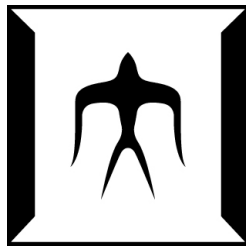


論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Automatic Tuning and Software Modularization Techniques for Optimized Large Scale Data Analytics
著者(和文)	MIYURUDAYARATHNA
Author(English)	Miyuru Dayarathna
出典(和文)	学位:博士（学術）, 学位授与機関:東京工業大学, 報告番号:甲第9347号, 授与年月日:2013年9月25日, 学位の種別:課程博士, 審査員:鈴木 豊太郎,徳田 雄洋,横田 治夫,村田 剛志,小林 隆志
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第9347号, Conferred date:2013/9/25, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Automatic Tuning and Software Modularization Techniques for Optimized Large Scale Data Analytics



A thesis submitted for the degree of
Doctor of Philosophy

Department of Computer Science
Tokyo Institute of Technology, Japan

Miyuru Dayarathna

Advisor: Prof. Toyotaro Suzumura

2013

Abstract

Large scale systems such as Exascale computing systems will be the key compute resources that enable high-end discoveries in the next half century of sciences. Data intensive large scale software systems need optimized software architectures focusing on concurrency and resiliency. Currently MPI, MapReduce, and stream programming models are famous for data intensive tasks. However, they are less optimal in terms of programmability, maintainability, and execution performance which are fundamental problems that need to be addressed in future systems. It is a technically challenging problem because it is very hard to achieve a single generic solution in variety of systems and programming environments. This thesis proposes new software modularization, automatic code transformation, and benchmarking techniques as solutions for improving productivity and execution performance of large scale data analysis systems. Specifically, it focuses on optimizing the current state of the art stream/graph data processing paradigms via the aforesaid techniques. First half of the thesis studies about automated techniques for performance improvement of stream processing systems. The thesis investigates on methods for workload characterization and benchmarking of large scale data analytics. The thesis evaluates performance of stream processing system architectures using System S, S4, and Esper. It introduces operator graph transformation, a notion for improving the concurrency of stream analytic tasks via automatic software tuning. The techniques were implemented on optimization frameworks called Hirundo, Tahitica, and Albatross.

The second half of the thesis describes graph analytics optimization. In order to understand performance features of graph database systems and also to create realistic benchmark framework for graph database systems, the thesis implemented XGDBench following the Multiplicative Attribute Graph model. The thesis describes the importance of having well-defined abstractions for batch/hybrid execution of large graph analysis tasks. Especially it presents optimized software abstractions, libraries, and middleware that can withstand software architectural challenges introduced by heterogeneous hardware. It created the architecture of ScaleGraph 1.0, a Partitioned Global Address Space (PGAS) library for analyzing graphs in the scale of billions of vertices and edges. Furthermore, it implemented a domain specific language for large graph analysis called Exedra, and showed how such languages can support specification of optimized graph data analysis tasks efficiently.

Thesis made contributions to large scale stream computing systems by introducing an automatic optimization mechanism for data stream programs. It created a scalable data generator (StreamFarm) and a benchmark (XGDBench) for supporting performance optimizations. Thesis contributed for the area of large scale graph analysis by introducing ScaleGraph library and Exedra which is a domain specific language for defining large graph analysis processes.

Thesis identifies future expansions in the areas of high productive software development, massive graph analysis, and software maintainability. Thesis describes the software architectural aspects of Hirundo, Albatross, XGDBench, ScaleGraph, and Exedra focusing on their contributions for performance optimization of large scale data analytics.

Key Words

Data Analytics, Stream Computing, Graph Analytics, Software Architecture, Distributed Systems, Graph Databases, Exascale, Benchmarking, Software Automatic Tuning, Software Performance, High Performance Computing

Dedication

To

My Mother Pathma Caldera, Father G. Dayarathna, and two Brothers
Nirmal and Chamika

For their endless love and understanding

Acknowledgements

I wish to thank Prof. Toyotaro Suzumura, my research advisor for his kindness, for great amount of mentoring, and for the freedom he provided for me to think logically and creatively. His support was always with me when I was facing difficult times in my student life. His insightful advice were so important for me to setting up my research carrier in large scale data analytics. I would also like to thank professors Takehiro Tokuda, Tsuyoshi Murata, Haruo Yokota, Takashi Kobayashi for being the thesis committee members of my Ph.D. thesis. Their comments have been of great support for improving the research presented in this thesis. I am very grateful to Prof. Takehiro Tokuda for acting as my secondary advisor during the final half of my stay at Tokyo Tech. I highly appreciate the mentoring provided by Prof. Yutaka Akiyama as my backup advisor.

I would also like to thank my co-authors at Suzumura Laboratory, Charuwat Houngkaew, Hidefumi Ogata, and Souhei Takeno for their support during the projects. Charuwat and Ogata were the two key members during the period of ScaleGraph 1.0 development project and contributed a lot for the library's development. Especially me and Charuwat had lots of discussions during the initial stages of the design of ScaleGraph 1.0. Takeno's support was instrumental for the success of the performance experiments conducted with stream processing systems. I would like to thank past lab members of Suzumura Laboratory Hiroya Matsuura and Atsushi Ishii for the supports they provided during first two years of my stay at Tokyo Tech. I would also like to thank the server team members of Suzumura Laboratory, Hiroya Matsuura, Koji Ueno, and Masaru Watanabe for various supports they provided as server administrators. I am so grateful to the Tokyo Tech Department of Computer Science's 9th floor secretaries Tomoko Takahashi, Tomomi Hotta, Yoko Takahashi, Tomomi Mitsudera, and Mayuko Takano for their endless supports during my three year student life at Tokyo Tech. I am also grateful to Reiko Yamamoto for all the supports she provided me as the secretary of the JST CREST project.

I am so grateful for Honjo International Scholarship Foundation for the financial support provided during my stay at Tokyo Tech. I would also like to thank the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing" for supporting the research conducted in this thesis.

Finally, I express my deepest thank to my mother Pathma Caldera, father G. Dayarathna, and two brothers Nirmal and Chamika for their encouragement, endless support, and patience.

Miyuru Dlishan Dayarathna
August 21st, 2013

Contents

List of Figures	x
------------------------	----------

List of Tables	xvi
-----------------------	------------

1 Data Analytics in Large Scale Settings	1
1.1 What is Large Data Analytics?	1
1.2 Types of Data Analytics	2
1.3 A Brief History of Data Analytics	3
1.4 Challenges for Large Scale Data Analytics	5
1.5 Issues of Current Data Analytics	6
1.6 Aim & Objectives of the Thesis	7
1.7 Solution through Autotuning and Software Modularization	7
1.8 Structure of the Thesis	8
1.9 Summary	10
2 State of the Art in Large Scale Data Analytics Optimization	11
2.1 Workload Characterization and Benchmarking of Stream Processing Sys- tems	11
2.2 Benchmarking of Graph Databases	13
2.3 Stream Processing Performance Optimization	14
2.4 Multiple Stream Query Performance Optimization	16
2.5 Stream Job Performance Maintenance	18
2.6 Graph Processing Libraries	18
2.7 Design and Implementation of Domain Specific Languages	20
2.8 Graph Data Storage and Management in Clouds	22
2.9 Summary	23
3 Technologies for Large Scale Data Analytics Optimization	25
3.1 Data Flow Languages and Their Importance for Large Scale Data Analytics	26
3.1.1 SPADE - An Operator-Based Stream Processing Language	26
3.2 Stream Processing Software	28
3.2.1 System S	30

3.2.2	S4	30
3.2.3	Esper	32
3.3	High Productivity Computing Systems Programming Languages	33
3.3.1	X10 - An Overview	34
3.4	Graph Databases	36
3.4.1	RDF Stores : AllegroGraph and Fuseki	37
3.4.2	Neo4j	38
3.4.3	OrientDB	38
3.4.4	Titan	38
3.5	Summary	39
I	Stream Processing Optimization	40
4	Workload Characterization and Benchmarking of Stream Processing Systems	41
4.1	Introduction	41
4.2	Methodology and Performance Metrics	42
4.2.1	Sample Programs and Data Sets	42
4.2.1.1	Micro-benchmark	42
4.2.1.2	VWAP	42
4.2.1.3	CDR	43
4.2.1.4	Twitter Topic Counter	43
4.2.2	Experimental Setup	44
4.2.3	Performance Metrics	44
4.2.4	Objectives and Methodology	45
4.3	Performance Evaluation	48
4.3.1	Job Throughput	48
4.3.2	CPU Usage	48
4.3.3	Memory Usage	49
4.3.4	Network I/O	51
4.4	Discussion	51
4.5	Conclusions	53
4.6	Summary	53
5	StreamFarm a Workload Generator Platform for Stream Computing Environments	54
5.1	Introduction	54
5.2	Methodology	55
5.3	Design and Implementation of StreamFarm	55
5.4	StreamFarm Query Language	57
5.5	Conclusion	60
5.6	Summary	60

6	Automatic Optimization of Stream Programs via Source Program Operator Graph Transformations	61
6.1	Introduction	62
6.2	Preliminaries	63
6.3	Methodology	64
6.4	Program Transformation	66
6.4.1	Example Stream Applications	66
6.4.2	Ray Transformation Algorithms	69
6.4.2.1	Tri-Operator Transformation	69
6.4.2.2	Transformer Patterns of Tri-Operator Transformation . .	71
6.4.2.3	Bi-Operator Transformation	73
6.4.2.4	Transformer Patterns of Bi-Operator Transformation . .	73
6.4.3	Program Transformation Algorithm	73
6.4.3.1	Program Transformation on Linear DAGs	75
6.4.3.2	Operator Blocks Fusion on Linear DAGs	78
6.4.3.3	Program Transformation on Multipath DAGs	79
6.4.3.4	Operator Blocks Fusion on Multipath DAGs	80
6.4.4	Preservation of Input Program Semantics	80
6.5	Transformer Blocks	81
6.5.1	Preservation of Sequential Semantics	82
6.6	Implementation	82
6.6.1	Program Structure Analyzer	82
6.6.2	Data Preparator	86
6.6.3	Parallel Compiler	86
6.7	Fault Tolerance of Hirundo	86
6.8	Sample Program Space Reduction and Ranking	87
6.8.1	Performance Prediction Mode	88
6.9	Evaluation	89
6.9.1	Experiment Setup and Methodology	89
6.9.2	Sample Program Naming Convention	90
6.9.3	Sample Program Performance	91
6.9.4	Performance Prediction	95
6.9.5	Performance Clusters	96
6.9.6	Comparison with Operator Fusion	99
6.10	Discussion	101
6.11	Conclusions	103
6.12	Summary	104
7	Automatic Stream Query Optimization in the Presence of Multiple Stream Jobs	105
7.1	Introduction	105
7.1.1	Input Applications	106
7.2	Sample Program Generation via Automatic Code Transformation	107

7.2.1	2D Transformation	108
7.3	Optimization of Multiple Stream Jobs	110
7.4	Implementation	111
7.5	Experimental Evaluation	111
7.6	Discussion	114
7.7	Conclusions	114
7.8	Summary	115
8	Stream Program Performance Recovery in Resource Limited Compute Clusters	116
8.1	Introduction	116
8.1.1	Resource Limited Stream Compute Clusters	118
8.2	Approach for Performance Maintenance	118
8.2.1	Program Ranking Algorithm	119
8.2.2	Program Switching Model	121
8.3	Load Shedding for Drastic Node Failures	121
8.4	Implementation	122
8.4.1	Runtime Orchestrator	123
8.4.2	Gateway	123
8.5	Elimination of Single Point of Failure	123
8.6	Experimental Evaluation	125
8.6.1	Experimental Setup	125
8.6.2	Evaluation of Stream Program Performance Variation	125
8.6.3	Evaluation of Performance Recovery Process	126
8.6.4	Evaluation of Single Point of Failure Recovery Process	127
8.7	Discussion	128
8.8	Conclusions	130
8.9	Summary	130
II	Graph Processing Optimization	131
9	Benchmarking Graph Database Systems	132
9.1	Introduction	132
9.2	Background for Graph Database Benchmarking	133
9.2.1	Yahoo! Cloud Serving Benchmark	134
9.2.2	Synthetic Graph Generators	135
9.2.3	An Overview of Multiplicative Attribute Graphs	135
9.3	Methodology of XGDBench	138
9.4	Requirements of XGDBench	138
9.4.1	Attribute Read/Update	138
9.4.2	Graph Traversal	139
9.5	Implementation	140

9.5.1	Graph Generator	142
9.5.2	Workload Executor and Graph DB Workload	142
9.5.3	Graph DB Interface Layer	143
9.5.4	Implementation of Traversal Operation	143
9.5.5	Implementation of Insert and Update Operations	144
9.6	Ensuring Reproducibility of Results Across Benchmarking Sessions	144
9.7	Modeling Workload Spikes in Graph Database Systems	146
9.7.1	Spike Generation Methodology	146
9.8	Evaluation	147
9.8.1	Properties of the MAG Data Generator	148
9.8.1.1	Power-law Distribution	148
9.8.1.2	Categorical Prominence Within Communities	148
9.8.2	Graph DB Performance Comparison	151
9.8.3	Performance Evaluation of Titan	153
9.8.4	Evaluation of Graph Generation Time	154
9.8.5	Evaluation of Workload Spike Generation	154
9.9	Discussion and Limitations	158
9.10	Conclusions	160
9.11	Summary	160
10	An X10 Library for Billion Scale Graph Analytics	162
10.1	Introduction	162
10.2	ScaleGraph Architecture	163
10.3	Implementation	166
10.3.1	Graph Structural Properties	166
10.3.1.1	Degree Distribution Calculation	166
10.3.1.2	Betweenness Centrality	167
10.3.1.3	Spectral Clustering	167
10.4	Evaluation	169
10.5	Discussion and Limitations	173
10.6	Conclusion	173
10.7	Summary	174
11	A Domain-Specific Language for Large Graph Analytics	175
11.1	Introduction	175
11.2	Exedra Domain-Specific Language	176
11.3	Design and Implementation of Dipper	179
11.4	Evaluation	182
11.5	Results and Discussion	183
11.6	Conclusion	185
11.7	Summary	185

III Conclusion and Future Directions	186
12 Conclusion & Further Work	187
12.1 Contributions	188
12.2 Discussion and Limitations	189
12.3 Further Work	190
12.3.1 Stream Processing Optimization	190
12.3.2 Graph Processing Optimization	191
12.3.3 Distributed Graph Database Engine for Large Scale Elastic Graph Data Management and Mining in Hybrid Clouds	192
12.3.3.1 Proposed System Design	193
12.3.3.2 Proposed Scheduling Methodology	194
12.3.3.3 Proposed Implementation Technology	194
12.4 Summary	195
References	196
A Data Flow Graphs of the High Performance Sample Applications	228
B Some Example Transformations for Quadri-Operator Transformation	231
C Representation of a Communication Network with ScaleGraph 1.0	233

List of Figures

1.1	Software for Large Scale Data Analytics	5
1.2	Solution through benchmarking, software automatic tuning, and software modularization	8
3.1	SPADE code for CDR Application.	27
3.2	Architecture of System S runtime environment	31
3.3	Architecture of S4	31
3.4	Architecture of Esper	32
3.5	Overview of X10 language features.	35
3.6	Architecture of Neo4j. (a) High level Architecture of Neo4j. (b) Logical view of the user-facing APIs in Neo4j.	38
4.1	Data flow graphs of sample applications.	44
4.2	Augmentation of sample application data flow graphs on System S and S4. Sub-figures (a) to (e) correspond to System S applications while (f) to (j) correspond to S4.	46
4.3	Augmentation of sample application data flow graphs for Esper.	47
4.4	Throughput comparison of sample applications. (a) System S Applications (b) S4 Applications (c) Esper Applications.	47
4.5	CPU usage on different nodes of System S and S4 while running CDR, VWAP, and Twitter applications.	49
4.6	CPU utilization by nodes when running micro-benchmark on System S and S4.	50
4.7	Breakdown of CPU usage by processes for CDR and Micro-benchmark.	50
4.8	Memory consumption of sample applications on System S and S4.	51
4.9	Network I/O for Manager node and Worker nodes while running CDR on S4 and System S.	52
5.1	Architecture of StreamFarm	55
5.2	Main classes of StreamFarm	56
5.3	An example set of StreamFarm queries that can be used to create Springs	57
5.4	Event classes of StreamFarm	58
5.5	A GUI interface for StreamFarm	59

5.6	An example set of StreamFarm queries for managing filters	59
6.1	Methodology of Hirundo	65
6.2	Data flow graphs of CDR, Twitter, and VWAP applications	67
6.3	Data flow graph of Apnoea detection SPADE application. Note that the operator outlines using dotted lines and the arrows that connect operators using dotted lines represent rays on the data flow graph. E.g., F2, AG2, and F5 are on a single ray.	68
6.4	Data flow graph of the Bargain calculation SPADE Application.	69
6.5	Sample transformations using Tri-OP transformation. The figure also shows categorization of possible transformations for a three operator block linear segment. (a) Input operator blocks (b) Result of applying 1-1-2 transformation (c) Result of transforming by 1-2-1 (d) Result of transforming by 2-0-2 (e) Result of transforming by 2-0-1 (f) Result of transforming by 2-2-2 (g) Result of transforming by 2-1-2 (h) Result of transforming by 2-2-1 (i) Result of transforming by 2-1-1 (j) Result of transforming by 1-2-2.	70
6.6	Sample transformations using Bi-OP transformation. The figure also shows categorization of possible transformations for two operator block linear segment. (a) Input operator blocks (b) Result of applying 1-2 transformation (c) Result of transforming by 2-0 (d) Result of transforming by 2-2 (e) Result of transforming by 2-1	74
6.7	An overview of Hirundo's program transformation algorithm.	74
6.8	How VWAP application's DAG is traversed by GENERATE() procedure. . .	75
6.9	Operator Blocks Fusion	78
6.10	Paths on the example data flow graphs. (a) Apnoea (b) Bargain (for n th portfolio item)	79
6.11	Some examples of Transformer Blocks used by Hirundo. (a) DEMUX-SINK (b) MUX-SINK (c) MUX-SOURCE (d) PARALLEL-SOURCE (e) MULTI-FUNCTOR (f) PARALLEL-SINK (g) DEMUX-STREAM (h) MUX-STREAM (i) DEMUX-FUNCTOR-N-TO-M (j) TRANSFORMER-N-TO-M (k) TRANSFORMER-MODULUS-N-TO-M	81
6.12	System Architecture of Hirundo.	83
6.13	A sample set of expressions of Hirundo's grammar with terminals.	83
6.14	A sample rule from Hirundo's grammar.	84
6.15	An example of how the rules get mapped to the data flow graph of CDR application. The complete source code of CDR application is given in Figure 3.1.	85
6.16	Comparison of three optimization runs (Opt run 1,2,3) with CDR application using equal optimization parameters (d=4, nodecount=4) on cluster X	91
6.17	Comparison of three optimization runs (Opt run 1,2,3) with CDR application using equal optimization parameters (d=4, nodecount=8) on cluster Y	92

6.18	Comparison of three optimization runs (Opt run 1,2,3) with Twitter application using equal optimization parameters ($d=4$, $nodecount=4$) on cluster X.	92
6.19	Comparison of three optimization runs (Opt run 1,2,3) with Twitter application using equal optimization parameters ($d=4$, $nodecount=8$) on cluster Y.	93
6.20	Comparison of three optimization runs (Opt run 1,2,3) with VWAP application using equal optimization parameters ($d=8$, $nodecount=4$) on cluster X.	93
6.21	Comparison of three optimization runs (Opt run 1,2,3) with VWAP application using equal optimization parameters ($d=8$, $nodecount=8$) on cluster Y.	94
6.22	Comparison of three optimization runs (Opt run 1,2,3) with Apnoea application using equal optimization parameters ($d=4$, $nodecount=4$) on cluster X.	94
6.23	Comparison of three optimization runs (Opt run 1,2,3) with Bargain application using equal optimization parameters ($d=4$, $nodecount=4$) on cluster X.	95
6.24	Results of applying K-Means clustering for VWAP sample program performance data.	97
6.25	Results of applying K-Means clustering for Twitter sample program performance data.	98
6.26	Results of applying K-Means clustering for CDR sample program performance data.	98
6.27	An example operator PE partitioning scheme for Apnoea data flow graph.	100
6.28	Average throughput of applications generated with different fusion combinations of VWAP application on cluster Y.	100
6.29	Average throughput of applications generated with different fusion combinations of CDR application on cluster Y.	101
7.1	Data flow graphs of example input stream applications	107
7.2	Methodology of Tahitica	107
7.3	2D Transformation of Tahitica	109
7.4	Overview of the program transformation of Tahitica.	110
7.5	Architecture of Tahitica	111
7.6	Comparison of three optimization runs (Opt run 1,2,3) of the Apnoea application using equal optimization parameters (transformation depth (d) = 2, traversal depth = 1).	112
7.7	Comparison of three optimization runs (Opt run 1,2,3) of the SCTR application using equal optimization parameters (transformation depth (d) = 4, traversal depth = 1).	113

7.8	Comparison of three optimization runs (Opt run 1,2,3) of the unified job created for Apnoea and SCTR applications. The optimization parameters were the same as in the previous two evaluations.	113
8.1	System Architecture of Albatross.	122
8.2	Recovery process from single point of failure.	124
8.3	Throughput of sample applications produced for VWAP application for six calibration runs.	126
8.4	How the sample program performance change when two nodes were crashed.	126
8.5	Recovering performance of Regex application during two consecutive node crashes.	127
8.6	Performance recovery process when (a) two nodes were crashed for VWAP (b) two consecutive node crashes for Apnoea	128
8.7	Output data rate variation of Apnoea during the recovery process when the main node has been crashed.	129
9.1	Architecture of YCSB Client.	134
9.2	Multiplicative Attribute Graphs (MAG) model	136
9.3	An example for attributes with binary values (yes/no) from Facebook online social network.	136
9.4	Architecture of XGDBench Client	140
9.5	Implementation of traverse operation on Titan client	145
9.6	Methodology used for workload spike generation on XGDBench. (a) Scenario of volume spike generation (b) Scenario of data spike generation.	147
9.7	Model of the workload spike.	147
9.8	Comparison of the degree distribution produced by MAG generator of XGDBench with a real world social network	149
9.9	An illustration of categorical prominence in a graph data set. In the graph data set the vertices are classified into two sets called A and B.	150
9.10	Data loading phase and transaction phases' throughput of graph database servers on Tsubame cloud	152
9.11	How XGDBench, Titan Rexster server, and Cassandra are deployed in the experiment node cluster on Tsubame.	153
9.12	Data loading phase of Titan on Tsubame cloud	155
9.13	Average latency for transaction phase of Titan on Tsubame cloud	156
9.14	Throughput and runtime for transaction phase of Titan on Tsubame cloud	156
9.15	Evaluation of graph generation time on XGDBench graph generator on Tsubame cloud. The numbers on the curve in (a) indicate the number of edges generated in millions (M).	157
9.16	How XGDBench, Titan Rexster server, and Apache Cassandra are deployed during the experiment on workload spike generation.	157
9.17	Network I/O	158
9.18	Disk access	158
9.19	System summary	159

10.1	ScaleGraph Architecture.	164
10.2	Medium scale and large scale configurations of ScaleGraph.	165
10.3	Package structure of ScaleGraph.	165
10.4	A code snippet of BC calculation on PlainGraph.	168
10.5	A code snippet of Spectral Clustering algorithm of ScaleGraph.	169
10.6	Elapsed time for calculating in-degree distribution of Kronecker graph. . .	170
10.7	Elapsed time for calculating in-degree distribution of LiveJournal graph. .	171
10.8	Elapsed time for obtaining BC up to 8 nodes.	172
10.9	Elapsed time for running Spectral Clustering up to 8 nodes.	172
11.1	Example Exedra code segments.	177
11.2	Example Exedra code for scenario 1.	178
11.3	Arrangement of compartments of Scenario 2.	178
11.4	Example workflow of Scenario 2.	180
11.5	Example Exedra code for scenario 2.	180
11.6	System Architecture of Dipper.	181
11.7	KDT version of the example scenario 2.	182
11.8	Experiment results of running Dipper.	183
11.9	Experiment results of running KDT workflow.	184
11.10	Experiment results of running Dipper distributed on 4 nodes.	184
12.1	Summary of the contributions.	188
12.2	High-level view of the relationships between the stream program optimization mechanisms and techniques described in this thesis.	190
12.3	Overview of the proposed Acacia hybrid graph database system	193
12.4	Proposed system architecture of Acacia	194
A.1	DAG of 8SCSV_16F_AG_F_SI sample application of VWAP which produced highest consistent performance compared to the original VWAP application on cluster X.	228
A.2	DAG of 6SCSV_48F_AG_F_SI sample application of VWAP which produced highest consistent performance compared to the original VWAP application on cluster Y.	229
A.3	DAG of 4SCSV_4U_2F_4AG_4SI sample application of CDR which produced highest consistent performance compared to the original CDR application on clusters X and Y.	229
A.4	DAG of U_2F_2F_F_AG_SI sample application of Twitter which produced highest consistent performance compared to the original Twitter application on cluster X.	229
A.5	DAG of U_4F_2F_F_AG_SI sample application of Twitter which produced highest consistent performance compared to the original Twitter application on cluster Y.	230

A.6	DAG of SCSV_F_AG_F_2F_2AG_4F_F_AG_F_J_J_SI sample application of Apnoea which produced highest consistent performance compared to the original Apnoea application on cluster X.	230
A.7	DAG of SCSV_FOR_S_2F_2AG_2F_F_J_F_SI_FOR_E sample application of Bargain which produced highest consistent performance compared to the original Bargain application on cluster X.	230
B.1	Some transformations possible using Quadri-Operator Transformation. . .	232
C.1	An example X10 code for representing a communication network with ScaleGraph 1.0.	234

List of Tables

2.1	A summary of graph analysis software.	22
3.1	Main sections of a SPADE program.	29
3.2	A summary of current graph databases.	37
6.1	Sample program space sizes.	87
6.2	Hardware specifications of experiment environments.	90
6.3	Summary of performance prediction runs.	96
8.1	Notation.	118
9.1	Basic operations of graph databases.	139
9.2	Core workloads of XGDBench.	141
9.3	Categorical Prominence	150
9.4	Specification of a single compute node	151
9.5	Specifications of Graph DBMSs	151
11.1	Experiment graph data sets	183

Data Analytics in Large Scale Settings

We live in an information age where massive amounts of data have been produced by various sources in various different forms that we never thought before. The knowledge that we gain by analyzing these data in an intelligent manner has become a fundamental means of improving the living conditions of human beings. These data can be categorized into different classes based on their attributes. This chapter provides background information and motivation for the thesis. An introduction to large scale data analytics is given in Section 1.1. Next, Section 1.2 explains the types of large scale data analytics and highlights the importance of stream processing and graph data analytics as subdisciplines of large scale data analytics. The thesis describes a brief historical background of large scale data analytics in Section 1.3. Challenges faced by large scale data analytics is described in Section 1.4. The research problem addressed by this thesis is given in Section 1.5. Next, the aim and the objectives of the thesis are listed in Section 1.6. Proposed solution of the thesis is explained in Section 1.7. The structure of this thesis is elaborated in Section 1.8.

1.1 What is Large Data Analytics?

The term *Data Analytics* refers to application of computer systems to analysis of large data sets for the support of decision making [1]. The field of data analytics has adopted aspects from many other disciplines such as computational intelligence, signal theory, machine learning, statistics, operations research, etc[1]. While data analytics has been described as a synonym for *Data Mining* [2], data analytics is more concerned about the technologies and tools used for data mining tasks. Data Mining on the other hand is the process of discovering patterns from massive amounts of data. Data mining extracts useful models of data. A model can be most extreme features of data or it can be a summary of the data [3].

Applications of large data analytics have appeared in various different areas of sciences. Genome sequence alignment is a popular example of large data analysis. Some single nucleotide variations present in human genomes can be associated with genetic disorders. To search such genetic disorders one may typically collect about 1 billion of short sequences of length 100 nucleotides at random from a personal genome and then align each of the one billion sequences to the reference human genome which consists about 3 billion nucleotides. The alignment task can be run in parallel with dividing the set of 1 billion sequences into subsets and by assigning each subset to a CPU core [4].

