that the Y axis scale is logarithmic.
Depending on the data size, number of queries and number of simultaneous sessions (e.g. number of simultaneous users), execution time can vary from a few seconds to a very significant number of hours or days, especially when considering large data sizes and simultaneous sessions or both. In these results, and referring to 10GB and 50GB, we see that an increase of 5x of the data size resulted in an increase of approximately 20x in response time. An increase in the number of queries of 5x resulted in an increase of approximately 2x in query response time.

Conclusions: In this section we stressed the ETL+Q pipeline limitations and scalability needs of a system without automated scalability features. By increasing the input data-rate, a single node could not handle the desired processing times and data could not be processed within the desired time window. Extraction, transformation, load and queries performance were tested by increasing the data set size (GB) and the number of simultaneous sessions. In all tested situations we noticed the need for scaling as the data set increased and query response times became larger.

8.4 Typical data-warehouse scenario

In this section we evaluate AScale in a scenario where, due to log sizes and limited resources, data load takes too long to perform if scalability is not applied.

We start with only two nodes (two physical machines), one for handling extraction and transformation, the other to hold the data-warehouse

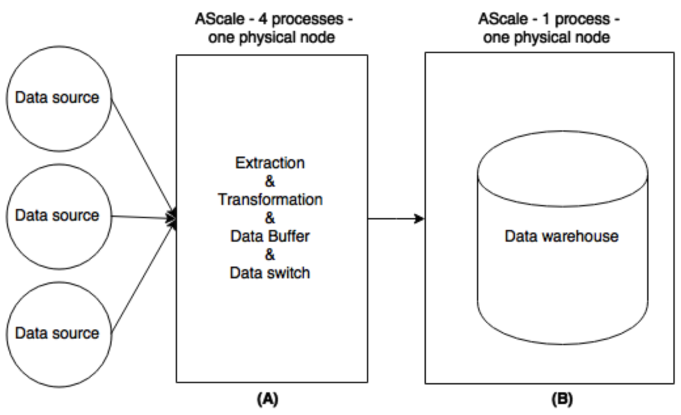

AScale - 4 processes - one physical node
AScale - 1 process - one physical node
Data source
Data source
Data source
Extraction & Transformation & Data Buffer & Data switch
Data warehouse
(A)
(B)

Figure 8.5: AScale for simple scenarios

as shown in Figure 8.5. AScale is setup to monitor the system and scale when needed.
Data is extracted from sources, transformed and loaded only during a predefined period (e.g. night), to be available for analysis the next day. The maximum extraction, transformation and load time, all together cannot take longer than 9 hours (e.g. from 0am until 9am). AScale was configured with an extraction frequency of every 24 hours and a maximum extraction duration of 4 hours, a transformation queue with a limit size of 10GB and data-warehouse loads were configured for every 24 hours, with a maximum duration of 9 hours.

Experimental results in Figure 8.6 show the total AScale ETL time using two nodes (two physical machines), one for extraction, transformation, data buffer and data switch (Figure 8.6 (A)), the other for the data-warehouse (Figure 8.6). Up to 10GB of log size, the ETL process can be handled within the desired time windows. However, when increasing to 50GB, 9 hours are no longer enough to perform the full ETL process. In this situation the data-warehouse load process (load, update indexes, update views) using one node (average load time 873 minutes) and two nodes (average load time 483 minutes) exceed the desired time window. When scaled to 3 nodes,

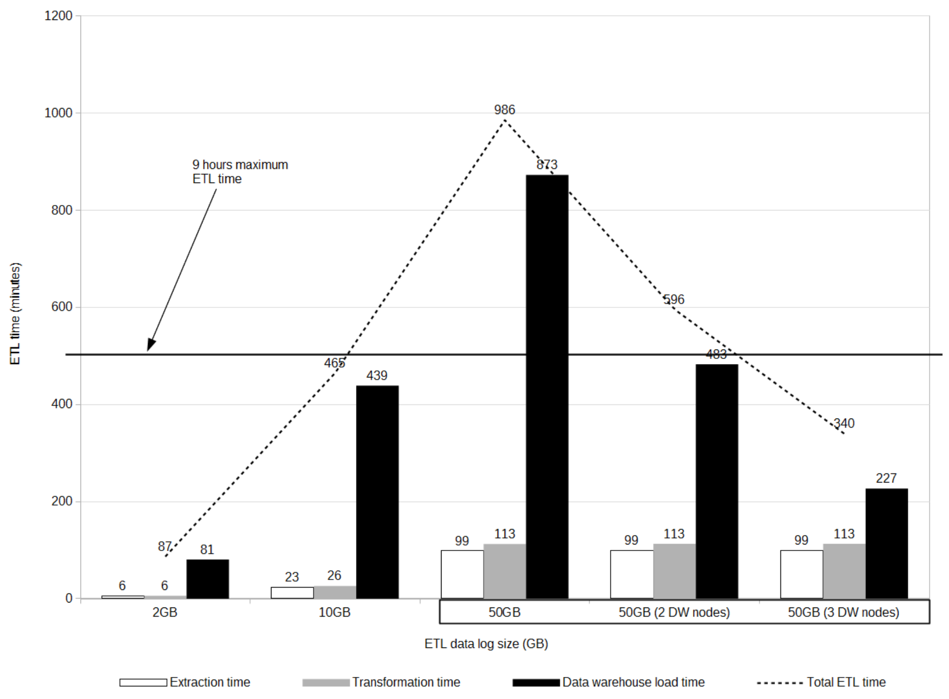


Figure 8.6: AScale, 9 hours limit for ETL

by adding another data-warehouse node (Figure 8.6 (B)), the ETL process returns to the desired time bound.

8.5 ETL with huge data rates and sizes

In this section we create an experimental setup to stress AScale under extreme data-rate conditions. The objective is to test scaling each part of the pipeline.

For this experiment we did the following configuration: E (extraction) was set to perform every 60 minutes with 30 minutes maximum extraction time, T (transformation) queue maximum size was configured to 500MB, and L (load) frequency to every 24 hours with a maximum duration of 5 hours.

Extraction: Figure 8.7 shows the AScale extraction process when us-

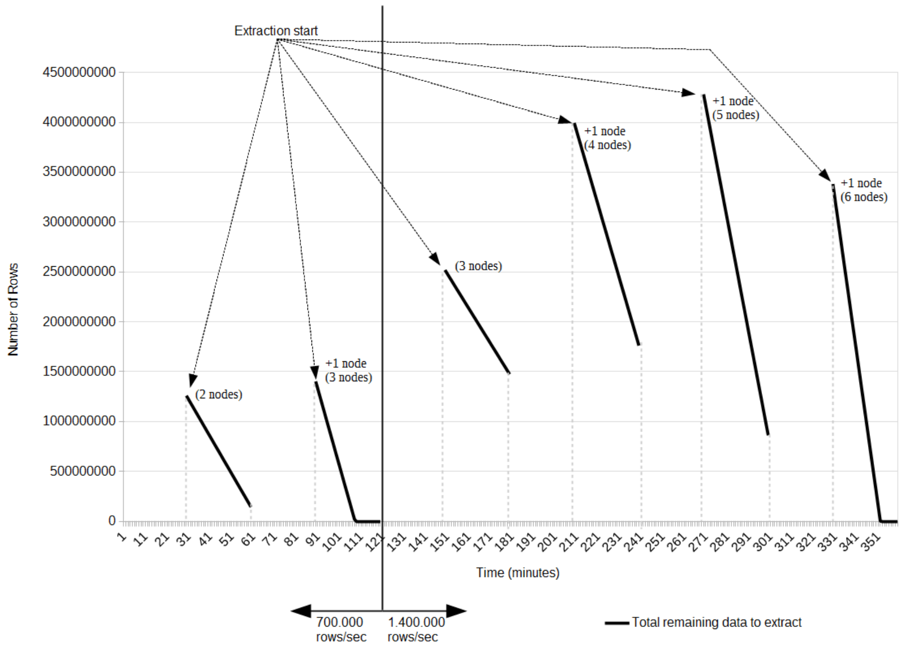


Figure 8.7: Extraction (60 minutes frequency and 30 minutes maximum extraction time)

ing an extraction frequency of 60 seconds and 30 seconds for the maximum extraction time. The figure is divided into two sections: first we use a data rate of 700.000 rows/sec and scale the extraction until all rows are extracted successfully; second we increase the data rate to 1.400.000 rows/sec and automatically scale until all rows are extracted within the configured time bound.

In Figure 8.7 we show: in the left Y axis, the number of rows to extract; in the X axis, the time in minutes; black line represents the total number or rows left to be extracted at each extraction period; and with labels the number of necessary nodes.

By analyzing results from Figure 8.7 we conclude that the extraction process is able to scale efficiently until all data can be extracted within the desired maximum extraction time. However, if the data-rate increases very fast (e.g. into the double) in a small time window, AScale requires additional extrac-

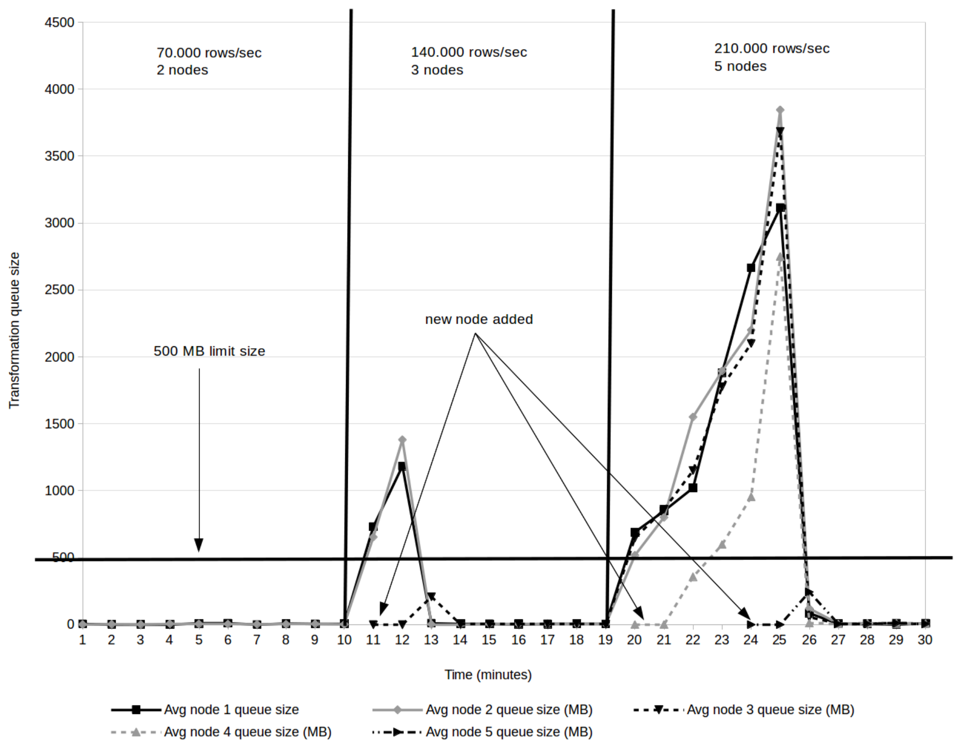


Figure 8.8: Transformation scale-out

tion cycles to restore the normal extraction frequency.

Transformation: In Figure 8.8 we test the transformation scale-out. The scale-out decision is based on monitoring the data queue size in each node. Every time a queue exceeds the maximum configured size, AScale scales that part automatically. The monitoring process allows to scale-out very fast, even if the data rate increases suddenly.

Figure 8.8 is divided in three parts, each one with a different data-rate. The data-rate is increased in each part in order to show AScale scaling-out the transformation nodes every time a queue reaches above the limit size, by adding one and two extra nodes.

For each scale-out it was necessary less than 1 minute until the node is processing data. The time delay refers to the copy and replication of the staging

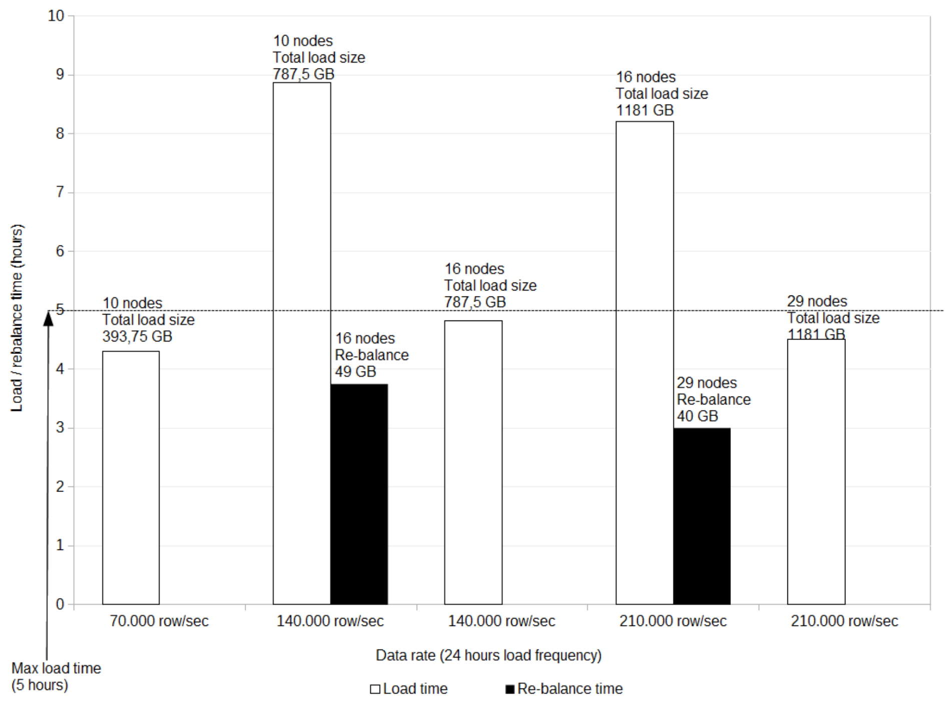


Figure 8.9: Data warehouse load and re-balance scale-out time

area.

Load: AScale load needs to be scaled at the end of a load cycle if the last load cycle did not respect the maximum load time. The number of nodes to add is calculated linearly based on previous load times. For instance, if the load time using 10 nodes was 9 hours, in order to be able to load in 5 hours, we need X nodes, Equation 8.1.

$$\frac{loadTime}{targetTime} \times n \tag{8.1}$$

Where "loadTime" represents the last load time, "targetTime", represents the desired load time and "n" represents the current number of nodes. Figure 8.9 shows the data-warehouse nodes load time for different data-rates over 24 hours generation, on top of each bar is represented the total loaded

data size and the number of nodes used. White bars represent load time and black bars represent the data re-balance time when more nodes were added. Every time the maximum load time was exceeded, more data-warehouse nodes were added and the data-warehouse was re-balanced.
We conclude that the data-warehouse nodes can be scaled efficiently in a relatively short period of time, given the large amounts of data being considered.

8.6 Near-Real-time and minimal downtime scenarios

In this section we assess the scale-out and scale-in abilities of the proposed framework in near-real-time scenarios, as well as scenarios where downtime should be minimized. Near-real-time scenarios require data to be always up to date and available to be queried (i.e. data freshness). In order to guarantee this, it is necessary to integrate new data in a predefined and very small time window.
The scenario was set-up as follows: E (extraction) and L (load) were set to perform every 2 seconds; T (transformation) was configured with a maximum queue size of 500MB; the load process was made in batches of 100MB maximum size. The ETL process is allowed to take 3 seconds at most.

Figure 8.10 and 8.11 show AScale scaling-out and scaling-in automatically, to deliver the configured near-real-time ETL time bounds while the data rate increases or decreases respectively. The X axis represents the data-rate, from 10.000 to 500.000 rows per second; the Y axis is the ETL time expressed in seconds; the system objective was set to deliver the ETL process in 3 seconds; the charts show the scale-out and scale-in of each part of AScale, obtained by adding and removing nodes when necessary; A total of 7 data sources were used/removed gradually, each one delivering a maxi-

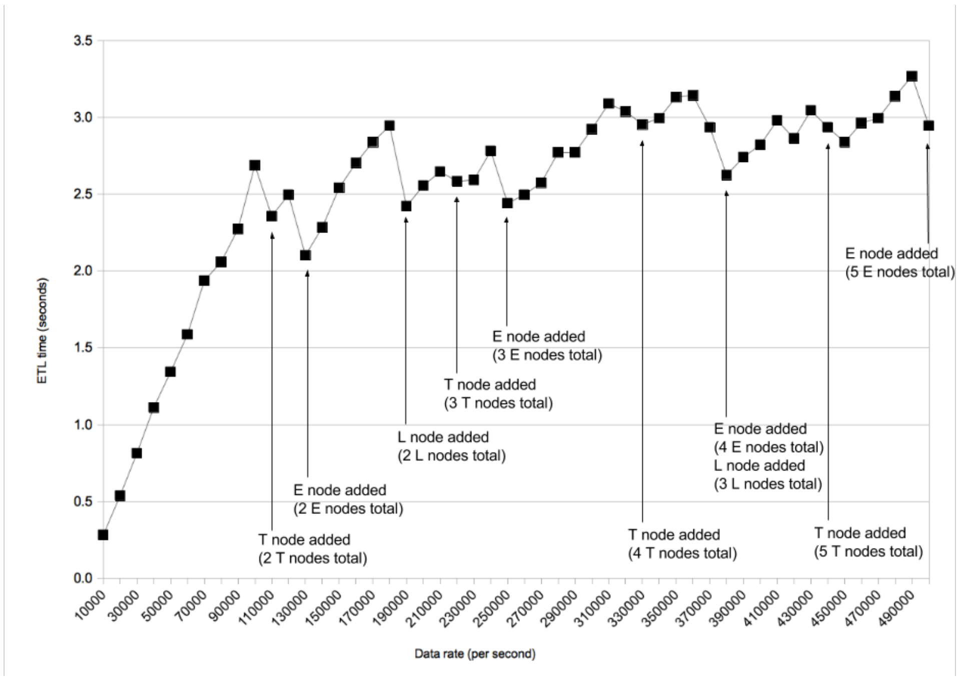


Figure 8.10: Near-real-time, full ETL system scale-out

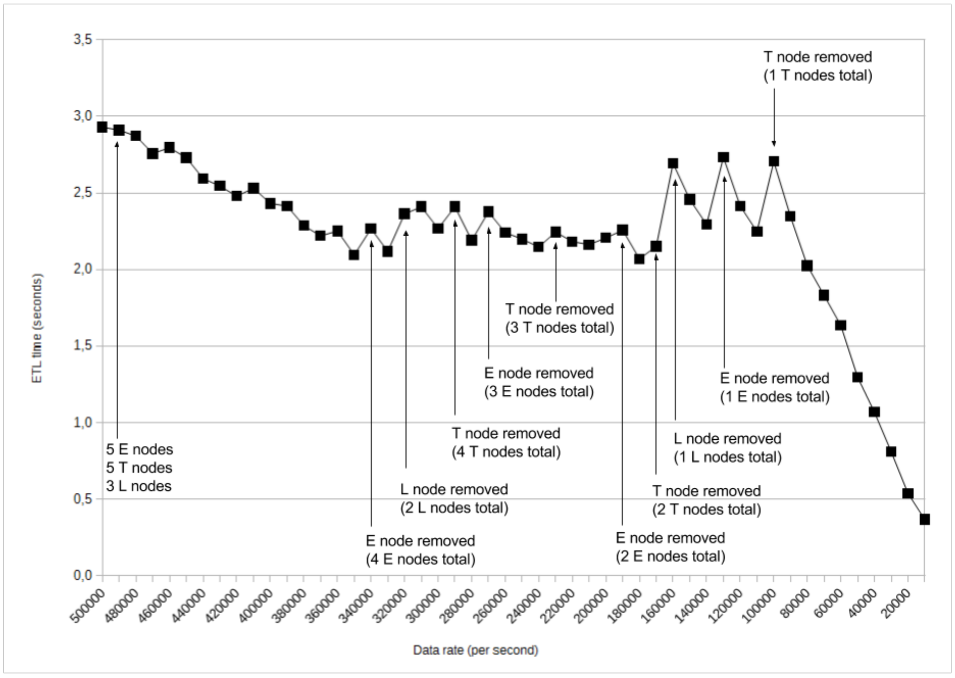


Figure 8.11: Near-real-time, full ETL system scale-in

mum average of 70.000 rows/sec; AScale used a total of 12 nodes to deliver the configured time bounds.