Another example for an application that generates large data sets is x-ray tomography. A tomography beam line such as the one produced by the Advanced Photon Source (APS) of Argonne National Lab may generate 1 terabyte of data per day. While many users may carry the experimental data on external storage devices the increasing data rates generated by future light sources will make such approaches infeasible for such increasing data sizes [5].

A third example for a situation where large scale analysis of data “on the fly” happens is large-field radio telescopes. Typically, array of antennas in such large radio telescope are connected by high speed communication links and may lead to an enormous rate of data collection. It was stated that Low-Frequency Array for radio astronomy (LOFAR) can produce observational data at rates as large as 13 Tbit/s for typical 200MHz sampling rate [6]. Such large data streams cannot be even transported in total [6]. In summary we need scalable techniques to analyze such data to gain valuable information for various different purposes.

1.2 Types of Data Analytics

Data analytics can be categorized into different classes based on the characteristics of the data mining activities. There are a variety of sub disciplines in data mining such as multi-relational data mining, text mining, web mining, stream mining, multimedia (or image, video) mining, spatiotemporal data mining, biological data mining, information network analysis, financial data analysis, etc [7]. Out of the various different types of data mining subdisciplines, this thesis focuses on analytics applied for data stream mining and graph data mining. These two categories of data analytics can be applied to a number of different data mining application scenarios.

Increasingly data in the form of continuous streams appear from various hardware and software sensors. E.g., Advances in Information Technology (IT) have lead to large flows of data across IP networks [8]. Unlike traditional data sets, data streams are organized as sequences of data instances that flow in and out of a system continuously with varying update rates [9]. Since the volumes of such data are often very large it is impractical for storing and conducting repeated analysis on such data. There is a growing need to gather, process, and analyze such continuous streams to extract insights in real-time. One of the solution technologies is *Stream Computing* (i.e., Stream Processing/Complex Event Processing [10]/Information Flow Processing [11]) where continuous streams of high volume data are processed online with very low latency. The middleware used

for this purpose are called *Stream Processing Systems*. Such systems are distinguished from similar middleware such as Business Activity Monitoring (BAM) or Business Process Monitoring (BPM) because the client of a stream processing application is often a program rather than a human. Therefore, volume and latency requirements of stream processing applications are much more stringent [12]. Stream processing is widely used in online real-time analytics in areas such as financial services (stock ticker analysis for market-making [13]), maintaining the state of massively multiplayer Internet games [12], real-time risk analysis, health care informatics (physiological sensor monitoring in intensive care units [14]), network monitoring, national security, traffic systems (highway toll processing) [15][16][17], process control for manufacturing [18], fraud detection [19], multi-modal surveillance for law enforcement, call detail record processing for telecommunications [20], advertising [19], radio astronomy [21][22], environmental monitoring [23][24], etc.

Batch processing has been the widely adapted data analytics for many decades. Batch processing is characterized by properties such as large unit of work, coarse-grained resource allocation, sequential access patterns, isolated execution, etc [25]. The popular relational databases [26][27][28] and MapReduce [29][30] model are examples for use of batch processing. Example software that implement batch processing paradigm include systems such as Hadoop [31]/Pig [32], SCOPE [33], MATE [34], Sawzall [35], Dryad [36]/DryadLINQ [37], etc. However, batch processing as it is cannot be applied for applications that need to operate on data in online basis.

In computer science and mathematics, graphs are used as mathematical structures to model pairwise relationships between objects from a certain collection [38]. More formally a graph $G = (V, E)$ consists of a set of vertices V along with a set of E of vertex pairs or edges [39]. The field of graph data mining had seen a rapid explosion in recent years because of new applications in areas such as computational biology [40][41], social and communication networking, national defense, politics [42], cheminformatics [43], medical informatics [44][45][46][47], etc. need to process large scale graph data sets. Mining these graphs to find deep insights require middleware and software libraries that could harness the full potential of large scale computing infrastructures such as cloud computing systems, and super computers [48]. Graphs are irregular data structures and the software systems that are aimed for solving graph problems must address the irregular nature of the graphs [49]. Large graph analysis is a dilemma faced by programmers due to complexity involved in specifying large graph analysis activities. For example, the famous social networking site Facebook had 1.06 billion monthly active users with more than 150 billion friend connections as of December 31st, 2012 [50].

1.3 A Brief History of Data Analytics

Roots of stream data analytics can be found in the areas of Active Database systems [11]. Active databases allowed for actions to automatically execute when given conditions are satisfied. Data Stream Management Systems (DSMS) evolved from this idea to perform query processing on continuous data streams [51]. Aurora developed by Brandeis, Brown,

and MIT is one of the earliest DSMSs [52]. Aurora experiences resulted in development of Borealis distributed stream processing engine [53]. STREAM is another contemporary DSMS for Borealis which appeared circa 2004 [54]. While the initial DSMS prototypes appeared in the academic community, they got huge attention from industry as well. This led to the development of System S [55], S4 [19], Esper [10] like DSMSs.

Roots of graph data analytics can be found in several areas such as graph theory [56]. The seminal work by the great mathematician Leonard Euler around 1736 on solving the seven bridges of Königsberg problem gave rise to the field of graph theory [57][39]. During Euler's time there were seven bridges linking the banks and two islands of Pregel river which flowed across the Prussian city of Königsberg. Euler modeled the problem as a multigraph with seven edges and four vertices [58]. Euler proved that there was no path that allows for walking over each of the seven bridges only once and returning home (Such path is called Eulerian cycle). Euler's tour technique has been influential in parallel graph algorithms even though it seems rather trivial in modern days [59]. There are many parallel graph algorithms that start by finding a spanning tree and then traversing the tree using the Euler tour technique [39].

Many computer scientists have implemented a variety of graph analysis algorithms as graph analysis packages. Most notable examples include Igraph [60], JUNG [61], GraphStream [62], The Boost Graph Library [63], GraphLab [64], etc. However, all of these software packages are intended for execution on single computer settings. This leads to the amount of processing that can be conducted using these software limited to the computer's hardware resources such as RAM, Hard Disk Drive (HDD) capacity, and CPU (number of threads, cache hierarchy). To increase the size of the graph problems handled by such software applications, we have to follow a vertical scaling approach (i.e., Add more hardware resources to the computer.). Still there are limitations to such vertical scaling when handling very large graph data sets (e.g., A graph dataset with more than 1 billion edges on a modern commodity class system). A solution is horizontal scalability in which we conduct parallel graph analysis using multiple compute nodes.

In the current times scalability of graph data analysis in computing systems has become a new frontier due to the massive sizes of graph data encountered compared to the Euler's time [65]. Graphs with millions (or even billion) of vertices and billions of edges are abundant in areas such as web, Internet, social networks [66][50], etc.

Parallel graph data analysis is currently new area in computer science. Such graph data analysis often happen through utilizing the existing software infrastructures such as MPI [67], MapReduce, etc. A notable, inspirational early work in this realm is PEGASUS which is a library built on top of Hadoop platform for mining graphs of scales of terabytes or petabytes of storage [68]. PEGASUS was built around a highly optimized primitive called *Generalized Iterative Matrix-Vector Multiplication* (GIM-V). There are three operations in the GIM-V called *combine2*, *combineAll*, *assign* which if customized separately gives a surprisingly large number of useful graph mining algorithms. However, alternative MapReduce graph analysis platforms have also been introduced such as Ex-MATE [69] which outperformed PEGASUS's performance figures.

Another approach for parallel graph analysis has been MPI based parallel graph

libraries. Parallel Boost Graph Library (PBGL) [70] is one of the most notable parallel graph processing libraries since it has been implemented by extending the famous Boost graph library. PBGL has been developed using C++/MPI. ParGraph [71] is a generic parallel graph library which is comparable to PBGL. Combinatorial BLAS is a high-performance software library written in C++/MPI for graph analysis and data mining [72].

While the aforementioned approaches are widely applicable to a range of hardware, there are several special classes of software libraries that utilize special hardware for conducting massive graph analysis. Examples include the work by Madduri *et al.* [73], Bader *et al.* [74][75], and Berry *et al.* (Multi Threaded Graph Library) [76]; which describe the ability of using massively multi-threaded machines to implement graph algorithms. While distributed memory application developers focus on maximization of locality to minimize interprocess communication, program developers for massively multi-threaded computers having large shared memory (e.g., Cray MTA-2) do not focus on locality or data exchange [77]. Some of the work of this domain (e.g., Multi Threaded Graph Library) have been extended to commodity processors yet with lesser performance [78].

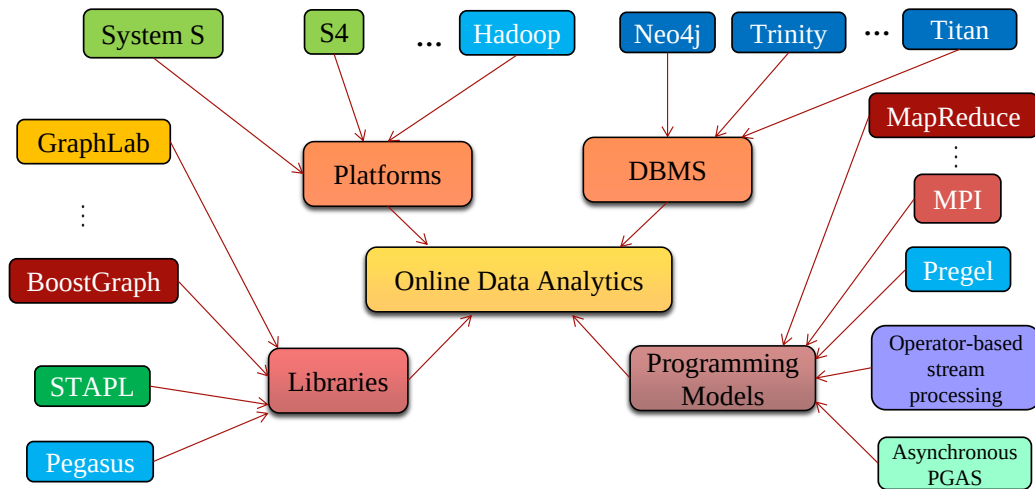


Figure 1.1: Software for Large Scale Data Analytics

1.4 Challenges for Large Scale Data Analytics

Exascale era which is expected to appear in 2018~2020 time frame will revolutionize many computing paradigms [79]. Transiting applications to new Exascale systems require very large investment in several areas of software research and development. Since power and cooling constraints limit increases in microprocessor clock speeds; hardware manufacturers are dramatically increasing on-chip parallelism to improve performance

[80]. New hardware architectures such as multi-core, many-core processors, heterogeneous hardware such as GPGPUs, FPGAs, 1000x increase of parallelism, increase in system faults, as well as complicated multicomponent software environment are some of the most challenging issues [81]. Applications and algorithms need to adapt for evolving machine architectures by managing locality and resilience to achieve high performance [82].

Efficient parallel code needs to focus on maximizing parallelism. With modern memory hierarchies, both sequential and parallel codes must focus on improved locality [83]. Extreme scale computers provide the opportunity to read in and react to an enormous amount of data in real-time. E.g., The ability to analyze in real-time one million sources, such as field sensors that can produce > 1 GB/s of visual data each. A compute node would have to be able to read in at a rate of one petabyte per second assuming there was no further computation involved [84]. Programming system needs to provide a simple execution model for the programmer. Simple execution models are needed which will reduce the application programming complexity required to achieve the goals of exposing all parallelism and locality [84] (Locality here means the logical or physical proximity of data to the units that perform computation on them). While the execution model may be defined at a high level of abstraction, the underlying programming systems need to enable data analytics programmers to give transparent guidance when needed and thereby providing a level of performance transparency.

The requirement for portable large scale data analysis parallelism demands that all of the intrinsic parallelism need to be exposed at all levels in the application. This is different from the current practice where programmers repeatedly rewrite applications to expose incrementally more parallelism for the next generation hardware [84]. A fundamental issue in the portable expression of parallelism is the need for data structures that allow themselves naturally to do data-parallel operations [84]. While *array* or *vector* data structures are very well suited for data parallelism *streams* represent another data structure that is well suited for parallelism. Yet, graph and other pointer-based data structures necessary for large scale data analytics in systems such as Exascale systems create additional challenges for expressing parallelism and locality.

Code generators used with large scale data analysis systems need to change in response to the need for agile code mappings that respond dynamically to execution context such as input data set properties, machine load, and power constraints. Rather than creating a single statically-generated implementation of a computation, the compiler needs to represent a space of possible implementations that are generated either a priori by predicting relevant features of the execution contexts or dynamically to respond to changing execution contexts.

1.5 Issues of Current Data Analytics

Powerful integrated performance modeling, prediction, measurement, analysis, and optimization capabilities are required to operate large scale data analysis system. However, application of current HPC software architectures as it is for large scale data analysis

is less optimal in terms of execution performance and programmability. The main reasons are lack of programmer productivity, lack of suitable benchmarks, rapid changes happening in the hardware technologies, and application scalability.

1.6 Aim & Objectives of the Thesis

Aim: The aim of this thesis is to use highly scalable software modularization, automatic code transformation, and benchmarking techniques to improve productivity and execution performance of data analysis software architectures for large scale systems.

Objectives:

1. To conduct performance analysis of large data analysis software systems to identify important performance characteristics
2. To implement automatic performance tuning of large scale data analysis applications
3. To create well-defined software modularization to improve programming productivity as well as support automatic performance tuning
4. To create data abstractions to hide complexities involved in performance tuning
5. To create supporting tools and libraries which utilize the aforementioned techniques
6. To evaluate these tools and techniques in post peta-scale environments

1.7 Solution through Autotuning and Software Modularization

The thesis proposes creation of new software automatic tuning, software modularization, and benchmarking techniques for large scale data analytics that can withstand the architectural challenges posed by large scale systems such as future Exascale systems. The abstract view of the proposed solution is shown in Figure 1.2.

Software Automatic Tuning is a systematic process for evaluating alternative implementations of a computation with the aim of selecting best performing solution for a particular system architecture [85]. Diversity and complexity of parallel architectures increases burden on application programmers for porting and tuning their codes.

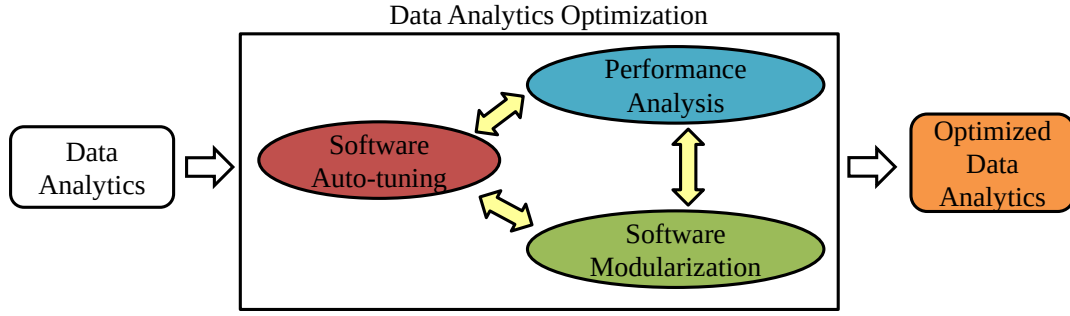


Figure 1.2: Solution through benchmarking, software automatic tuning, and software modularization

1.8 Structure of the Thesis

The thesis describes the work conducted in design and implementation of optimized data analysis architectures. The thesis consists of 12 chapters which span across three main sub sections. The rest of the chapters of this thesis are organized as follows.

Chapter 2 describes the related work in the area of large scale data analytics. The thesis reviews the work done on performance characterization and benchmarking of large scale data analysis systems. Related work on performance optimization of large scale data analysis systems and stream program performance maintenance is described next. The thesis surveys state-of-the-art on stream job performance optimization. Finally, related work on creation of modularized software architectures for large scale data analysis is described including work on graph processing libraries, domain specific languages, and graph data management and mining.

Chapter 3 provides a background for the technologies used for implementing the data analytics optimization techniques described in this thesis. The chapter describes the IBM System S and its associated SPADE programming language. The chapter also describes Yahoo S4, and EsperTech’s Esper which are two popular stream processing software. Furthermore, the chapter provides a brief introduction to X10 which is a PGAS (Partitioned Global Address Space) language that promotes creation of high performance computing applications in a scalable and productive manner. Finally, the chapter concludes by providing an overview for graph database systems.

Part I of the thesis is dedicated for performance optimization of stream data analytics. Chapter 4 describes workload characterization of stream processing systems conducted with the aim of identifying software architectural properties that affect the performance of stream processing systems. Chapter 5 describes StreamFarm which is a workload generator platform for stream computing environments. StreamFarm has been used as the workload generator during the experiments of Albatorss mechanism. Chapter 6 explains Hirundo which is a mechanism for automated production of optimized data stream programs. Hirundo’s auto-tuning mechanism is centralized around code transformation algorithms based on parallel streams pattern which represents a core

contribution of this thesis. The chapter describes three transformation algorithms called Uni-Operator, Bi-Operator, and Tri-Operator transformation as well as the concept of operator fusion. Chapter 7 describes automatic stream query graph optimization in the presence of multiple stream jobs. It explains the multi-data stream optimization algorithm proposed by this research and its implementation on a mechanism called Tahitica. Chapter 8 describes Albatross which is a mechanism for performance maintenance of stream computing systems. Albatross uses the same code transformation algorithm of Hirundo; However, Albatross's aim is to apply such automatic code transformation techniques for performance maintenance of large data analysis systems that eliminate the need for keeping additional backup compute nodes.

Part II of the thesis describes performance optimization of graph data analytics. Chapter 9 describes performance aspects that need to be considered when implementing graph database benchmarks. The chapter describes XGDBench which is a graph database benchmarking platform intended for full filling the need for realistic benchmarking of property graphs. The chapter elaborates on characteristics of different types of workloads proposed by the benchmark as well as performance characterizations conducted on graph database systems. Furthermore, the chapter describes how workload spikes can be modeled and generated using XGDBench. Next, the thesis describes different software modularization techniques carried out for defining optimized data analysis processes. These software modularization techniques can be utilized in combination with the software auto-tuning techniques described in Part I for creating optimized large scale data analysis workflows. Chapter 10 describes the design of ScaleGraph 1.0 which is a library for billion scale graph data analysis. The library's design abstracts out the complex graph data structures and offline graph analysis algorithms into simple API calls. This enables creation of graph analysis workflows that can be autotuned using the software auto-tuning techniques described in Part I. Chapter 11 describes Exedra which is a domain specific language for large graph data analysis created with the aim of applying the software autotuning techniques presented in this thesis. Exedra allows for creating unified graph analysis processes using different graph analysis software which enables creation of rich workflows that cannot be created using single software.

Part III describes the conclusions arrived as well as further work for thesis. Chapter 12 describes the overall achievements quantitatively. First, the chapter describes the overall observations made during performance analysis conducted on stream processing systems. Then the chapter provides the overall conclusions arrived by the experiments conducted using Hirundo, Tahitica, and Albatross in the context of stream data analytics optimization. Next, the chapter elaborates on the overall conclusions made on ScaleGraph 1.0, Exedra. Furthermore, the chapter describes the problems encountered and the limitations of the solutions in a general context. Next, the chapter describes some further work for creating large scale optimized data analysis architectures. The chapter describes application of software automatic tuning techniques for graph analysis workflows. Furthermore, the chapter describes Acacia, a distributed graph data base server intended to operate in hybrid clouds. Acacia abstracts out the online graph analysis activities that can be conducted on a data analysis workflow. Acacia addresses completely

different aspect of large graph analysis compared to the ScaleGraph 1.0 because of this reason.

1.9 Summary

The first chapter of this thesis focused on providing an introduction to the materials presented in the next chapters. The chapter provided an explanation about the term Large Data Analytics in Section 1.1. The chapter described types of data analytics described in this thesis 1.2. Next, the chapter made a brief survey of the historical background of large scale data analytics in Section 1.3. Challenges faced by data analytics in future large scale HPC systems such as Exascale systems is described in Section 1.4. Issues of current data analytics software was described in Section 1.5. It specifically pointed out the problem, that is the current large scale data analysis applications are less optimized in terms of execution performance and programmability. Next, the aim and the objectives of the thesis were presented in Section 1.6. The proposed solution was described in Section 1.7. An insight to the structure of this dissertation was provided under the Section 1.8. Next chapter of this thesis surveys current state-of-the art in creating optimized large scale data analytics.

State of the Art in Large Scale Data Analytics Optimization

The aim of this thesis is to introduce new techniques and concepts for highly scalable software modularization, automatic code transformation, workload characterization, and benchmarking to improve productivity and execution performance of data analytics software architectures for large scale systems. The work described in this thesis builds up on a number of previous work scattered in various areas in large scale data analytics such as stream query performance optimization, highly availability and fault tolerance of stream processing systems, workload characterization and benchmarking of stream processing systems as well as graph databases, middleware, libraries for large graph analysis. This chapter reviews the research efforts in these areas and explains how the work in this thesis improves over them. The chapter begins with a survey of related work in the area of workload characterization and benchmarking of stream processing systems in Section 2.1. Next, chapter describes benchmarking of graph database systems in Section 2.2. Efforts in stream processing performance optimization and multiple stream query performance optimization are described in Sections 2.3 and 2.4 respectively. Related work on stream job performance maintenance are described in Section 2.5. Current state-of-the-art in design and implementation of graph processing libraries is described in Section 2.6. Design and implementation of domain specific languages and related research are described in Section 2.7. The chapter ends with a survey of related work on graph data storage and management in clouds in Section 2.8.

2.1 Workload Characterization and Benchmarking of Stream Processing Systems

There have been several previous work on characterizing performance of stream processing systems. Among them, Mendes *et al.* have conducted a performance evaluation

of three event processing systems by running several micro-benchmarks [86]. Their intention was to provide a first insight into the performance of event processing systems. In this thesis we try to delve more deep into the performance characteristics of such systems. Mendes *et al.* have used Esper similar to this thesis, however, their study has been conducted in single node settings. Different to them, this thesis implemented all the benchmarks using Esper in distributed settings using Java Messaging Service (JMS) [87]. Furthermore, the performance experiments described in this thesis were on different node configuration from 1 to 12 nodes for all the three stream processing software.

Suzumura *et al.* made a performance study considering ETL (Extract-Transform-Load) scenario of System S [88]. However, this thesis's intention is completely different from their work. While they evaluate performance of System S in the context of ETL applications, the thesis aims for identifying the characteristics of stream processing systems in a more general context. The work done by Parekh *et al.*, and Zhang *et al.* study methods for characterizing resource usage profiles, and characterizing the resource usage of Processing Elements (PEs) respectively [89][90]. Both these works are based on System S, and their aim is to provide solid foundation for modeling, and predicting resource usage of stream programs which is different from the motivation of this thesis which is on performance characterization.

Arasu *et al.* described a stream data management benchmark [15] which has been originally used by members of Aurora [91], and STREAM [92] projects to compare performance characteristics of Data Stream Management Systems (DSMS) (i.e., stream processing systems). Recently Zeitler *et al.* implemented the Linear Road benchmark [15] on SCSQ DSMS [93]. However, this thesis does not concentrate on single concrete benchmark, rather the thesis uses a collection of applications which is another distinguishing point of this work from their's. Yet Linear Road can be considered as a possible avenue for extending the work described in this thesis.

On their paper introducing S4 [19], Neumeyer *et al.* have conducted two experiments (online, and offline) on S4 with a Streaming Click-Through Rate computation application. While they conducted their experiments maximum at 20000 input events per second rate, this thesis conducts the experiments with several magnitudes higher input data rates to compare performance of S4 with respect to System S.

Mendes *et al.* described FINCoS [94] Framework which is a Java-based framework for load submission and performance measurement of Complex Event Processing systems. They extended their work on FINCoS to create an event processing benchmark called Pairs [95]. The key difference of their work with StreamFarm described in this thesis is that StreamFarm goes beyond acting as a platform that delivers support for running user defined benchmark applications. StreamFarm provides a flexible interface for controlling the workload generation as well as infrastructure management. This allows StreamFarm to be used by third party performance optimization applications (e.g., Albatross) easily. There have been previous similar efforts for introducing query languages for controlling distributed systems [96].

Thies *et al.* conducted an empirical characterization of large collection of stream programs written in StreamIt [97] programming language [98]. However, the characteri-

zation of the large set of stream programs has not been conducted in distributed settings. Their focus was on architecture-independent program characteristics that are inherent properties of an algorithm rather than the program’s mapping to any given machine. But the workload characterizations conducted in this thesis are made on stream applications written for executing in distributed environments which is a significant difference from the work done by Thies *et al.*.

2.2 Benchmarking of Graph Databases

Relational databases have standard benchmark suites such as TPC [99] benchmark suite. However, the emerging field of Graph DBMSs still lacks such support [100]. This section reviews current landscape of graph database benchmarking and explains how XGDBench: the graph database benchmark described in this thesis improves over the present state-of-the-art.

HPC Scalable Graph Analysis Benchmark represents a compact application with multiple analysis techniques that access a single data structure representing a weighted, directed graph. This benchmark is composed of four separated operations (Graph construction, classification of large vertex sets, graph extraction with Breadth-First search, and graph analysis with Betweenness Centrality [101]) on a graph that follows a power-law distribution [102]. However, this benchmark does not evaluate some features that are inherent to graph databases such as object labeling, attribute management, etc. [102] a feature that will dominate future graph database systems. Furthermore, HPC Scalable Graph Analysis Benchmark does not evaluate the OLTP features of graph DBMSs.

A survey of graph database performance on the HPC Scalable Graph Analysis Benchmark was conducted by Dominguez-Sal *et al.* [103]. They evaluated the performance of four graph databases/stores Neo4j [104], Jena [105], HypergraphDB [106], and DEX [107]. However, a limitation of their work is that it only implements HPC Benchmark and does not consider attribute graphs. In a different work Dominguez-Sal *et al.* provided a discussion on graph database benchmarking [108]. They reviewed application of graph databases in different application areas such as social network analysis, proteomics, recommendation systems, travel planning and routing. They explained that social network analysis would be a representative application for general graph database applications. Their work motivated us for using Multiplicative Attribute Graph (MAG) [109] as the synthetic graph generator model for XGDBench.

Recently, a benchmark for graph traversal operations on graph databases was described by Ciglan *et al.* [110]. They designed their graph database benchmark focusing on traversal operations in a memory constrained environment where whole graph cannot be loaded and processed in memory. Similar to them, XGDBench implements graph traversal as one of the workload items. However, unlike them the study described in this thesis is purely based on graph database servers. The thesis does not focus on benchmarking embedded graph databases (i.e., graph databases that run on the same process with the user application). It is because XGDBench is a benchmarking framework rather than a benchmark specification. Further explanation on this decision is given in Section

9.5.3.

Holzschuher *et al.* made a performance comparison of graph query languages [111] using Cypher [112], Gremlin [113], and native access of Neo4j [114]. Their study is focused around Apache Shindig [115] which is a container for hosting social applications. They replace the traditional relational back-end of Shindig with Neo4j and compare alternatives for querying data with different back-ends in terms of their performance characteristics. Their work is different from this thesis's because the thesis's focus is on creation of a graph database benchmarking platform rather than characterizing the performance of graph query languages. However, the graph database benchmarking platform proposed in this thesis can be used to create such performance characterizations.

There are popular benchmarks for graph data stores from semantic web community such as Lehigh University Benchmark (LUBM) [116], Berlin [117], DBpedia [118], SP2Bench, [119] etc. None of these benchmarks employ statistical graph generator model which allows very large scale, realistic synthetic graphs.

Vicknair *et al.* compared performance of Neo4j graph database and MySQL [120] for graph data storage. However, their study did not focus specifically cloud environments. Rohloff *et al.* conducts an evaluation of triple-store technologies for large data stores [121]. Triple-stores also have been used as graph database management systems in various occasions. This thesis uses AllegroGraph: a famous triple-store (which is also popular as a graph database) for evaluation of XGDBench due to this reason. However, Rohloff *et al.*'s work was conducted using the LUBM and their study focused on evaluating triple store technologies. Another similar work on benchmarking RDF stores has been conducted by Thakker *et al.* [122]. However, they used the University Ontology Benchmark (UOBM) [123] for this purpose.