Scale-out results in Figure 8.10 show that, as the data-rate increases and parts of the ETL pipeline become overloaded, by using all proposed monitoring mechanisms in each part of the AScale framework, each individual module scales to offer more performance where and when necessary.

Scale-in results in Figure 8.11 show the instants when the current number of nodes is no longer necessary to ensure the desired performance, leading to removal of some nodes (i.e. set as ready nodes in stand-by, to be used in other parts).

The next (sub)sections detail how each part of the ETL and queries scale-out in the near-real-time scenario.

8.6.1 Scalability of data extraction

Considering data-sources generating high-rate data and extraction-nodes to extract the generated data, when the data flow is too high, a single data node cannot handle all ingress data. In this section we study how the extraction nodes scale to handle different data-rates, using the data setup similar to the one used in the previous section. The maximum allowed extraction time was set to 1 second ($\max_{extractionTime} < \max_{desiredExtractionTime}$); and extraction frequency was set to every 3 seconds ($\max_{extractionTime} < ExtractionFrequency$).

Figure 8.12 shows the extraction nodes automatic sacalbility in near-real-time. The left Y axis represents average extraction time in seconds, the right Y axis represents the number of nodes used; the X axis represents data-rate; the black line represents extraction time; the grey line represents

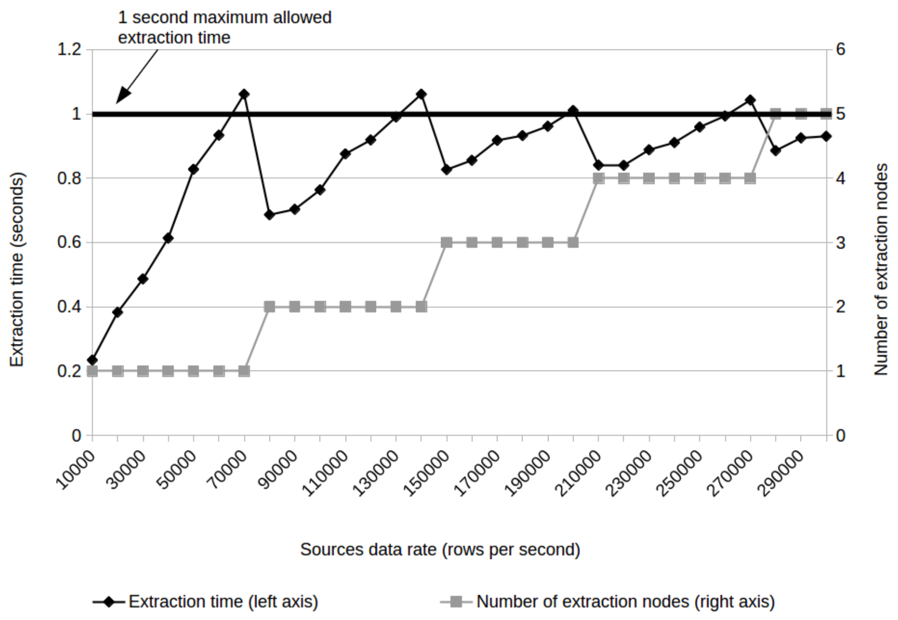


Figure 8.12: Extraction scalability

the number of nodes as they scale-out. Every time the extraction time is above the configured threshold limit of 1 second, a new node is automatically added (from the pool of ready-nodes). After the new node is added, more nodes are being used to extract data from the same number of sources, improving the extraction performance.

8.6.2 Transformation scalability

During the ETL process, after data is extracted, it is set for transformation. Because this process can be computationally heavy, it is necessary to scale the transformation nodes to ensure that all data is processed without delays. Data ingress transformation queues are monitored. Once it is detected that a queue is full above a certain configured threshold AScale scales the transformation process.

The transformation nodes scale-out mechanisms were set to limit the queue memory size to a maximum of 5GB before swapping data into disk,

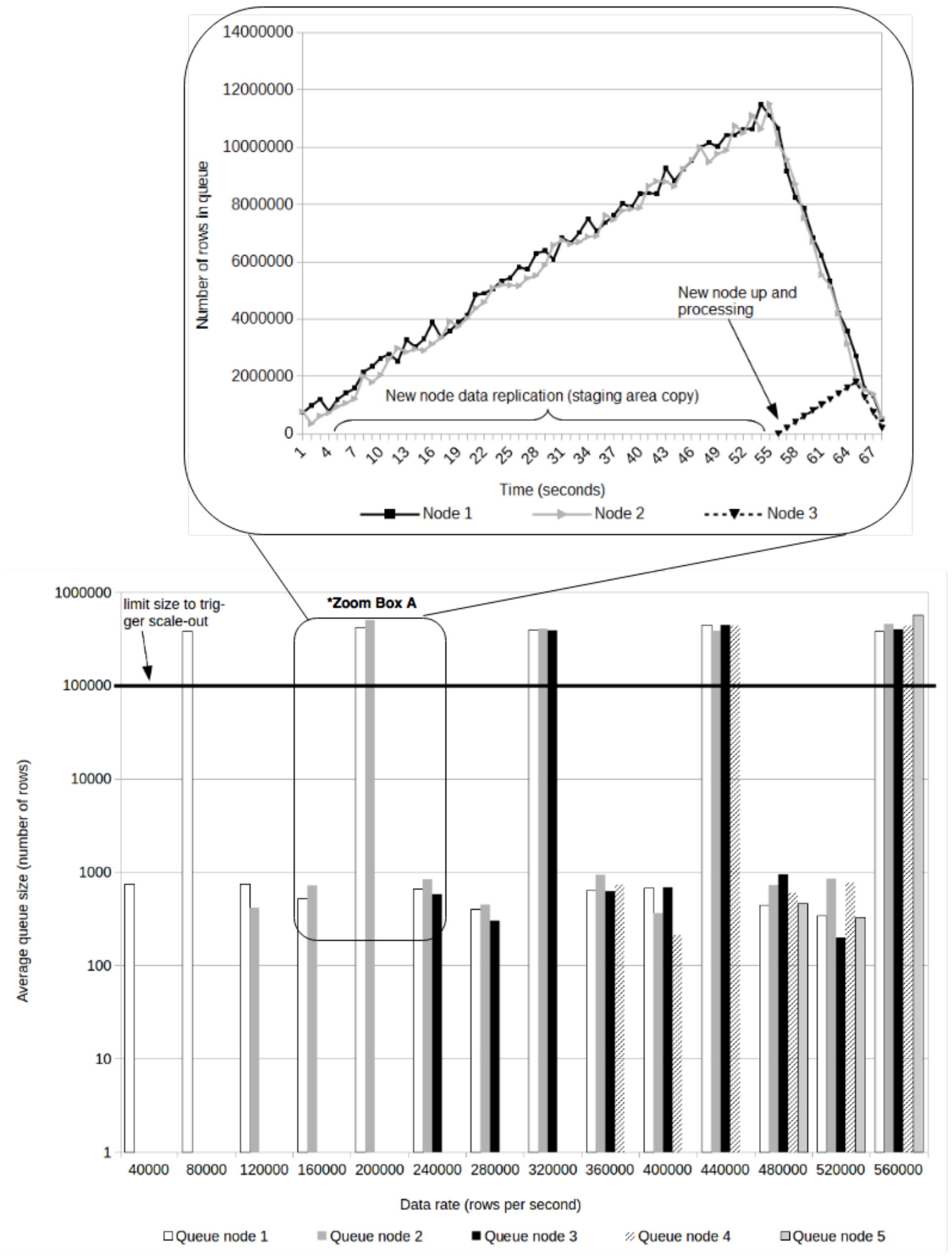


Figure 8.13: Automatic transformation scalability

and 500MB as the limit to trigger the scaling mechanism (corresponding to 100.000 rows. Figure 8.13 shows the addition of new transformation nodes, as queue sizes increase above the configured limit. The Y axis represents average queue size in number of rows, the X axis shows the data rate expressed in rows per second. Each plotted bar represents the average transformation queue size (up to 4 nodes). As can be seen in the figure, as soon as the data rate reaches 80.000 rows/sec, AScale detects overload in the queue (queue size above 100.000 rows) and triggers addition of a new node. The new node can be seen in the 120.000 rows/sec bar, where two queues allow the system to handle the data rate satisfactorily. After that, as the data rate reaches 200.000 rows/sec the maximum queue size is reached again triggering addition of one more node. The scale-out can be seen in the 240.000 rows/sec bar, where three queues are again able to handle the increased load. A fourth and fifth node are added again to scale at 320.000 rows/sec and 440.000 rows/sec respectively, as can be seen in the figure.
in the Figure 8.13, "Zoom Box A" shows the detail of what happens when the queue sizes of transformation nodes increase above the configured limit. As overload is detected a new transformation node is added. The new node takes approximately 1 minute to be ready (see zoom box A), the delay corresponds to the necessary time to replicate/synchronize the node and respective staging area. After the node is fully working, data is distributed using a least renaming work (LWR) algorithm, which makes the queue of the new node balance balance the sizes of all nodes.

8.6.3 Data Buffer nodes

Data buffer nodes hold the transformed data until it is loaded into the datawarehouse. For this experiment the data buffers were configured as follows: we consider only the data generation/producer, there is no data "consumer", so the buffer must hold all ingress data; the generation data-rate speed was

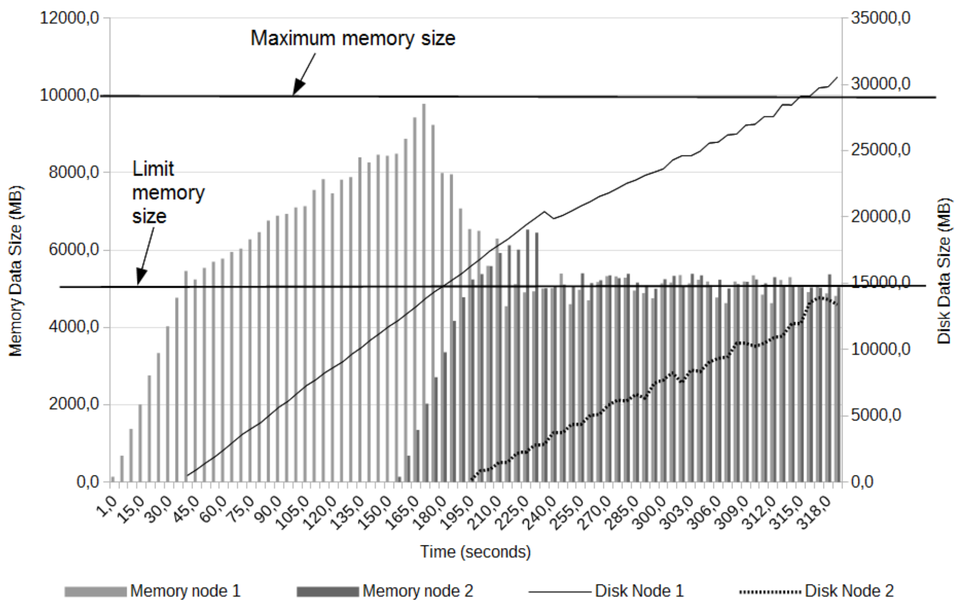


Figure 8.14: Data buffer swap into disk and scaling

set to 1.500.000 rows per second (i.e. transformation output data rate) in such way that the disk speed cannot swap all data fast enough, leading the memory to increase until its maximum; available memory storage was set to 10GB; memory storage limit before data swap into disk was set to 5GB; available disk storage was 1TB.

When the limit memory size is reached (5GB), data starts being swapped into disk. However, because the disk speed cannot handle all ingress data-rate, the memory reaches the maximum limit size (10GB). At that moment a new data buffer node is added. After the new node is added, data is distributed, using LWR, making the queue of the newly added node increase faster. After a while, the data volume in each data buffer returns to normal.

Figure 8.14 shows an extreme scenario where the data buffer write speed from memory into disk (swap) is not fast enough. This scenario leads to a scale-out of the data buffer node. The figure shows the data queue size increasing until the ”limit memory size” (5GB), at that moment data starts being swap into disk in an attempt to release memory space. However, because the data rate is too high the memory continues increasing until the "maximum memory size". Once the memory is at the "maximum memory size" another node is added and data is distributed by both nodes. In the figure we also show the data memory queue of the second node increasing and the swap process occurring again. Although, because now there are two nodes handling the ingress data rate, the data swap speed can free the memory.

8.6.4 Data-warehouse load and query scalability

In this section we test the data-warehouse scalability, which can be triggered either by the load process (because it is taking too long), or because query execution is taking more time than the configured response time bounds.
To test the load scalability we create the setup: loads are from batch files, each approximately size 100MB each; the maximum allowed load time is set to 60 seconds; each time a data-warehouse node is added we show the data size that was moved into the new node and the required time in seconds to re-balance data; all load and re-balance times include the execution of pre-load tasks (i.e. drop all indexes and views) and pos-load tasks (i.e. build all indexes and views).

Load scalability: Experimental results in Figure 8.15 show the data-warehouse scaling when the data size to be loaded increases and as consequence the load time also increases above the predefined bound: the left Y axis represents average load time in seconds; the right Y axis shows the number of data-warehouse nodes; the X axis represents the data batch size in MB; the horizontal bar at Y = 60 seconds represents the maximum configured load time; at each scale-out moment there are notes specifying the data re-balanced size and time to perform it; the black plotted line represents average load time; the grey plotted line represents number of data-warehouse

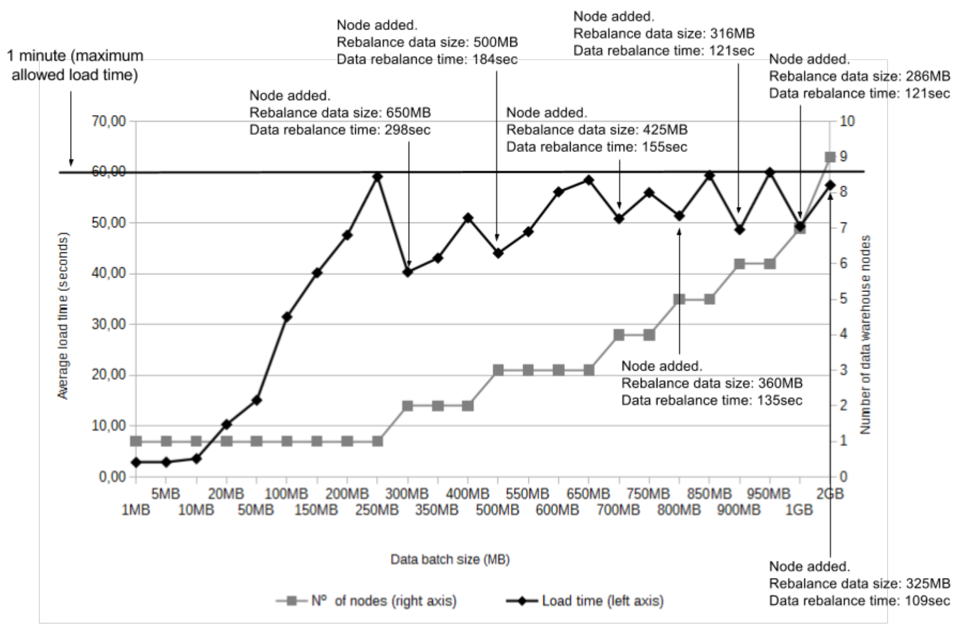


Figure 8.15: Data warehouse load scalability

nodes.

Results in Figure 8.15 show how load performance degrades as the data size increases and how it improves when a new node is added. After a new node is added, performance improves to meet the maximum configured load limit.

Query scalability: When running queries, if the maximum desired query execution time is exceeded, the data-warehouse is set to scale-out in order to offer better query execution performance. The following workloads were considered to test AScale query scalability:

- Workload 1 (WL 1);
  - 50GB total size;
  - Execute Q1.1, Q2.1, Q3.1, Q4.1 randomly chosen;
  - Desired execution time per query: 1 minute (60 seconds).
- Workload 2 (WL 2)- as workload 1 but, 1 to 8 simultaneous sessions

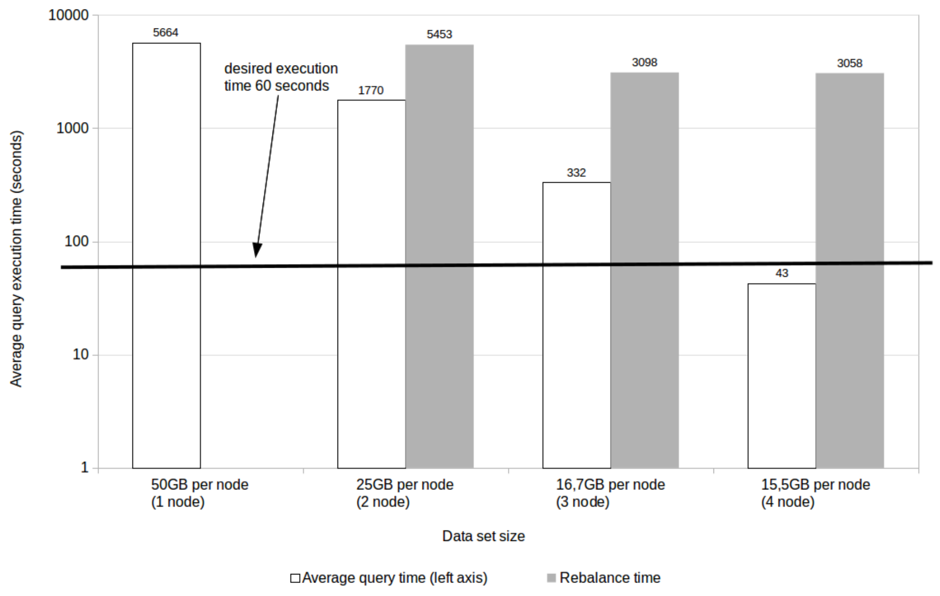


Figure 8.16: Data warehouse scalability, workload 1, 50GB data set

used;

Workload 1 studies how the proposed mechanisms scales-out the data-warehouse when running many queries. Workload 2 studies AScale scalability running simultaneous sessions (e.g. number of simultaneous users). Both workloads were set with the objective of guaranteeing the maximum execution time per query of 60 seconds.

Query based scalability, WL 1: Figure 8.16 shows the experimental results for workload 1, where: the Y axis represents the average execution time in seconds; X axis represents the data size per node and the current number of nodes; the horizontal line over 60 seconds represents the desired query execution time; white bars identify the total workload time and grey bars the re-balance time (i.e. extract data, load into nodes, rebuild indexes and views).Results shows that every time a query is executed and the average query time is not inferior to the maximum configured query execution

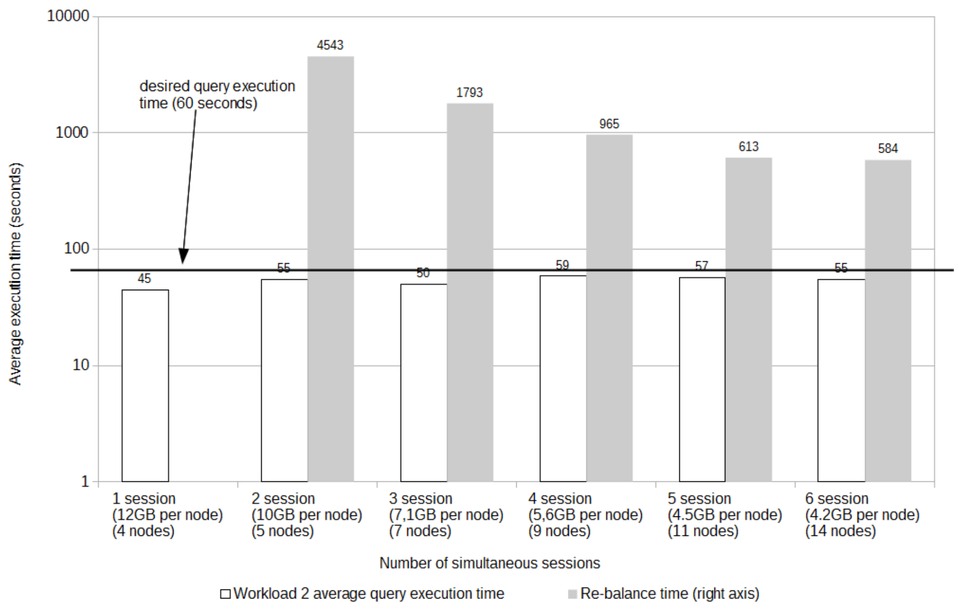


Figure 8.17: Data warehouse scalability, workload 2, 50GB data set

time, one extra node is added (scale-out). In each scale-out the re-balance time represents the necessary time to extract data from existing nodes, re-distribute it and rebuild indexes and views. Once the average query time becomes lower than the configured desired execution time, the framework stops scaling the data-warehouse nodes.

Simultaneous Session query scalability, WL 1: Figure 8.17 shows how the data-warehouse scales when simultaneous sessions are executing. The figure shows: the left Y axis represents average query execution time in seconds; X axis shows the number of sessions, the data size per node and the number of nodes; grey bars represent the data re-balance average time in seconds (i.e. extract from nodes, load into new node, rebuild indexes and views); white bars show average query execution time. Results in Figure 8.17 show that, the number of simultaneous sessions increases, the system scales the number of nodes in order to provide more performance. Thus, average query execution time follows the configured parameters.

Note that since both loads and query execution are performed against the data-warehouse and the data-warehouse is scaled, AScale query execution performance improves at the same time the data-warehouse load performance improves and vice versa.

8.6.5 Freshness and dynamic-data-warehouse scalability

In this section we evaluate the capability of the dynamic-data-warehouse (D-DW) component to scale. The D-DW is responsible to provide always fresh data and instant integration of new data into results.

Figure 8.18 shows the dynamic-data-warehouse nodes scaling to offer faster data integration while the data-rate increases in time. Data integration requires inserts of arrival data into the D-DW component, after ET (extraction and transformation). The ingress data queue holds the data while it is waiting for its time to be inserted. If this queue exceeds a certain size a new D-DW node is added to scale. In the left Y axis we represent the queue size expressed in number of rows. In the X axis is represented the data rate increasing over time (every 5 seconds). Each plotted line represents a node queue. The horizontal line on 1000 rows represents the configured limit number of unprocessed rows before scaling-out.
Results in Figure 8.18 show the overload detection moment and the addition of new nodes. D-DW nodes scale every time a data queue reaches the configured limit size. When a new node is added, data starts immediately being (re)distributed across all available nodes using a LWR policy.

Dynamic-data-warehouse query scalability: We tested the dynamic-data-warehouse query scalability considering two factors. When executing queries and with simultaneous sessions. Workload 1, tests the scalability based on queries execution. Workload 2, tests the scalability based on the

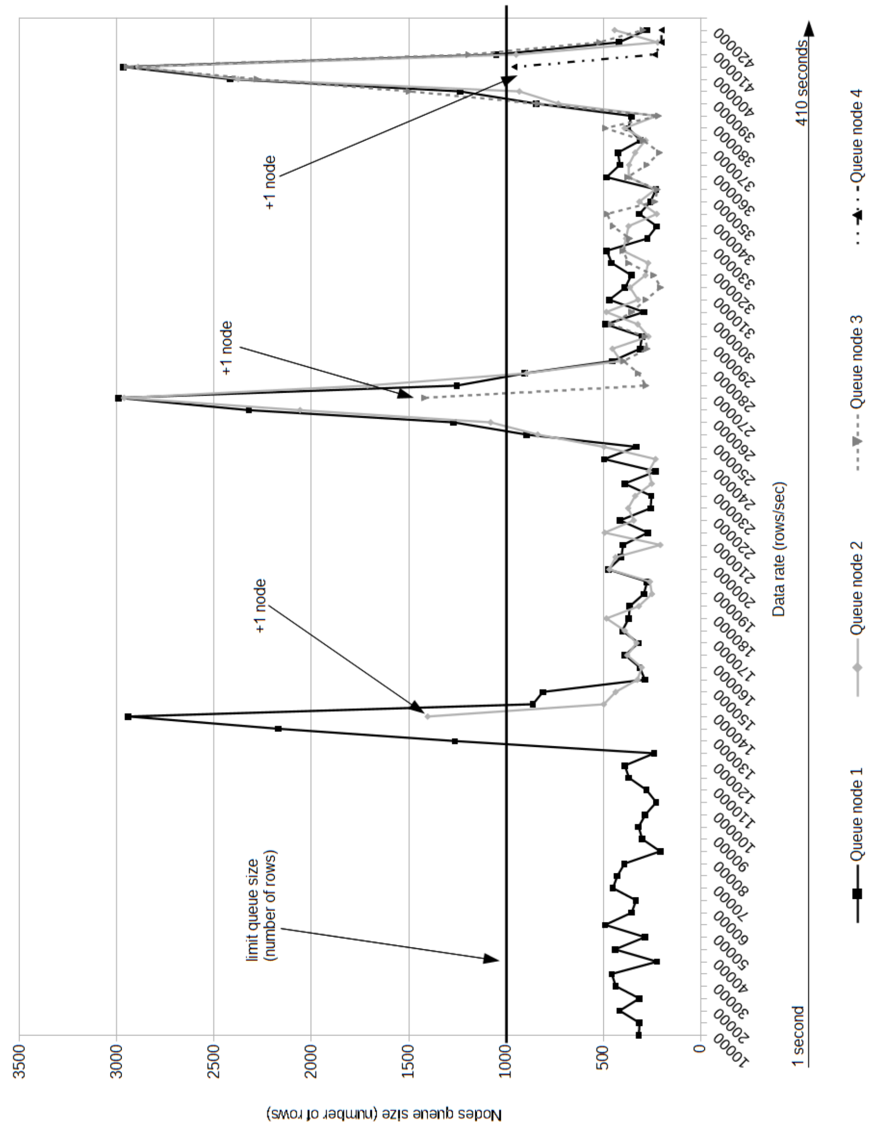


Figure 8.18: Automatic freshness scalability

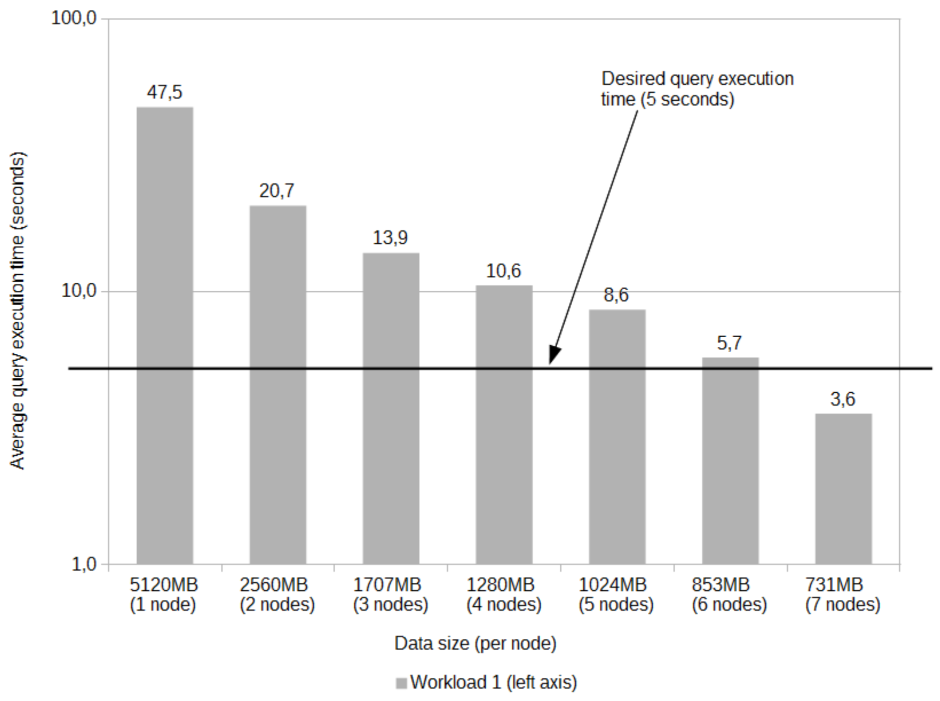


Figure 8.19: Automatic freshness scalability, workload 1, 5GB

number of sessions (e.g. simultaneous users):

- Workload 1;
  - **5GB** total size, and growing (online load - new data is being inserted while executing queries);
  - Execute Q1.1, Q2.1, Q3.1, Q4.1 randomly chosen;
  - Desired execution time per query, 5 seconds.
- Workload 2 - as workload 1 - but 1 to 4 simultaneous sessions;

Figure 8.19 shows experimental results using workload 1. In the chart, the Y axis represents average query execution time in seconds; the X axis represents the data size per node and the number of nodes; the horizontal line over 5 seconds represents the desired query execution time; plotted bars represent the average execution time per query.

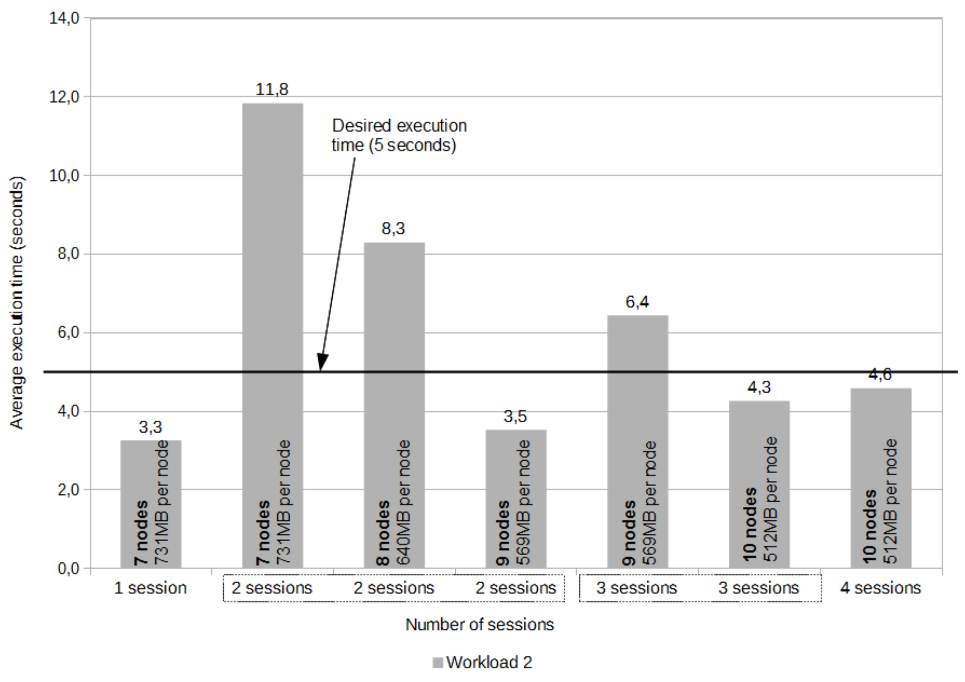


Figure 8.20: Automatic freshness scalability, workload 2

Figure 8.20 shows the experimental results using workload 2 based on simultaneous sessions. The chart shows: in the Y axis the average query execution time in seconds; the X axis represents the number of simultaneous sessions; line over 5 seconds represents the desired query execution time; plotted bars represent the query average execution time.
At each scale-out instant data is re-balanced (extracted from existent nodes and submitted to the new node, all in near-real-time) each scale-out took an average of 20 to 40 seconds.

8.7 Continuous results

AScale is also capable to analyze streams of continuous results in almost real-time (e.g. to issue alarms, detect frauds), and at the same time allowing CEP queries to access the data-warehouse and dynamic-data-warehouse to validate and record events.

In this section, the data set and query workloads, regarding the "Complex Event Processing" nodes concern a telecommunications scenario, where call-detail-records are constantly being fed to the P/C queue from distributed data sources. The used call detail records were from Asterisk [81]. An example of a call detail record can be found in Appendix B.

Queries output results periodically. They are assumed to be ad-hoc, in the sense that their processing weight is not known prior to running, and simple parameters such as output periodicity or selection conditions can result in very different query weights. These alternatives are considered in the query workload which includes:

- Light Queries (**Ql**), "light" per-tuple filters;
- Heavy in memory queries (**Qh-m**), time-consuming in-memory (e.g. scans of data);
- Heavy queries with database (**Qh-db**), stream processing queries with database sub-queries;
- Heavy queries with memory look-ups and updates (**Qh-mlu**);
- Killer queries (**Qdb-2**), stream processing queries that include database sub-queries and "kills" the node. Qdb-2 database query and CEP results output window are too demanding, resulting in too heavy processing.

Queries are frequently registered and de-registered dynamically and in a random fashion. Queries are described in more detail in Appendix B.

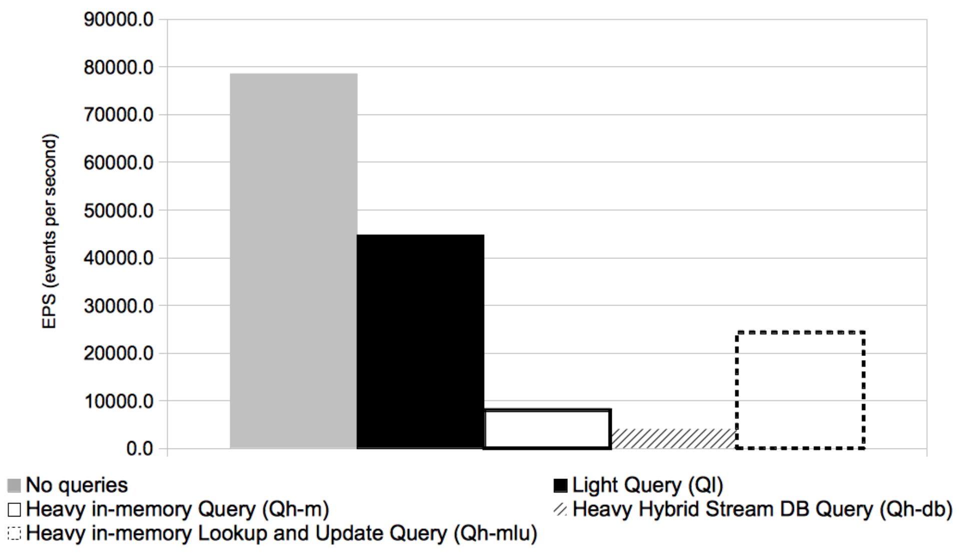


Figure 8.21: Base line comparison of performance architecture.