Graph 500 is a benchmark suite intended for benchmarking data intensive super-computing applications [124]. Similar to HPC Scalable Graph Analysis Benchmark the intended application scenario of Graph 500 is a compact application that has multiple analysis techniques accessing a single data structure representing a weighted, undirected graph. Current implementation of Graph 500 does not consider different application scenarios such as graph databases. Its focus is on benchmarking super computer performance. Graph 500 has been implemented in C++ using MPI. In contrast to Graph 500, XGDBench framework has been implemented in X10 which provides support for productive programming of future Exascale systems compared to use of MPI.

2.3 Stream Processing Performance Optimization

Optimization of data flow graphs has been widely addressed research issue. Earlier efforts in automatic parallelization of sequential programs studied methods for automatic data partitioning and distribution of computations in distributed-memory multicomputer systems [125][126][127]. PARADIGM is a compiler that makes data partitioning decisions for Fortran 77 procedures. PARADIGM had the feature of decomposition of the data partitioning problem into a number of sub-problems, each working on a different distribution parameter for all the arrays [128]. However, the distributed computing

model handled by these works differ from stream computing model. The Hirundo system described in this thesis concentrates more on computations that are data-intensive, and does not conduct any static code optimizations like these works.

Automatic composition of workflows has been addressed by Qin *et al.* [129] and Liew *et al.* [130]. Compared to them, this thesis's focus is on automatic optimization in the context of stream computing, and Hirundo ensures the optimization process does not get affected by node failures. This issue has not been addressed by these works.

MPI programs has been autotuned by various previous work. ATune is an automatic performance tuning tool that uses machine learning techniques to identify program-specific optimal settings for a subset of the Open MPI's runtime parameters [131]. Similar to the work conducted in this thesis, Faraj *et al.* describe that collective communication routines must be able to adapt to the system architecture and use different algorithms for different situations. Their proposed system adapts to a given platform and constructs routines that are customized for the platform [132][133]. A layout aware optimization strategy for collective I/O has been described by Chen *et al.* [134].

Hirundo introduces use of *Transformer Blocks* during its data flow graph transformations in the context of stream computing. There has been similar use of recurring patterns for optimizing workflows by Yaikhom *et al.* [135] and Hall *et al.* [136].

There have been work on performance prediction of parallel applications by partial execution [137] using skeletons [138] etc. A major bottleneck in software autotuning is the computation time required to test a large collection of possible code variants [139][140]. There have been previous work on reducing this time by creation of improved search techniques that are capable of finding high-performing parameter configuration quickly [140]. Furthermore, recent relational database servers use empirical cost information for producing optimized query plans [141][142][143]. Yet Hirundo follows a different approach for identifying optimized sample programs by integrating results from partial execution of sample programs with empirical data.

Subquery optimization of relational database systems by Bellamkonda *et al.* [141] has similarity to what Hirundo does since both the approaches use code transformation as the means of optimization. Table partitioning in relational databases is a technique used for optimizing SQL query performance [144]. This technique is analogous to Hirundo's Tri-OP Transformation.

Stream graph partitioning [145][146] tackles the problem of stream program performance optimization at lower levels of stream processing environment compared with the approach followed by this work. Hirundo approaches the solution from the source program level of a stream application.

Recently there has been interest of automatically optimizing programs written for MapReduce systems [147][148][149]. Similarly compiler of DryadLINQ [150] system performs static optimizations which enables automatic synthesis of optimized LINQ code. However, these systems do not perform high level code transformations like Hirundo does during the process of optimization. Hirundo outputs a ranked list of optimized sample programs, whereas these systems performs their optimizations in lower levels.

Evolutionary algorithms [151] (e.g., Genetic Programming [152]) are closely related

to our stream program optimization approach. Similar to Hirundo an evolutionary algorithm maintains a population of sample programs solutions for a given problem. Each program is evaluated by a fitness function, which measures the quality of its corresponding candidate solution [151]. Yet the main difference from this thesis's work is that each program evolves to better and better versions by cross over mutations with each other. In Hirundo only the input stream program is mutated and the generated sample programs are not used for further mutations. This is because our focus is on obtaining higher performance versions of a given stream application within a short period of time (i.e., within few minutes) which can be achieved by transforming the input program.

There is a class of programming languages which are also designed and implemented with the motivation for supporting program construction with stream programming model. Brook [153], StreamIt [97][154][155], Spidle [156], Cg [157] are examples for such languages which follow stream programming model. However, all of these languages are intended for programming only on single node settings. They exploit the hardware parallelism available in multicore systems. StreamIt is a portable programming language intended for programming high performance signal processing applications [158]. Gordon *et al.* demonstrated how StreamIt compiler can attain robust multicore performance in the face of varying application characteristics [159]. They exploited task, data, and pipeline parallelism available in stream programming model for improving performance of StreamIt applications. However, different from them the work described in this thesis is focusing on distributed data stream processing systems' job performance optimization. Furthermore, they do not conduct any source code level program transformations. In a different work Lamb *et al.* demonstrated that multiple algorithmic transformations on Digital Signal Processing (DSP) which are traditionally hand tuned by DSP experts can be completely autotuned by the compiler [160]. Castro *et al.* described design space exploration technique for reducing the minimal memory requirement of running stream applications on Multi-Processor Systems on Chip (MPSoC). Similar to Hirundo, they explore a large number of equivalent program variants and select a set of variants that minimizes a buffer evaluation metric [161]. They made a formalism of stream data flow graphs which is very close to StreamIt using Cyclo-Static Data Flow (CSDF) graphs [162]. However, both the aforementioned works are conducted in the context of embedded systems.

2.4 Multiple Stream Query Performance Optimization

Multiple stream query optimization is relatively new area of stream query optimization that descends from the broad and well established field of database query optimization. Most of the work described in previous section are also related to multiple stream query performance optimization.

This thesis's work is closely related to the area of optimization of multiple continuous queries. Seshadir *et al.* described a distributed stream query service that takes the advantage of the opportunities to reuse the same distributed operator for multiple and different continuous queries [163][164]. Their intent is similar to this thesis's in the sense

of sharing compute resources. However, unlike them this thesis do not reuse SPADE operators in its optimizations because BUILT-IN operators of SPADE are lightweight operators and therefore those can be easily instantiated rather than reusing. Nonetheless, UDOPs which are in general heavy operators can be considered for reuse between stream graphs.

Similar to this thesis's use of parallel pattern for stream program optimization [165], Schneider *et al.* described a compiler and runtime system that automatically extracts data parallelism for distributed stream processing [166]. However, their parallelization methodology requires support from the underlying runtime. Furthermore, through SPADE annotation scheme introduced through Hirundo [165] this thesis is able to conduct transformations of UDOPs as well while they follow strict rules in code transformation and does not allow such transformations. Wu *et al.* described a framework that supports parallelization of stateful operators. However, their method needs a state sharing mechanism called CAPSULE [167] which introduces an additional overhead on the cluster. The optimization mechanism described in this thesis creates a partitioned state whenever its applicable. If not we do not transform such stateful operators. They consider the fact whether parallelizing an operator could be beneficial or not during their optimization. Similar to them, we determine which operator(s) to be parallelized based on profiling results.

Wolf *et al.* described System S scheduler which is known as SODA (Scheduling Optimizer for Distributed Applications) [168]. The scheduler maximizes a utility-theoretic function based on the "importance" measured at the terminal streams of the data flow graphs in scheduling the stream data flow graphs. SODA is concerned with resource allocation for each PE for the jobs that will be run in the environment. SODA tries to balance the load across all resources (such as network bandwidth, node processing capacity) in the system. Compared to query performance optimization mechanisms such as Tahitica, the operations of SODA happens at the lower level of the stream processing middleware. Since SODA is just a scheduler it does not involve in automatic code transformations such as the ones done by Hirundo and Tahitica.

Operator fusion has been used as a method for optimizing SPADE applications in System S. Khandekar *et al.* described COLA: an optimization scheme that is applied at the compile time that groups operators in a SPADE program into appropriate run-time software units called Processing Elements (PEs) [145]. This thesis's methodology is different from them in two aspects. First, they follow a top-down fusion approach where all the operators are fused together and then the large PE is split into two separate smaller PEs. The optimization is continued until a suitable operator to PE partitioning is met. This thesis follows a bottom-up method of fusion where we initially place all operators into distinct PEs and iteratively fuse them into larger PEs till some criteria is met. Such approach is followed because in the case of multiple stream query optimization, it is rather impractical for fusing all the jobs' operators into single PE at first. Gedik *et al.* follows a similar bottom-up approach with their fusion strategy [169].

2.5 Stream Job Performance Maintenance

Stream program performance maintenance is associated with work conducted on highly availability, and fault tolerance of stream processing systems. Hwang *et al.* [170][171], and Gu *et al.* [172][173] described two main approaches for highly availability called Active Standby (AS) and Passive Standby (PS). In AS two or more copies of a stream job run independently on different nodes. In PS a primary copy periodically checkpoints its state to another node and uses that copy for recovery during failures. Both these approaches involve replication. The method described in this thesis is completely different from AS since, our method is based on automatic code generation, and does not require backup machines. However, we share a common feature with PS since our method keeps incoming data tuples in main memory during the crash recovery period to avoid data losses.

An in-situ MapReduce architecture was described by Logothetis *et al.* that mines data (i.e., logs) on location where it appears [174]. They use load shedding techniques to improve fidelity under limited CPU resources. Albatross, the mechanism proposed for performance maintenance in this thesis also incorporates incremental load shedding similar to these works that allows Albatross to maintain performance during drastic failures. However, different from them our approach does not conduct load shedding from the point it detects where there is resource deficiency. Instead the load shedding is done only when the automatic code generation cannot withstand the workload.

As mentioned previously Khandekar *et al.* [145] and Wolf *et al.* [168] discussed optimizing stream job performance in the context of operator fusion (COLA) and operator scheduling (SODA) respectively. However, their approaches do not focus on maintaining an agreed level of performance. Instead their focus is on performance improvement and distributed scheduling of PEs. Furthermore, COLA works on finding the optimal fusion of processing elements (using compiler outputs) of a single program whereas, Albatross constructs and uses many versions of input program(s) during its operation. This is because Albatross needs different versions of sample programs with different performance levels during performance maintenance process. Furthermore, Albatross uses parallel pattern based code transformation which is different from fusion used in COLA.

Wang *et al.* [175] uses a generalized solution to avoid Single Point of Failure (SPOF) in master-slave systems. They propose highly availability as a service (HAaaS) to provide the high availability for several subsystems in an unstructured data management system. Similar to this thesis they use Zookeeper [176] to elect a new leader whenever needed.

2.6 Graph Processing Libraries

Construction of graph processing libraries with support for variety of graph algorithms has been a widely studied area. One of the famous examples for such graph libraries is igraph [60]. Igraph has been heavily used by complex network analysis community. It has support for classic graph theory problems such as Minimum Spanning Trees, and Network Flow. Core of the igraph has been written in C. There are two extensions for

igraph, one in R and another in Python. Lee *et al.* created Generic Graph Component Library (GGCL) [177] which is a library built on C++ STL. Graph algorithms on GGCL do not depend on the data structures on which they operate. Stanford Network Analysis Package (SNAP) [178] is a general purpose network analysis and graph mining library developed by Leskovec *et al.* SNAP version 2011-12-31 supports maximum 250 million vertices and 2 billion edges [178]. The library calculates structural properties, generates regular and random graphs. Similar to ScaleGraph library described in this thesis, SNAP supports attributes on nodes and edges and has been used to analyze large graphs with millions of nodes and billions of edges. However, compared to ScaleGraph, one of the major limitation of all the above mentioned libraries is that they are made to run on workstations.

Boost Graph Library (BGL) is a C++ STL library for graph processing [179][180]. Part of the BGL is a generic graph interface that allows access to the graph's structure while hiding the details of the implementation. A parallel version of the library (PBGL) [70] has been developed using MPI. However, if the user is not well versed in use of C++ and STL, the learning curve of the BGL becomes very steep [181]. Hence BGL and PBGL might not be an acceptable solution for application programmers at large. Note that by using the term "Application Programmer" this thesis represents not only high-end parallel application programmers but also application programmers on next generation systems such as SMP-on-a-chip and tightly coupled Blade servers [182]. ParGraph [71] is a generic parallel graph library which is comparable to PBGL. ParGraph is written in C++ and it has similar syntax to PBGL. Different from BGL, PBGL, and ParGraph; ScaleGraph 1.0 requires less amount of code for specifying a graph computation requiring lesser programming effort. This way ScaleGraph promotes productivity in HPC graph analysis.

Standard Template Adaptive Parallel Library (STAPL) [183] is a generic library with similar functionality to Boost. STAPL targets scientific and numerical applications, also intended for exploiting parallelism for graph algorithms. However, like Boost libraries, STAPL does not define broadly applicable abstractions for graphs [184].

Java Universal Network/Graph (JUNG) is a comprehensive open-source graph library [61][185]. It supports variety of representations of graphs such as directed and undirected graphs, multi-modal graphs, hypergraphs, and graphs with parallel edges (i.e., multi-edges). Since JUNG has been developed using Java, it offers the interoperability with rich third party libraries written in Java. Current distribution of JUNG includes a number of graph algorithms related to data mining and social network analysis [61]. Current version of JUNG does not support distributed implementation of algorithms which is a limitation in applying it to distributed graph processing scenarios.

Combinatorial BLAS is a high-performance software library written in C++/MPI for graph analysis and data mining [72]. Knowledge Discovery Toolkit (KDT) which is a Python graph analysis package has been introduced recently utilizing Combinatorial BLAS as the underlying graph processing infrastructure [186]. Different from them, ScaleGraph 1.0 library is completely written in X10 which is a PGAS language aimed for productive programming on multi-core, multi-node systems.

While there has been large scale graph algorithm implementation on main stream parallel architectures such as distributed memory machines; there are some other studies focusing on specific machine architectures which are currently less popular. Examples include the works by Madduri *et al.* [73], Bader *et al.* [74][75], and Berry *et al.* (MultiThreaded Graph Library (MTGL)) [76]; which describe the ability of using massively multithreaded machines to implement graph algorithms. While distributed memory application developers focus on maximization of locality to minimize interprocess communication, program developers for massively multithreaded machines having large shared memory (e.g., Cray MTA-2) do not focus on locality or data exchange [77]. Some of the work of this domain (E.g., MTGL) have been extended to commodity processors yet with lesser performance [78]. Different from them, this thesis concentrates on productivity of specifying graph computations in distributed settings while maintaining scalability aspects in commodity machines ranging from developer laptops to super computers.

There have been prior work on specifying graph computations on X10. Cong *et al.* worked on creating fast implementations of irregular graph problems on X10 [187][188]. They also worked on creating an X10 Work Stealing framework (XWS) with the aim of solving the problem of present software systems not supporting irregular parallelism well. However, both these works do not focus on creating a Graph API with well-defined abstractions for representing graphs. This thesis emphasizes creation of such library since we believe this will support X10 application developers to directly use the support provided by the library for graph computations.

While ScaleGraph 1.0 creates a large graph analysis library in the domain of PGAS languages, Pregel [77] is a computational model for analyzing large graphs with billions of vertices and trillions of edges. Pregel focuses on building a scalable and fault tolerant platform with an API that is flexible in expressing arbitrary graph algorithms using vertex centric computations. Similar to PBGL, Pregel's C++ API described in [77] requires more programming effort compared to ScaleGraph 1.0's API which is also targeted for users outside the HPC domain.

Kulkarni *et al.* mentioned that concurrency should be packaged, when possible, within syntactic constructs that makes it easy for programmer to express what might be done in parallel, and for the compiler and runtime system to determine what should be done in parallel [189]. Hall *et al.* emphasized the importance of abstractions for big data analytics [190]. ScaleGraph 1.0 follows the same concept and tries to introduce well-defined abstractions for massive graph analysis scenarios.

2.7 Design and Implementation of Domain Specific Languages

This section described related work for Exedra which is a domain specific language for large graph analysis (presented in Chapter 11). Exedra provides a unique programming interface for large graph analysis with the emphasize on programmer productivity. It descends from large graph analysis software on High Performance Computing (HPC)

environments yet significantly deviates from the current approaches. Several other research have been conducted on creating DSLs for graph analysis, and increasing graph analysis productivity. Yet Exedra is a DSL for creating unified large graph analysis workflows to increase graph analyst's productivity targeting large scale systems.

Green-Marl is a DSL designed specifically for graph analysis algorithms [191][192]. Compared to Exedra which is an interpreted language, Green-Marl is a compiled language. Green-Marl code gets compiled to C++ code while Exedra statements are directed to the underlying library/middleware. Furthermore, graph data structures in Green-Marl are immutable. Yet, in Exedra graph's content can be changed during execution. Currently Green-Marl does not cover broad set of analytics such as clustering, community detection, etc. which are available in Exedra.

We adapt similar syntax to Igraph/R [60][193] package when creating Exedra DSL. Igraph is written in C and we use Igraph/R to denote its R interface [193]. The main reason for this design decision was due to the fact that Igraph/R package is currently widely used by the community of complex network analysts and it would be easier for domain experts to utilize the same syntax when they work with Exedra. However, there are limitations for the Igraph/R grammar such as specification of graph traversal operations that is addressed in Exedra.

Gremlin [194][195][196] and SPARQL [197] are two examples for DSLs that can be used for specifying graph analysis tasks. Gremlin is a graph traversal language and is completely oriented toward specifying queries over graph data. Yet, Gremlin does not have constructs to express specific algorithms such as degree distribution calculation, graph clustering, pattern matching, etc. Exedra improves over Gremlin in this aspect which is very important for allowing users to specify a variety of graph analysis algorithms. SPARQL on the other hand is the norm for accessing RDF graphs. Yet it also lacks the support for expressing graph analysis algorithms. SPARQL's syntax is closer to SQL while Exedra adopts R syntax used in Igraph/R package.

As mentioned in Section 2.6 Knowledge Discovery Toolbox (KDT) is graph analysis library written in Python which aims for high level exploratory graph analysis by domain experts and lower level specification of graph operations by the developers of graph analytics [186]. KDT is built on top of the CombinatorialBLAS [72] which is a C++ library for combinatorial computational kernels. Current version of KDT is interfacing only CombinatorialBLAS, and it has not been extended for other back ends. Hence KDT can only utilize the computational functions offered by Combinatorial BLAS.

Neptune is a DSL based on Ruby programming language which automates configuration and deployment of scientific software frameworks over disparate cloud computing systems [198]. Neptune integrates support for MPI, MapReduce, Unified Parallel C (UPC) [199], X10 [200], etc. Neptune is closely related to Exedra, however Exedra's focus is only on large graphs.

Das *et al.* created Ricardo [201] a data analysis platform which allows analysts to work on huge data sets using R language. Yet, Ricardo does only integration of R with Hadoop. Dipper integrates multiple programming libraries/middleware for large graph analytics.

Ansel *et al.* presented PetaBricks [202][203] which is an implicitly parallel language and compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. PetaBricks have a similar aim to Exedra although Exedra is specifically for graph analysis where as PetaBricks is applicable to other areas of computing.

Kennedy *et al.* described a strategy called Telescoping languages [204] where they developed a framework for automatic generation of optimizing compilers for new domain-specific languages. They generate a library aware optimizer which uses the information gathered during preprocessing to carry out fast and effective optimization of high level scripts. Different from them our focus is only on one single domain-specific language. However, we do a library aware optimization, a similarity that we share with their approach.

A summary of graph analysis software described in this section is shown in Table 2.1.

Table 2.1: A summary of graph analysis software.

	Supports Distributed Processing?	Mutable Graph Data Struc- ture?	Interpre- ted?	Graph analysis library?	Multiple graph analysis software
Igraph	No	Yes	No	Yes	No
KDT	Yes	Yes	Yes	Yes	No
ScaleGraph	Yes	Yes	No	Yes	No
SPARQL	No	Yes	Yes	No	No
Gremlin	No	Yes	Yes	No	No
Green-Marl	No	No	No	No	No
Exedra	Yes	Yes	Yes	No	Yes

Note that the last column of Table 2.1 indicates whether the graph analysis software supports utilization of multiple graph analysis software systems during its operation. More details about this feature are available in Chapter 11.

2.8 Graph Data Storage and Management in Clouds

Graph data storage and management has been a rigorously addressed issue by database, complex network analysis, and cloud computing communities in recent times [205]. Many of these efforts have materialized in to solid implementations of graph database servers such as Neo4j [206], OrientDB [207], AllegroGraph [208], DEX [107], etc. but all of these run on single workstations only. Li-Yung Ho *et al.* [209] described a distributed graph

database architecture for large social computing. Their focus was on creation of a distributed graph database server that runs completely on either public or private clouds. They use Neo4j as their storage back end. However, their underlying communication has been implemented using MPI. Barguno *et al.* described the architecture of Parallel-GDB which is a graph database that uses random graph partitioning to avoid complex partitioning methods on graph topology [210]. Trinity is a graph engine for analysis of large scale graphs which incorporates graph storage functionality [211][212][213]. Their in-memory distributed database supports both online query processing, and offline analytics on graphs. Graph databases such as Trinity, Titan [214], InfiniteGraph [215], FlockDB [216] etc. can operate distributedly. However, no known studies exist that describe their capabilities for operating in hybrid cloud environments.

There is substantial amount of research literature [217][218][219] devoted for stream processing in hybrid cloud environments. Gulisano *et al.* combined elasticity with dynamic load balancing to minimize the computational resources utilization [217]. Kleiminger *et al.* presented a combined stream processing system that adaptively balances workload between a dedicated local stream processor, and a cloud stream processor [218]. Ishii *et al.* described a method and an architecture to use Virtual Machines (VMs) in a cloud environment elastically to satisfy real-time processing requirements.

Yan-hua *et al.* described a cloud database route scheduling algorithm using a combination of the genetic algorithm, and ant colony algorithm [220]. Das *et al.* mentioned that elasticity, scalability, fault tolerance, and self-manageability are fundamental requirements for database management systems [221]. They presented ElasTraS a database management system designed for this purpose. ElasTraS borrows from the design philosophy of scalable Key-Value stores to minimize distributed synchronization and remove scalability bottlenecks. Yu *et al.* describe that multi-tenancy is one of the key features of database-as-a-service (DBaaS) offering, where a large volume of databases with different SLA requirements are co-located in one environment [222]. However, in this thesis our focus is on single-tenancy. Minhas *et al.* describe a highly available database management system that is based on VM replication, that leverages the capabilities of the underlying virtualization layer [223].

TIRAMOLA is a cloud-enabled framework for automatic provisioning of elastic resources on any NoSQL platform [224]. They apply this framework on top of a fully distributed RDF store back-end by an elastic NoSQL database. Chen *et al.* developed a novel graph partitioning framework to improve the network performance of graph partitioning itself, partitioned graph storage, and vertex oriented graph processing [225].

2.9 Summary

This chapter described related work for constructing optimized large scale data analytics. The chapter described related work for performance characterization of stream processing systems in Section 2.1. State-of-the-art techniques on graph database benchmarking and other related work for graph database benchmarking was described in Section 2.2. Next, stream processing optimizations related to Hirundo were described in Section

2.3. Related work on multiple stream query optimization and stream job performance maintenance were described in Sections 2.4 and 2.5 respectively. State-of-the-art graph processing libraries were surveyed in Section 2.6. Related work on the design and implementation of domain specific languages was presented in Section 2.7. Related work on graph data storage and management in clouds were described in Section 2.8. Next chapter provides background for the technologies used in the rest of the thesis.

3

Technologies for Large Scale Data Analytics Optimization

This chapter provides brief introduction for the technologies adapted for conducting the work mentioned in this thesis. From the description provided in Chapter 1 on the challenges faced by large scale data analytics architectures and the state-of-the-art techniques for automatic optimization of large scale data analysis systems it was obvious that the need for having well-defined abstractions and software modularizations for constructing optimized data analytics scenarios. E.g., The programming model's design of the programming languages plays a big role in making many optimizations easier or difficult to apply [226]. The advantage of data flow languages compared to many contemporary approaches for data analytics is that data flow languages provide better abstractions for implementing software autotuning techniques compared to use of general purpose languages such as C, C++, Java, etc [227].

The thesis compares the features of the alternative technologies by comparing their features. However, the thesis do not explain technical details of those alternative technologies for maintaining brevity. Section 3.1 gives an introduction to data flow languages, one of the key programming paradigms that allows for creation of optimized large scale data analytics. A description of stream processing software and systems is provided in Section 3.2 which covers the details of System S, S4, and Esper. Next, an explanation on what is meant by High Productivity Computing Systems (HPCS) programming languages is provided in Section 3.3. An introduction to what is meant by a Graph Database is provided in Section 3.4. The section highlights the key reasons for choice of graph databases Neo4j, OrientDB, Titan, AllegroGraph, and Fuseki for benchmarking experiments described in this thesis.

3.1 Data Flow Languages and Their Importance for Large Scale Data Analytics

Data flow languages are driven by the fundamental concepts of data flow graphs [228]. A data flow graph is a graph model of computer programs which expresses possibilities for concurrent execution of program parts [229][230]. In a data flow graph, vertices are called actors that represent operations(functions) and predicates to be applied to data objects, and edges represent channels of data objects to move from a producing actor to a consuming actor. This allows control and data aspects of a program to be represented in one unified model. In this model of operation there is no current operation, and each operator is free to execute when its data arrives. This promotes data flow languages to be an ideal candidate for large scale data analytics because there are many data that need to be processes online as and when those appear to the system.

3.1.1 SPADE - An Operator-Based Stream Processing Language

The *Stream Processing Application Declarative Engine (SPADE)* is a language as well as a framework for developing stream processing applications [231]. One of the main design goals of the SPADE language is to reduce the burden associated with designing and implementing complex, large scale, scalable applications that run on the System S [232].

Stream programs developed using SPADE are organized as data flow graphs consisting of operators and streams [145]. Operators are the smallest possible building blocks that are required to deliver the computation an application is supposed to deliver. Streams are directed edges that connect pairs of operators, and carry data from source to destination operators. An example SPADE application's code (Call Detail Record Processing (CDR)) is shown in in Figure 3.1 which contains five different types of operators.

Any SPADE application can be composed of up to six main sections called Application meta-information, Type definitions, External libraries, Node pools, Program body, Function debugging [231][233]. The application shown in Figure 3.1 consists of three of these sections (Application meta-information, External libraries, Program body). In general only two main (mandatory) sections (Application meta-information, and Program body) are sufficient to create a SPADE data flow graph. Our main use of SPADE in this thesis is data flow graph optimization. Therefore, the applications described in this thesis consist of two (or maximumly three) main sections. These main sections of SPADE applications and their purpose are listed in Table 3.1.

SPADE language (the latest version is referred to as Stream Processing Language (SPL) [234][235]) consists of two types of operators called composite and primitive operators [234]. A composite operator consists of a reusable stream subgraph that can be invoked to define streams. Primitive operators are basic building blocks of composite operators. Primitive operators can be further categorized in to built-in operators (BIOP), user-defined operators (UDOP), raw UDOPs, and user-defined built-in-

[Application]

CDR

[Libdefs]

```
package "/var/software/sqlite/include/sqlite3.h"  
libpath "/var/software/sqlite/lib"  
libs "sqlite3"
```

[Program]

```
stream SourceXDR(Username : String, AcctSessionId : String,  
NASIPAddress : String, CallStationid : String, AcctInputPackets : Long,  
AcctOutputPackets : Long, AcctSessionTime : Long, Timestamp : Long)  
:=  
Source()["file:///INFILE.csv", nodeLays, csvFormat]{2,4,5,9,14,15,20,21}
```

#Hirundo-meta:UnFroze:Start

```
stream CAFCheckedXDR(Username : String, AcctSessionId : String,  
NASIPAddress : String, CallStationid : String, AcctInputPackets : Long,  
AcctOutputPackets : Long, AcctSessionTime : Long, Timestamp :  
Long, type:Integer) := Udop(SourceXDR) ["CAFCheck"]{}
```

#Hirundo-meta:UnFroze:End

```
stream ValidXDR(CallStationid: String, AcctSessionId: String,  
AcctPackets: Long, Timestamp : Long) := Functor(CAFCheckedXDR)[  
{ AcctPackets := AcctInputPackets + AcctOutputPackets }
```

```
stream AggregateXDR(CallStationid: String, AcctSessionId: String,  
AcctPackets: Long, Timestamp : Long) := Aggregate(ValidXDR  
<time(20),time(1)>) [CallStationid] { Any(CallStationid),  
Any(AcctSessionId), Max(AcctPackets), Max(Timestamp) }
```

```
Nil := Sink(AggregateXDR) ["file:///output.csv", noDelays, csvFormat]{}
```

Figure 3.1: SPADE code for CDR Application.

operators (UBOP). In Figure 3.1 Source, Functor, Aggregate, and Sink operators are examples for BIOPs. There is one UDOP in the example SPADE code. In the example application the streams that flow into an operator are enclosed within the parenthesis that appear just after the operator type declaration. The streams that flow out from an operator are marked by using the *stream* key word.