8.7.1 Gold run

This experiment evaluated throughput of a single node when running each type of query (alone). These results provided a baseline for comparison purposes.

The results shown in this section are measured in throughput, as Events-Per-Second (EPS). Figure 8.21 shows per-node throughput results obtained in the gold run. From those we can see that it was able to process about 80K events per second (EPS) without any query registered into it. A light query reduced the throughput to 45K EPS, while a heavy query reduced the throughput much further (e.g. the stream-DB query Qdb tested in this run achieved a throughput of less than 4K EPS).

8.7.2 Degradation run

In this experiment we overloaded a single node by submitting high-rate data and registering a heavy hybrid stream-DB query with small output time window, Qdb-2. It shows that the node dies, i.e. becomes overloaded

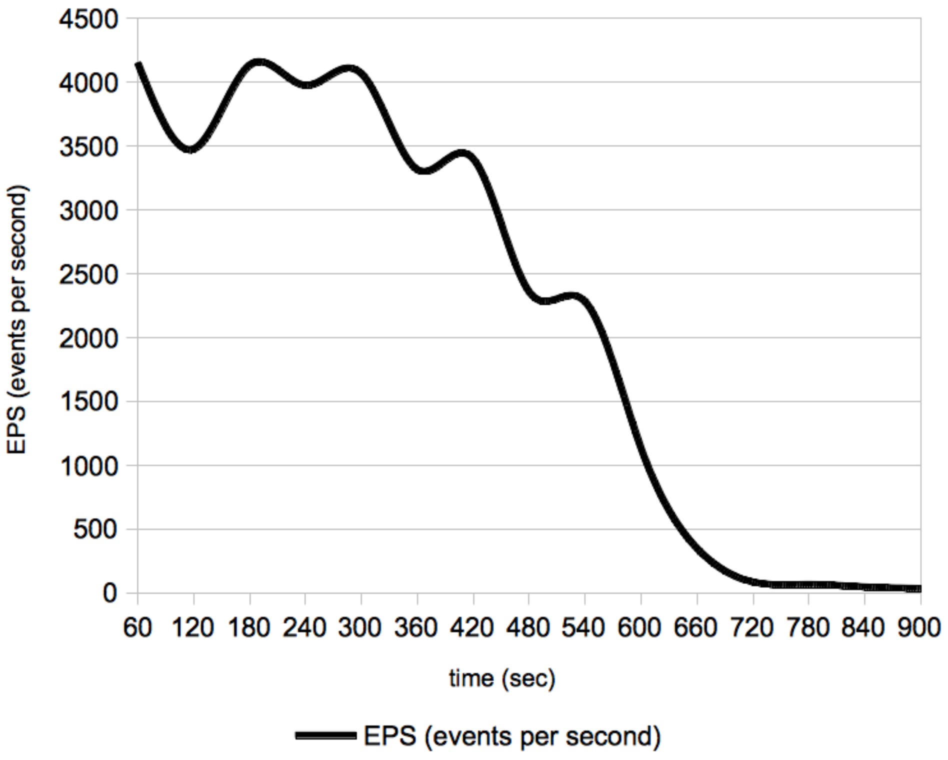


Figure 8.22: Overall performance degradation due to heavy queries (Qdb-2).

and incapable of producing results – after some rounds of heavy querying and processing. This motivates the need for data admission control and for adequate query balancing.

The results in Figure 8.22, which concerns execution of the stream-DB query Qdb-2, show that the throughput degrades severely over time, up to a situation in which the node is unable to accept new data (dead node). Besides processing the stream, this query aggregates over a 500MB data table for computing outputs, and must return the outputs every 10 seconds.

Figure 8.23 illustrate an example why this happens. The query time exceeds the output time period that was defined in the CEP query, thus the system fills-up (queue increases, memory and swap overload happens), to a point when the engine stops being able to consume input data. This behavior is as expected, since conceptually a system cannot serve at a smaller

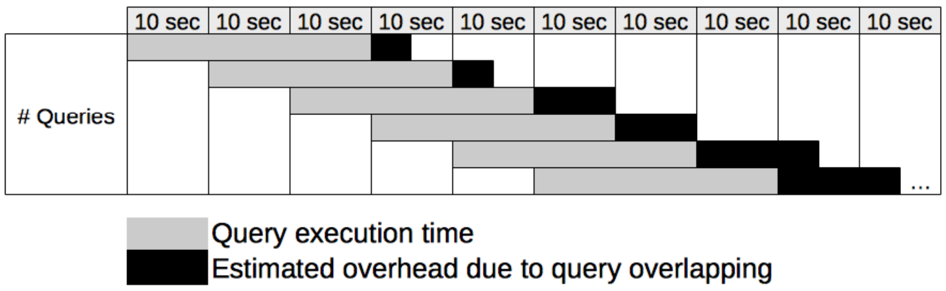


Figure 8.23: Query overlapping and estimated overheads in time.

rate than the data arrival rate indefinitely without reaching a point at which it needs to discard data. AScale mechanisms are able to identify these situations and prevent them from happening (e.g. buffer queues getting overloaded, scheduling queries for other nodes, using ready-nodes, load shedding).

In Figures 8.24 and 8.25 we show the system behavior (10 processing nodes) when systematically adding more in-memory light queries, Figure 8.24, and more in-memory heavy queries, Figure 8.25, with AScale. At each 3 queries added, a Qdb-2 was also added. As shown in the charts, when using AScale the performance decreases almost linearly when adding more processing tasks, and the system avoids dead nodes by rejecting the addition of the Qdb-2, which is achieved by relocation and admission control. Without AScale, some overloaded nodes stop answering because of overloading (dead nodes).

8.7.3 Workload experiments

These experiments evaluate balancing algorithms and admission control against different query workloads. We run: AScale with least-weight (LW) balancing; AScale with the balancing algorithm changed to be round-robin (RR); AScale with the balancing algorithm changed to be LW based in the number of remaining queries (LWRn); No queries admission control, RR

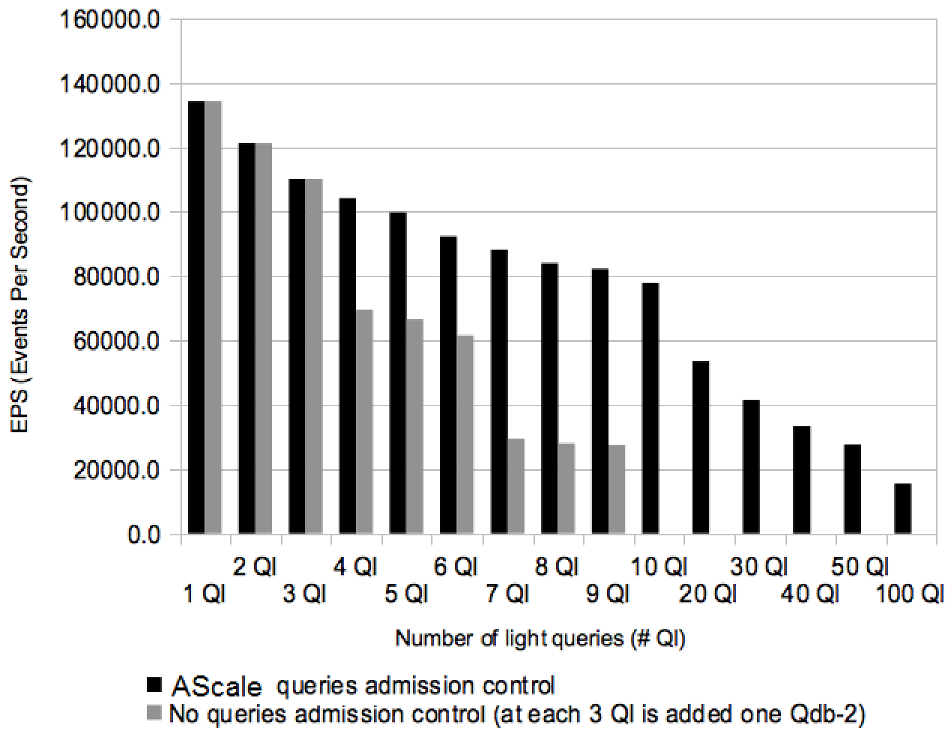


Figure 8.24: AScale admission control vs no admission control, with light queries, Ql.

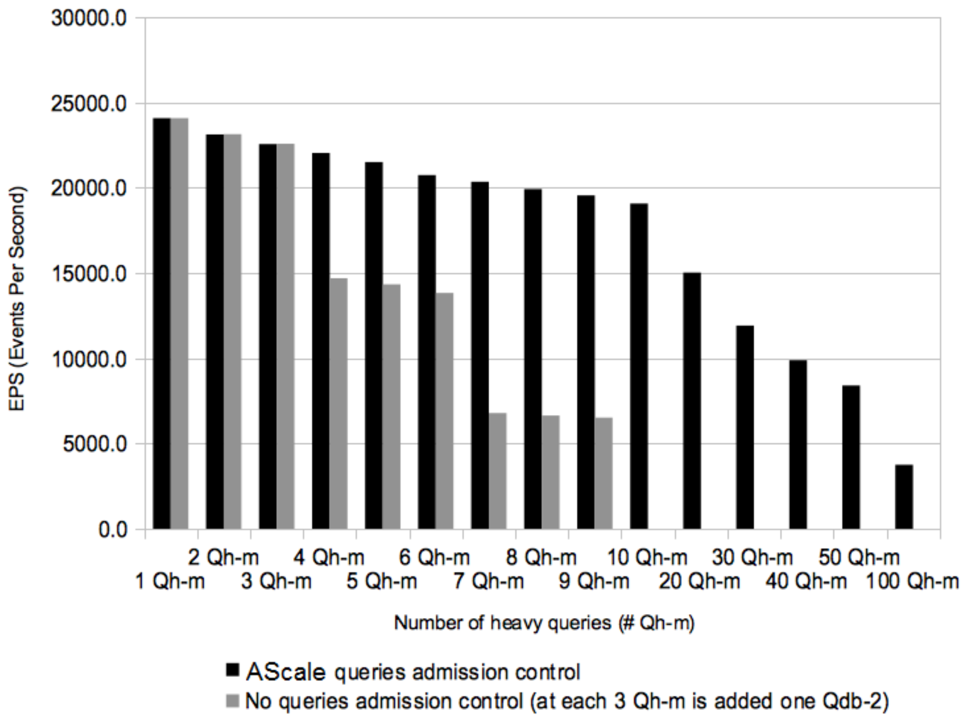


Figure 8.25: AScale admission control vs no admission control, with heavy queries, Qh-m.

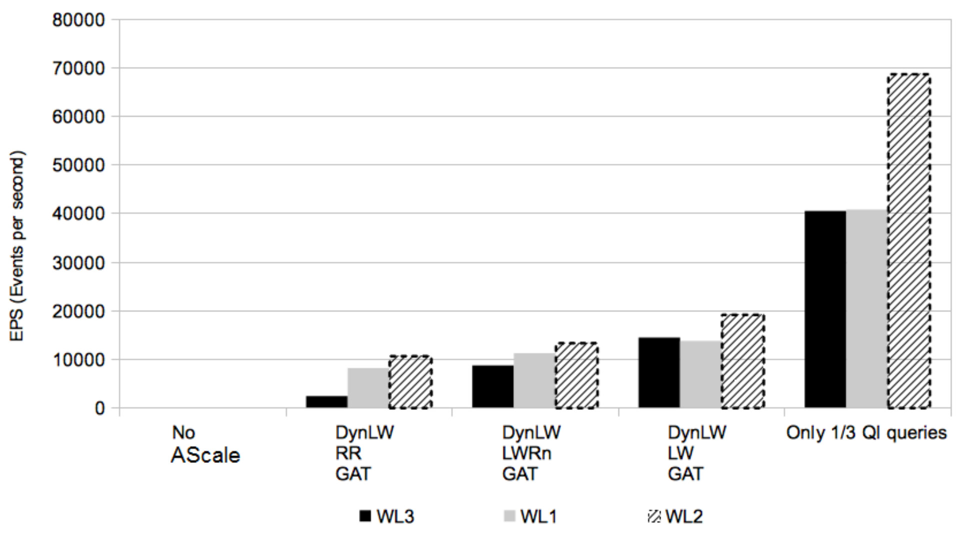


Figure 8.26: Workloads 1, 2 and 3

processing without admission control. The experiments were ran in a setup using a total of AScale 3 CEP nodes.

Figure 8.26 show the results obtained from running workloads WK1 (27 Qh-m, 54 Ql), WK2 (15 Qh-db, 30 Ql) and WK3 (45 Qh-mu, 90 Ql), where (-m) means in-memory and (-mu) means in-memory and updates (for each output, an in-memory list was scanned and values were update values).

For the comparison between balancing algorithms, we define a metric called GAT – *Guaranteed Achievable Throughput.* Data may be arriving at any rate (high, low or not arriving at all). The GAT is defined as the maximum data input rate that the algorithm or systems are able to guarantee as the input data rate is stressed. It corresponds to processing the data with the worst balancing case for that algorithm.

The results will show that admission control avoids node overloading and that least weight balancing is able to constantly maintain a higher throughput than the alternative algorithms.

The first bar group (No AScale) from left to right in each group concerns a run without admission control of queries. It shows that the throughput drops to zero when admission control is not applied. The main reason is due to degradation of performance because of very heavy queries (Qdb-2) being allowed to enter the processing system.

Comparing the observed throughput's of AScale with RR, LWRn and AScale with least weight, we can conclude that AScale with least weight supports a much higher throughput. We also conclude that without AScale admission control and no (re)balancing the system dies.

For reference purposes, the last bar shows the throughput that is achieved by a single node running 1/3 of the queries, but when all those queries are light. This throughput cannot happen in practice when considering the whole system with a mix of light and heavy queries (WK1, WK2 or WK3), since if a node runs only light queries, the remaining nodes will be overloaded with heavy queries, resulting in a small system throughput or nodes death.

8.8 Conclusions

This chapter presented an experimental evaluation of AScale. First by demonstrating ETL+Q scalability importance and how a single server is incapable of ETL+Q processing when the data-rate increases.
The second experiment evaluates AScale for use in typical data-warehouse scenarios, where the load process is slow and needs to scale. Results show how the pipeline parts (extraction, transformation and load) scale-out when necessary for performance reasons.
The third scenario tested a case with huge amounts of data. In this scenario

the extraction and transformation process occurs every hour and the load process starts every 24 hours. To avoid several lengthy cycles of scaling one node at a time, we applied a proportional scale-out formula to estimate the necessary number of nodes, in order to optimize performance.

The fourth experiment evaluated AScale scaling for two alternative scenarios: near-real-time, with high-rate data and strict freshness requirements, and minimal downtime. These tests demonstrated AScale scaling-out and -in when ETL was configured to take only 3 seconds. We show that AScale monitoring can scale each individual ETL pipeline module, assuring the necessary resources to keep performance within bounds.

A fifth experiment evaluated scalability of the continuous result processing CEP add-on, from which we conclude that AScale with CEP integration performs efficiently to deliver not only the configured performance, but also to (re)balance and avoid overloaded CEP nodes.

Overall, AScale was able to scale all parts in the ETL+Q pipeline.

Chapter 9

Conclusions and future work

In this thesis we proposed an Automated Scalability Framework for Data Warehouses, named AScale.

Using the proposed approach, data warehouse designers focus in the conceptual design of the data warehouse schema, transformation operations and queries. Then, based on integration primitives and configurations, and on performance configurations, AScale monitors and scales any part of the ETL+Q pipeline that might need more or less parallel resources. We present the first proof of concept, with the purpose to describe how connection and integration with other tools can be made.

We extended AScale in three directions. First, to support automatic ETL and query processing scalability, the concept assumes that the developer only focuses on data transformations and the data warehouse schema, then AScale automatically scales the ETL process by replicating and adding more resources. The second concept is data freshness. The concept is to automatically include into data analysis (i.e. queries) the most recent transformed data that was not yet integrated into the main data-warehouse. For that we introduce the Dynamic-data-warehouse, to hold only the most recent data, and we also show how to scale that part. Third, AScale also support continuous results processing (CEP) involving a different nature of queries.

Data is stored temporally in a data window (e.g. 1 minute) and results of registered queries are kept updated.

The AScale pipeline model was designed to support scalability of all modules independently. On top, we added support for data freshness and later support for CEP. For each module, we defined scalability mechanisms, based on queues/buffers monitoring and time performance. Also, alternate data transfer policies between modules were defined, such as, least work remaining, round-robin, manual and scheduled based.

Finally the chapter "Experimental evaluation" proves AScale concept and efficiency though a series of experimental results and analysis.

9.1 Future Work

There are a number of interesting directions for future work.

The AScale framework can be compared, both in terms of approach, usability and performance, with an implementation using Map-Reduce modules. We plan to not only test performance, but also the learning curve of using both approaches, implementation costs and required hardware to offer comparable performance.

Implementation of proactive and predictive scalability mechanisms are other future work direction.

Another future work direction would be to provide extended support for more features, visual tools (drag-and-drop), tools to build schemas and to configure ETL processes.

During our tests we assumed a node as a physical machine instead of a process with affinity to a core. This derives from AScale limitations regarding machines resources monitoring. Future work will also explore resources monitoring of physical machines (i.e. nodes) for automatic process based scalability.

Each AScale module communicates based on an IP:PORT, future work includes the exploration and applicability to the cloud for elastic scalability. Full availability, "24/7", should be explored in detail. Future work includes a comprehensive analysis of the scalability approaches, load methods, data balance and querying in a 24/7 always on case.

Another interesting research topic would be to include some of the scalability mechanisms inside a database engine. For instance, the Dynamic-DW and Stream-DW can reside in the same hybrid database engine. Such system was already projected and development started, we named it "VarDB" [82]. This approach would support a column oriented storage for more data flexibility, supporting hybrid in-memory and disk based storage, the schema flexibility of no-SQL, column data partitioning (i.e. by size, in-memory, in-disk, etc), and standard SQL script language support.

More future work directions would be to add transformation properties/operators to the hybrid database engine or to AScale itself. At the end, the whole AScale framework could reside in a single storage system.

Appendix A

Background work

In this section we provide a global overview to the main topics associated with the work of this thesis for achieving ETL, data warehousing and query execution performance and scalability. Data warehousing (DW) is a technology that helps data analysis for decision making in large organizations. Data warehouse parallelization and partitioning is already a well-known technique, explored by many works for parallel queries processing and data distribution for support of efficient parallel and distributed processing [83], [27], [84].

In the following sections, we give a global overview of the database processing approaches and related concepts that we explore in this thesis. In Section A.1 we introduce the concept of High Availability, followed by Section A.2 where we introduce how databases can be replicated for performance and availability at the same time. Section A.3, introduces the concepts of scale-out and scale-up. In Section A.4, we explain the concept of Elastic Scalability and parallelization. Section A.7 we introduce the ETL process and query processing. Section A.5 introduces the most common distributed architectures and Section A.6 explains the concept of Parallel databases with some examples. Because virtualization of processors, cores, disks and full computers is widely used to take advantage of computers resources, in Section A.8

we introduce the concept of Virtualization. And also since Map-Reduce is a very recent paradigm to scale systems, in SectionA.9 we introduce the Map-Reduce paradigm. Afterwards in Section A.10 we present some limitations of Map-Reduce paradigm. Because storage models and data access requirements change, in Section A.11 we describe the no-sql paradigm. Section A.12 compares the main differences between parallel database systems and Map-Reduce. Finally in Section A.13 we focus on industry and academic database engines.

A.1 High Availability

High availability refers to the ability of a system to remain accessible for users permanently in the presence of high intensive activity, software maintenance, or hardware failures (e.g. an application may crash or a network card may go down or an entire physical machine can fail). Various studies have been shown that downtime (e.g. for updates) is a "revenue killer". Small to medium sized businesses lose 1% of revenue per year to system downtime, while 40% to 50% of businesses never fully recover after a major system outage [85].
The requirements to provide high availability is no longer limited to mission critical applications. Even the most simple applications require high availability, meaning that high availability is no longer an option but rather a requirement. Indeed, most businesses now implement some form of high availability for their IT infrastructure. Database systems also need to provide high availability, where the database remains accessible and consistent in the presence of failures or intensive load periods, with little or no downtime.

A.2 Database Replication

Replication is a technique where a database is fully or partially replicated locally or at a remote site to improve performance and data availability [86]. The original copy of the data is referred to as the primary (or master) replica while the replicated copy is called a secondary replica. If the primary replica becomes unavailable, the data still remains accessible through the secondary replica. We can also have N-way replication where a primary replica is copied to N-1 other replicas. However, this additional replication comes at a higher cost. Replication may be implemented within a single database system or between database systems across machines possibly distributed across geographical boundaries. Changes from the primary replica are periodically propagated to the secondary replica.

Synchronous or asynchronous replication are the two options for keeping the secondary replica up-to-date with the primary replica.

Synchronous replication [87] [88] is usually implemented through a read any, write all mechanism. Where a read transaction can read data from any replica but a write transaction must update all replicas. For this type of replication, the updating transaction must acquire exclusive access to all the replicas which might involve lock requests across remote sites. Also, in order to update transaction for successfully commit, all the replicas must remain accessible during the transaction. If the replicas are distributed among remote and local sites, a two-phase commit protocol is required to ensure that each transaction either commits at all replicas or is aborted. Synchronous replication offers the advantage of keeping all the replicas strongly consistent. However, due to the high operational costs associated with acquiring locks, and excessive message passing for two-phase commit, it is rarely used in practice [87].

On the other hand, asynchronous replication [87], [89], [90], [91], propagates the changes from the primary to the secondary through transaction log shipping or snapshots. The changes in the log records or a snapshot are then applied to the secondary copy.

Asynchronous replication offers a trade-off in terms of minimizing overhead of normal operation and data consistency. With asynchronous replication, it is possible for a transaction to get slightly different results when accessing different replicas since they are updated periodically. In a typical setting, asynchronous replication is run with two servers, a primary (master), and a secondary. with the secondary server's database state being transactionally consistent to that of primary server with some replication lag. Moreover, typically, only the primary server can update the replicated data (e.g., in primary site asynchronous replication). When the primary server fails, the secondary server takes over execution losing some work, for example, in-flight transactions are aborted and restarted on the secondary server. Also, the secondary server needs to be brought up-to-date after a failure, i.e., the transaction log of committed operations performed on the primary server that were not yet propagated to the secondary server needs to be replayed at the secondary. After a failure, the primary server can be recovered while the backup server acts as the primary. Later, the roles of the two serves can be switched again, or the backup server can remain as the primary, and vice versa.

A.3 Scale-out and scale-up

There are two distinct methods of scaling a system, horizontal (scale-out) or vertical (scale-in).

- Scale-out, means to add more computer nodes to a system for the distributed software take advantage of the new resources. The greatest advantage is that computers prices are lowering down while performance increases. However, larger number of computers means increased management complexity, as well as a more complex programming model.

- Scale-up, means to add more hardware, more memory, more disk, more CPUs. In radical cases completely replace the old hardware by new and more powerful.

A.4 Elastic scalability and data warehouse parallelization

A software system is said to be scalable if it is able to handle increasing load by using more computing resources. A system can be scaled up by adding more computing resources to the physical running machine for example, by adding more hardware (i.e. memory, CPU, disk, graphics). Scale-out permits a system to handle even larger workloads by adding more physical machines, like a cluster. Systems that are elastically scalable are able to respond to changes in load by growing and shrinking their processing capacity on the fly. In Elastic scalability, ideally at any given time, an application deployed should be using exactly the amount of required resources to handle its load, even as this load fluctuates [92]. Such elastic scalability results in significant cost savings for the applications. This way resources can be set offline to save energy or redirected to process other tasks.

Database systems are difficult to scale since they are stateful – they manage a large database. It is important when scaling to multiple server machines, provide mechanisms so that these servers can collaboratively manage the database and maintain its consistency. Database partitioning is often used to solve this problem, by making each server responsible for one or more partitions.

Parallel DBMSs (column-wise or row-wise) capable of running over shared-nothing clusters exist since the eighties. These types of systems support common relational tables and SQL script language. Although the data is stored in multiple machines, the data is accessed by the user through SQL language transparently. The two key aspects that make parallel execution possible are:

- Tables may be partitioned though the nodes of the cluster, allowing;
- ... The system to use an optimizer that is capable to translate the SQL queries to an execution plan of queries that is divided by multiples nodes.

Thus programmers only need to specify their objectives in a high-level language, not having to bother with details of how data is stored, indexing options, or strategies for joining and processing the data (whereas Map-Reduce [93] systems do need to specify all those details).
Consider a SQL statement that filters the records of a table T1 based on a predicate, and that makes a join with a second table T2 with an aggregate result after the join. A very simple sketch of how this type of command would be processed in a parallel DBMS would consist of three phases [94].

1. T1 is stored and distributed by some nodes and partitioned by some attribute, the filter of the sub query runs first in parallel. Somehow, similar to what is done on a Map function of the Map-Reduce paradigm;

2. Afterwards a join algorithm is applied, based on the size of the data tables. For example if the number of records in the table T2 is smaller, then the DBMS can replicate it on all nodes when the data is read for the first time. Therfore it is possible that the join is executed in parallel on all nodes;

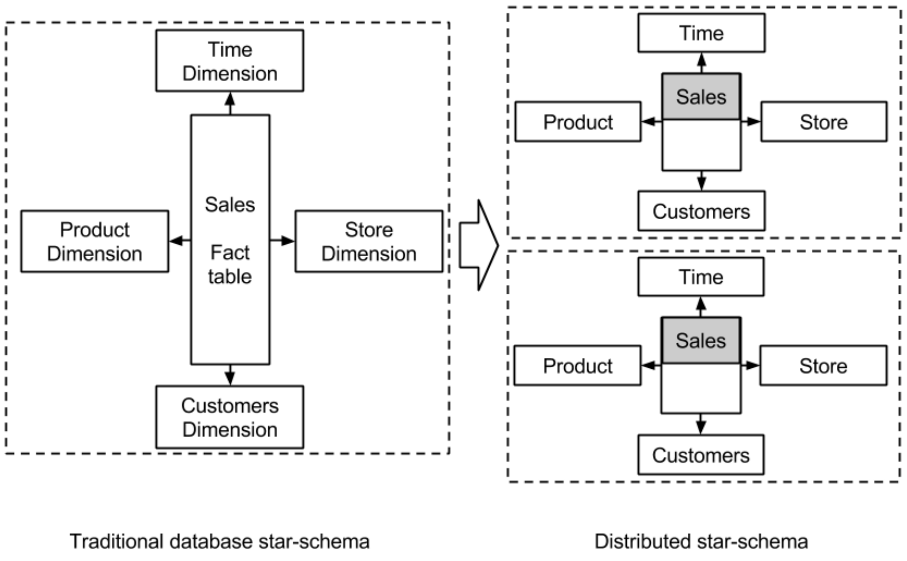


Figure A.1: Example of how to distribute data in a start schema.

3. Then, each node makes the computation of each aggregation using its piece of the intermediate result. A final step is necessary to put everything together and give the final answer.

The process of redistribution of data from tables T2 and/or T1 is similar to the process that occurs between the Map and Reduce tasks in architectures such as Hadoop Map-Reduce.

Figure A.1 shows how a simple start schema can be partitioned in such a way that it becomes highly scalable. While the dimension tables are replicated (because they have small size), the fact table(s) is distributed by the available nodes following known partitioning and placement strategies [95], [96], [97], [98], [99], [26], [15], [100]. Horizontal partitioning of data is also often used for scalability [101]. When scaling, tables are replicated or partitioned across all nodes. Other alternatives, available in previously cited works, can be considered to partition data. For instance all tables can be fully replicated by the data warehouse nodes, or partitioned using round-robin, or manually distributed. Queries must be decomposed, to run in the partitioned model and the results from each partition needs to be merged into the final result [77], [102], [103], [104], [96], [105], [106], [107], [108],[26]. Other approaches for scaling are such as based on complete de-normalization, hash-partitioning, range-partitioning or distributed file systems.

A.5 Common distributed architectures

Due to the large storage requirements and performance, big data warehouses (DW) find efficiency by parallelizing their systems.
There is a wide ranges of parallelizing architectures, including: share-nothing, share-disk, share-memory and hybrid models, which are used by state-of-the-art servers that contain many processors [27]. There are three large taxonomies of models as described in [105].

1. *Shared-Memory*: the architecture of shared memory or the "Shared-everything" is a system where all existing processes are shared between global memory addresses as well as other devices;

2. *Shared-Nothing*: the architecture in which nothing is shared is composed of multiple autonomous processing units, each with their storage units and carry copies of the DBMS. Communication between the processing nodes is done through messages passed over the network. Each of processing nodes can be composed of one or more processors and storage units;

3. *Share-Disk*: the architecture of shared-disk is characterized by having multiple processing units like the share-nothing architecture. However, in this case the architecture has a global storage unit that can be accessed by all nodes in the DBMS.

All architectures mentioned here have their advantages and disadvantages. The main advantage/disadvantage of greater importance is the scalability of the system vs. cost, fault tolerance and availability. Share-nothing and hybrid architectures can scale at a low cost (by simply adding more machines), and fault tolerance is achieved through replication.

A.6 Parallel Database Systems

Parallel database systems typically running on clusters of physical machines, are a popular choice for implementing high availability for database systems. A parallel (or clustered) database system executes database operations (e.g., queries) in parallel across a collection of physical machines or nodes (where a node can be either a computer, a core or a virtual machine). Each node runs a copy of the database management system. The data is either partitioned among the nodes where each partition is owned by a particular node or is shared by all nodes. Failure of individual nodes in the cluster can be tolerated and the data can remain accessible. Such high availability and fault-tolerance is a primary reason for deploying a parallel database system, in addition to improved performance. Nowadays, Parallel database architectures are mainly divided into shared nothing and data sharing.

In a shared nothing system, data is partitioned among nodes where each node hosts one or more partitions on its locally attached storage. Access to these partitions is restricted to the node that hosts the data and therefore, only the owner of a data partition can cache it in memory. This results in a simple implementation because data ownership is clearly defined and there is no need for access coordination.

Shared nothing database systems have been a huge commercial success because they are easier to build and are highly scalable [109]. Examples of shared nothing database systems include IBM DB2 [110], Microsoft SQL Server [111], and MySQL Cluster [112].

In a data storage sharing system nodes share access to all the data. Data is usually hosted on shared storage (e.g., by using Storage Area Networks).

Since any node in the cluster can access and cache any piece of data in its memory, access coordination is needed in order to synchronize access to common data for updates. This requires distributed locking, cache coherence, and recovery protocols which add to the complexity of a data sharing system. However, data sharing systems do not require a database to be partitioned and thus have more flexibility in doing load balancing.

A.7 ETL parallelism

Data produced by sources is extracted by external data sources an integrated into a common staging areas for transformation and cleansing before being loaded into the data warehouse. This process known as the ETL.
A vast number of commercial ETL tools available: IBM with InfoSphere server [113]; Oracle builder [114]. Nevertheless, in-house ETL development is preferred in many projects regarding data warehouse, because of the involved costs of purchasing and maintain such ETL systems. According with [115] when using such tools, ETL development time takes up to 70% or 80% of the development time in a data warehouse project.
Data warehouses are typically assembled from a variety of data sources with different formats and purposes. As such, ETL is a key process to bring all the data together in a standard, homogeneous environment.
Some ETL systems have to scale to process terabytes of data to update data warehouses with tens of terabytes of data. Increasing volumes of data may require designs that can scale from daily batch to multiple-day micro batch for integration with message queues or real-time changing data for continuous transformation and update.
ETL applications can implement different types of parallelism such like:

- Data: By splitting a single sequential file into smaller data files to provide parallel access.

- Pipeline: Allowing the simultaneous running of several components on the same data stream. For example: looking up a value on record 1 at the same time as adding two fields on record 2.

- Component: The simultaneous running of multiple processes on different data streams in the same job, for example, sorting one input file while removing duplicates on another file.

A.8 Virtualization

Machine virtualization is a technique that allows the resources of a physical machine to be shared among multiple partitions known as virtual machines (VMs). Each virtual machine runs an independent operating system and the associated set of applications. A virtual machine monitor manages the physical resources and provides a mapping between the physical resources and the abstractions known as virtual devices. A virtual device corresponding to each physical resource (e.g., CPU, disk, memory, and network) is exported to every virtual machine. Since a virtual machine is nearly an exact replica of the underlying hardware, this makes it possible to run applications unmodified inside a virtual machine. Xen [116] and VMware [117] are popular examples of virtual machine monitors.
Some benefits of virtualization include:

- Performance isolation – each virtual machine will use only its allocated share of resources, thus will not affect the performance of other virtual machines running on the same physical machine.

- Fault isolation – a software bug or a security flaw in one virtual machine does not affect other virtual machines.

- Dynamic resource allocation – the share of physical resources allocated to each virtual machine can be adjusted on the fly.

- Live migration – a virtual machine running on one physical host can be migrated to another physical host with minimum downtime.

A.9 Map-Reduce paradigm

Map-Reduce is Google's and Yahoo's distributed solution for web-scale data processing [93]. The major purpose is to automatically run parallel jobs in a cluster.

Programs are written as map functions that read individual items in the original data set and produce an intermediate tuple, then a reduce function merges these intermediate tuples into the final output. All map operations can run in parallel, after the map phase completed.

Map Reduce employs basic strategies for load balancing and fault tolerance, respectively:

1. Workers are assigned by map and reduce tasks as quickly as they complete them using a "greedy" strategy, leading to dynamic load balancing. Therefore, slow machines will be assigned with less work as they complete their tasks more slowly and also faster machines will be assigned with more work.

2. Map-Reduce uses the file system (e.g. Google file system (GFS), Hadoop file system (HFS)) to distribute the original data and intermediate results among the cluster machines. By doing that it is assured that, in case of a node failure the data is still available for other worker to resume the task.

Map function reads data from an input file and makes the desired filtering and/or processing of data. Then a function "split" partitions the records into R disjoint groups, separating them by applying a key function to each record. This function is typically a hash function. Each group of data generated by the Map ends with the production of R output files, one for each data group.

In general, there are multiple instances of map function to run on different nodes in a cluster. The term instance is used to refer to a single invocation of the function or the Map-Reduce. Each instance of Map is allocated to an input portion by the scheduler MR for processing. If there are M different parts of the input file, then there will be R files stored on disk for each of the M instances of map, thus creating a total of M x R files. Note that all functions map must use the same hash function, so all output records have the same hash value in the same location of the input.

In the second phase of implementation of the MR there are R, reduce instances, where R is typically the number of nodes. The files at this stage are transferred through the network from the nodes that have done the Map. Note that once again all records of the Map output stage with the same hash value are consumed by the same instance, regardless of which one produced the Map. Each Reduce process will combine the data assigned to somehow, and then write them to an output file (in the Distributed File System), which will be a part of the final output.

The input data exist as a collection of one or more partitions in the distributed file system. It is the MR scheduler responsibility to decide how many instances of the map function should run and how they allocate the available nodes. Similarly, the scheduler must also decide on the number and location of nodes that will run the reduce functions. The central controller of the MR is thus responsible for coordinating the activities of each map and reduces functions.

A.10 Map-Reduce limitations

When it comes to processes large amounts of data, MR model is well suited for development environments with a small number of programmers and limited field of applications. The absence of restrictions can make this paradigm inappropriate for large-scale and long term projects, since like everything has to be programmed and there is a high flexibility, which leads to great difficulties to reuse code.

Other difficulty is related with the transfer of data between Map and Reduce functions. Recall that N instances of map functions produce M output files, each one destined to a reduce instance. These files are written locally on disk for each node that runs the Map function. If N is 1000 and M is 500, the operation produces 500.000 map local files. When the Reduce stage starts, each one of the 500 Reduce instances need to read 1000 input files, and have to get them from the Map nodes that created them. With hundreds of reduce instances running simultaneously, it is inevitable two or more reduce functions try to read the same file from the same node, generating large amounts of I/O on disk (disk seek) and thus increasing the transfer time.