Out of the BIOPs available with SPADE, Source operator is used to create a stream from data flowing from an external source [231]. In the example code in Figure 3.1 the Source operator reads a file called INFILE.csv stored in application's data directory. Source operator is capable of parsing, creating tuples as well as interacting with external devices [231]. A Sink operator converts a stream of data from the program into a flow of tuples that can be used by external entities (See Figure 3.2 for an example data flow graph). Both the Source and Sink operators in the example code read/write tuples in CSV format. Functor operator on the other hand is used for performing tuple-level manipulations such as filtering, projection, mapping, attribute creation, transformation, etc. In the sample code the Functor operator sums up the total input and output packet counts mentioned in the data tuple it processes. Aggregate operators are used for grouping and summarization of incoming tuples. Join operator correlates two streams using join predicates. Split operator is used for splitting a stream into multiple output streams using a split condition that determines which of the output streams a tuple will be forwarded to.

SPADE language supports defining stream schemas with Byte, Short, Integer, Long, Float, and Double numerical types; Boolean logical type, and String for representing sequence of characters. SPADE language uses `#` character to create single line comments while multi-line comments are created by surrounding the intended code with `/*` and `*/`.

Another notable feature of SPADE language is its extensibility. There are three ways for extending the language.

- User-Defined Operators (UDOPs) - These operators can be used to enclose external libraries and to provide customized/domain specific processing.
- User-Defined Built-in-Operators (UBOPs) - These are fully templated operators [236]. UBOPs should be created when new, general purpose analytics are identified.
- User-Defined Functions - These are functions declared in C++ header files, and are imported using the *package* key word under the `[Libdefs]` header.

This Section described the most important, and essential features of SPDAE language that are related to the content presented in this thesis. More detailed information of SPADE language can be found from [231] and [233].

3.2 Stream Processing Software

Stream processing software are the middleware/libraries that enable extraction of new insights from data in online satisfying real-time constraints. Several stream process-

Main section	Header	Description
Application meta-information	[Application]	Lists the application name followed by debug/tracing level desired when running the application.
Type definitions	[Typedefs]	Programmer can create aliases for built-in data types to the types they intend to use in the program.
External libraries	[Libdefs]	Programmer can include references to header files that describe the interfaces to external libraries and the file system paths to any library object code files that are used by user-defined operators.
Node pools	[Nodepools]	Allows for defining a pool of UNIX hosts.
Program body	[Program]	This is the place where application itself is defined.
Function debugging	[FunctionDebug]	Under this section, programmer may enter expressions for testing purposes only.

Table 3.1: Main sections of a SPADE program.

ing platforms have been developed in the recent past [237][53][238][239][19][240]. These software provide support for streaming and stored data ingestion, flexible and extensible analytics on streaming data, system support for distributed, scalable, dynamic, and high-performance stream processing. Out of a number of stream processing systems available for conducting research on stream processing, we have chosen IBM System S, Yahoo S4, and EsperTech Esper during the studies conducted in this thesis. Implementation of the data analytics automatic optimization solutions described in this thesis have been conducted on System S. However, S4 and Esper which are relatively new stream processing software have been used for studying performance features of stream processing applications.

3.2.1 System S

System S is an operator-based, large scale distributed data stream processing middleware [232]. The project was initiated in 2003 and it went extensive development at IBM Research [168]. System S comprises of a middleware runtime system and an application development framework intended for supporting large scale, scalable structured as well as unstructured data stream processing. Furthermore, System S can execute a large number of long-running applications in the form of data flow graphs where operators carryout portions of the data processing analytics by consuming and producing new streams. When an application is compiled, a set of runnable processing elements are created. Data flow graphs in System S consist of a set of Processing Elements (PEs) connected by streams, where each stream carries a series of data tuples [237]. The data stream analytics are implemented in Processing Elements (PEs) and they are basic execution containers for portions of the flow graph that are distributed over the compute nodes. The PEs communicate with each other through input and output ports that are connected by streams. The PEs belonging to an application can be logically grouped together to form jobs. The user can then startup the application by submitting it to the middleware runtime system, thereby creating one or more jobs. The System S management tools allows the jobs to be monitored, moved, and canceled. System S provides background services such as scheduling, fault tolerance, storage services, distributed job management, security etc [237]. Architecture of System S runtime is shown on Figure 3.2.

3.2.2 S4

S4 is an open source operator-based stream processing system released by Yahoo Inc. in October 2010 [19]. S4 follows Actors model which makes it considerably different from System S. S4 has a decentralized, and symmetric architecture where all nodes share the same responsibilities. Furthermore, S4 uses a pluggable architecture that keeps the design generic, and customizable to a greater extent.

In S4, the computations are performed by PEs, and communication between the PEs is done in the form of data events. A sequence of events in S4 is defined as a stream. Event emission, and consumption is the only mode of communication between PEs.

System S runtime environment

System S single instantiation is referred to as a Streams instance. Collection of the following services work together to facilitate the running of SPADE applications.

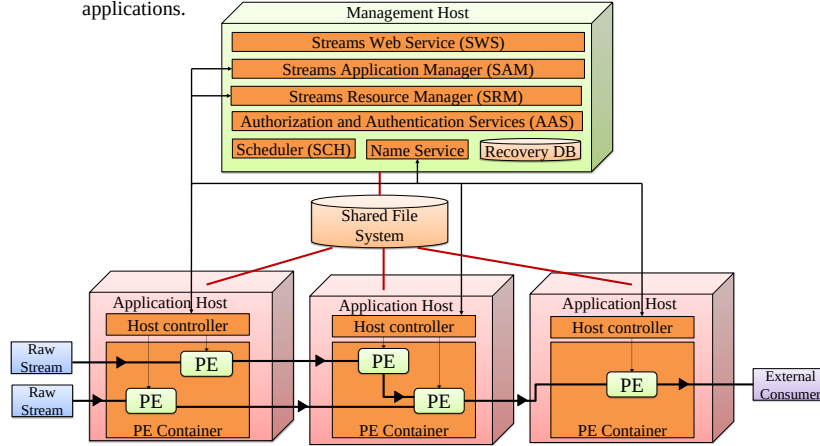


Figure 3.2: Architecture of System S runtime environment

Current version of S4 provides several PEs for standard tasks such as count, aggregate, join, etc. However, custom PEs can be easily created by extending the classes provided by S4 API. All PEs consume exactly the events which corresponds to the values on which they are keyed.

Programming model of S4 has been created in such a way that developers write PEs in Java programming language, and the PEs are assembled into applications using Spring framework. Developers need to essentially implement input event handler `processEvent()`, and `output()` which implements the output mechanism [19]. Architecture of S4 is shown in Figure 3.3.

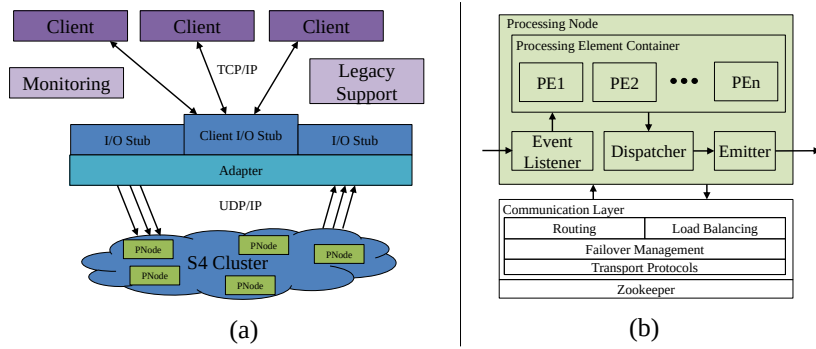


Figure 3.3: Architecture of S4

3.2.3 Esper

Esper is a software component for stream processing developed by EsperTech Inc. [241]. Architecture of Esper is shown in Figure 3.4. The software is available in Java with the name Esper, and in.NET as NEsper. However, this thesis uses only Esper, since the aim is to compare performance of S4 with a Java based stream processing software. One of the important aspects of Esper which differentiates it from System S, and S4 is that it is just a software library. Therefore, important features of stream processing systems such as distributed processing, fault tolerance, etc. has to be coded manually. However, since Esper is a software library it can be easily integrated with variety of applications such as J2EE web servers, distributed caches, web browsers, etc. Moreover, Esper has a Domain Specific Language (DSL) of relational syntax called the *Event Processing Language (EPL)* which makes Esper's programming model considerably different from System S, and S4 which have operator-based programming models.

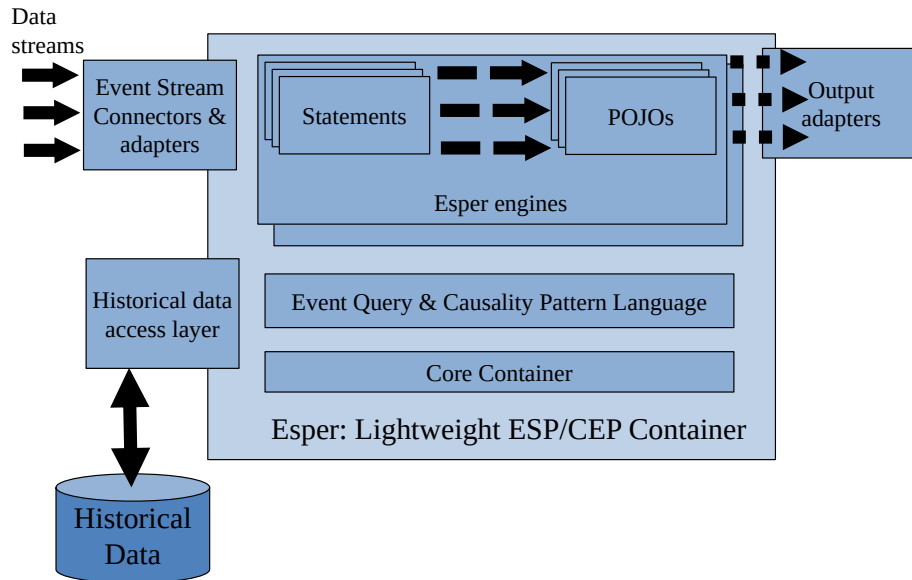


Figure 3.4: Architecture of Esper

Since Esper is just a software library, the performance evaluations described in this thesis implemented the distributed event processing functionality by using Esper with ActiveMQ [87] which is an open source message broker implementation of the Java Messaging Service (JMS) specification.

3.3 High Productivity Computing Systems Programming Languages

The DARPA's High Productivity Computing Systems (HPCS) program was initiated with the vision of developing economically viable high productivity computing systems [242][243]. The initial phase of the project started in June 2002 with one-year study contracts awarded to Cray, HP, IBM, SGI, and Sun Micro Systems. The end result of the HPCS program was defined as gaining the ability for efficiently run a broad spectrum of applications and programming models in support of the national security and industrial user communities. Two of the key areas that DARPA HPCS emphasized were: Performance and Programmability. The project set achievement of performance goals of 10X to 40X improvement of performance over 2002's capability. Also the project set programmability goal of reduced time to develop, operate, and maintain HPCS application solutions to one-tenth of 2002's cost. Another important project goal was to make available the research and operational HPCS application software that is independent of specific hardware and software architectures.

In the second phase of DARPA HPCS project, three vendors - Cray, IBM, and Sun Microsystems - were commissioned to develop new programming languages that would optimize performance on each vendor's hardware as well as optimize program development time. The languages developed by these vendors - Cray's Chapel [244][245], IBM's X10 [246], and Sun's Fortress [247] - provide a global view of data, together with a more dynamic model of processes and sophisticated synchronization mechanisms [242]. These languages have different sequential bases. X10 language uses an existing object-oriented language, Java while Chapel and Fortress use their own, new object-oriented languages [242].

Partitioned Global Address Space (PGAS) languages are a type of programming languages used to program high performance systems [248]. Examples for such languages include Unified Parallel C (UPC), Co-Array Fortran, Titanium which extends C, Fortran, and Java respectively. These languages address the programmer productivity problem in distributed-memory machines by providing a simpler, shared memory-like programming model where the address space is partitioned, and the programmer has control over the data layout [249]. The HPCS languages mentioned previously investigated use of asynchronous computation in the PGAS model which gave rise to the concept of Asynchronous PGAS (APGAS). HPCS languages follow the PGAS memory model and also provide mechanisms for asynchronous execution. These languages allow to create an activity and return immediately (An activity is a computational unit that can run in parallel with the main program).

Out of the three HPCS languages, Fortress project was wrapped up in 2012 due to implications, technical challenges involved in creating a Fortress compiler targeted for JVM [250]. Therefore, only two options were remaining as the implementation choices for optimized large scale data analysis: X10 and Chapel. In this thesis we use X10 for creating optimizations of large scale data analytics because X10 provides seamless interoperability with famous programming languages such as C/C++/Java. There are

very few work conducted with Java/C/C++ interoperability with Chapel [251].

3.3.1 X10 - An Overview

This section explains what is X10 and briefly describe the X10 language constructs which have been used to develop modularizations for optimized graph data analytics below. More information on X10 language syntax is available from X10 language specification [246] and from X10 web site <http://x10-lang.org>.

X10 is an experimental PGAS language currently being developed by IBM Research in collaboration with academic partners [252][200]. The project started in 2004, and tries to address the need for providing a programming model that can with stand architectural challenges posed by multiple cores, hardware accelerators, cluster, and supercomputers. The main role of X10 is to simplifying the programming model in such a way that it leads to increase in programming productivity for future systems [182] such as Extreme Scale computing systems [79]. X10 has been developed from the beginning with the motivation of supporting hundreds of thousands of application programmers and scientists with providing ease of writing HPC code [200]. Previous programming models use two separate levels of abstraction for shared-memory thread-level parallelism (e.g., pthreads, Java threads, OpenMP, etc.) and distributed-memory communication (e.g., JMS, RMI, MPI, etc.) which results in considerable complexity when trying to create programs that follow both the approaches [253]. X10 addresses this problem by introducing the notion of *Places*. Every activity in X10 runs in a place which is collection of non-migrating mutable data objects and the activities (similar to threads) that operate on the data [253]. Therefore the notion of Places includes both shared-memory thread level parallelism as well as distributed-memory communication which makes the life of the programmer easier. Supporting both concurrency and distribution has been the first class concerns of the programming language's design [254]. X10 is available freely under open source license another motivation behind use of X10 for data analytics optimizations described in this thesis.

X10 is a strongly typed, object-oriented language which emphasizes static type-checking and static expression of program invariants. The choice of static expression supports the motivation of improving programmer productivity and performance. X10 standard libraries are designed to support applications to extend and customize their functionality which is a supporting factor for X10 library developers.

The latest major release of X10 is X10 2.3 and it has been constructed via source-to-source compilation to either C++ or Java [254]. The C++/Java language specific tools are then used to compile the translated code to platform specific versions. In the case of C++ a platform specific C++ compiler is used to create an executable. In the case of Java, compiled class files from a Java compiler are ran on a JVM. These two methods of X10 language implementations are termed as *Native X10* and *Managed X10* [254]. The design ScaleGraph 1.0 library is motivated for high performance rather than portability advantages provided by Java, hence ScaleGraph 1.0 library has been developed targeting the Native X10. Furthermore, by choosing Native X10, ScaleGraph

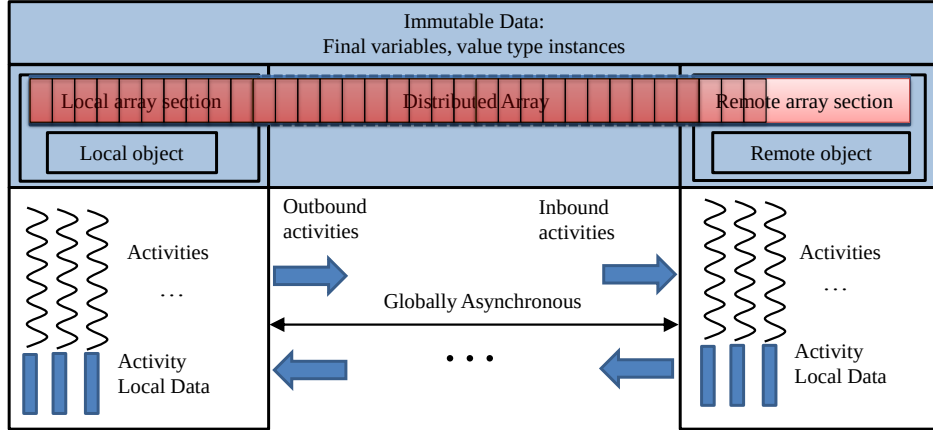


Figure 3.5: Overview of X10 language features.

1.0 is able to integrate with many scientific libraries which are typically available via C APIs [254]. However, in the case of XGDBench graph database benchmarking platform the thesis uses managed X10 because XGDBench has been implemented as an extension to YCSB which is written in Java. Use of X10 in Acacia is also motivated by the fact that Acacia inter-operates with third party Java libraries. From X10 version 2.3 onwards X10 provides very smooth interoperability with Java which indicates strong relationship of X10 with Java [255].

X10 programmers can write code that get compiled and run on GPUs [256]. The Native X10 has been extended to recognize the language constructs of CUDA code and produce corresponding kernel code. The software modularization techniques described in this thesis do not use the GPU programming features available with X10.

One of the fundamental language constructs of X10 is *Place* (See Figure 3.5). A place in X10 corresponds to a processing element with attached local storage [257]. A place can also be viewed as an address space [254]. Asynchronous activities (i.e., *async*) work as a single abstraction for supporting a wide range of concurrency constructs such as threads, message passing, direct memory access, streaming, data prefetching [246]. Activities specify logical parallelism using structured and unstructured constructs such as *ateach*, *async*, and *future*. Throughout its lifetime, an activity executes at the place where it got spawned and has access only to the data stored at that place. An activity may spawn new remote activities which get executed asynchronously at remote places using *at*. Termination detection of such spawned activities can be done using *finish*. Activities can be coordinated using *clocks* and lock free synchronization *atomic*.

The thesis uses distributed arrays (*DistArray*) in creating the graph abstractions. Every element in a distributed array is assigned to a particular place by following the array’s distribution. X10 uses an annotation system to allow the compiler to be extended to new static analyses and new transformations [246]. Annotations are created by an “@” followed by an interface type. X10 provides full interoperability with C++ and

Java through `@Native(lang, code)` annotation on classes, methods and blocks [257]. ScaleGraph 1.0 uses `@Native(lang, code)` annotation for implementing certain C++ language specific functions which are not currently supported by X10. For example, directory listing is currently not supported by X10. As a solution, ScaleGraph 1.0 developed an X10 class and linked it to a C++ code that does directory listing. X10 has a special struct type called *GlobalRef* which is a global reference to an object at one place that might be passed to a different place. The research in this thesis uses *GlobalRef* as a support for coordinating activities between different places.

3.4 Graph Databases

Graph databases address one of the key areas of large data analytics that is online analysis (i.e., online transaction processing (OLTP)) of graph data [211]. This thesis uses the term “Graph Database” as a short form to represent the term “Graph Database Management System”. Currently general-purpose graph databases have become a reality allowing mainstream users to obtain the benefits of large data analytics on connected data without having to invest in building their own graph infrastructure [258]. A unique feature of graph databases is that they elevate relationships to be the first-class citizens of the data model compared to the famous relational data model [259]. Unlike relational databases which represent relationships just as a runtime constraint, graph databases consider relationships on the same level as the data items themselves. The definition of graph database model can be stated as a database model in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors [260]. A graph database can be defined as a storage engine which supports a graph data model backed by native graph persistence, with access and query facilities for the graph primitives to make querying the stored data pleasant and performant [258].

Resource Description Framework (RDF) is a language for making statements about resources [261] focused for representing information about resources in the World Wide Web [262]. RDF allows for forming statements in the forms of triples. A triple is an element $\langle s, p, o \rangle$ interpreted as a statement where “object o stands in relationship p with subject s”. Therefore, the first element s is called the subject of the triple, p is called the predicate, and o is called the object. The RDF graph model describes this set of triples as a graph. A data store that follows such data model is usually called triple store or RDF database [263]. RDF stores (also referred as *triple stores*) are purpose-built databases for the storage and retrieval of any kind of data expressed in RDF. Generally, in graph databases an entity is represented as a vertex and its associated set of attributes. The relationships between the entities are represented as edges and their attributes. Graph databases have close similarity to RDF (Resource Description Framework) stores because RDF stores use triples which could be used to represent entities and the relationships between them. Therefore, RDF stores can also be categorized as graph databases.

Currently there are many popular graph databases in use (Details of some of them are listed in Table 3.2). Out of the graph databases listed on Table 3.2 this thesis have chosen

Allegrograph, Fuseki, Neo4j, OrientDB, and Titan graph databases for the experiments described in the thesis. The main reason for choosing these databases is that they are published as open source software. Furthermore, their license allow for performance comparisons. Furthermore they represent some of the leading graph databases in the industry. A brief description of each chosen graph database server is given below.

Table 3.2: A summary of current graph databases.

	Data Model	Supports Dis-tributed Process-ing?	Published as Open-source?	Language	Thrid-party back-end
AllegroGraph	RDF	No	No	LISP	None
BigData	RDF	Yes	Yes	Java	None
FlockDB	Network	Yes	Yes	Java	MySQL
Fuseki	RDF	No	Yes	Java	None
Neo4j	Network	No	Yes	Java	None
OrientDB	Network	No	Yes	Java	None
Titan	Network	Yes	Yes	Java	BerkelyDB/ Cassandra/HBASE
Trinity	Network	Yes	No	C#	None

3.4.1 RDF Stores : AllegroGraph and Fuseki

AllegroGraph is a commercial RDF-Triplestore introduced by Franz Inc. [264][265] for storage and querying of RDF data. AllegroGraph can be deployed as a standalone database server and offers interfaces for accessing the server data remotely via HTTP [266]. AllegroGraph has been written using LISP and there are client connectors available for Java, Python, List, Clojure, Ruby, Perl, C#, and Scala [266]. AllegroGraph's generic storage model allows any form of graphs whose elements can be represented as tuples with node elements *s* and *o*, and an edge *p*, and additional data *g* [267][268].

Fuseki is a SPARQL server that provides a REST-style SPARQL interface [269]. Fuseki is completely written in Java and is running as non-distributed. Fuseki is a component of the Jena framework.

3.4.2 Neo4j

Neo4j is an open source graph database from Neo Technology, Inc [270][258]. Neo4j emphasizes on ACID properties for graph databases and transactional support is one of the strong points of Neo4j which differentiates it from the majority of NoSQL solutions for graph data management [104]. Neo4j implements an object oriented API, a native disk-based storage manager for graphs, and a framework for graph traversals [271]. Neo4j architecture is shown in Figure 3.6. Neo4j stores graph data in a collection of store files. Each store file contains the data for a specific part of the graph (e.g., nodes, relationships, properties).

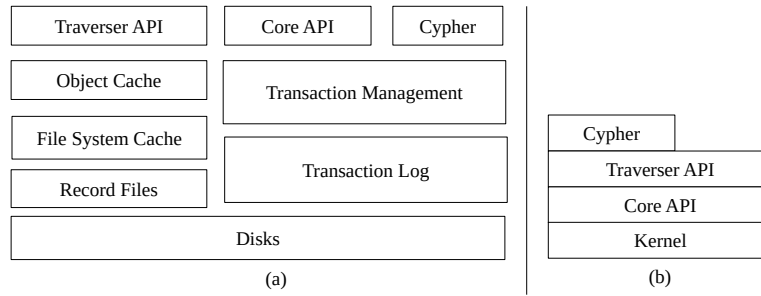


Figure 3.6: Architecture of Neo4j. (a) High level Architecture of Neo4j. (b) Logical view of the user-facing APIs in Neo4j.

Neo4j has its own high-level query language called Cypher that can be used to specify many graph query tasks. The lower level programmatic APIs which Cypher uses are also available for developers as well (See Figure 3.6 (b)).

3.4.3 OrientDB

OrientDB is an open source NoSQL database with both features of Document and Graph DBMSs [207]. OrientDB can work both in schemaless mode as well as schema-full or mix of both modes. OrientDB supports ACID transactions, fast indexes, native and SQL queries. OrientDB uses an indexing algorithm called MVRB-Tree which has been derived from the Red-Black Tree and from B+Tree which offers benefits of both the algorithms [207].

3.4.4 Titan

Titan is a distributed graph database server written in Java [272]. Titan supports ACID and eventual consistency and it can be configured to work with different storage backends: Apache Cassandra, Apache HBase [31], and Oracle BerkeleyDB [273]. All the data of Titan gets persisted in a storage backend. The choice of backend determines the transactional guarantees and scalability of a Titan graph instance. During the benchmarking experiments conducted with titan, the thesis used Apache Cassandra backend because

of need to have highly available service. Titan is natively integrated with TinkerPop graph stack including Gremlin [113] graph query language, Frames [274] object-to-graph mapper, Rexster [275] graph server, Blueprints [276] standard graph API, etc.

3.5 Summary

Technology selection plays a key role in success of implementation of optimized large scale data analysis architectures. This chapter explained the key technologies such as programming languages and software systems used to implement the data analytics optimization mechanisms. Data analytics optimizations described in this thesis are developed around data flow languages. Section 3.1 described about the data flow languages used in this thesis and explained the syntax of SPADE language in Section 3.1.1. Section 3.2 described the architectures of stream processing software used in this thesis. Next, in Section 3.3 the thesis described HPCS languages which are considered one of the important highly productive programming paradigms for implementing large scale data analytics. Out of the three HPCS languages (Chapel, Fortress, and X10) the data analytics optimizations described in this thesis have been implemented using X10. The reasons for such decision are explained in Section 3.3. Online graph data analysis represents an important segment of data analytics. Currently, graph database systems are used as the main technology for implementing OLTP on graph data. Hence Section 3.4 elaborated on what is meant by a graph database and also it listed details on Neo4j, OrientDB, Titan, Allegrograph, and Fuseki which have been used as the representative graph data stores during the experiments described in this thesis in Sub-sections 3.4.1 - 3.4.4. Chapter 4 which describes workload characterization and benchmarking of stream processing systems starts the Part I of the thesis.

Part I

Stream Processing Optimization

Workload Characterization and Benchmarking of Stream Processing Systems

Before conducting any data analytics optimizations it is essential to identify the performance characteristics of the data analysis software systems. This chapter addresses the issue of workload characterization of stream processing systems that represent a key category of data analysis software systems currently being used. Chapter provides an introduction to stream application workload characterization and benchmarking in Section 4.1. Methodology and performance metrics used for the evaluations are described in Section 4.2. Next, a description of the benchmarking applications used during the workload characterization is given in subsection 4.2.1. Details on performance evaluation is provided in Section 4.3. The chapter provides a discussion on the performance results in Section 4.4. Conclusions arrived by conducting the workload characterization of stream processing systems is given in Section 4.5.

4.1 Introduction

There have been extensive studies for characterizing the workload, and performance implications of computing systems. However, there has not been sufficient amount of such studies carried out in the area of stream computing. This chapter describes stream processing software performance characterization using System S, S4, and Esper; which are currently three prominent stream processing software in the field [277][278]. As explained in Section 3.2 of this thesis, decision to choose these three software was stimulated due to their unique architectural designs. While System S is developed following a manager and worker model, S4 has a decentralized and symmetric architecture and follows Actors model [19]. Esper is completely different from System S and S4 because

it is just a component for stream processing [241]. However, Esper provides a complete software suite for stream processing which has been used by popular software vendors for their event processing back-ends. Furthermore, current implementation of S4's operators are purely based on Java and also Esper is a pure Java library, whereas System S allows for both C/C++ and Java versions, which leads to interesting results. The licenses of these software were checked beforehand and was confirmed that they allow for publishing performance comparisons.

In achieving the aforementioned objectives 70 different experiment scenarios were created using three real-world application benchmarks and a micro benchmark. Performance characteristics such as Job throughput, CPU usage, Memory Usage, and Network I/O were observed in arriving at conclusions about the design of optimized stream processing system architectures.

4.2 Methodology and Performance Metrics

The workload characterization methodology described in this chapter is based on two level benchmarking. The chapter uses a micro-benchmark to get basic characterization of the three software's performance. Next, three different stream programs (Application-specific benchmarks) are used for obtaining further insights to the characteristics of the stream processing applications and software systems. These programs and the features of their associated data sets are described below.

4.2.1 Sample Programs and Data Sets

4.2.1.1 Micro-benchmark

We use a three operator micro-benchmark program to get a basic understanding of the behavior of the stream processing software. This program's structure is shown in Figure 4.1 (d). It has only one functor (F1) that increments an integer value it receives from the Source operator (S). The result from F1 gets stored in `result.dat` via Sink operator (SI). The choice of this program was made because of its simple nature which allows for revealing the behavior of the stream processing systems during heavy work loads.

A synthetic data set of two digit integer values (in CSV format) was used during the experiment. The data set contained 10 million records.

4.2.1.2 VWAP

Calculation of Volume-Weighted Average Price (VWAP) is a real world application scenario of stream processing in the financial services domain. VWAP is calculated as the ratio of the value traded and the volume traded within a specified time horizon. This can be depicted as,

$$VWAP = \frac{\sum_{i=1}^n (P_i \cdot V_i)}{\sum_{i=1}^n (V_i)}$$

where V_i represents each traded volume and P_i represents the corresponding traded price in a series of n transactions. Figure 4.1 (a) shows the data flow graph of the VWAP application used for performance evaluation. This application is part of a larger financial trading application described at [13]. The first functor (F1) filters the tuples for valid records and the aggregate operator (AG) keeps a tuple window of 4 based on the ticker id it receives from F1. Furthermore, it calculates the $P_i.V_i$ portion of the VWAP formula for each tuple. AG outputs a new tuple each time it receives a tuple from F1. Finally, F2 calculates the VWAP value and transfers to Sink operator (SI) which stores the result in `result.dat` file.

The dataset available with System S's sample VWAP application was used as the sample data set during the experiments, but the data set was magnified to a larger data set of 1 million tuples.

4.2.1.3 CDR

A Call Detail Record (CDR) is a piece of information produced by a telephone exchange containing details of a phone call passed through it. Telecommunication networks process massive amounts of CDR events, another application area for stream processing software systems. Furthermore, it allows for testing how well different stream processing software scale under massive data rates if programmers do not worry about producing optimized version of their codes. These factors motivated the use the four operator CDR processing application (shown in Figure 4.1 (b)) in this study to evaluate the three stream processing software. First functor (F1) splits the input tuples read by Source operator to different routes based on a hash value of call station ID (which identifies each user). The Aggregate operator (AG) calculates the total packet count by adding input, and output packet values mentioned in the tuple. Finally the result is stored in the disk via Sink (SI) operators.

A synthesized data set with 2 million data tuples was used during the experiments with each tuple having 22 fields for this experiment. The data set had information of 0.1 million users.

4.2.1.4 Twitter Topic Counter

A Twitter hash tag count application was used as the third application specific benchmark. Twitter users can label the key words of their tweets using # (e.g., #car) which indicate to which conversations the messages relate to. As shown in Figure 4.1 (c) a six operator SPADE program was developed with similar functionality to the Twitter Topic Counter application (here onwards referred to as Twitter application) available with S4 and used both these applications during our performance evaluations.

A real data set with 90507 tweets gathered from twitter on 4th June 2011 in the period 00:00-01:00 JST was used during this experiment.

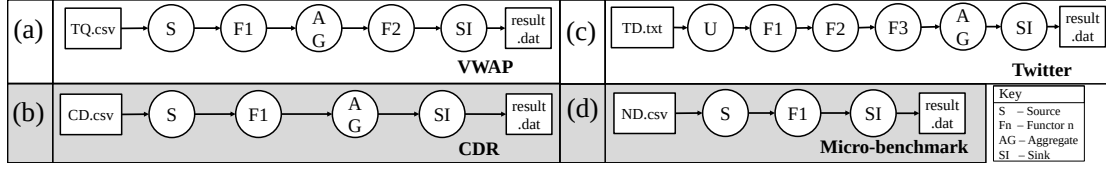


Figure 4.1: Data flow graphs of sample applications.

4.2.2 Experimental Setup

The experiments were conducted on 12 compute nodes each with AMD Opteron™ Processor 242, 1600MHz 1MB L2 cache per core, 250GB hard drive. Seven out of twelve nodes had 8GB RAM (Nodes labeled as **sa0<n>**, n is from 1 to 7) while the remaining five (Nodes labeled as **sb0<m>**, m is from 1 to 5) had 4GB RAM. From the profiling results that were obtained using Oprofile [279], and Nmon [280] it was observed that the difference of main memory in the nodes did not affect the experiments. All the nodes were installed with Linux Cent OS release 5.4, IBM Infosphere Streams Version 1.2, and JRE 1.6.0, and were connected via 1Gigabit Ethernet. S4 version 0.3.0.0, Esper version 4.5.0. were used during the experiments by setting both initial Java heap sizes and maximum Java heap sizes to 3GB to avoid CPU time being spent on increasing Java heap memory during the experiments.

4.2.3 Performance Metrics

Job throughput, CPU usage, network I/O, and memory usage of each sample application (i.e., job) were used as the metrics for the evaluations. While throughput is an important metric for data intensive applications; CPU usage, network usage, and memory usage information gathered via two profiling tools (Oprofile and Nmon) were used to further understand performance characteristics of the System S and S4 applications.

- **Job Throughput :** Measures number of input tuples processed by each stream processing system. On System S we measured this in Tuples per second while on S4, and Esper it was measured in Events per second.
- **CPU usage (%) :** Overall CPU utilization during the experiments. This metric provides information on which nodes are busy processing data. Hence it allows for identifying stream processing system's node level bottlenecks.
- **CPU usage by process (%) :** Measures what processes contribute to overall CPU utilization and the amount of their contribution. This allows for identifying where the performance bottlenecks exist (i.e., in stream processing system level or OS level) and which processes are responsible for such behavior.
- **Memory usage (MB) :** The amount of main memory (RAM) use during application execution. It is essential to maintain high percentage of free main memory

for proper functioning of stream applications. Because unlike most batch processing tasks, stream processing keeps all the required data in main memory during the program execution and does not persist incoming stream data on secondary storage. Hence, it is one of the important metrics for this study.

- Network I/O (KB/s) : Network I/O of each node of stream processing system cluster was measured during the experiments to quantitatively understand what kind of communication overhead exists between the nodes.

4.2.4 Objectives and Methodology

Measurement of throughput was done in two different ways for S4, Esper, and System S. In S4 and Esper time required to process a specific amount of events was measured while in System S time between processing first tuple and a last tuple was measured.

By measuring throughput the intention was to understand how well the applications scale in both stream processing systems. Scalability of current S4 applications are characterized by S4 runtime. The runtime determines what number of PEs to be created to suit for a particular node allocation under different workloads. User of S4 cannot explicitly allocate PEs for different nodes. Hence we used automatic node allocation for S4. However, SPADE allows for allocating a node pool (a collection of hosts in a stream processing cluster) and attaching different operators to specific nodes programmatically. This feature was used to distribute workload of System S jobs among nodes. The Esper applications had to be manually allocated for different nodes which provided us finer grain control over their distributed execution. Furthermore, the sample programs shown in Figure 4.1 were augmented in such a way that the modified versions allow for attaching more nodes with the job. The objective was to use more nodes than the operators (PEs) available in the data flow graph, and allocate multiple operators to each node. E.g., If we used 12 nodes we need a data flow graph with at least 12 operators which is not possible with any sample program layout shown in Figure 4.1. How the augmentation of data flow graphs of sample applications developed for System S has been carried out is shown in Figure 4.2.

Apart from the four augmented sample applications, an optimized version of the CDR application (shown in Figure 4.2 (c)) was used in order to observe how the introduction of multiple sources affects the throughput of a stream program. However, this program was used only for this purpose, but we used normal augmented version of CDR application (in Figure 4.2 (b)) in order to make the comparison a fair one. In order to supply four source files to CDR Optimized data flow graph, the data file used for CDR was splitted equally into four files using Linux's `split` command and resulting files were checked to make sure that the first and last packets were in the correct format.

As shown in Figure 4.2, a split operator (SP) was introduced for each sample program that splitted the data stream from source operator to several sub graphs. The splitting was based on a hash value produced for each incoming packet. It was made sure the packets are evenly splitted among sub-graphs by choosing an appropriate field to hash and by generating the input data file with proper distribution of tuples. Furthermore,

4.2 Methodology and Performance Metrics

two Functor operators were introduced (FS (at the beginning) and FE (at the end)) which recorded the time of receiving start packet and a desired last packet.

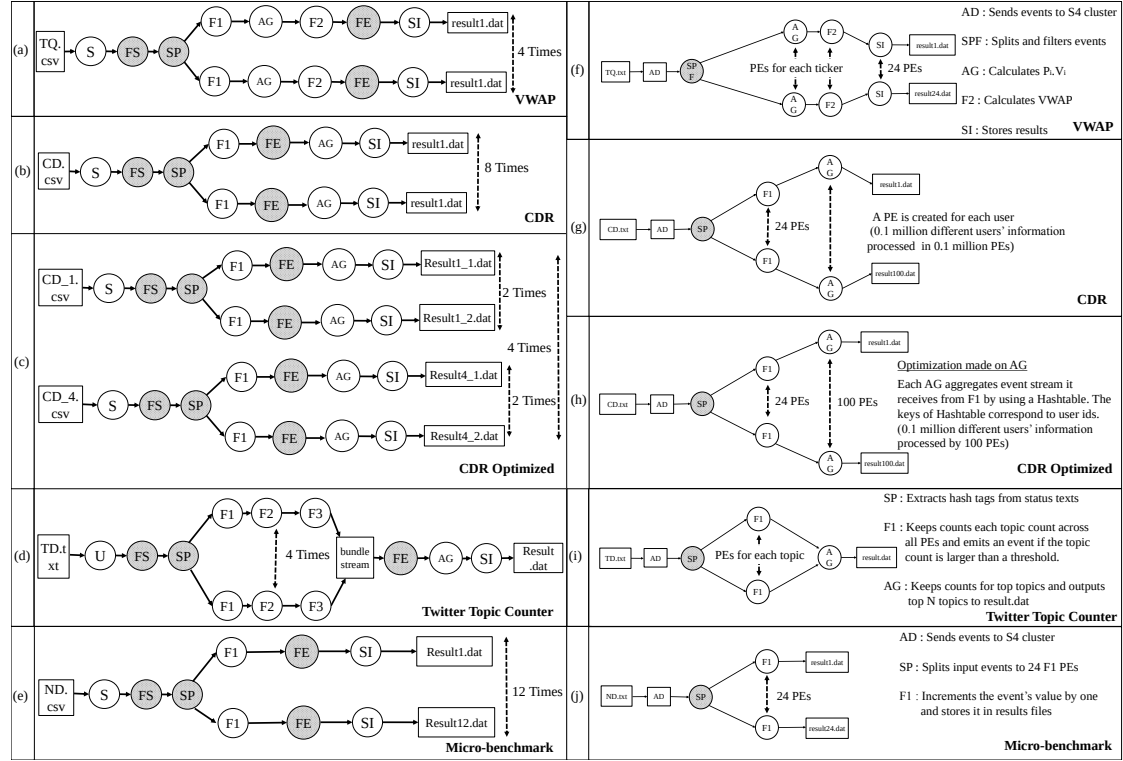


Figure 4.2: Augmentation of sample application data flow graphs on System S and S4. Sub-figures (a) to (e) correspond to System S applications while (f) to (j) correspond to S4.

In the case of VWAP a data set of 1 million data tuples was used and the last packet was attached with trade ticker “GMD”. The time it took to receive this packet at one of the four PEs was taken as the total runtime and throughput was calculated. For CDR, the input data file had 2 million data tuples and the time it takes to receive the last packet was measured (i.e., 2 millionth packet) by one of the FEs. The same method was employed to measure the runtime of CDR optimized version.

The data flow graph of Twitter application is significantly different from other augmented versions in order to keep the similarity of program design with S4 version of Twitter application. It should be noted that in CDR, CDR Optimized, and Twitter application the measurement of last packet was done before Aggregate operators that outputs only a subset of tuples they receive. The measurements of the throughput was conducted on the corresponding Aggregate operators on S4 versions, hence this ensured that the measurements were taken on the same operator on both System S and S4.

How the sample applications have been implemented on S4 is shown in Figure 4.2 (f) to (j). For data flow graphs shown in Figure 4.2 (g), (h), (i), and (j) the storing of results was done without use of a Sink (SI) PE. However, a separate SI PE was defined

4.2 Methodology and Performance Metrics

for VWAP since data output involved an additional step of conversion to CSV format. Furthermore, as shown in Figure 4.2 (g), (h) two versions of CDR application was developed for S4 since the original version shown in Figure 4.2 (g) resulted in inefficient use of system resources by spawning 0.1 million PEs. The solution was to reduce the number of PEs to 100 by using a hash table on each PE with user id as the key. All the PEs were distributed equally to all nodes during the experiments that ran on S4.

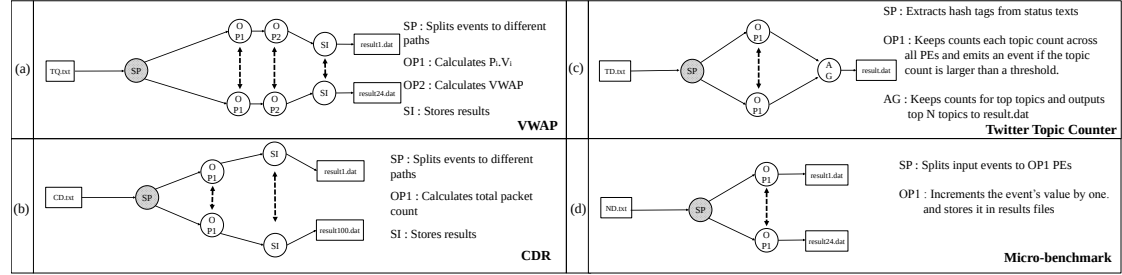


Figure 4.3: Augmentation of sample application data flow graphs for Esper.

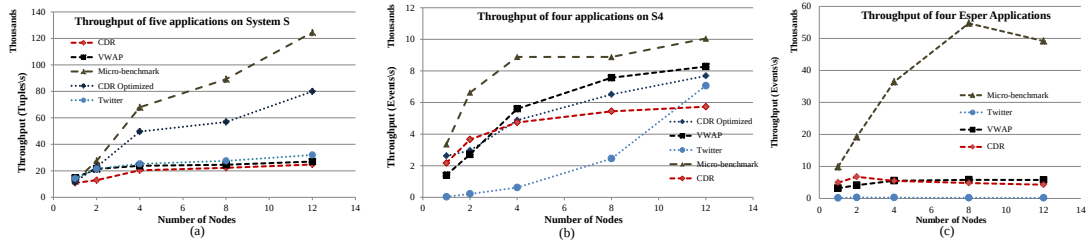


Figure 4.4: Throughput comparison of sample applications. (a) System S Applications (b) S4 Applications (c) Esper Applications.

Augmented versions of the Esper applications are shown in Figure 4.3. Each operator denoted by OP (e.g., OP1) is attached to a JMS server running on the same node.

For each augmented sample program we allocated different number of operators per node as the number of worker nodes involved in the processing vary. Each version of the sample program was run three times for System S, Esper, and S4. Average value of the running times was taken for calculating the throughput to improve the accuracy of end results. Furthermore, both Nmon and Oprofile were ran on each node associated with the experiment well before one of the experiments during three experiment runs begin. The two daemons were shutdown after the experiment completed keeping enough delay from the experiment end time.

4.3 Performance Evaluation

4.3.1 Job Throughput

The throughput results obtained from running the four sample applications and the micro-benchmark based on System S, S4, and Esper are shown in Figure 4.4 (a), (b), and (c) respectively. Note that a tuple in System S corresponds to an event in S4 and Esper. Since most research literature on System S uses the term tuple to represent a data event the thesis uses the same terminology. The performance curves on Figure 4.4 (b) show that S4 achieves sub-linear scalability for CDR, VWAP, and micro-benchmark because doubling the number of nodes did not result in a corresponding doubling of the system's performance. Two different curves for CDR on S4 were drawn due to the optimization made on CDR application. The curve marked with CDR corresponds to the data flow graph shown in Figure 4.2 (g) which is a naive unoptimized version while the curve CDR Optimized is obtained by running the optimized program shown in Figure 4.2 (h). The latter was created to avoid inefficient resource usage as described in Section 4.2.4.

Out of the System S versions of these applications, micro-benchmark and CDR Optimized version produced almost linear throughput curves. CDR, VWAP, and Twitter applications indicated sub-linear scalability (saturated) because doubling the number of nodes did not result in a corresponding doubling of the system capacity. However, considering the throughput characteristics obtained from optimized version of CDR, it is apparent that having a single source operator in the data flow graph results in such sub-linear scalability.

Esper reported more than 1.5 times higher performance for single node applications compared to S4. This indicates that Esper is much suited for single node event processing scenarios. However, the distributed versions of the Esper applications did not scale well, compared to S4 or System S. It can be conjectured that the reason for such less scalability was because of the overhead associated with object serialization.

Nmon and Oprofiler tools were used in order to get more details of System S and S4 middleware. The results are discussed below.

4.3.2 CPU Usage

CPU utilization of manager nodes and worker nodes of both S4 and System S are shown in Figure 4.5. The node on which the S4 adapter component run during the experiment is referred as Manager node and the remaining nodes as Worker nodes in the case of an S4 cluster. In the case of System S the node on which Streams Application Manager (SAM) [232] resides is referred as Manager node and the other nodes are taken as Worker nodes.

To increase the legibility of results and to distinguish between application-level and micro-level benchmarks the results of micro-benchmark experiments' CPU utilization are listed in Figure 4.6 separately.

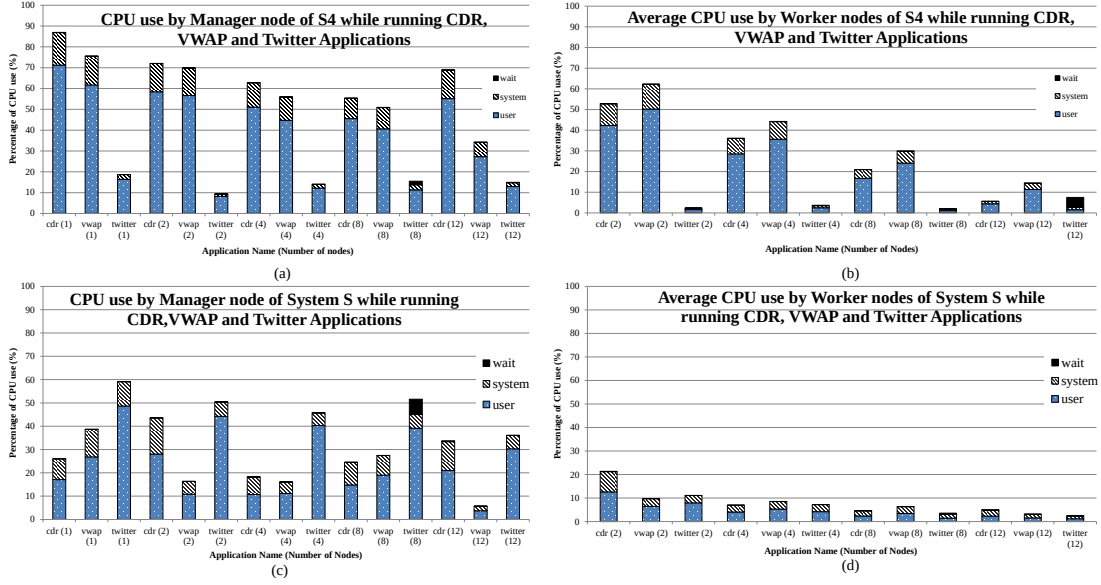


Figure 4.5: CPU usage on different nodes of System S and S4 while running CDR, VWAP, and Twitter applications.

It was clear that System S's workers used (in Figure 4.5 (c), (d)) less CPU compared to S4 with VWAP and CDR applications. However, in the case of Twitter which operated relatively low input data rate, S4 workers and manager node reported relatively less CPU usage compared to their System S counter parts. Furthermore, in System S, a considerable amount of processing has been performed by the system processes rather than by user processes as can be observed from Figure 4.5 (c) and (d).

The system profiling results of Oprofile were used to construct a clear picture of what happens in system and user processes in the case of the CDR and micro-benchmark experiments with 12 nodes for both System S and S4. The results are shown in Figure 4.7. Each graph of Figure 4.7 shows the share of CPU_CLK_UNHALT events used by different processes. It can be observed that in both experiments involved with S4 (Fig 4.7 (a), (b)), in most of the nodes the Java virtual machine (java) had used slightly less than 30% of CPU_CLK_UNHALT events which is little less than the amount consumed by the operating system's kernel. However, in the case of System S when running both CDR and micro-benchmark, the System S components had consumed roughly 20% to 30% CPU_CLK_UNHALT events. However, unlike the case with S4, nodes' kernel consumed 50% to 60% CPU_CLK_UNHALT events. This gives an explanation for why a considerable amount of processing conducted by system processes in Figure 4.5.

4.3.3 Memory Usage

Memory consumption of different nodes while running the four augmented applications on S4 and System S are shown in Figure 4.8. It can be observed that worker nodes

4.3 Performance Evaluation

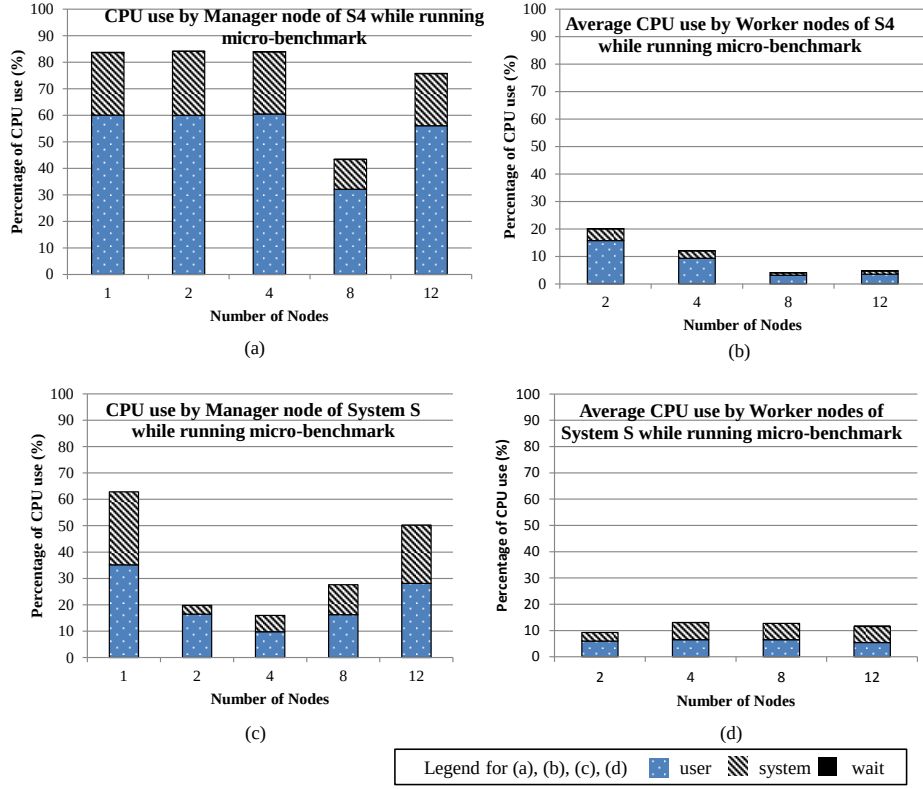


Figure 4.6: CPU utilization by nodes when running micro-benchmark on System S and S4.

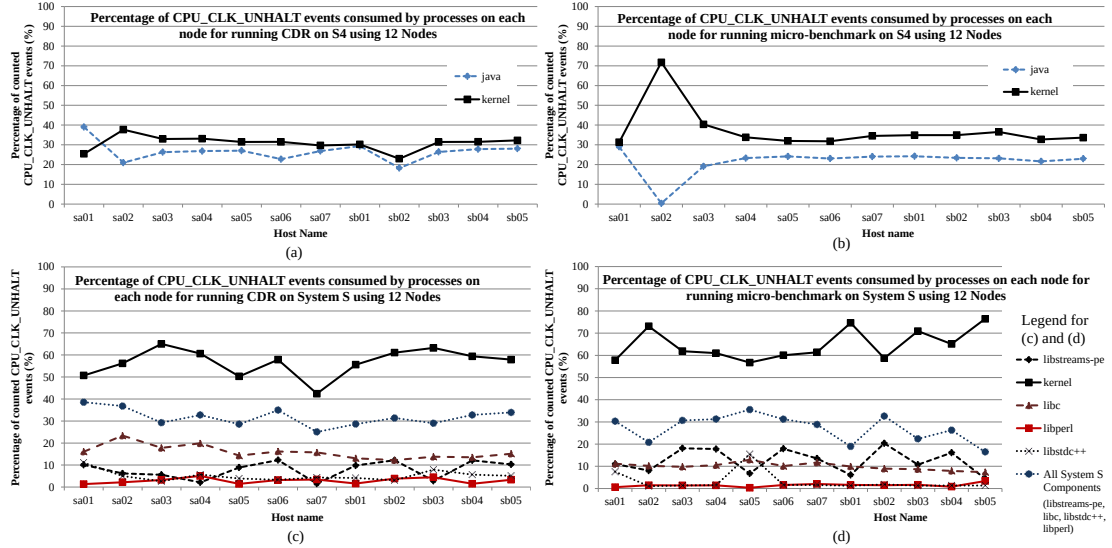


Figure 4.7: Breakdown of CPU usage by processes for CDR and Micro-benchmark.

of System S consumed less memory compared to S4 workers. Twitter application introduced relatively less data transfer rate among the nodes since it only injected 90507 events. This is a reason for why S4 nodes indicate less memory use with Twitter application compared to other applications. Furthermore, representation of data tuples in the form of Java events in S4 requires more memory compared to System S, another reason for why such curve appear. Huge amount of memory consumption can be observed on manager node (sa01). It should be noted that the node named sa01 was used as the master node in all the experiments carried out in this chapter.

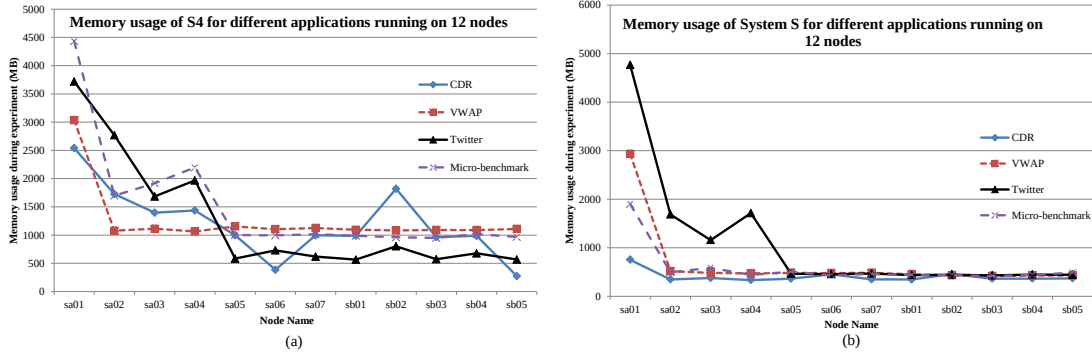


Figure 4.8: Memory consumption of sample applications on System S and S4.