Another issue is related with the widespread use of network, because beyond normal information flux, and network spikes related to when reduce functions start. Also the redundancy copies must be taken into account to assure the fault tolerance, which will contribute in increasing network traffic.

The limitations of Map-Reduce architecture is leading many researches to study better the optimization in this architecture [118]. In [119] the authors focus on optimization to the HDFS and fault tolerance replicas, for faster upload and access to data, improving this way the execution of queries and indexation creation with third part middlewares.

A.11 No-Sql paradigm

NoSQL is a term for database management systems where it is not required a fixed schema to storage data. Usually NoSQL avoids operations like joins and transactions, and therefore it is able to scale horizontally. Thus by avoiding transactions there are no locks, by forbidding joins data will easily distribute and by many nodes all data is stored and indexed based in a key value, making it easy to distribute the keys and data across many nodes. Researchers usually refer to these databases as structured storage [120] [121]. While traditional database systems are designed for generic workloads and support large (and growing) features sets, NoSQL is designed to solve specific problems, and trade features (e.g. joins, transactions) for performance. No joins will mean that the need for complex indexes is reduced. The chances of index/query mismatches are lower. Disk I/O is less complex and therefore much faster.

NoSQL is a very oriented system, designed to serve heavy workloads like, indexing documents; serving pages; high-traffic websites; deliver streaming media [122]. In real-world NoSQL deployments include Digg's 3TB, Facebook's 50TB for inbox search and eBay 2PB for overall data.

The provided architecture has weak consistency guarantees such as eventual consistency, or transactions restricted to single data items. Some systems however provide full ACID guarantees in most cases by adding supplementary middleware layers (e.g. CloudTPS). Several NoSQL systems use distributed architectures with the data replicated on different servers, by using distributed hash tables, so the system can scale out by adding more servers and the failure of a server can be tolerated [123] [120].

Although NoSQL is very scalable and fast, the secret of NoSQL is not about eliminating SQL from the databases systems.To be precise , systems like Bigtable, HBase, Hypertable, Cassandra, Dynamo, SimpleDB and others based on key-value store, are mostly concerned with scalability. However, it turns out to be difficult to scale traditional ACID compliant relational database systems on shared-nothing architectures. Thus these systems decide to drop some of the ACID guarantees in order to be able to achieve shared-nothing scalability giving the option to the application developer the responsibility and complexity of programming a non-ACID system. So in

this way, NoSQL really means NoACID.
In order to NoSQL achieve scalability and high availability ACID characteristics were weakened in two ways:

- Systems like Bigtable, SQL Azure, and key-value stores support atomicity and isolation only when each transaction accesses data within a subset of the database (a single tuple in Bigtabel, or a single partition in SQL Azure). This removes the need for expensive distributed protocols at the cost of losing atomicity and isolation in transactions.
- Weak replication schemas contribute in the sacrifice of consistency. The main objective is to reduce data write and increase network performance.

In the end the programmer must implement any additional ACID functionality at the application level.

A.12 DBMSs Vs. Map-Reduce

Parallel DBMSs require data to be entered into a structure consisting of a relational model of rows and columns. In contrast, the MR data model does not require data (files) to comply with a fixed relational model. Thus the programmer using an MR model is free to use (program) structure as desired, or even not having a data structure. The absence of a rigid data model automatically makes the MR a preferable option. For example, in SQL the programmer has to specify the data schema. Moreover in MR the programmer writes his own parser, which is at least equivalent to a working definition of the data schema. But there are other potential problems of not using a data schema for large amounts of information.
Whatever the structure that exists in the MR and independent of the implementations provided for controlling keys and values, the programmer has to

explicitly write support for more complex data structures such as indexes, composite keys, and constraints. If there is a second developer sharing the application, he will have to first decode the code that was written by the previous programmer. Therefore the approach taken by the DBMSs using SQL and separating the schema is clearly more advantageous.

Even taking into account that for instance in the MR the schema of data is separate from the application, like in SQL based DBs, developers would have to agree on a single model schema. Obviously this requires a commitment to some kind of data model. In a way this is happening. Slowly, some languages begun to appear for the MR in the style of SQL, to optimize the processes of implementation, as is the case of Yahoo's Pig Latin [124] or Hive [33]. Yet, because they do not require fixed structures, there is the issue of violation of restrictions (constraints), which can be easily violated due to the freedom of defining structures (e.g. employees salaries cannot be negative). Thus in order for the DBMSs to separate the application of such restrictions, they allow automatically definition of data integrity rules in the schema without taking on additional programming work.

Typically, the MR tool is more sophisticated than parallel DBMSs with regard to fault tolerance. Although both systems use a particular type of replicas to handle failures, the MR is more efficient dealing with those failures. In an MR system, if a work unit fails, then the MR scheduler can automatically restart the task in an alternative node. Part of this flexibility arises from the fact that the output files from the Map phase are always materialized locally instead of being transferred to the reduce task. This flexibility of the MR is possible due to the creation (typically by default) of three replicas for each file, supported mainly by the distributed file system. This process of fault tolerance is quite different from that used in parallel DBMSs, which have large amounts of work (e.g. transactions) which take time to run, and in the event of any failure have to be restarted. Part of the reason for this approach is because DBMSs avoid saving intermediate results to disk whenever possible (avoid intermediate materialization). Thus, if one node fails during a long query, all queries related must be completely restarted.

A.13 Current related industry and academic technology focus

There have been many efforts at the present time of many companies (e.g. Google, Yahoo, Vertica, ExaSol, Teradata, IBM DB2, Oracle, VoltDB) in order to find solutions for processing large-scale data. So there are emerging approaches such as Map-Reduce (MR) [93], vertical data models (e.g. Vertica) [125], memory based databases (e.g. VoltDB), with optimization strategies, such as multiple redundancy and general specializations of DBMSs parameters for analytic or transactional processing, in order to allow better performance optimization.
Current trends point to five major groups of models for processing data in parallel:

- Row-wise, traditional [126], where data is stored in the form of rows (e.g., Oracle, PostgreSQL, MySQL).
- Column-wise, where the data are all stored in the form of columns (e.g. Vertica, ParAccel, MonetDB, SadasDB).
- Map-Reduce, using on two simple functions over the MR sructure: one maps the data, the other gathers information and joins it [28], [127], [31], [93] (e.g. BigTable, Hadoop, Apache Spark).
- Main-memory based systems, which operate and depend mainly on memory rather than disk. The disk is then, in some cases, only used

to maintain consistency of ACID transactions and data durability (e.g., Oracle TimesTen, ExaSol, RAIMA, VoltDB).

- NoSQL, next generation of databases, mostly addressing some of the following points: being non-relational, distributed, open-source, horizontal scalable, schema-free, easy replication support, simple API, eventually consistent (not ACID), supporting huge data amounts, (e.g. Google's BigTable, Amazon's Dynamo, Apache Cassandra, Hypertable, HBase).

In addition to these models, usually there are combinations of two models named hybrids. The aim is to combine all the advantages of the best of each approach. Two examples of these models are HadoopDB, which is the junction of the MR and PostgreSQL, and also the new version of Vertica, which consists on Vertica with a MR layer on top. There are even hybrid models that arise from systems based solely on memory, and make use of the hard-drive to get better performance with large amounts of data, such as Altibase, which is widely used in the telecommunications industry.
Several commercial database systems have emerged that include Map-Reduce functionality into existing database systems, Greenplum transforms Map-Reduce code into a query plan that its proprietary distributed SQL engine can execute on existing SQL tables. AsterData implements something similar, where Map-Reduce functions can be loaded into the database and invoked from standard SQL queries in Aster's distributed engine.
There is also HadoopDB, which is not commercial that uses a middleware approach, simultaneously using PostgreSQL server in their database layer, but uses Hive and Hadoop for query transformation, job scheduling and replication [127].
Teradata, IBM DB2, MySQL Cluster, Oracle Real Application Cluster, Oracle Exadata, Teradata, generally use the traditional approaches to parallelization and clustering. In these systems there are several nodes connected

together, which are most often "share nothing", "share memory", or "share disk". Different nodes are in charge of managing different components and functions of the DB. The attributes that stand out are the use and existence of front-end managers, load balancers and optimizers, which together manage the way information is distributed through the nodes, accessed, handled and processed. This type of system is commonly used in large businesses, but new technologies tend to change this. New models of data storage and new processing methods are emerging, and promise to bring great advantages in terms of performance and scalability.

Recent DBMSs such as BigTable and Hadoop raised interest in the Map-Reduce paradigm due to its flexibility in processing data in a distributed and parallel mode (with replicas of data) across multiple nodes. However, as is shown in studies of the area [28], the MR has performance problems when compared with column-oriented DBMSs such as Vertica, and even with row oriented DBMSs.

There is also the question of better programming model, MR vs. SQL. Some [28] argue that the MR is more flexible (everything is implemented by the programmer) and more failure tolerant. While in DBMSs, there are schemas for structuring data, and SQL is used, which generates easy implementation, code sharing (e.g. portability), optimization and organization/changes in data models (schema).

Parallel columnar DBMSs, such as Vertica or ParAccel, have performed extremely well when compared both with MR and with row oriented DBMS [28] [127]. In order to bridge the gap between the MR and the DBMS, emerging studies in the field involve the use of hybrid models, which make use of the best technologies of each model, such as HadoopDB, AsterData, HyperTable, Hyrise. Thus it is possible, for example, to have the fault tolerance of MR, among other advantages, and simultaneously to use SQL language and schemas.

In the case of a HadoopDB, studies show that the use of PostegreSQL with Hadoop, although in general not performing better than vertical or row models, improves results when compared with Hadoop Map-Reduce [127].
With the emergence of hybrid models and prototypes of academic success (e.g. HadoopDB, MonetDB, Hyrise), Vertica announced it is preparing a new version of their DBMS (Vertica Analytic Database 3.5 broadens reach with added FlexStore architecture) that, among other technologies, also integrates a layer of MR, offering the flexibility to process large amounts of data at high speeds, and offering support for SQL and data models. Also, Oracle has already prepared a version (Oracle 11gR2) of a DBMS that supports column-orientation, which promises to greatly improve performance. Oracle's future plans involve a slow and transparent replacement of "old" row-wise by the new column-wise engines. Researchers in the field point out that the old DBMSs are obsolete and that the column-oriented DBMSs may replace the row-oriented DBMSs [128].
ExaSol, VoltDB, and others are engines based on memory and oriented to columns, obtaining these way performance improvements over DBs such as Oracle, Microsoft, ParAccel, at a lower or similar cost. Some of these types of systems use special customized hardware; the data processing is done in clusters primarily using memory, in which case the disk is only used to ensure data persistence (ACID).

Appendix B

Continuous results queries and data schema

This appendix explains the data set used to process Call Detailed Records (CDRs) used for continuous query results processing.

B.1 Data set

The data set is composed of CDRs containing the following information regarding clients: towerID, accountCode, timestamp, callerNumber, calledNumber, callerID, duration, billSec, progressSec, progressMedSec, flowBillSec, mDuration, billmsec, progressMsec, progressMedMsec, flowBillMsec, uDuration, cellCompany, planType, roaming, relativeDistance, callStatus, sms, mms, dataMB, dataGB, timestamp, typeOfData, deviceModelID, comunicationType, comSize, GPS.

The data set also contains the following information regarding location: country, city, county, towerID, timestamp, hour, min, sec, msec, day, month, year, GMT.

In what regards towers, the data set contains the following information: towerID, country, city, county, address, GPS, timestamp, GMT, temperature, workTime, avgLoad, freeLoad, usedLoad, maxLoad, minLoad, dataUsage, smsUsage, clientsConnected, networks, frequency, energy, powerStatus, maintenaceHour, avgConnectionTime.

Regarding the data warehouse, it is designed as a star schema, which includes: all clients information, accounts information, cell phones models, bills, detailed invoices, cell phone plan types, statistical clients info, clients usage logs, and past statistical data.

B.2 CEP queries

Light queries **(Ql)** consist of queries that make arithmetic calculus, all in memory, regarding global key parameters of the CDRs (i.e. calls duration, average duration, total minutes on use, average incoming, so on). No join filters are included. No specific orders of output and no grouping of data is considered. The considered data window was 300 sec, and each output was set to be done in periods of 10 seconds.

Listing B.1: Light queries

```
//Average duration of all calls
select sum(duration), avg(duration)
from CDR.win:length(300)
output last every 10 second

//Average duration of calls by user and order by user name
select callerID , sum(duration), avg(duration)
from CDR.win:length(300)
group by callerID
output last every 10 second
order by callerID desc
```

```

//Average duration grouped by cell plan (50\% of calls)
select avg(duration), sum(duration)
from CDR.win:length(300)
where planType = 0
output last every 10 second
order by avg(duration) desc

//Average duration grouped by cell plan (25\% of calls)
select avg(duration), sum(duration)
from CDR.win:length(300)
where planType = 1
output last every 10 second
order by avg(duration) desc

//Average duration grouped by cell plan (25\% of calls)
select avg(duration), sum(duration) "
from␣CDR.win:length(300)
where␣planType␣=␣2
output␣last␣every␣10␣second
order␣by␣avg(duration)␣desc

//Towers␣average␣load
select␣towerID,␣count(callerID)
from␣CDR.win:length(300)
group␣by␣towerID
output␣last␣every␣10␣second

//Status␣of␣calls␣by␣provider,␣calls␣rejected
select␣callerID,␣count(callStatus)
from␣CDR.win:length(300)
where␣callStatus␣=␣0
group␣by␣callerID
output␣last␣every␣10␣second␣,

//Status␣of␣calls␣by␣provider,␣calls␣accepted
select␣callerID,␣count(callStatus)
from␣CDR.win:length(300)
where␣callStatus␣=␣1
group␣by␣callerID
output␣last␣every␣10␣second␣,

//SMS's by provider
select callerID , timestamp , sum(sms) , sum( billSec )
from CDR. win : length (300)
where sms = 1 AND cellCompany = 0
group by callerID
output last every 10 second
order by timestamp ,

//SMS's by provider
select callerID , timestamp , sum(sms) , sum( billSec )
from CDR. win : length (300)
where sms = 1 AND cellCompany = 1
group by callerID
output last every 10 second ,

//SMS's by provider
select callerID , timestamp , sum(sms) , sum( billSec )
from CDR. win : length (300)
where sms = 1 AND cellCompany = 2
group by callerID
output last every 10 second ,
```

Heavy in-memory queries **(Qh-m)**, process information regarding clients (i.e. total call duration, average calls duration, total to pay, number of messages, data usage, maximums and minimums). The results are grouped by client, ordered by name, plan type and phone number. The considered data window was 300 sec, and each output was set to be done in periods of 10 seconds. To achieve high cardinality of queries, we changed randomly the clients age interval range of each query. Results were kept on memory for comparison.

Listing B.2: Heavy in-memory queries

```
//Count number of sms's and average of sms by client
select callerID , sum(sms)
from CDR. win : length (300)
where sms = 1
```

```
group by callerID
output last every 10 second

//Count total of sms group by tower
select towerID , sum(sms)
from CDR.win:length(300)
where sms = 1
group by towerID
output last every 10 second

//Count total of sms group by planType
select planType , sum(sms)
from CDR.win:length(100)
where sms = 1
group by planType
output last every 10 second

//Client spending for calls
select callerID , sum(duration) * 0.20
from CDR.win:length(60)
where sms = 0 AND callStatus = 1
group by callerID
output last every 10 second

//Client for sms
select callerID , count(*) * 0.10
from CDR.win:length(60)
where sms = 1
group by callerID
output last every 10 second

//Client with max minutes
select callerID , towerID , planType , MAX(duration)
from CDR.win:length(200)
group by callerID
output last every 10 second
order by callerID desc , towerID ,
```

Heavy queries with access to database **(Qh-db)**. These queries are oriented to manage the clients billing for update of current recorded information (i.e. complete report on communications expenditure, data pack used, recommended plan type, most called numbers, maximums and minimum times). The database is used along side with the CEP processing to obtain extra information regarding, plan type, calls cost depending on the time and day, calls and data usage limits. Then, based on the processed information, further information inside the database is updated. This query also includes filters to group the output, filter and order the output results.

Listing B.3: Heavy queries with access to database

```
//SIM cards fraud based on distance between towers
//Alarms are logged into memory
select callerID , max(relativeDistance), min(relativeDistance)
from CDR.win:length(500)
group by callerID
having (max(relativeDistance) - min(relativeDistance)) > 5
output last every 10 second
order by callerID asc

//Insert calls into the Dynamic Data warehouse
//All calls are registered into the Dynamic Data Warehouse
select accountCode, avg(duration)
from CDR.win:length(180)
group by accountCode
output last every 1 second
order by accountCode desc,

//Get caller information from the data warehouse and
//dynamic data warehouse to check the available balance
select callerID , count(*) * 0.10
from CDR.win:length(180)
where sms = 1
group by callerID
output last every 10 second
order by callerID desc

//compare duration with last month duration
//data from the last month is retrieved from the data
```

```
//warehouse.
select callerID, sum(duration), avg(duration)
from CDR.win:length(200)
group by callerID
output last every 1 second
order by sum(duration) desc,

//get best recommended plan
//get client information form data warehouse
//to compare suggestions with current plan
select callerID, towerID, planType, max(duration), avg(duration)
from CDR.win:length(300)
where planType = 1 OR planType = 2
group by callerID, planType
having max(duration) <= 150
output last every 10 second
order by callerID
```

Heavy queries with in-memory lookups with updates **(Qh-mlu)**, regard the management of cell towers. These queries collect all information of each cell tower (i.e. total usage, available resources, operation temperature, used energy, number of clients connected, average time each client is connected, communications speed) and set it for update of in-memory structures. Filters for grouping towers and order them are set. The considered window of data was 300 sec, and each output was set to be done in periods of 5 seconds.