4.3.4 Network I/O

The network I/O happening between nodes of a stream processing system is a key criterion that determines its performance. Figure 4.9 shows the network I/O of different nodes when running CDR application on both S4 and System S. When considering the `eth0-write` at master nodes and `eth0-read` at worker nodes, it is apparent that the amount of data transferred between the master and the worker nodes is considerably larger in S4 compared to System S. For the data shown in Figure 4.9, the average `eth0-write` rate of Manager node of S4 is 60% larger than the average `eth0-write` rate of System S's Manager node. Moreover, average rate of `eth0-read` at worker nodes of S4 is 120% larger than the average `eth0-read` at worker nodes of System S. However, both System S and S4 applications used the same 2 million data set. This indicates that S4's data transfer protocol consumes more memory bandwidth compared to System S. The network I/O for eight nodes while running CDR for both S4 and System S (shown in Figure 4.9 (c) and (f)) confirms the aforementioned fact.

4.4 Discussion

The results obtained by running fourteen different applications (totaling 70 applications considering 1,2,4,8,12 node scenarios including System S and S4 source code optimization) on System S, S4, and Esper gave deep insight to the systems' performance. It

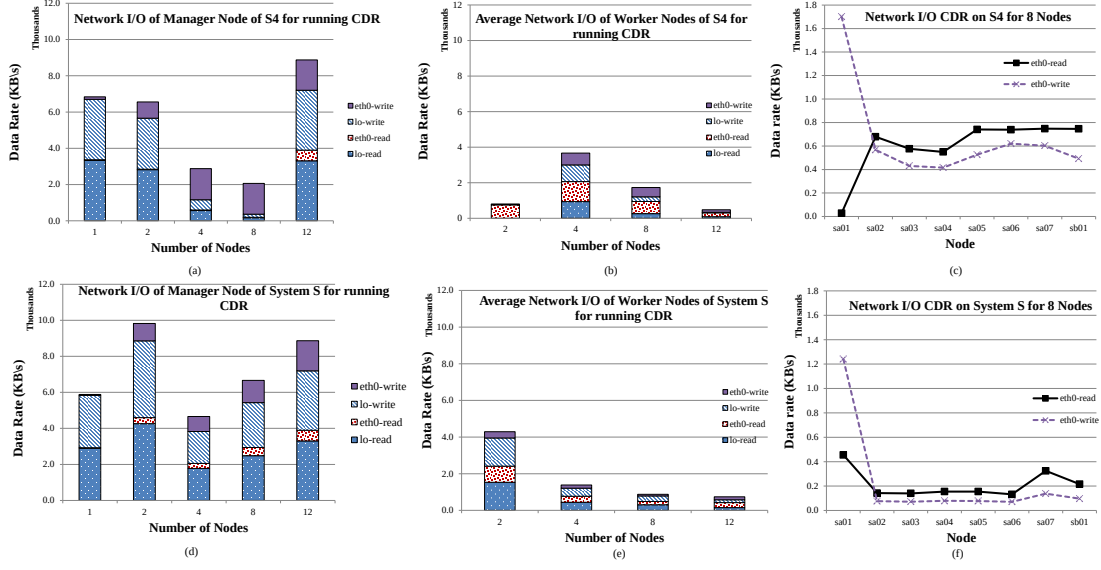


Figure 4.9: Network I/O for Manager node and Worker nodes while running CDR on S4 and System S.

became clear from the throughput comparison made on Figure 4.4 that the stream programming models that allow/require programmers to write optimized codes should be used carefully to maximize the throughput. E.g., A SPADE program written with single source operator might not scale well in different hardware configurations. Also a S4 application that generates huge numbers of PEs for incoming events cannot scale well with limited `PEContainer` queue size. Yet introduction of multiple source operators resulted in a 3.2 times speedup for CDR application on System S and a 1.34 times speed up for reducing 0.1 million Aggregator PEs of S4 to 100 PEs (See Figure 4.4) which indicated possible avenues for performance improvements in different stream programming models.

While Java based stream processing system architectures are gaining considerable attention due to their portability, system designers and programmers have to think carefully before choosing the desired solution. E.g., A light weight input event rate job could be easily processed using S4 with few amount of PEs (e.g., Twitter application run on S4). However, a large scale application with high commercial importance such as VWAP and CDR might produce millions of PEs since S4 dynamically generates PEs for each new data events it receives.

As with any other JVM related optimizations, setting of maximum and minimum heap size values plays a key role in determining the performance of Java based stream processing systems. However, in the case of S4 and Esper based stream processing applications, system administrators need to be vigilant about the characteristics of the data handled by their S4/Esper applications properly.

Furthermore, when using stream processing systems like S4; programmers, requirements engineers, software designers need to have a clear understanding of the highest

data rates that will be faced by their applications and they need to design the applications to avoid event losses [281]. They need to fine tune S4 codes for such scenarios. Also they need to develop their application codes in such a way that their applications distribute processing overhead evenly among the nodes. Avoiding the bottlenecks in the data flow graphs and the use of increased number of nodes tend to alleviate the problems faced by S4.

Esper applications's throughput results indicate that scalability is one of the key challenges faced by JMS based distributed Esper applications. While there was a slight scalability advantage scaling from one node to two nodes for all the benchmarks, the performance tend to degrade when the application is scaled to more nodes. The reason for such behavior can be traced to the slowness in network communication and serialization. However, the Esper micro benchmark application had considerably higher performance compared to its S4 counterpart.

While it was observed a heavy usage of network bandwidth by S4, by using optimized protocols and techniques such as Java New I/O, InfiniBand Remote Direct Memory Access [282] the conditions could be improved.

4.5 Conclusions

This chapter presented a performance study on three stream processing software: System S, S4, and Esper. Four application benchmarks were ran covering different domains of stream processing. A micro-benchmark was used for further clarification of the performance of these software systems. The study used job throughput, CPU usage, memory usage, network usage of each node as the performance metrics. By analyzing the throughput, and profiling results it was observed that carefully designed stream applications results in high throughput. Another conclusion arrived was that choice of a stream processing system need to be made considering factors such as performance, platform independence, size of the jobs. Furthermore, the operating system kernel was observed as a key factor that determines the performance of stream processing systems.

4.6 Summary

This chapter conducted a workload analysis of stream processing software which represent one of the key categories of large scale data analytics. Section 4.1 provided an introduction to performance characterization of stream processing systems. Methodology and performance metrics used in this study were listed in Section 4.2. Performance evaluation experiment details were provided in Section 4.3. A discussion of the results and the conclusions arrived are listed in Sections 4.4 and 4.5 respectively. Next chapter describes StreamFarm which is a scalable stream workload generator platform with a flexible control interface.

StreamFarm a Workload Generator Platform for Stream Computing Environments

Stream computing (also known as Complex Event Processing [283][10]) has created a paradigm shift in data analytics by introducing an effective means of dealing with vast amounts of high volume, continuous data produced by various sensors. This chapter describes design and implementation details of StreamFarm which is a scalable workload generator platform for stream processing system performance analysis and benchmarking [284]. Section 5.1 provides an introduction to stream workload generation and StreamFarm. Section 5.2 describes the methodology followed by StreamFarm for implementing a scalable workload generation. Implementation details of StreamFarm is given in Section 5.3. StreamFarm’s query language is described in Section 5.4. Conclusions from the experiment results of StreamFarm are described in Section 5.5.

5.1 Introduction

Although stream computing is currently a burgeoning technology with lots of future applications, there has been little work done in stream related performance tuning, measurements, and benchmarking [16]. While there are stream processing benchmark suites such as FINCoS Framework [15] exist, these benchmark suites do not provide flexible architecture that can be easily integrated with stream workload generation scenarios. For example most of the current stream workload generator suites do not allow for flexible integration of the workload generator platform with third party applications/frameworks such as the performance optimization mechanisms described in this thesis: Hirundo, Tahitica, Albatross, etc. The workload generator suite should be easily attachable with such performance optimization mechanisms. StreamFarm addresses this

issue by allowing it to be automatically configured to use with optimization mechanisms such as Albatross by using the StreamFarm’s query language. It should be noted that the experiments described on Albatross has been conducted completely using StreamFarm platform [284].

5.2 Methodology

The workload generation need to be conducted in a distributed and scalable manner in order to emulate real world stream data sources. Therefore, architecture of StreamFarm (shown in Figure 5.1) has been designed in a component oriented manner that can be deployed distributedly. *FarmManager* is the main component of StreamFarm platform. It instantiates *FarmWorkers* in the workload generation cluster which essentially carries out the workload generation tasks. FarmWorkers create Springs and Sinkholes during a workload generation session.

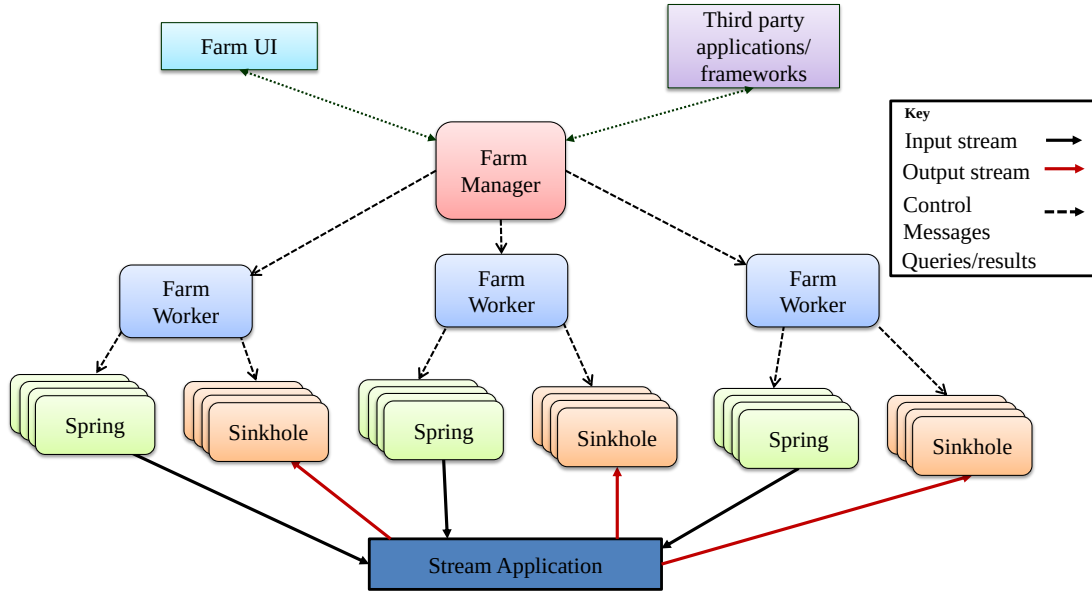


Figure 5.1: Architecture of StreamFarm

5.3 Design and Implementation of StreamFarm

StreamFarm has been designed from the ground up following object oriented system design concepts. The main classes of StreamFarm are shown in Figure 5.2.

The main entities of StreamFarm that are **FarmManager** and **FarmWorker** implement **Farmer** interface. As mentioned earlier, stream data generation and consumption has been conducted on two types of objects whose classes implement Spring and Sinkhole

5.3 Design and Implementation of StreamFarm

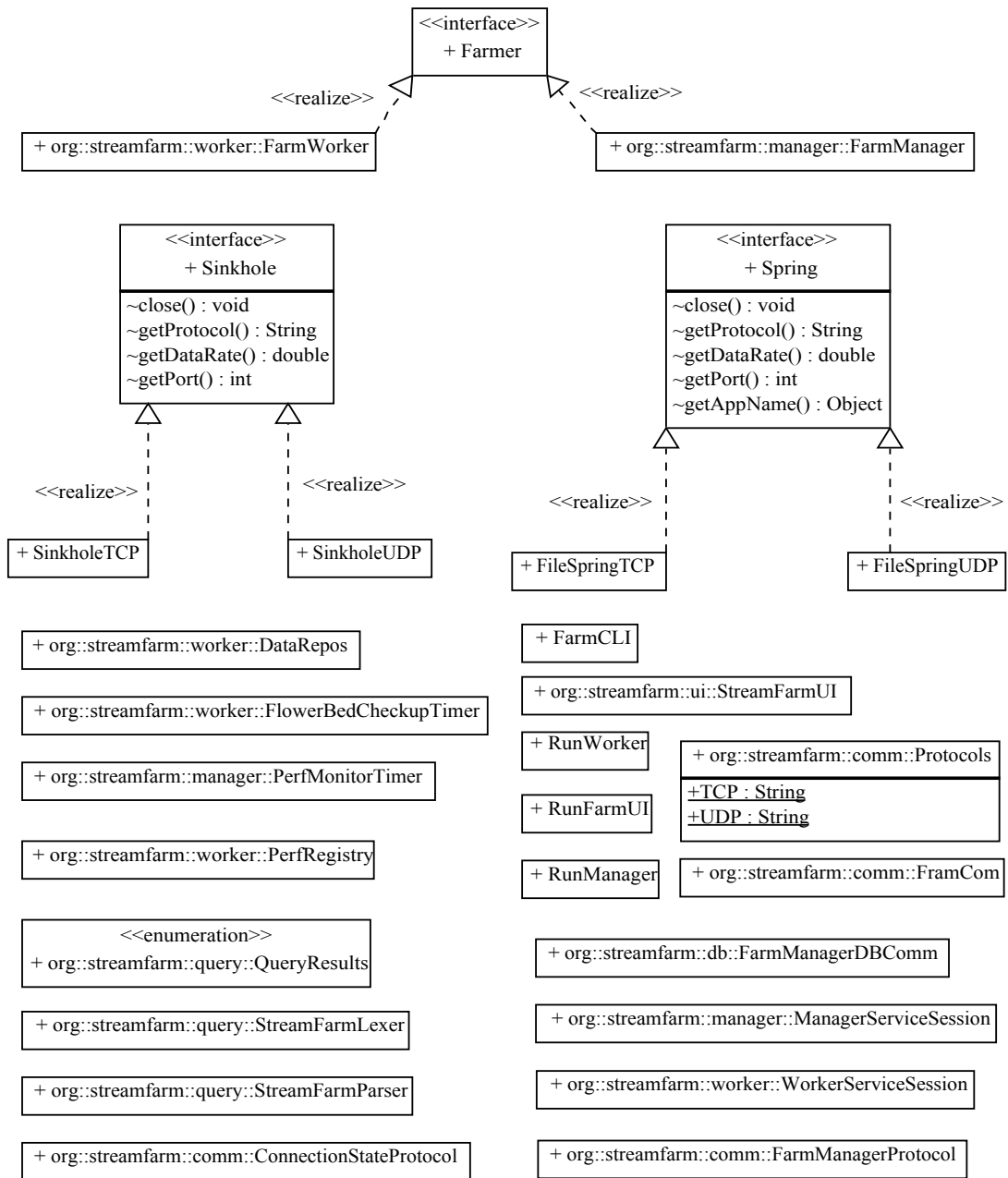


Figure 5.2: Main classes of StreamFarm

interfaces. Currently there are two main types of Springs and Sinkholes: TCP types and UDP types. The class names represent the type of protocol that these objects use to communicate with the stream application. E.g., `SinkholeTCP` class creates Sinkhole objects that communicate through TCP/IP protocol while `SinkholeUDP` class creates objects that communicate through UDP protocol.

StreamFarm utilizes an HSQLDB [285] in-memory database to store meta information during its execution. Coordination between the HSQLDB instance and the rest of the components of StreamFarm has been conducted using `FarmManagerDBComm` class. `RunWorker`, `RunFarmUI`, and `RunManager` are three classes with Java `main()` methods that act as the starting points of the platform. Furthermore, classes for stream performance monitoring and measurement, StreamFarm query language are also shown in Figure 5.2.

(a)	CREATE 4 TCP P Springs FROM "/nfs/home/miyuru/Desktop/VWAP/data/TradesAndQuotes.csv" ON localhost AS "VWAP"
(b)	CREATE 8 TCP P Springs FROM "/nfs/home/miyuru/Desktop/VWAP/data/TradesAndQuotes.csv" ON st01,st02,st03,st04 AS "VWAP"
(c)	CREATE 1 TCP P Springs FROM "/nfs/home/miyuru/Desktop/VWAP/data/TradesAndQuotes.csv" ON st01,st02,st03,st04 AS "VWAP" WITH MAXDRATE 100
(d)	CREATE 1 UDP P Springs FROM "/nfs/home/miyuru/Desktop/wc/data/words.dat" ON localhost AS "VWAP"

Figure 5.3: An example set of StreamFarm queries that can be used to create Springs

StreamFarm has been designed with a collection of **Event** classes that model different events which happen throughout the life cycle of an instantiation of StreamFarm. These classes and their **Listener** classes are shown in Figure 5.4. It should be noted that these events do not communicate between different components running in different JVMs but rather used for communication and coordination happening between objects in the same JVM process. E.g., When a Worker is informed by the Manager that the StreamFarm is shutting down, the Worker emits a **FarmShutdown** event which shuts down all the Springs and Sinkholes that were instantiated by the Worker.

Current version of StreamFarm has been implemented completely in Java. The StreamFarm query language (described in the next Section) has been implemented using Antlr parser generator [286]. As can be observed from Figure 5.2; `StreamFarmParser` and `StreamFarmLexer` utilize this grammar for parsing StreamFarm queries. StreamFarm includes a Java CLI interface as well as a Java SWING user interface (a screen shot is shown in Figure 5.5) for management purposes.

5.4 StreamFarm Query Language

One of the unique features of StreamFarm is that it uses a custom query language for life cycle management purposes. The language includes constructs for querying a

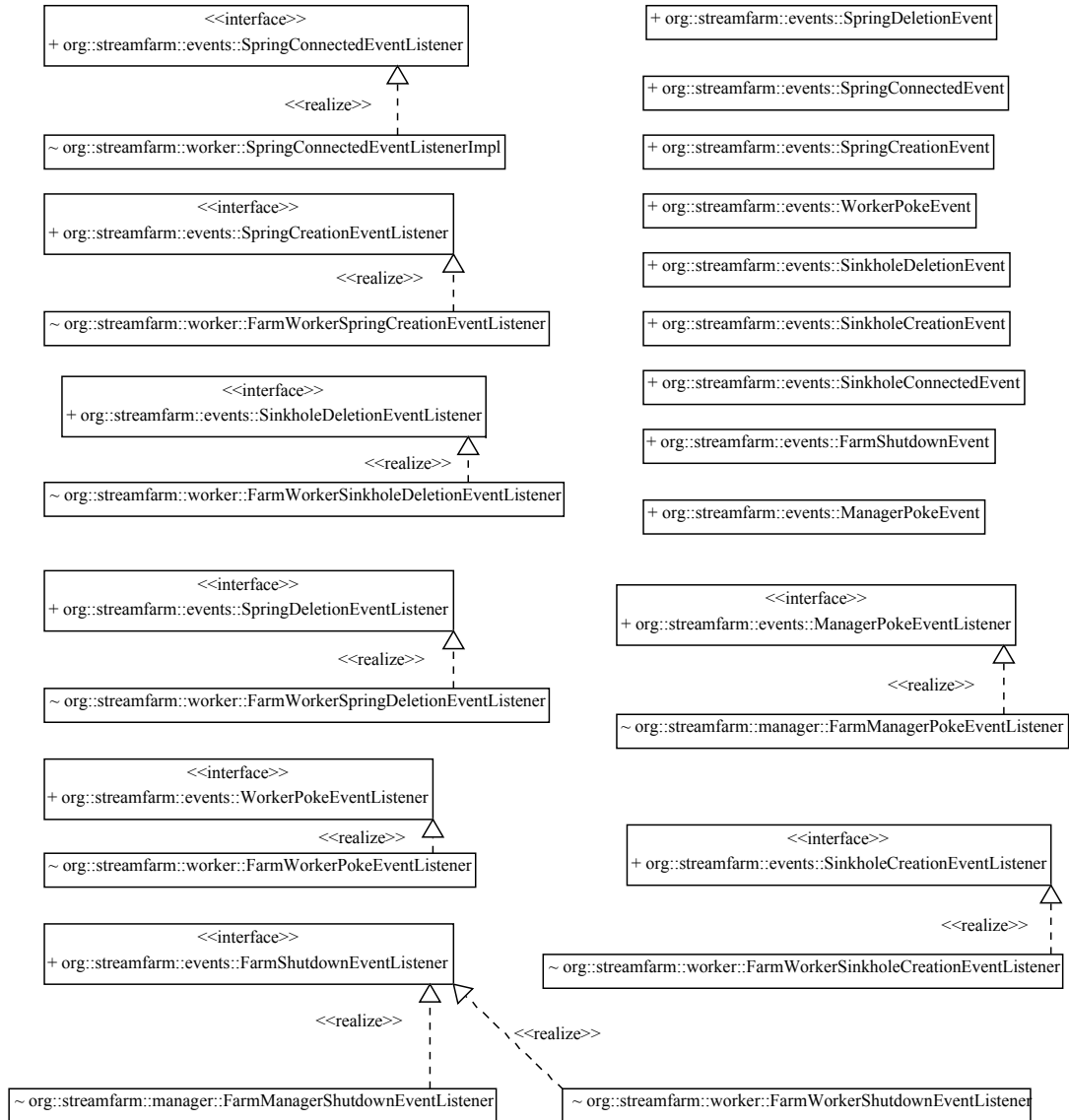


Figure 5.4: Event classes of StreamFarm

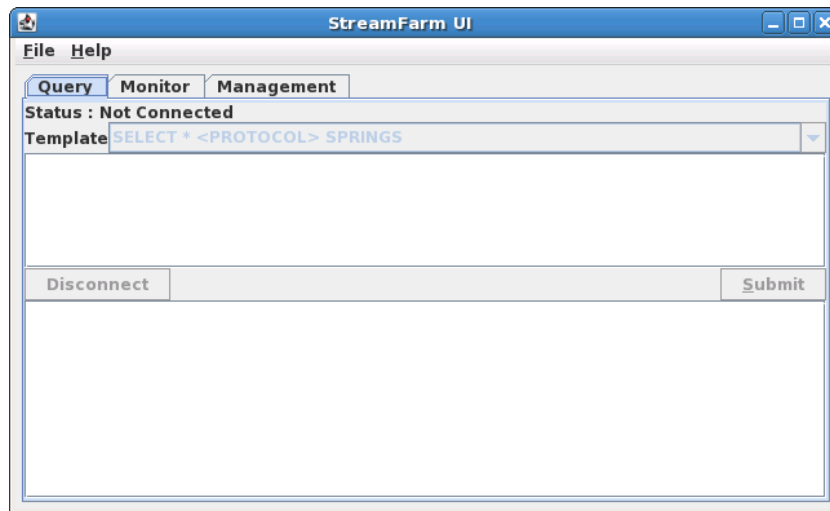


Figure 5.5: A GUI interface for StreamFarm

StreamFarm instance’s operational statistics such as data rate information from different Springs/Sinkholes, number of Springs/Sinkholes running on the platform and their host, port information, etc. Furthermore, the language provides support for creating new Springs/Sinkholes, deleting existing Springs/Sinkholes, etc. The statements shown in Figure 5.3 are an example for StreamFarm queries used to create Springs.

(a)	CREATE FILTER “ $y=x^2$ ” AS “Q1” WITH DRANGE 0,1000 IN PERIOD 10
(b)	CREATE FILTER “ $y=\sin(x)$ ” AS “S1” WITH DRANGE 0, 3000 IN PERIOD 10
(c)	CREATE FILTER PULSEWAVE AS “S1” WITH DRANGE 0, 3000 IN PERIOD 10
(d)	SET FILTERS “Q1”, “N1”, “S1” ON TCP SPRINGS 567 ON st01 LIMIT 1
(e)	SET FILTERS “Q1” ON TCP SPRINGS * ON st01,st02,st08 LIMIT -1
(f)	REMOVE FILTERS “Q1”, “N1”, “S1” ON TCP SPRINGS 567 ON st01
(g)	DELETE FILTERS *

Figure 5.6: An example set of StreamFarm queries for managing filters

There are four main types of Springs in the current design of StreamFarm. These Springs are called *Pure Springs* (PSPRINGS), *Graph Springs* (GSPRINGS), *Emulator Springs* (ESPRINGS), and *Amplifier Springs* (ASPRINGS). StreamFarm allows for creation of data streams using some sample data set files. When queries such as shown in Figure 5.3 is executed, FarmManager fetches the file from the specified location and sends it to all the Workers. Each line of the file is read and is pumped as a data tuple through the network to the appropriate application by the Springs configured to do so. The language allows for creation of any number of springs (e.g., In Figure 5.3 (b) the

query creates 8 TCP springs).

Another important feature of the language is its support for definition of *filters*. The filters allow for modifying the characteristics of data streams generated by StreamFarm. Figure 5.6 shows some example filter related queries.

5.5 Conclusion

This chapter presented StreamFarm which is a stream workload generator platform for benchmarking stream processing systems. Unlike most of the current approaches for stream workload generation, StreamFarm provides a flexible interface for controlling the workload generator platform which allows it to be used in multiple optimization mechanisms directly.

5.6 Summary

StreamFarm which is a workload generator platform for benchmarking and performance characterization of stream processing systems was described in this chapter. Section 5.1 provided an introduction to the field of stream workload generation. Methodology followed in StreamFarm was described in Section 5.2. Design and implementation of StreamFarm was explained in Section 5.3. StreamFarm query language was presented in Section 5.4. The section provided some examples for StreamFarm queries. The conclusions arrived are described in Section 5.5. Next chapter (Chapter 6) describes construction of software automatic tuning techniques for stream application performance optimization.

Automatic Optimization of Stream Programs via Source Program Operator Graph Transformations

As mentioned in Chapter 1 large scale data analysis applications need to be optimized (i.e., fine tuned) for the environment they are intended to operate in. Tuning of a data analysis system can be conducted at different levels such as application code level, middleware level, operating system level, network level, etc. This thesis focuses mainly on the data analytics optimization techniques conducted at the application code level. This chapter describes application of code transformation and software auto-tuning techniques for optimizing stream applications. Section 6.1 provides an introduction to stream data analytics optimization via auto-tuning. Terminologies used in the chapter are described in Section 6.2. An overall view of the methodology followed is presented in Section 6.3. Program transformation including Ray transformation algorithms are described in Section 6.4 while supporting software modularizations called Transformer Blocks are presented in Section 6.5. Next, in Section 6.6 implementation details of Hirundo mechanism which implements the software auto-tuning techniques is presented. Fault tolerance characteristic of Hirundo is very important for conducting accurate optimization sessions which is described in Section 6.7. Sample program space reduction and ranking techniques used to find the optimized sample applications in a faster manner are described in Section 6.8. Evaluation details of software-auto tuning technique and the Hirundo mechanism are given in Section 6.9. A discussion of the evaluation results are presented in Section 6.10 before presenting conclusions in Chapter 6.11.

6.1 Introduction

High performance of a stream program is determined not only by its structure, but also by the topology and performance characteristics of the stream processing system on which it runs. However, stream programs deployed on most of the stream processing systems produce low performance since they are less optimized for the target runtime environment [165] which makes the target cluster’s CPU, memory, and network resources to be under utilized. Such under utilized compute clusters result in wasted CPU cycles, in efficient electricity usage, etc.

A solution for this issue is manual fine tuning of the program to consume the unused system capacity and to produce efficient execution of streaming applications [88][287][288]. Yet, this requires tremendous amounts of programmer’s time and expertise since there are various different ways a stream program’s flow graph can be constructed that gives the same semantics but completely different performance characteristics. Many organizations in different countries face the problem of lack of highly skilled programmers for developing, deploying, and managing applications [20]. Moreover, in production environments run-time topology may change quite significantly [289]. For example, some of the existing nodes may be brought down for maintenance. Therefore, manual fine tuning costs a lot for organizations and it might not be practical in many production environments.

A second solution for this problem is to conduct a profile driven optimization [290]. Results obtained from profiling can be used to characterize the run-time behavior of operators and an optimization model can be constructed to map operators to specific node/process which enables the stream processing job to perform better. However, such approach does not necessarily make any changes to the structure of the input program which is an important factor that contributes to the performance of the stream processing job. Hence we provide a solution from the point of view of the programmer because source level design decisions could affect the entire application’s performance even if profile driven optimization is used. We concentrate on performance optimization of stream jobs at the level of query plan (i.e., DAG). Although our focus in this chapter is on System S, our approach can easily be generalized to different operator based stream processing systems due to the aforementioned reason.

Put in concrete terms our goal of performance optimization is to increase resource utilization (CPU/Memory/Network) of stream applications that leads to increased throughput. This in turn results in less idling CPU cycles of the system leading to reduced CPU cost. We achieve our goal by automatically identifying and creating parallelized sections of the stream application code that can utilize the idling CPU and memory resources.

We first identify the structure of the input stream program and represent it as a Directed Acyclic Graph (DAG) [39]. In the constructed DAG vertices represent operators and edges represent the streams that make connections between them. Next, the input SPADE application is profiled to obtain its operator CPU usage. If the DAG is a multi-path DAG the profile information is used to rank the directed paths from Source to Sink of the DAG. The path with the highest CPU cost is selected for transformation. Next,

the best transformation patterns are identified which could be applied to the selected path using the path's CPU usage profile information. The chosen code transformations are applied to the input application's flow graph to obtain sample programs. This chapter introduces Tri-Operator Transformation and Bi-Operator Transformation which are novel methods for transforming operator-based data flow graph of a stream program without violating their semantics. Bi-Operator Transformation is essential when transforming multipath DAGs because often there arises the need for transforming two operator block segments in multipath DAGs.

Out of the generated sample programs, a subset is chosen and ran in the target distributed environment. A ranked list of sample programs are selected as the output. The proposed performance optimization algorithms were implemented on a mechanism called Hirundo¹ [291][165]. We observed that our methodology is capable of producing higher performance versions of a stream program within a short period (e.g., five minutes) of time.

6.2 Preliminaries

This section provides formal definitions of a stream data flow graph.

Definition 1 (Stream Data Flow Graph). : A stream data flow graph (sdf-graph) G is a triple $G = (N, E, n_0)$, where:

- N = Set of nodes representing operators
- E = Set of edges ($E \subseteq N \times N$) representing streams of data that flow between operators.
- (N, E) = A directed graph
- n_0 = Is the initial node, there is a path from n_0 to each node in N .
- U_s = Source set
- U_i = Sink set

All the sources (input operators) of a stream data flow graph belong to Source set while all the Sinks (output operators) belong to Sink set.

Definition 2 (Fragment). : Any graph M being part of the G is called its fragment.

¹Hirundo is a bird genus name used for representing a group of passerines in the family Hirundinidae.

Definition 3 (Vertex predicates). : If (n_1, n_2) pair is in E , we denote the edge by $n_1 \rightarrow n_2$, and label n_1 as a predecessor of n_2 and n_2 as a successor of n_1 . Predecessors of vertex x is denoted by $PRED(x)$. Successors of vertex x is denoted by $SUCC(x)$. Arcs entering vertex x is denoted by $INARCS(x)$. Arcs leaving vertex x is denoted by $OUTARCS(x)$.

Definition 4 (Path). : A path in graph G is a sequence of nodes $n_1, n_2, \dots, n_k, k \geq 1$, such that $n_i \rightarrow n_{i+1}$ for all $i, 1 \leq i < k$ ($\forall i \forall j, n_i \neq n_j, i \neq j$).

Definition 5 (Ray). : A simple path $L = (l_1, \dots, l_n), n \geq 1$ on the stream graph G is called a linear part (i.e., basic block/ray) if $SUCC(l_i) = 1$ for any $i \in [1:n]$ and $PRED(l_j) = 1$ for any $j \in [1:n]$

Definition 6 (Operator Block). : An operator block B is a fragment of G ($B \subset G$).

Definition 7 (Transformed Operator Block). : An operator block B_T that is produced by transforming an operator block B is called the transformed operator block of B .

Definition 8 (Cardinality of an operator block). : The number of operator blocks of type B present in a given ray of a stream application is called the operator block B 's cardinality on that ray.

Definition 9 (Transformation Depth). : The maximum number of operator blocks that can be present at the beginning/ending of a Transformed Operator Block during an optimization session (i.e., run) is called the optimization session's transformation depth and is denoted by d ($d > 1, d \in \mathbb{N}$).

6.3 Methodology

Input for Hirundo is a stream program, a sample data file, and a collection of meta-information stored in configuration files. The input program is analyzed using a custom grammar to identify operator blocks. After Hirundo identifies all the operator blocks present within the input program it constructs a DAG where each operator is a node and streams that connect operators are edges. Note that current version of Hirundo optimizes stream programs having flow graph structure of DAGs.

This DAG along with the operator profiling information are used during the optimization process. User can specify the optimization goal before the optimization process begins in terms of expected throughput increase (e.g., 2 times, 3 times, etc.). During run time Hirundo generates a sample program space with initial transformation depth value (d). Transformation depth (defined in Section 6.2) is an integer (>1) specified prior starting the optimization process and it plays a central role in determining the size of the sample program space. An explanation of the relationship between the transformation depth and the sample program space size is given in Section 6.8. Next, Hirundo runs the

input application in the environment for a short period and obtains each operator's profile information (CPU usage, Memory usage, and network I/O data rates). These profile information along with previous optimization run information are used to select a subset of sample programs (U_1). The selected programs are compiled in parallel in distributed node cluster. Then each sample application in U_1 are run in the intended environment for a time window of W_t . A ranked list of programs R_1 ($R_1:R_1 \subset U_1$; $|R_1|=m$; $m < n$) is selected based on the performance results obtained by running the programs. Based on the performance results obtained, Hirundo determines whether the optimization goal is met or not. If the optimization goal is not met, it increases the transformation depth by 2 and transforms the input application using the same transformations used to generate the R_1 sample programs. The generated sample programs are compiled as previously, run in the environment for W_t time to obtain their performance. Similar to previous run, a ranked list of programs are selected and is compared whether the optimization goal is achieved or not. This process is continued until the performance optimization goal is met or a less performance value was encountered for newly generated sample applications. The performance optimization process is shown in Figure 6.1.

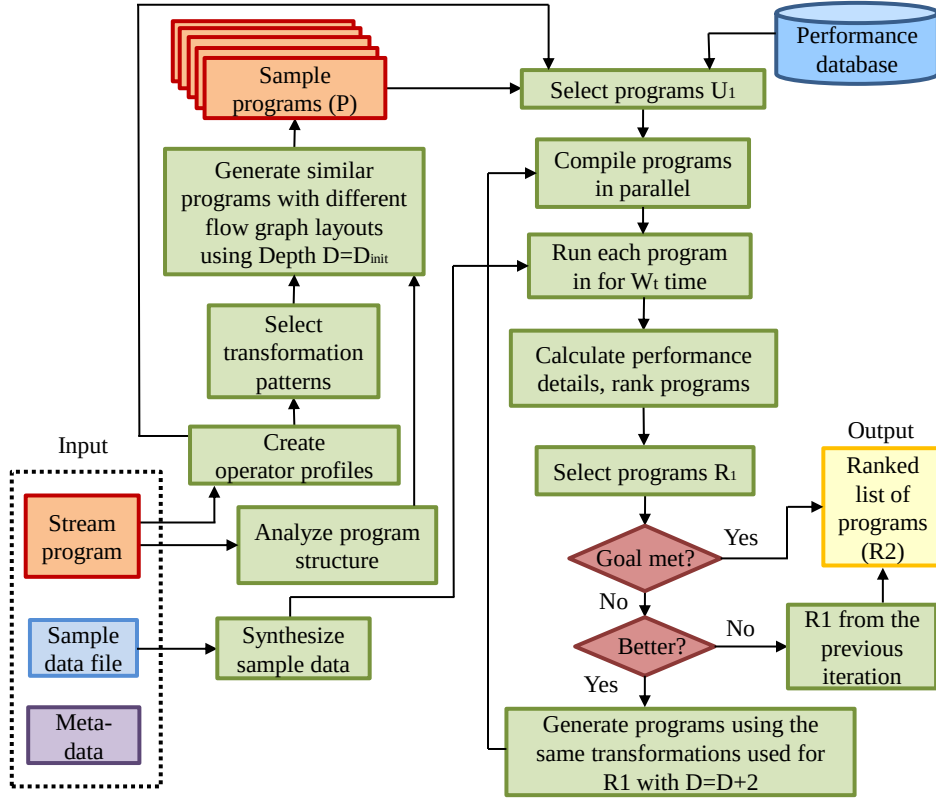


Figure 6.1: Methodology of Hirundo

The performance optimization methodology consists of two main components, Program Transformation and Sample Program Space Reduction which are described in two

separate sections (Section 6.4 and Section 6.8) of this chapter in order for better organization of the chapter's content. While more details are provided on the next sections; an overview of the program transformation and sample program selection is provided in this section for the completeness of the description of methodology.

Stream program optimization at the source code level can be conducted following three main techniques; operator parallelization, pipelining, and fusion. We chose operator parallelization (also known as parallel pattern/Fission [158]) [233][292] for implementing the code generator because in general stream applications are written as collections of light weight operators. Pure use of pipelining and fusion have drawbacks of increased communication overhead and I/O bottlenecks. In pipelining, when an operator's code is divided into separate stages of execution, it adds an additional communication path to the flow graph which carries the same amount of information between them. In the parallel pattern, the data are divided among streams which eliminates this problem. We compared our approach with purely fusion based method in Section 6.9.6, and show that our method is better in improving stream application performance.

During the optimization process, the input application is profiled for a short period before sample programs are generated to identify the resource usage information of each operator. These profile information are used to select the sample programs to be used for sample program ranking. The program transformation algorithms used in our method (Tri-OP and Bi-OP transformations) are aligned with the goal of optimizing the flow graph by eliminating the performance bottlenecks. This is further explained in Figures 6.5 and 6.6.

Based on the transformation categorization presented in Figures 6.5 and 6.6, we select half of the sample programs in U_1 . The other half of the sample programs are selected using the results from previous optimization runs. The selection is made considering the performance information of previous runs of the application as well as applications which were optimized previously which had similar structure to the input application. Results obtained from program ranking and selection are listed in Section 6.9.4.

6.4 Program Transformation

We describe a methodology for transforming a stream program to a variety of sample applications which are used as sample programs for optimization process. First, we describe example applications that are used during the rest of the chapter.

6.4.1 Example Stream Applications

This chapter uses five real world stream applications from different application areas such as healthcare services, financial markets, web analytics, etc. to evaluate effectiveness of the proposed approach.

The first application is Call Detail Record (CDR) processing application. A Call Detail Record (CDR) is a piece of information produced by a telephone exchange containing details of a phone call passed through it. Telecommunication networks process massive

amounts of CDR events, an application area for stream processing systems. CDR application has five operators Source operator (S) injects tuples to the flow graph. For each tuple received, UDOP (U) connects with a customer database and checks whether the customer is blacklisted or not. The Functor (F) counts the total number of packets, while Aggregate (AG) operator aggregates tuples by the call station ID.

The second application is a Twitter hash tag counter application (*Twitter*). Twitter users can label the key words of their tweets using # (e.g. #rose) which indicates to which conversations the message relates to. Twitter streams may contain thousands of tweets per second [293] which could be analyzed using a stream processing system. This is a six operator application (shown in Figure 6.2 (b)) including one UDOP (U), three Functors (F1, F2, and F3), and an Aggregate operator (AG). UDOP reads data from a dump of twitter data. F1 tokenizes the tweets and F2 extracts hash tags from the words. F3 eliminates empty tuples. AG aggregates hash tag tuples, emits the has tag count for every 10 tuples it receives, and emits summary of results for each tuple it receives.

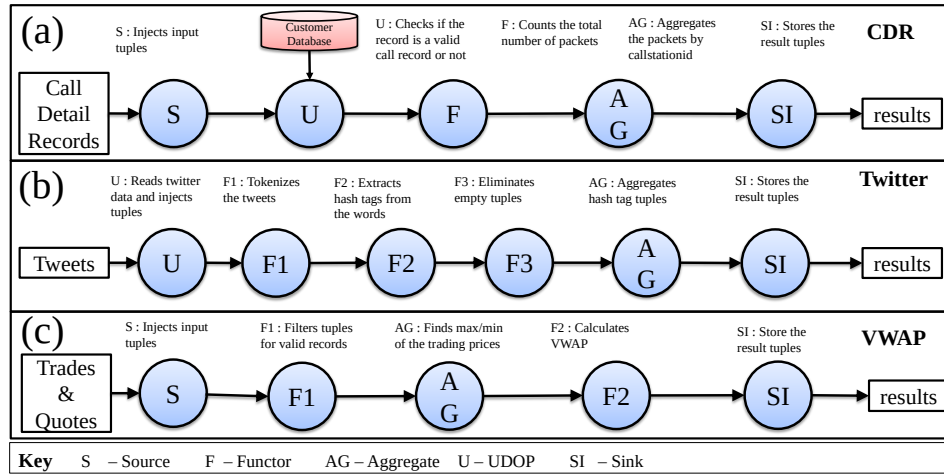


Figure 6.2: Data flow graphs of CDR, Twitter, and VWAP applications

The third application is a Volume Weighted Average Price Application (VWAP). VWAP is calculated as the ratio of the value traded and the volume traded within a specified time horizon. The data flow graph of VWAP contains five operators including two functors (F1 and F2) and an Aggregate operator (AG).

Note that the VWAP, CDR, and Twitter applications are used to demonstrate how linear data flow graphs (linear DAGs) are transformed by Hirundo. Also CDR and Twitter are examples for transforming stream applications with UDOPs. The remaining two applications (Apnoea and Bargain) are used to show how stream program optimization is conducted on multipath DAGs.

The fourth application is named as Apnoea. Apnoea is the most common problem of ventilatory control in premature infants [294]. Apnoea makes infants to be hospitalized frequently with the need for cardiopulmonary monitoring. Catley *et al.* [294] implemented a framework for the real-time analysis of time series physiological data

streams extracted from a range of medical equipment to detect clinically significant conditions (such as apnoea) using System S. Their framework has been implemented on top of Artemis analytics system [14]. This chapter implements the same apnoea detection stream application (shown in Figure 6.3) described by them.

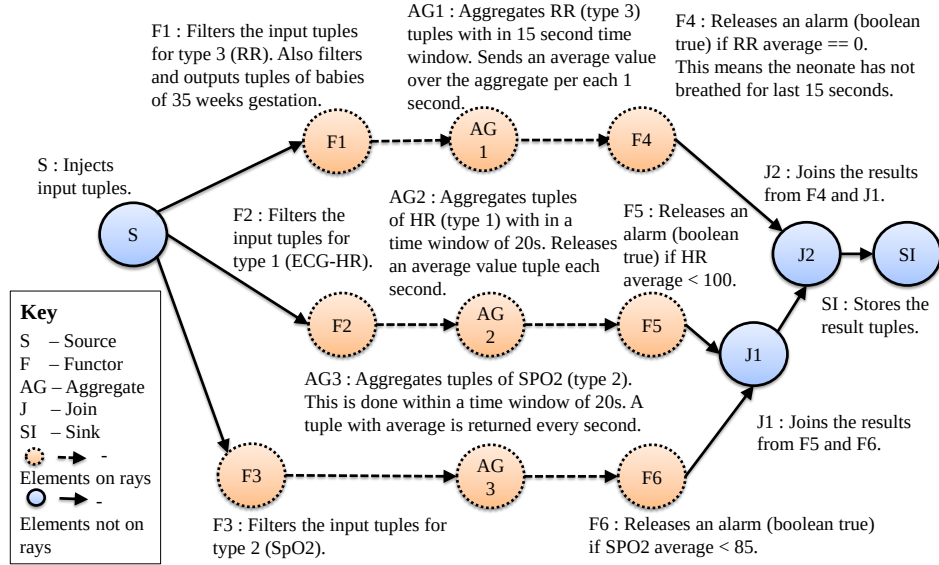


Figure 6.3: Data flow graph of Apnoea detection SPADE application. Note that the operator outlines using dotted lines and the arrows that connect operators using dotted lines represent rays on the data flow graph. E.g., F2, AG2, and F5 are on a single ray.

The fifth application is a bargain detection application (here onwards Bargain) which has been described in [231][13]. This application (shown in Figure 6.4) detects bargains as stock quotes flow into the system. A bargain is simply how much cheaper someone is willing to sell a stock versus its moving average price as seen in older trades. Bargain application includes the Volume Weighted Average Price (VWAP) calculation application that is described in this chapter. Optimization of Bargain is much complex than VWAP because Bargain application is a multipath DAG. The application ingests a stream of tuples, obtains the configuration for the portfolio for which it should be looking for a bargain, computes the moving averages on trades, compares the moving average price with quotes, and computes and exports a bargain index to an algorithmic trading application.

Programmer of Bargain can specify the list of n portfolio items (e.g., IBM, HPQ, MSFT) during application compilation. Then the application gets translated in to a tree like structure with n number of branches (e.g., Three branches in the above case).

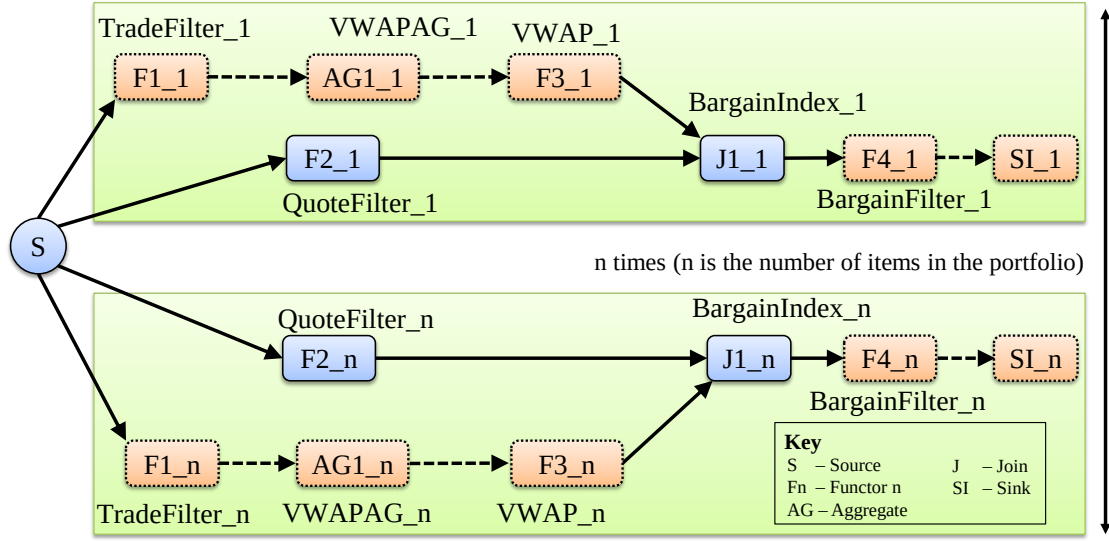


Figure 6.4: Data flow graph of the Bargain calculation SPADE Application.

6.4.2 Ray Transformation Algorithms

Hirundo’s program transforming algorithm generates many sample applications using the input application’s DAG by transforming its rays. We use two ray transformation algorithms that we call *Tri-Operator Transformation* (Tri-OP transformation) and *Bi-Operator Transformation* (Bi-OP transformation). If a ray consists of at least three operator blocks it applies Tri-Operator transformation. If the ray consists only two operator blocks Hirundo applies Bi-Operator transformation. The next half of this section explains how Tri-Operator transformation and Bi-Operator transformation are performed on rays.

6.4.2.1 Tri-Operator Transformation

Lets consider three adjacent operator blocks (an operator block is a collection of operators) A, B, and C in a ray (Shown in Figure 6.5(a)). From here onwards such operator blocks are denoted as A_B_C. Note that this thesis uses the term “Operator Block” to denote each A, B, and C as well as A_B_C because A_B_C itself is a collection of operators.

The aim of the Tri-Operator transformation algorithm is to generate a variety of data flow graphs for a given stream program preserving the transformed operator blocks’ semantics. We have chosen to transform three adjacent operator blocks at a time due to several reasons. Transforming three operator blocks at a time provides more control over the transformation process. By use of the term “more control” here means that size of the transformed operator space does not increase drastically. Transformation logic should not increase the number of operators exponentially and use of three operator

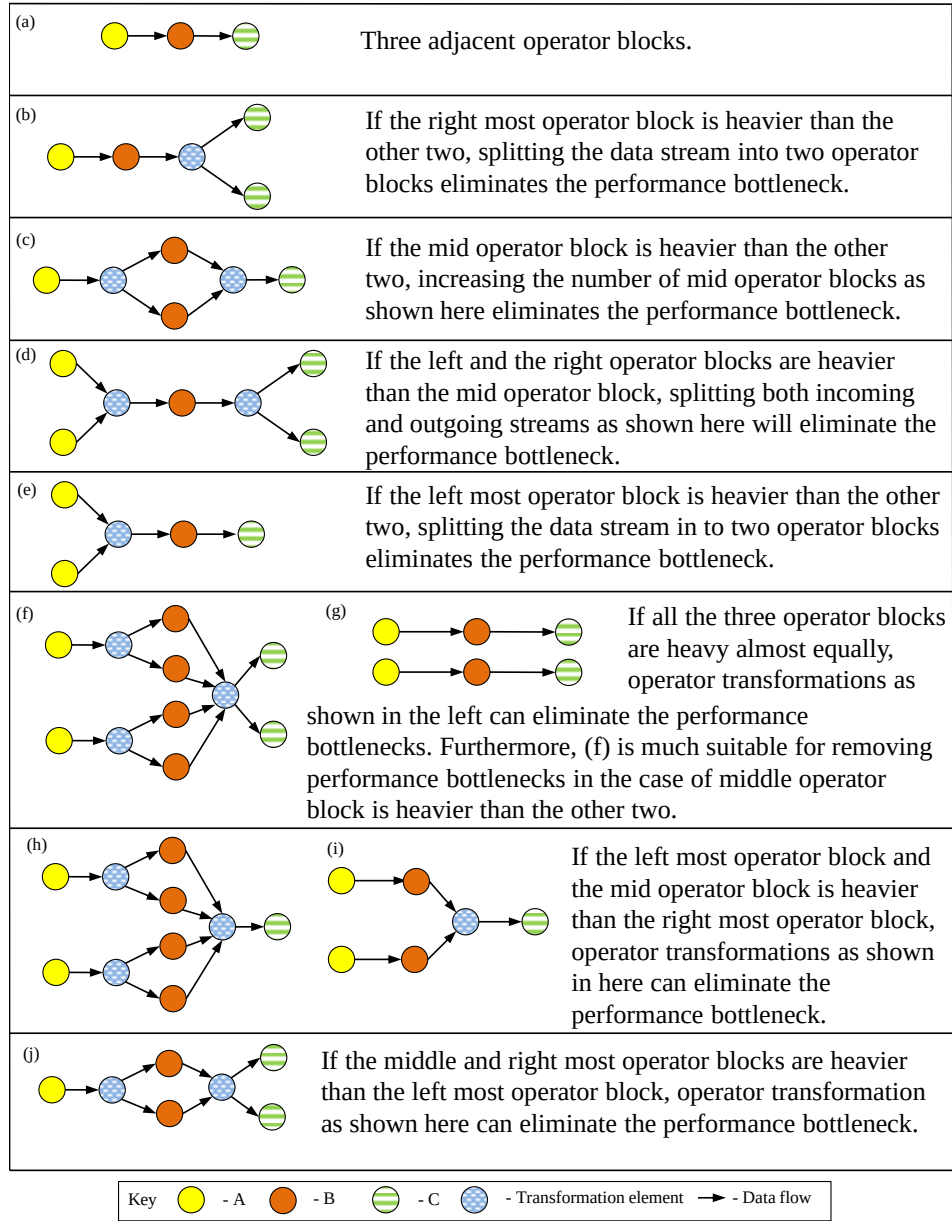


Figure 6.5: Sample transformations using Tri-OP transformation. The figure also shows categorization of possible transformations for a three operator block linear segment. (a) Input operator blocks (b) Result of applying 1-1-2 transformation (c) Result of transforming by 1-2-1 (d) Result of transforming by 2-0-2 (e) Result of transforming by 2-0-1 (f) Result of transforming by 2-2-2 (g) Result of transforming by 2-1-2 (h) Result of transforming by 2-2-1 (i) Result of transforming by 2-1-1 (j) Result of transforming by 1-2-2.

blocks at a time ensured this characteristics. We have shown the number of sample applications generated by Tri-OP transformation do not grow exponentially in Table 6.1. Changing the number of middle operator blocks (i.e., operator block B in Figure 6.5(a)) in a three operator blocks combination, we can achieve the controllability of the transformation process. Some of the transformation patterns that could have been produced if the transformations were conducted four operator blocks at a time (*Quadri-Operator Transformation(Quadri-OP transformation)*) are shown in Appendix B. However, as explained in Appendix B it may produce an enormously large transformed operator block space even at small transformation depth value compared to Tri-OP transformation. Furthermore, specifying Transformer Patterns (described in next sub-section) at the level of Quadri-OP is relatively complex task because it is difficult to express concepts such as *Accelerated Transformation* in Quadri-OP transformation. However, many of the transformations of the Quadri-OP transformation can be achieved using Tri-OP transformation and the concept of operator block fusion. Therefore, we conduct ray transformation in Hirundo using up to Tri-OP transformation algorithm.

Similar to Palermo *et al.* we are also motivated by the two-dimensional Fast Fourier Transformation's (FFT) use of butterfly communication networks [126] in creating Tri-OP transformation. During the Tri-OP transformation process we utilize a special set of transformer blocks that work as adapters that support operator parallelization. These operator blocks will be described more in detail in Section 6.5.

6.4.2.2 Transformer Patterns of Tri-Operator Transformation

In this thesis the notations i-j-k (i,j,k are positive integers including 0) are used to denote a transformation pattern. In simple terms the i,j, and k values represent the number of parallel operator blocks that will be created in three operator block pattern. Psuedocode of `transform.triop()` procedure in Algorithm 1 explains how an operator block A_B_C is transformed by an i-j-k pattern.

Tri-OP transformation does not do any change to A_B_C operator blocks if it transforms using the pattern 1-1-1. The meaning of 1-1-1 can be described as follows. Keep one A, increase the number of Bs to 1×1 , and map transformed B operator blocks to one C. Tri-OP transformation does not consider any other patterns having only 0s or 1s (e.g., 0-0-0, 0-0-1) other than the pattern 1-1-1 to avoid duplication. Transformation pattern 1-2-1 transforms A_B_C in to A_2B_C (See Figure 6.5(c)). This means that keep one A, increase the number of Bs to two, and map the two Bs to one C. This is an example for increase of middle operator blocks. Minimum number of middle operator blocks is 1.

Tri-OP transformation creates a single B when it finds $j=0$. E.g., When transformation pattern 2-0-2 is applied to A_B_C, it results in 2A_B_2C (Shown in Figure 6.5(d)). In this example two As are mapped to a single B. Then streams from B are mapped to two Cs. The value of j plays an important role in describing the structure of the resulting operator block. Lets take the scenario of applying 2-1-2 transformation to A_B_C. This is an example for $i = k, j = 1$ in Line 13 of Algorithm 1. It will result in 2A_2B_2C,

Algorithm 1 transform_triop(A_B_C, i, j, k)	Algorithm 2 transform_biop(A_B, i, j)
<pre> /* Note : transformation indices are i (left), j (mid), and k (right); operators are A (left), B (mid), and C (right)*/ 1: if all the transformation indices are 1 then 2: return A_B_C 3: end if 4: if all transformation indices are 0 or 1 then 5: return 6: end if 7: if both beginning and ending indices are 0 then 8: return 9: end if 10: if mid transformation index is 0 then 11: return iA_B_kC /* Keep mid operator as it is and 12: increase left and right operator count*/ 13: if mid index is 1 with left and right indices are equal then 14: return parallel(iA_iB_iC)/* Increase all the operators by equal 15: amount, connect the operators in parallel*/ 16: end if 17: return iA_(i×j)B_kC /* Increase left and right operators by corresponding transformation indices. Also, increase mid operators by the result of multiplying left and the mid transformation indices*/ </pre>	<pre> /* Note : transformation indices are i (left), and j (right); operators are A (left), and B (right) */ 1: if all the transformation indices are 1 then 2: return A_B 3: end if 4: if all transformation indices are either 0 or 1 then 5: return 6: end if 7: if only left transformation index is 0 then 8: return 9: end if 10: if only right transformation index is 0 then 11: return iA_B 12: end if 13: if left transformation index is greater than 1 14: and right transformation index is 1 then 15: return parallel(iA_iB)/* Increase all the operators by 16: equal amount, connect the 17: operators in parallel*/ 18: end if 19: return iA_jB /* Increase left and right operators by the corresponding transformation indices. */ </pre>

having two operator blocks from each category (shown Figure 6.5(g)). Furthermore, these operator blocks will be connected in parallel. Yet when transforming by 2-1-1 it will result in 2A_2B_C, which changes cardinality of A and B operators to 2 and keeps single C operator. Similarly 2-2-2 (shown in Figure 6.5(f)) and 2-2-3 transformation patterns result in 2A_4B_2C and 2A_4B_3C transformed operator blocks respectively. Transformation patterns 2-2-2 in Figure 6.5(f) is an example for an *Accelerated Transformation* that happens in Tri-OP transformation. The term *Accelerated* is used because we get $(i \times j)$ number of transformed operator blocks for mid operator block of the input operator block.