Listing B.4: Heavy queries with in-memory lookups with updates

```
//look up tower location and status into mem
//update in memory data regarding each tower
select towerID, towerLocation, ...
from CDR.win:length(300)
group by towerID
output last every 5 second ,
order by towerID desc

//total of money spent on calls
```

```
//update logs into memory
select callerID , avg(duration), sum(duration)
from CDR.win:length(300)
where planType = 2
group by callerID
output last every 5 second
order by callerID desc

//get cell company calls count
//update calls made by other companies
select cellCompany , count(*), sum(duration)
from CDR.win:length(300)
group by cellCompany
output last every 5 second
order by cellCompany

//Count tower use by cell company
//update tower load made by other companies
select towerID , cellCompany , count(*)
from CDR.win:length(300)
group by towerID , cellCompany
output last every 5 second
```

The queries used to demonstrate the degradation of performance **(Qdb-2)** consists on doing a full analysis of the clients to promote the best clients (e.g. for promotions and bonus) during a configurable time period. These queries include heavy join tasks in the database side, as well as updates and inserts both in the database and memory, which increase the processing weight. A part of join tasks also relate with the analysis of the friends (most called numbers) of each client, to offer them bonus. The considered window of data was 300 sec. and each output was set to be done in each 60 seconds.

Listing B.5: Heavy queries with in-memory lookups with updates

```
//Analyze best costumer for promotion
//Get from the data warehouse users profile and
//profile of the most called numbers. Notify each
```

```
//friend, logs are created for notifications.
//Update balance of costumers into memory.
select callerID, sum(duration), avg(duration), max(duration)
from CDR.win:length(300)
group by callerID
output last every 60 second
order by max(duration) asc,
```

Appendix C

Data warehouse queries

This appendix introduces the used queries to test the data warehouse performance and scalability. For more information regarding the used queries please consult the SSB Benchmark documentation [80].

Listing C.1: Query 1.1

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder , date
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between1 and 3
and lo_quantity < 25;
```

Listing C.2: Query 1.2

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder , date
where lo_orderdate = d_datekey
and d_yearmonthnum = 199401
and lo_discount between4 and 6
and lo_quantity between 26 and 35;
```

Listing C.3: Query 1.3

```sql
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_weeknuminyear = 6
and d_year = 1994
and lo_discount between 5 and 7
and lo_quantity between 26 and 35;
```

Listing C.4: Query 2.1

```sql
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Listing C.5: Query 2.2

```sql
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 between 'MFGR#2221' and 'MFGR#2228'
and s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Listing C.6: Query 2.3

```sql
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 = 'MFGR#2221'
and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

Listing C.7: Query 3.1

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_region = 'ASIA' and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Listing C.8: Query 3.2

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_nation = 'UNITED_STATES'
and s_nation = 'UNITED_STATES'
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Listing C.9: Query 3.3

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED_KI1'
or c_city='UNITED_KI5')
```

```
and (s_city='UNITED_KI1'
or s_city='UNITED_KI5')
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Listing C.10: Query 3.4

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED_KI1' or
c_city='UNITED_KI5')
and (s_city='UNITED_KI1' or
s_city='UNITED_KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

Listing C.11: Query 4.1

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

Listing C.12: Query 4.2

```
select d_year, s_nation, p_category,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
```

```
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1'
or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

Listing C.13: Query 4.3

```
select d_year, s_city, p_brand1,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_nation = 'UNITED_STATES'
and (d_year = 1997 or d_year = 1998)
and p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1;
```

Appendix D

Data warehouse indexes and views

This section presents the created indexes to optimize the data warehouse queries and created views, which were updated after each load process.

Listing D.1: Indexes part 1

```
CREATE INDEX L_ORDERKEY_idx ON LINEITEMORDER (L_ORDERKEY)

CREATE INDEX L_LINENUMBER_idx ON LINEITEMORDER (L_LINENUMBER)

CREATE INDEX L_KEYS_idx ON LINEITEMORDER (L_ORDERKEY, L_LINENUMBER,
L_CUSTKEY, L_PARTKEY, L_SUPPKEY)

CREATE INDEX L_LINENUMBER_L_ORDERKEY_idx ON LINEITEMORDER (L_LINENUMBER,
L_ORDERKEY)

CREATE INDEX L_CUSTKEY_idx ON LINEITEMORDER (L_CUSTKEY)

CREATE INDEX L_PARTKEY_idx ON LINEITEMORDER (L_PARTKEY)

CREATE INDEX L_SUPPKEY_idx ON LINEITEMORDER (L_SUPPKEY)

CREATE INDEX L_ORDERDATE_idx ON LINEITEMORDER (L_ORDERDATE)

CREATE INDEX L_ORDPRIORITY_idx ON LINEITEMORDER (L_ORDPRIORITY)
```

Listing D.2: Indexes part 2

```
CREATE INDEX L_SHIPPRIORITY_idx ON LINEITEMORDER (L_SHIPPRIORITY)

CREATE INDEX L_DISCOUNT_idx ON LINEITEMORDER (L_DISCOUNT)

CREATE INDEX L_SHIPMODE_idx ON LINEITEMORDER (L_SHIPMODE)

CREATE INDEX L_REVENUE_idx ON LINEITEMORDER (L_REVENUE)

CREATE INDEX L_ORDTOTALPRICE_idx ON LINEITEMORDER (L_ORDTOTALPRICE)

CREATE INDEX C_CUSTKEY_idx ON CUSTOMER (C_CUSTKEY)

CREATE INDEX C_NAME_idx ON CUSTOMER (C_NAME)

CREATE INDEX P_PARTKEY_idx ON PART (P_PARTKEY)

CREATE INDEX P_NAME_idx ON PART (P_NAME)

CREATE INDEX P_CATEGOTY_idx ON PART (P_CATEGOTY)

CREATE INDEX P_COLOR_idx ON PART (P_COLOR)

CREATE INDEX P_SIZE_idx ON PART (P_SIZE)

CREATE INDEX S_SUPPKEY_idx ON SUPPLIER (S_SUPPKEY)

CREATE INDEX S_CITY_idx ON SUPPLIER (S_CITY)

CREATE INDEX S_NATION_idx ON SUPPLIER (S_NATION)

CREATE INDEX S_REGION_idx ON SUPPLIER (S_REGION)

CREATE INDEX S_PHONE_idx ON SUPPLIER (S_PHONE)

CREATE INDEX S_ADDRESS_idx ON SUPPLIER (S_ADDRESS)

CREATE INDEX D_DATEKEY_idx ON DATETIME (D_DATEKEY)

CREATE INDEX D_DATEID_idx ON DATETIME (D_DATEID)

CREATE INDEX D_DATE_idx ON DATETIME (D_DATE)

CREATE INDEX D_MONTHSEQ_idx ON DATETIME (D_MONTHSEQ)

CREATE INDEX D_WEEKSEQ_idx ON DATETIME (D_WEEKSEQ)

CREATE BITMAP INDEX D_QUARTERSEQ_idx ON DATETIME (D_QUARTERSEQ)

CREATE INDEX D_WEEKDAY_idx ON DATETIME (D_WEEKDAY)

CREATE INDEX D_YEARMONTHNUM_idx ON DATETIME (D_YEARMONTHNUM)

CREATE INDEX D_WEEKNUMINYEAR_idx ON DATETIME (D_WEEKNUMINYEAR)

CREATE BITMAP INDEX D_HOLIDAY_idx ON DATETIME (D_HOLIDAY)

CREATE BITMAP INDEX D_WEEKEND_idx ON DATETIME (D_WEEKEND)
```

The created views for update are as follow.

Listing D.3: View 1.1

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder , date
where lo_orderdate = d_datekey
and lo_quantity < 25
```

Listing D.4: View 1.2

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder
Where lo_quantity between 26 and 35
```

Listing D.5: View 1.3

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder , date
where lo_orderdate = d_datekey
and lo_discount between 5 and 7
```

Listing D.6: View 2.1

```
select sum(lo_revenue), d_year , p_brand1
from lineorder , date, supplier
where lo_orderdate = d_datekey
and s_region = 'AMERICA'
```

Listing D.7: View 2.2

```
select sum(lo_revenue), d_year , p_brand1
from lineorder , date, supplier
where lo_orderdate = d_datekey
and s_region = 'ASIA'
```

Listing D.8: View 2.3

```
select sum(lo_revenue), d_year , p_brand1
from lineorder , date
where lo_orderdate = d_datekey
and s_region = 'EUROPE'
```

Listing D.9: View 3.1

```
select c_nation , d_year , sum(lo_revenue) as revenue
from customer , lineorder , date
where lo_custkey = c_custkey
and d_year >= 1992 and d_year <= 1997
```

Listing D.10: View 3.2

```
select c_city , s_city , sum(lo_revenue) as revenue
from customer , lineorder , supplier
where lo_custkey = c_custkey
and c_nation = 'UNITED_STATES'
and s_nation = 'UNITED_STATES'
```

Listing D.11: View 3.3

```
select c_city , s_city , d_year , sum(lo_revenue) as revenue
from customer , lineorder , supplier , date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and d_year >= 1992 and d_year <= 1997
```

Listing D.12: View 3.4

```
select c_city , s_city , sum(lo_revenue) as revenue
from customer , lineorder , supplier
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and (c_city='UNITED_KI1' or
 c_city='UNITED_KI5')
```

Listing D.13: View 4.1

```
select c_nation , sum(lo_revenue - lo_supplycost) as profit
from customer , supplier , part , lineorder
where lo_custkey = c_custkey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
```

Listing D.14: View 4.2

```
select s_nation , p_category
from customer , supplier , part , lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and p_category = 'MFGR#14'
```

Listing D.15: View 4.3

```
select d_year , s_city , p_brand1 , sum(lo_revenue - lo_supplycost) as profit
from date , customer , supplier , part , lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and s_nation = 'UNITED_STATES'
and (d_year = 1997 or d_year = 1998)
```

Appendix E

API specification

This appendix describes AScale API configuration for each part of the framework pipeline.

E.1 Data Sources

An AScale module ”data loader” must be configured in the ”data source” side to allow connection from AScale extraction module.

Listing E.1: AScale maximum data chunk size

```
aScaleSetMaxDataChunkSize(
50MB);   //maximum data chunk size
```

Listing E.1 shows the configuration of the maximum size of the data chunks that are transferred along the AScale modules.

Listing E.2: Data loader configuration

```
dataLoaderSetLog(
”log1”,            //log ID
"../home/,          //log storage path
"stock",            //log file name
"-");               //log file prefix before sequential numbering
```

Listing E.2, shows "Data Loader" configuration for AScale extraction. Data extraction is made following the file sequence (e.g. stock-0, stock-1, stock-2). To avoid read and write conflicts, a file n is only loaded when n+1 is created. After data extraction completed the file is deleted.

Listing E.3: Data loader max memory configuration

```
dataLoaderMaxMemory(
"1GB");
```

Listing E.3, shows the memory size configuration for the data loader nodes. This memory is used to read the available data into memory and make it available for extraction by the data sources.

E.2 Extraction nodes

Data is extracted from data sources using a default approach based on AScale scheduler extraction algorithm. A Scheduler module collects the available data size information of each data source. Then an extraction node (the one that is free to work) is selected to extract data from a specific data source.

Listing E.4: Extraction, logs configuration

```
extractSetDataSourceLog(
"source1",                              //data source ID
{"logA","logB"},                          //log files ID
{"* 1 * * * * *", "* 2 * * * * *"},       //extraction frequency
{"7h", "7h"});                            //maximum extraction time
```

Listing E.4 shows how to configure "data sources" data logs formats. Note that the extraction frequencies (e.g. every 5 minutes) are configured using the Unix crontab format, where the stars mean: second (0-59), minute (0-59), hour (0-23), day of month (1-31), month (1-12), day of week (0-6), year (1970-2099).

Listing E.5: Extraction data source format

```
extractSetDataSourceFormat(
"source1",      //source ID
"logA",         //source log file ID
"|",            //column separate character
"\n");          //row separate character
```

Listing E.5, shows the configuration of the data sources data format.

Listing E.6: Extraction policy, definition

```
extractSetExtractMode(
"schedulerBased");
```

Listing E.6, shows the definition of the default extraction policy to be applied to extract data from the data sources. Other policies can be applied such as manual or round-robin.

Listing E.7: Extraction policy, manual

```
extractSetExtractFrom(
"extractID",     //extraction node ID
"source1",       //data source to extract data from
"logA");         //data log to be extracted
```

Listing E.7 shows the required configuration for manual extraction policy.

Listing E.8: Extraction send policy configuration

```
extractSetSendPolicy(
"LWR");
```

Listing E.8, shows the configuration of the data policy used to send data from the extraction nodes into the transformation nodes. By default lest-work-remaining data distribution is used. However others can be used such as, manual data distribution, or round-robin data distribution.

Listing E.9: Extraction configure manual send policy

```
extractSetSendTo(
"extract1",             //extract node ID
"transformNode1");      //transformation node ID
```

Listing E.9 shows how to configure manual data distribution policy.

E.3 Transformation configuration

In this section we describe how to configure the the transformation nodes.

Listing E.10: Transformation, configuring queue size

```
transformSetMaxSize(
"16GB",          //maximum allowed memory size
"5GB",           //limit free memory size, before data swap into disk
"500GB");        //maximum disk storage size
```

Listing E.10, shows how to configure the data queue size.

Listing E.11: Transformation API get data

```
array[] = transformGetExtractData();
```

Listing E.11 shows AScale library and web service API to get data.

Listing E.12: Transformation API submit data

```
transformSetOutput(
"112|Pedro|30|Portugal\n", //transformed data
"|",                       //column separate character
"\n",                      //row separate character
"users");                  //data-warehouse schema corresponding table
```

Listing E.12 shows the API to set data back to AScale after transformations applied.

Listing E.13: Transformation data send policy

```
transformSetSendPolicy(
"LWR");
```

Listing E.13 shows how to configure the data distribution policy from the transform nodes into the data buffer nodes. By default LWR is applied.

However other policies can be applied such as manual data distribution, or round-robin.

Listing E.14: Transformation, manual distribution policy

```
transformSetSendPolicy(
"trasnform1",            //transform node ID
"dataBufferNode1");      //destination data buffer ID
```

Listing E.14, shows how the manual data distribution policy can be configured.

E.4 Data buffer

The "data buffer" functions as a data storage to hold recent transformed data until the next load instant.

Listing E.15: Data Buffer size configuration

```
dataBufferSetSize(
"dataBuffer1",  //data buffer Id name
"10GB",                  //maximum memory size
"5GB",                   //limit memory size before data swap
"1TB",                   //maximum disk size
"D:")                    //disk storage location
```

Listing E.15 shows the configuration parameters to set the data buffer sizes that will allow it to scale if necessary.

E.5 Data Switch

Data switch nodes extract data from the data buffers and set it into the data warehouse nodes to be loaded. Note that by default the extraction policy from the "data buffer" nodes is commanded by the scheduler.

Listing E.16: Data Switch maximum supported data-rate

```
dataSwitchSetDataRate(
"dataSwitch1",  //data switch Id name
"80000l/s"      //maximum supported data-rate
"5m");          //maximum allowed time at the maximum data rate
```

Listing E.16 shows the data switch configuration regarding the maximum supported data-rate.
If this value (data-rate) is reached for the duration of a certain time frame, AScale scales-out the data switch nodes.

Listing E.17: Data Switch replication configuration

```
dataSwitchSetSchemaReplication(
"nation",       //data-warehouse schema table
true);          //true - for data replication by all nodes
```

Listing E.17 shows the replication configuration command line. The developer must specify which tables, present in the data warehouse schema, should be replicated.

Listing E.18: Data Switch extraction policy configuration

```
dataSwitchSetExtractPolicy(
"schedulerBased");
```

Listing E.18 shows the data switch data extraction configuration API. By default a scheduler based extraction is applied. However others can be configured, "round-robin", "manual", "LWR" from the data buffers nodes holding more data.

Listing E.19: Data Switch manual extraction policy configuration

```
dataSwitchSetExtractFrom(
"dataSwitch1",            //data switch ID
"buffer1");               //data buffer ID to extract data from
```

Listing E.19 shows the manual data extraction configuration parameters.

Listing E.20: Data Switch manual load policy configuration by table referencing

```
dataSwitchSetManualLoadPolicyByTable(
"nation",        //data warehouse schema table name
"DWnode1");      //data warehouse destination node ID
```

Listing E.20 shows the data load policy configuration to select specific data to be stored into specific data warehouse nodes.

E.6 Data warehouse

Listing E.21: Data warehouse connection

```
dataWarehouseSetConnection(
"oracle",        //vendor engine ID
"AScaleDB"       //database name
"orcl"           //database user name
"12345");        //database password
```

Listing E.21 shows the connection configurations for the database.

Listing E.22: Data warehouse load configuration

```
dataWarehouseSetLoad(
"* 30 1 * * *", //load frequency (UNIX cron tab format)
"5h",            //maximum load time
"100MB");        //maximum batch file size
```

Listing E.22 shows, the data warehouse load configuration.

Listing E.23: Data warehouse schema configuration

```
dataWarehouseSetSchema(
"/.../script");
```

Listing E.23 shows the data warehouse schema configuration. The schema must be submitted in standard SQL format. The indexes, views and key identifiers must be defined. All nodes have the same schema.

Listing E.24: Data warehouse pre-load and post-load configurations

```
dataWarehouseSetPreLoadTasks(
"../*/preLoad.sql");

dataWarehouseSetPostLoadTasks(
"../*/posLoad.sql");
```

Listing E.24 shows the pre-load and post-load configurations to be applied before loading data and after the loading process. A set of tasks can be set to be performed, such as: destroy indexes, remove/add/clean tables, create/destroy views.

E.7 Queries

Queries can execute in the data warehouse and/or dynamic data warehouse. For all queries a maximum acceptable execution time is considered. If this time is not fulfilled the data warehouse nodes scale-out.

Listing E.25: Maximum query execution time configuration

```
querySetMaxDWQueryExecutionTime(
value); //maximum desired execution time

querySetMaxD-DWQueryExecutionTime(
value); //maximum desired execution time
```

Listing E.25 shows the API to configure the maximum query execution time.

Listing E.26: Query execution

```
array[] = querieSetRun(
"SELECT * FROM customer", //query in SQL format
"DW");            //data warehouse to use "DW" or "D-DW" or both
```

Queries can run at the DW and/or D-DW.

Output parameters: an array in text format with the query results.

E.8 Freshness

The specification of the API to configure and support fresh results is as follows.

Listing E.27: Configuring the dynamic data warehouse buffer queue

```
dynamicDWSetScale(
"10GB",            //maximum queue size
"1GB");            //limit size before scaling-out
```

Listing E.27 shows the configuration command to set the maximum dynamic-data-warehouse memory size and the limit memory size to trigger the scaling mechanisms.

Listing E.28: Configuring the dynamic data warehouse load size

```
dynamicDWSetLoad(
"batch",           //load method
"10MB");           //the number of lines or size

dynamicDWSetLoad(
"row-by-row",      //load method
10);               //the number of lines or size
```

Listing E.28 shows the two different method supported at the dynamic-data-warehouse to load data, batch based or row-by-row.

Listing E.29: Configuring full availability, 24/7

```
loaderSet24Replication(
true/false);
```

Listing E.29 shows the configuration to achieve full availability, 24/7 query answering can be achieved by replication of the data warehouse, and D-DW nodes. When data is being loaded into one node the other is used to answer queries, and vice-versa.

Appendix F

XML Configuration

In this appendix we show an example of a possible XML configuration file.

Listing F.1: XML configuration file sample

```
<XML>

<maxDataChunkSize>50MB</maxDataChunkSize>

<DataSource>

        <ID>source1</ID>
        <Location>192.168.1.1:1234</Location>
        <MaxMem>500MB</MaxMem>
        <columnCharacter>|</columnCharacter>
        <lineCharacter>\n</lineCharacter>

        <DataSourceLog>
                <name>logA</name>
                <baseFileName>stock</baseFileName>
                <sequenceSeparateCharacter>_</sequenceSeparateCharacter>
                <frequency>*/10 * * * * * *</frequency>
                <maxDuration>5s</maxDuration>
        </DataSourceLog>

        <DataSourceLog>
                <name>logA</name>
                <baseFileName>sales</baseFileName>
                <sequenceSeparateCharacter>_</sequenceSeparateCharacter>
                <frequency>*/10 * * * * *</frequency>
                <maxDuration>5s</maxDuration>
        </DataSourceLog>

</DataSource>

<Extraction>

        <ExtractPolicy>schedulerBased</ExtractPolicy>
        <SendPolicy>LWR</SendPolicy>

</Extraction>

<Transformation>

        <MaxMem>10GB</MaxMem>
        <limitMem>5GB</MaxMem>
        <SendPolicy>LWR</SendPolicy>

</Transformation>

<DataBuffer>

        <ID>DataBuffer1</ID>
        <MaxMem>10GB</MaxMem>
        <swapMemLimit>5GB<swapMemLimit>
        <MaxDisk>1TB</MaxDisk>
        <location>../home/</location>
        <limitMem>5GB</MaxMem>

</DataBuffer>

<DataSwitch>

        <performance>
                <ID>dataSwitch1</ID>
                <maxDataRate>80000</maxDataRate>
                <maxDataRateTime>2m</maxDataRateTime>
        </performance>

        <ExtractMode>schedulerBased</ExtractMode>

        <replicate>
                <name>nation</name>
                <name>time</name>
                <name>clients</name>
        </replicate>

        <distribute>
                <name>lineitem</name>
        </distribute>

</DataSwitch>

<DataWarehouse>

        <frequency>* 30 1 * * *</frequency>
        <maxDuration>5h</maxDuration>
        <batchSize>100MB</batchSize>
        <schema>../home/schema.sql</schema>
        <preLoad>../home/preLoad.sql</preLoad>
        <posLoad>../home/posLoad.sql</posLoad>

        <connection>
                <ID>dw1</ID>
                <dbEngine>oracle</dbEngine>
                <db>AScaleDB</db>
                <usename>orcl</username>
                <pass>12345</pass>
        </connection>

</DataWarehouse>

<query>

        <maxDurationDW>5m</maxDurationDW>
        <maxDurationD-DW>1m</maxDurationD-DW>

</query>

</XML>
```

Bibliography

[1] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 2007.

[2] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[3] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[4] Bert Scalzo. *Oracle DBA guide to data warehousing and star schemas*. Prentice Hall Professional, 2003.

[5] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos Sellis. View selection for designing the global data warehouse. *Data & Knowledge Engineering*, 39(3):219–240, 2001.

[6] Alkis Simitsis, Panos Vassiliadis, and Timos Sellis. State-space optimization of etl workflows. *Knowledge and Data Engineering, IEEE Transactions on*, 17(10):1404–1419, 2005.

[7] Alkis Simitsis, Panos Vassiliadis, and Timos Sellis. Optimizing etl processes in data warehouses. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 564–575. IEEE, 2005.

[8] Alexander Albrecht and Felix Naumann. Metl: Managing and integrating etl processes. In *VLDB PhD workshop*, 2009.

[9] Anastasios Karagiannis, Panos Vassiliadis, and Alkis Simitsis. Scheduling strategies for efficient etl execution. *Information Systems*, 38(6):927–945, 2013.

[10] Guiteng Wang and Chaozhen Guo. Research of distributed etl engine based on mas and data partition. In *Computer Supported Cooperative Work in Design (CSCWD), 2011 15th International Conference on*, pages 342–347. IEEE, 2011.

[11] Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen. Mapreduce-based dimensional etl made easy. *Proceedings of the VLDB Endowment*, 5(12):1882–1885, 2012.

[12] Christian Thomsen and Torben Bach Pedersen. pygrametl: A powerful programming framework for extract-transform-load programmers. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 49–56. ACM, 2009.

[13] Xiufeng Liu. *Data warehousing technologies for large-scale and right-time data.* PhD thesis, dissertation, Faculty of Engineering and Science at Aalborg University, Denmark, 2012.

[14] Alkis Simitsis, Chetan Gupta, Song Wang, and Umeshwar Dayal. Partitioning real-time etl workflows, 2010.

[15] Panos Vassiliadis and Alkis Simitsis. Near real time etl. In *New Trends in Data Warehousing and Data Analysis*, pages 1–31. Springer, 2009.

[16] Pentaho. Pentaho, 2014-10-07. URL `www.pentaho.com`.

[17] Mahdi Abdelguerfi and Kam-Fai Wong. *Parallel database techniques.* IEEE Computer Society Press, 1998.

[18] Hongjun Lu, KL Tan, and Beng-Chin Ooi. *Query processing in parallel relational database systems*. IEEE Computer Society Press, 1994.

[19] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

[20] Pedro Furtado. Model and procedure for performance and availability-wise parallel warehouses. *Distributed and Parallel Databases*, 25(1-2): 71–96, 2009.

[21] Pedro Furtado. Efficient and robust node-partitioned data warehouses. page 203. IGI Global, 2007.

[22] Ladjel Bellatreche, Michel Schneider, Hervé Lorinquer, and Mukesh Mohania. Bringing together partitioning, materialized views and indexes to optimize performance of relational data warehouses. In *Data Warehousing and Knowledge Discovery*, pages 15–25. Springer, 2004.

[23] Ladjel Bellatreche, Kamel Boukhalfa, and Pascal Richard. Data partitioning in data warehouses: Hardness study, heuristics and oracle validation. In *Data Warehousing and Knowledge Discovery*, pages 87–96. Springer, 2008.

[24] Ladjel Bellatreche, Rima Bouchakri, Alfredo Cuzzocrea, and Sofian Maabout. Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 208–210. ACM, 2013.

[25] Raquel Almeida, Jorge Vieira, Marco Vieira, Henrique Madeira, and Jorge Bernardino. Efficient data distribution for dws. In *Data Warehousing and Knowledge Discovery*, pages 75–86. Springer, 2008.

[26] Jorge Bernardino and Henrique Madeira. Experimental evaluation of a new distributed partitioning technique for data warehouses. In *Database Engineering and Applications, 2001 International Symposium on.*, pages 312–321. IEEE, 2001.

[27] Pedro Furtado. A survey of parallel and distributed data warehouses. *International Journal of Data Warehousing and Mining (IJDWM)*, 5 (2):57–77, 2009.

[28] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. pages 165–178, 2009.

[29] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1): 64–71, 2010.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. USENIX, 2012.

[32] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD Record*, 40(4):11–20, 2012.

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham

Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[34] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[35] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[36] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[37] Li Chen, Wenny Rahayu, and David Taniar. Towards near real-time data warehousing. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 1150–1157. IEEE, 2010.

[38] M Nguyen Tho and A Min Tjoa. Zero-latency data warehousing for heterogeneous data sources and continuous data streams. In *5th International Conference on Information Integrationand Web-based Applications Services*, pages 55–64, 2003.

[39] Youchan Zhu, Lei An, and Shuangxi Liu. Data updating and query in real-time data warehouse system. In *Computer science and software engineering, 2008 international conference on*, volume 5, pages 1295–1297. IEEE, 2008.

[40] Florian Waas, Robert Wrembel, Tobias Freudenreich, Maik Thiele, Christian Koncilia, and Pedro Furtado. On-demand elt architecture for right-time bi: Extending the vision. *International Journal of Data Warehousing and Mining (IJDWM)*, 9(2):21–38, 2013.

[41] Janis Zuters. Near real-time data warehousing with multi-stage trickle and flip. In *Perspectives in Business Informatics Research*, pages 73–82. Springer, 2011.

[42] Oracle. Oracle data integrator, 2014. URL `http://goo.gl/cFCnu3`.

[43] Oracle. Oracle goldengate, 2014. URL `http://goo.gl/i2AODL`.

[44] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[45] Umar Farooq Minhas. *Scalable and Highly Available Database Systems in the Cloud*. PhD thesis, University of Waterloo, 2013.

[46] Raul Castro Fernandez, Peter Pietzuch, Joel Koshy, Jay Kreps, Dong Lin, Neha Narkhede, Jun Rao, Chris Riccomini, and Guozhang Wang. Liquid: Unifying nearline and offline big data integration. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 01/2015 2015. ACM, ACM.

[47] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[48] Justin Langseth. Real-time data warehousing: Challenges and solutions. *DSSResources. com*, 2(08):2004, 2004.

[49] Alfredo Cuzzocrea, Nickerson Ferreira, and Pedro Furtado. Real-time data warehousing: A rewrite/merge approach. In *Data Warehousing and Knowledge Discovery*, pages 78–88. Springer, 2014.

[50] Nickerson Ferreira, Pedro Martins, and Pedro Furtado. Near real-time with traditional data warehouse architectures: factors and how-to. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 68–75. ACM, 2013.

[51] Janis Zuters. Near real-time data warehousing with multi-stage trickle and flip. In *Perspectives in Business Informatics Research*, pages 73–82. Springer, 2011.

[52] Tanvi Jain, S Rajasree, and Shivani Saluja. Refreshing datawarehouse in near real-time. *International Journal of Computer Applications*, 46, 2012.

[53] Nickerson Ferreira and Pedro Furtado. Real-time data warehouse: a solution and evaluation. *International Journal of Business Intelligence and Data Mining*, 8(3):244–263, 2013.

[54] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *Proceedings of the 2008 international symposium on Database engineering & applications*, pages 49–58. ACM, 2008.

[55] David Luckham. *The power of events: An introduction to complex event processing in distributed enterprise systems.* Springer, 2008.

[56] Thomas Heinze. Elastic complex event processing. page 4, 2011.

[57] ESPER Complex Event Processing. Esper complex event processing. URL `http://esper.codehaus.org/`.

[58] I StreamBase. Streambase: Real-time, low latency data processing with a stream processing engine, 2006.

[59] CEP Oracle. Oracle cep homepage, 2008.

[60] Neal Leavitt. Complex-event processing poised for growth. *Computer*, 42(4):17–20, 2009.

[61] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. An overview of business intelligence technology. *Communications of the ACM*, 54(8):88–98, 2011.

[62] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[63] Theodore Johnson, S Muthukrishnan, and Irina Rozenbaum. Monitoring regular expressions on out-of-order streams. In *ICDE*, pages 1315–1319, 2007.

[64] Anirban Majumder, Rajeev Rastogi, and Sriram Vanama. Scalable regular expression matching on data streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 161–172. ACM, 2008.

[65] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

[66] RuleCore. Rulecore complex event processing. URL `http://www.rulecore.com/`.

[67] Adrian Paschke. Design patterns for complex event processing. *arXiv preprint arXiv:0806.1100*, 2008.

[68] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. Gates: a grid-based middleware for processing distributed data streams. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 192–201. IEEE, 2004.

[69] Alan J Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, volume 7, pages 412–422, 2007.

[70] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[71] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 4. ACM, 2009.

[72] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[73] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: the stanford stream data manager (demonstration description). pages 665–665, 2003.

[74] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[75] Andrey Khorlin and K Mani Chandy. Control-based scheduling in a distributed stream processing system. In *Services Computing Workshops, 2006. SCW'06. IEEE*, pages 55–64. IEEE, 2006.

[76] Ahmed M Aly, Asmaa Sallam, Bala M Gnanasekaran, L Nguyen-Dinh, Walid G Aref, Mourad Ouzzani, and Arif Ghafoor. M3: Stream processing on main-memory mapreduce. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1253–1256. IEEE, 2012.

[77] Pedro Furtado. Efficiently processing query-intensive databases over a non-dedicated local network. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 72a–72a. IEEE, 2005.

[78] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006.

[79] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[80] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick O'Neil, and Elizabeth O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 361–372. ACM, 2013.

[81] Asterisk. Asterisk call detail record example. URL `www.asteriskdocs.org`.

[82] Pedro Martins, João Costa, José Cecílio, and Pedro Furtado. Vardb: high-performance warehouse processing with massive ordering and binary search. In *Data Warehousing and Knowledge Discovery*, pages 184–195. Springer, 2011.

[83] Anindya Datta, Debra VanderMeer, Krithi Ramamritham, and Bongki Moon. Applying parallel processing techniques in data warehousing and olap, 1998.

[84] Ladjel Bellatreche, Kamalakar Karlapalem, and Mukesh Mohania. Olap query processing for partitioned data warehouses. In *Database Applications in Non-Traditional Environments, 1999.(DANTE'99) Proceedings. 1999 International Symposium on*, pages 35–42. IEEE, 1999.

[85] Dell technology brief. Better business protection through virtualization, 2007. URL `http://www.dell.com/downloads/global/power/ps4q06-20070169-Ziff.pdf`.

[86] Hui-I Hsiao and David J DeWitt. A performance study of three high availability data replication strategies. *Distributed and Parallel Databases*, 1(1):53–79, 1993.

[87] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM, 1996.

[88] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[89] Todd Anderson, Yuri Breitbart, Henry F Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *ACM SIGMOD Record*, volume 27, pages 484–495. ACM, 1998.

[90] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S Seshadri, and Avi Silberschatz. Update propagation protocols for replicated databates. In *ACM SIGMOD Record*, volume 28, pages 97–108. ACM, 1999.

[91] Parvathi Chundi, Daniel J Rosenkrantz, and SS Ravi. Deferred updates and data placement in distributed databases. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 469–476. IEEE, 1996.

[92] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004.

[93] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[94] David J DeWitt and Robert H Gerber. Multiprocessor hash-based

join algorithms. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[95] Clement T. Yu, Keh-Chang Guh, David Brill, and Arbee L. P. Chen. Partition strategy for distributed query processing in fast local networks. *Software Engineering, IEEE Transactions on*, 15(6):780–793, 1989.

[96] Pedro Furtado. Workload-based placement and join processing in node-partitioned data warehouses. In *Data Warehousing and Knowledge Discovery*, pages 38–47. Springer, 2004.

[97] Pedro Furtado. Efficient, chunk-replicated node partitioned data warehouses. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 578–583. IEEE, 2008.

[98] Daniel C Zilio, Anant Jhingran, and Sriram Padmanabhan. *Partitioning key selection for a shared-nothing parallel database system.* IBM TJ Watson Research Center, 1994.

[99] Daniel C Zilio. *Physical database design decision algorithms and concurrent reorganization for parallel database systems.* PhD thesis, Citeseer, 1998.

[100] Chengwen Liu and Clement Yu. Validation and performance evaluation of the partition and replicate algorithm. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 400–407. IEEE, 1992.

[101] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136. ACM, 1982.

[102] Pedro Furtado. Algorithms for efficient processing of complex queries in node-partitioned data warehouses. In *Database Engineering and Applications Symposium, 2004. IDEAS'04. Proceedings. International*, pages 117–122. IEEE, 2004.

[103] Pedro Furtado. Hash-based placement and processing for efficient node partitioned intensive databases. In *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*, pages 127–134. IEEE, 2004.

[104] Pedro Furtado. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*, pages 23–30. ACM, 2004.

[105] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[106] Pedro Furtado. Model and procedure for performance and availability-wise parallel warehouses. *Distributed and Parallel Databases*, 25(1-2): 71–96, 2009.

[107] Pedro Furtado. A survey of parallel and distributed data warehouses. *International Journal of Data Warehousing and Mining (IJDWM)*, 5 (2):57–77, 2009.

[108] Daniel C Zilio. Modeling on-line rebalancing with priorities and executing on parallel database systems. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 43. IBM Press, 1996.

[109] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[110] Jonas S Karlsson, Amrish Lal, Cliff Leung, and Thanh Pham. Ibm db2 everyplace: A small footprint relational database system. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 0230–0230. IEEE Computer Society, 2001.

[111] Jason Buffington. Microsoft sql server. *Data Protection for Virtual Data Centers*, pages 267–315, 2010.

[112] Mikael Ronstrom and Lars Thalmann. Mysql cluster architecture overview. *MySQL Technical White Paper*, 2004.

[113] Howard Nasgaard, Bugra Gedik, Mary Komor, and Mark Mendell. Ibm infosphere streams: event processing for a smarter planet. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 311–313. IBM Corp., 2009.

[114] Stephen Fenner, Lance Fortnow, Stuart Kurtz, and Lide Li. An oracle builder's toolkit. 120:131, 1993.

[115] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for etl processes. pages 14–21, 2002.

[116] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[117] Vmware, 2014. URL `http://www.vmware.com/`.

[118] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[119] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. volume 5, pages 1591–1602. VLDB Endowment, 2012.

[120] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[121] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2): 4, 2008.

[122] Rakesh Agrawal, Anastasia Ailamaki, Philip A Bernstein, Eric A Brewer, Michael J Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J Franklin, Hector Garcia-Molina, et al. The claremont report on database research. *ACM SIGMOD Record*, 37(3):9–19, 2008.

[123] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5. ACM, 2009.

[124] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. pages 1099–1110, 2008.

[125] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[126] Miguel C Ferreira. Compression and query execution within column oriented databases. Master's thesis, Massachusetts Institute of Technology, 2005.

[127] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[128] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.