The size of the entire transformation space possible with Tri-OP transformation can be calculated as follows. The collection of the transformations shown in Figure 6.5 represents the entire possible transformation space from Tri-OP transformation. If we specify transformation depth d ($d > 1$) at the beginning of a transformation session, sample transformations shown in Figures 6.5 (b), (c), (d), (e), (f), (g), (h), (i), and (j) can produce $(d-1)$, $(d-1)$, $[(d-1)(d-1)]$, $(d-1)$, $[(d-1)(d-1)(d-1)]$, $(d-1)$, $[(d-1)(d-1)]$, $[(d-1)(d-1)]$, and $[(d-1)(d-1)]$ transformations respectively. The summation of these transformation totals to $d^3 + d^2 - d - 1$. In the case of transformation shown in Figure 6.5 (i) we conduct the calculation assuming multiple right most operators scenario as well. Also we eliminate the scenario having equal number of operators because that is addressed by (g). But $(d-1)$ was added to get the total of $[(d-1)(d-1)]$ for Figure 6.5 (i) because we need to consider the scenario having single right-hand side operator block. If we define transformation space function $f_{tri}(d)$ to represent the total number of possible transformations, we get $f_{tri}(d) = d^3 + d^2 - d - 1$ (Note that the subscript *tri* denotes that this function is for Tri-OP transformation.). This means that transformation depth values 2, 3, and 4 will produce $f_{tri}(2) = 9$, $f_{tri}(3) = 32$, $f_{tri}(4) = 75$ transformed operator

blocks from Tri-OP transformation respectively.

6.4.2.3 Bi-Operator Transformation

Similar to Tri-OP transformation, the aim of Bi-OP transformation is to transform a given ray which has only two operator blocks to a variety of flow graph patterns. Bi-OP transformation is essential for transforming rays of the graphs which have only two operator blocks. For example, the operator blocks F4_n and SI_n shown in Figure 6.4 is an example for such two operator block region which needs the Bi-OP transformation. However, as can be observed from Figure 6.6 Bi-OP transformation does not result in a much variety of transformed operator blocks such as produced by Tri-OP transformation. Algorithm 2 describes how an operator block A_B is transformed by an i-j pattern. Note that Bi-OP transformation cannot be considered as a special case of Tri-OP transformation because Tri-OP transformation has the ability of producing unique transformations based on the cardinality of the mid operator block.

6.4.2.4 Transformer Patterns of Bi-Operator Transformation

Similar to Tri-OP transformation, Bi-OP transformation does not do any change to A_B operator blocks if it transforms using 1-1. Transform pattern 1-2 transforms A_B in to A_2B. This transformation is Shown in Figure 6.6(b). When the transformation pattern 2-0 applied to A_B it creates 2A_B (shown in Figure 6.6(c)). However, the transformation pattern 2-1 transforms A_B to 2A_2B (shown in Figure 6.6(d)). This is an example for parallel pattern. Any transformation pattern i-j ($i > 1, j > 1$) applied to A_B results in iA_jB. For example, the patten 2-2 transforms Figure 6.6(a) to Figure 6.6(e).

Similar to Tri-OP transformation, the size of the entire program transformation space from Bi-OP transformation can be calculated as follows. A transformation depth d ($d > 1$) specified at the beginning of Bi-OP transformation will produce $(d-1)$, $(d-1)$, $[(d-1)(d-1)]$, $(d-1)$ transformed operator blocks for the general transformation patterns shown in Figures 6.6 (b), (c), (d), and (e). If we define transformation space function $f_{bi}(d)$ to represent the total number of possible transformations, we get $f_{bi}(d) = d^2 + d - 2$. Transformation depth values 2, 3, and 4 will produce $f_{bi}(2) = 4$, $f_{bi}(3) = 10$, $f_{bi}(4) = 18$ transformed operator blocks from Bi-OP transformation respectively.

6.4.3 Program Transformation Algorithm

This section explains how program transformation is conducted on the CDR, Twitter, VWAP, Apnoea, and Bargain example applications that were described in Section 6.4.1.

After Hirundo accepts an input program it parses the program using its Program Structure Analyzer module. The Program Analyzer identifies each operator block and creates a graph (G) which represents the structure of the input program. The transformation process is summarized in Activity diagram on Figure 6.7. The graph is checked to see whether it is a linear one (i.e., ray/linear DAG) or a graph with multiple paths (i.e., multipath DAG).

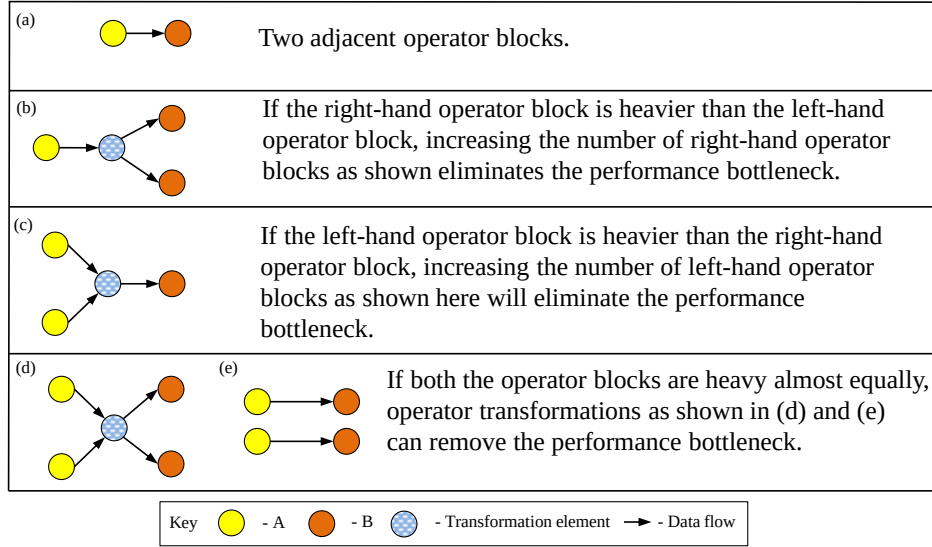


Figure 6.6: Sample transformations using Bi-OP transformation. The figure also shows categorization of possible transformations for two operator block linear segment. (a) Input operator blocks (b) Result of applying 1-2 transformation (c) Result of transforming by 2-0 (d) Result of transforming by 2-2 (e) Result of transforming by 2-1

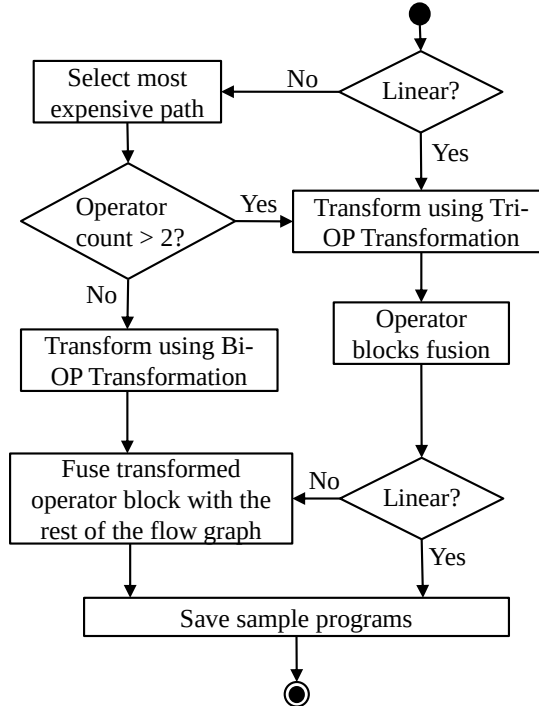


Figure 6.7: An overview of Hirundo's program transformation algorithm.

Algorithm 3 tri_op_transform(G, d)	Algorithm 4 generate(G, i, j, k)
<pre> 1: oblist ← emptylist 2: for i ← 0 to d do 3: for j ← 0 to d do 4: for k ← 0 to d do 5: outdictionary ← generate(G, i, j, k) 6: oblist.add(outdictionary) 7: end for 8: end for 9: end for 10: weld(oblist, len(G)) </pre>	<pre> 1: m ← 0 2: v ← getroot(G) 3: while v has next do 4: inarray ← getnextthreevertices(v) 5: if length[inarray] = 3 then 6: tarray[m] ← transform(inarray, i, j, k) 7: m ← m + 1 8: v ← inarray[2] 9: end if 10: end while 11: return (tarray) </pre>

If the graph is a linear DAG the transformations are conducted using Tri-OP transformation. If not (the DAG is multipath) the most expensive path on the DAG is selected. If the number of operator count is more than 2, Tri-OP transformation is applied. If not the path is transformed with Bi-OP transformation. If the graph is multipath, the transformed operator blocks from the operator fusion need to be fused back with the original flow graph to construct the final sample applications. Finally, the generated sample programs are saved in to different source files on the file system.

6.4.3.1 Program Transformation on Linear DAGs

As mentioned previously, if the graph is a linear DAG, it is fed to `transform.triop()` procedure (shown in Algorithm 3) along with a depth value `d`. Depth value `d` is a positive integer that determines to what extent the input program will be transformed.

Initially an empty operator block list (oblist) is created. An operator block is represented as a dictionary object. The procedure traverses graph `G` in steps of 3 operator blocks. This can be observed from the three `for()` loops (Lines 2-4, Algorithm 3). Each pass generates an `i-j-k` pattern.

The `tri_op_transform()` procedure calls `generate()` procedure (see Algorithm 4) passing the graph `G` and the `i, j` and `k` values. Functionality of the `generate()` procedure can be visualized using Figure 6.8.

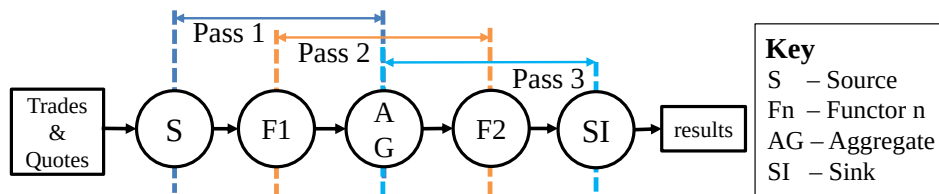


Figure 6.8: How VWAP application's DAG is traversed by GENERATE() procedure.

First, `generate()` procedure moves to the root node of the graph (in the example it is the source operator (S)). Then it selects three adjacent operator blocks from root (S, F1, and AG) by calling `getnextthreevertices()` and applies the transformation of the i-j-k pattern by calling `transform()`. This is termed as the pass1 in Figure 6.8. Then `generate()` procedure moves to the neighbor of the root (that is F1), picks three operator blocks (F1, AG, and F2) and applies the transformation of i-j-k pattern to them (pass2). Note that we chose the second operator block rather than the fourth operator block because to enable transformation of graphs with vertex counts not belonging to multipliers of three. Finally, it applies transformation of i-j-k pattern to operator blocks AG, F2, and SI (pass3). If there are n operator blocks present in a linear DAG, `generate()` procedure does the i-j-k pattern transformation $n-2$ times. The resulting $n-2$ operator blocks are saved in a dictionary. These resulting operator blocks are named as *Transformed Operator Blocks*. The keys (i.e., labels) of the transformed operator blocks are created using i,j,k values and the input operator block names (i.e., A, B, and C). The dictionary gets saved in the oblist (See line 6 of Algorithm 3).

Finally, `tri_op_transform()` procedure calls `weld()` procedure (shown in Algorithm 5) by passing the transformed operator blocks list (oblist) and the input program graph length (glen) to `weld()` (glen is 5 for the VWAP application shown in Figure 6.8). The `weld()` procedure selects matching operator blocks from oblist and fuses them to create sample programs. As shown in Algorithm 5, `weld()` procedure traverses the list of transformed operator blocks that it received. If a particular operator block has one or more source operators (i.e., It is a source operator block), the procedure creates a matched operator blocks list (fList). Then it finds matching operator blocks for source operator block by calling `findoblist()` (Line 5 in Algorithm 5) and stores in another operator block list (tList). The pseudocode of `findoblist()` procedure is shown in Algorithm 7. If tList is not empty it means that there are matching operator blocks for source operator. In this case transformed operator blocks list received from `generate()` procedure (oblist), tList, fList, maximum depth of traversal (glen - 3) are fed to a recursive procedure called `getmatchob()` as shown in Line 7 of Algorithm 5.

The pseudocode of `getmatchob()` is shown in Algorithm 6. It finds matching operators for every operator in the transformed operator block list (oblist) and adds them to fusionList, which is a globally defined list that holds the final result. Next, it appends operator block ob to fList. A call to `getmatchob()` procedure returns a matched operator block list which keeps lists of matching operator blocks (fusions) in sequential order. After `getmatchob()` completes execution, the duplicate operator blocks in fusionList are removed by calling `filter()` procedure (Line 11 of Algorithm 5). Then `weld()` procedure fuses each and every fusion that is stored with in the fusionList to construct sample programs. A fusion in Lines 12-13 in Algorithm 5 refers to a sample program with operator blocks which are not fused yet. Merging of operator blocks located in each individual fusions is done by `fuse()` procedure (Line 13).

Algorithm 5 weld(oblist, glen)	Algorithm 6 getmatchob(oblist, tList, fList, depth)
<pre> 1: fList ← emptyList 2: for all oblock in oblist do 3: if oblock is sourceoblock then 4: fList ← add(fList, oblock) 5: tList ← findoblist(oblist, oblock) 6: if tList not empty then 7: getmatchob(oblist, tList, fList, glen - 3) 8: end if 9: end if 10: end for 11: filter() 12: for all fusion in fusionList do 13: resprog ← fuse(oblist, fusion) 14: save(resprog) 15: end for </pre>	<pre> 1: for all ob in tList do 2: if ob is sinkoblock then 3: fList.add(ob) 4: fusionList.add(fList) 5: else 6: if depth ≤ 0 then 7: return 8: end if 9: kList ← findoblist(oblist, ob) 10: if kList is not empty then 11: for all obk in kList do 12: fList.add(obk) 13: return getmatchob(obList, kList, fList, depth - 1) 14: end for 15: end if 16: end if 17: end for </pre>

Algorithm 7 findoblist(oblist, ob)
<pre> 1: tList ← emptyList 2: for all oblock in oblist do 3: if ismatch(ob, oblock) then 4: tList.add(oblock) 5: end if 6: end for 7: return tList </pre>
Algorithm 8 ismatch(opb1, opb2)
<pre> 1: if sopb(opb1) = fopb(opb2) and topb(opb1) = sopb(opb2) then 2: inopb1 ← getIndexes(opb1) 3: inopb2 ← getIndexes(opb2) 4: if inopb2[1] = 1 then 5: if topb(opb2) = 'SI' and inopb2[2] = inopb2[3] then 6: return true 7: else if inopb1[3] = (inopb2[1]*inopb2[2]) then 8: return true 9: else 10: return false 11: end if 12: end if 13: end if </pre>

6.4.3.2 Operator Blocks Fusion on Linear DAGs

A typical stream program may consist of minimum three operators. They are Source, a computational operator (e.g., Functor/Aggregate etc.), and Sink. Transformation of such program directly creates sample programs since the three operators represent three operator blocks. However, in most of the scenarios more than three operator blocks are present in stream programs. In such occasions more than one transformed operator blocks are created by `generate()`. These transformed operator blocks need to be stitched together meaningfully (i.e., with same semantics) to produce sample programs. Synthesis of sample programs in such occasions is called *Operator Blocks Fusion*. This process is shown in Figure 6.9.

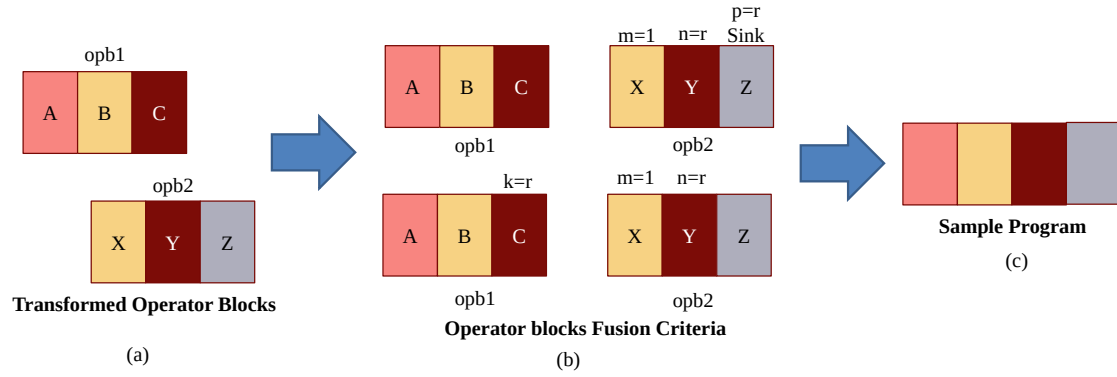


Figure 6.9: Operator Blocks Fusion

A fusion mentioned in the previous section is a list of operator blocks which are arranged in a sequential order which makes a complete sample program (semantically equivalent to input program) if they are concatenated. Ordering of operator blocks is done based on the decision given by `ismatch()` procedure call shown in Algorithm 8. The `ismatch()` procedure determines whether two operator blocks opb1 (iA_jB_kC) and opb2 (mX_nY_pZ) should be fused or not based on their operator types (A,B,C,X,Y,Z) and the transformation pattern values (i,j,k,m,n,p). For two transformed operator blocks opb1 and opb2 to match with each other, first and the second operators of opb2 should be the same as the second and third operators of opb1 (See Line 1 of Algorithm 8). (i.e., $X \equiv B$ and $Y \equiv C$). This is the primary requirement for opb1 to match with opb2. Next, if the opb2's last operator block (i.e., Z) is a Sink operator block then if $m = 1$ and $n = p$ then opb1 and opb2 matches with each other (See Lines 4 to 6 of Algorithm 8). If opb2's last operator block is not a sink operator block then if the conditions $k = (m \times n)$ and $m = 1$ holds, opb1 and opb2 are considered matching with each other. (Note the function names `fopb()`, `sopb()`, and `topb()` correspond to first operator block, second operator block, and third operator block respectively).

6.4.3.3 Program Transformation on Multipath DAGs

The program structure analyzer decides a DAG (G) as a multipath graph if more than one out going edge is present on a vertex ($\exists x: \text{OUTARCS}(x) > 1$). If the program is categorized as multipath DAG (e.g., Apnoea, Bargain, etc.) Hirundo compiles the program in the target runtime and obtains each operator's profile information. The SPADE compiler's SPADE application profiling functionality has been used for this purpose [231]. SPADE application profiler returns a DAG (W) which essentially represents the SPADE application's structure with each vertex (operator) contains the percentage CPU usage as a property of its corresponding vertex.

Next, the graph G is analyzed to identify rays (The rays identified in the example graphs are marked using pink color vertices and dotted arrow edges in Figures 6.3 and 6.4). The identified rays are the fundamental element of the DAG on which the transformations are conducted on. If a ray contains at least three operator blocks, then Tri-OP transformation is performed on it. If the ray contains only two operator blocks then, Bi-OP transformation is applied.

Next, each individual directed path on the graph G are identified. For Apnoea DAG these paths are S-F1-AG1-F4-J2-SI, S-F2-AG2-F5-J1-J2-SI, S-F3-AG3-F6-J1-J2-SI. For Bargain application for the n^{th} portfolio item ($n > 0$) the paths are S-F2_n-J1_n-F4_n-SI_n and S-F1_n-AG1_n-F3_n-J1_n-F4_n-SI_n. These paths are shown in Figure 6.10.

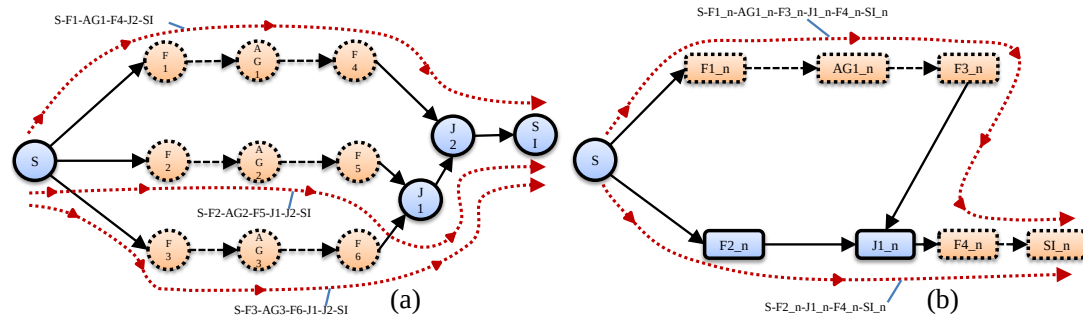


Figure 6.10: Paths on the example data flow graphs. (a) Apnoea (b) Bargain (for n^{th} portfolio item)

The paths are ranked based on the operator cost values present in the graph W and the path with the highest execution cost is chosen as the candidate path for transformation. Furthermore, if there are more than one ray on a particular path (E.g., Both the paths on Bargain application shown in Figure 6.10 (b) have two rays.) the most expensive ray is chosen for transformation. This way we are able to reduce the size of the possible sample program space. The rays on the candidate path are transformed using the Tri-OP/Bi-OP algorithms.

6.4.3.4 Operator Blocks Fusion on Multipath DAGs

Operator blocks fusion is equally applicable to multipath DAGs as well because the process of multipath DAGs transformation involves transformation of a linear DAG. If a ray contains more than three operator blocks it generates multiple transformed operator blocks. In such situations the same fusion algorithm is used to fuse the transformed operator blocks mentioned before. However, if the two ends of a fused ray has cardinality more than one they are attached with the main DAG using transformer blocks such as `DEMUX-STREAM` or `MUX-STREAM`. This is because the two ends of the ray may not be necessarily have similar cardinality as with the case of operator block fusion.

6.4.4 Preservation of Input Program Semantics

Several key measures have been taken to ensure the semantic equivalence of the sample programs to their input program. We believe these measures preserve the semantics of all the input programs processed by Hirundo. First, Operator Block Fusion uses the fusions that have the same operation type sequence similar to the input program's corresponding ray. Next, we make sure by default that UDOPs are not transformed by our code generator module. This is because currently Hirundo does not employ any technique that can determine whether a UDOP is stateful or not. We have implemented a SPADE program annotation scheme for the expert users who want to override the default rule set of code transformations. If a UDOP is enclosed within *UnFroze* annotation tags `#Hirundo-meta:UnFroze:Start` and `#Hirundo-meta:UnFroze:End`, Hirundo then applies the code transformations on the UDOP as well.

Hirundo transforms Aggregate operators using the concept of partitioned state. For example in the Apnoea application, path S-F3-AG3-F6-J1-J2-SI was selected as the costliest path with operator graph profiling. Then the fragment F3-AG3-F6 was transformed with Tri-Operator transformation. In this case the AG3 operator aggregates tuples with the patient id (pid). When transformation has been applied for this fragment, Hirundo introduces an additional Split operation to split the tuple stream to multiple paths based on the pid. Therefore, tuples of a particular pid travels through only one fragment on the transformed graph. Hence the Aggregate operator that is on a particular path receives only a subset of pids consistently. Therefore, the Aggregate operator maintains a separate state for each pid (i.e., tuples are grouped based on the pid) in a partitioned manner. The state is partitioned across multiple parallel Aggregate operators, and it is safe in conducting the DAG transformation in this manner.

Join operators are not transformed in the current version of Hirundo; because Hirundo conducts transformations only along the rays of multipath DAGs. However, Join operator's state can be partitioned similar to Aggregate operators. The Tahitica mechanism described in the next chapter conducts transformation of Join operators as well.

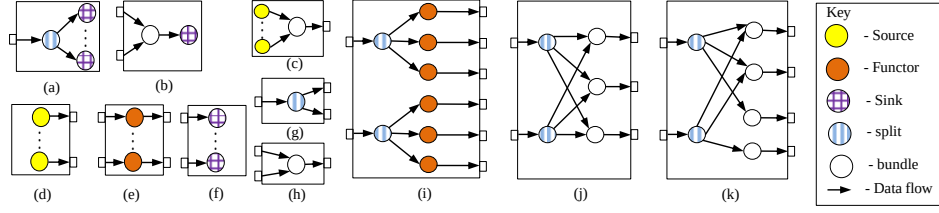


Figure 6.11: Some examples of Transformer Blocks used by Hirundo. (a) DEMUX-SINK (b) MUX-SINK (c) MUX-SOURCE (d) PARALLEL-SOURCE (e) MULTI-FUNCTOR (f) PARALLEL-SINK (g) DEMUX-STREAM (h) MUX-STREAM (i) DEMUX-FUNCTOR-N-TO-M (j) TRANSFORMER-N-TO-M (k) TRANSFORMER-MODULUS-N-TO-M

6.5 Transformer Blocks

Hirundo uses a set of generic operator blocks called *Transformer Blocks* during transformation of a data flow graph. These are introduced in between the identified operator blocks to create coupling between resulting operator blocks (transformed operator blocks) of Tri-OP transformation.

DEMUX-SINK is a transformer block that de-multiplexes its input stream into N number of sink operators (shown in (a) of Figure 6.11). The opposite of this operation, multiplexing of several streams in to a single sink operator is done by MUX-SINK (see (b) in Figure 6.11). MUX-SOURCE transformer block merges N number of source operators in to one single stream. Yet, multiple streams from N number of source operators can be obtained using PARALLEL-SOURCE transformer block. MULTI-FUNCTOR transformer block creates N number of functor blocks. Similar to PARALLEL-SOURCE, PARALLEL-SINK transformer block creates N number of sink operator blocks. A stream is converted to multiple streams using DEMUX-STREAM transformer block and MUX-STREAM merges several homogeneous streams to a single stream. DEMUX-FUNCTOR-N-TO-M transformer block creates $N \times M$ number of functor blocks accepting N input streams. TRANSFORMER-N-TO-M is a transformer block that accepts N number of input streams which can output M number of streams (see (j) of Figure 6.11). A slightly different transformer block to TRANSFORMER-N-TO-M called TRANSFORMER-MODULUS-N-TO-M accepts N input streams and outputs M streams. However, in the latter scenario, symmetry of the internal operators is not preserved. This can be observed from (k) of Figure 6.11.

Transformer blocks are created as supporting primitives for Tri-OP transformation process. They are reusable and useful when Hirundo is updated to support new operator types in future. While Tri-OP transformation algorithm concentrates on increasing/decreasing number of operator blocks in a data flow graph, transformer blocks concentrate on solving the problem of how to make links (i.e., streams) between the operators in transformed operator blocks. Transformer blocks should not be confused with similar constructs such as *Composite Operators* of IBM Stream Processing Language [234].

Furthermore, the decision of mapping output streams from split operators of transformer blocks such as TRANSFORMER-N-TO-M, TRANSFORMER-MODULUS-N-TO-M, etc. has

been taken in order to preserve the isometry of data flow graph. Isometry of a data flow graph is an important factor for highly availability of a stream application [145].

6.5.1 Preservation of Sequential Semantics

Certain stream applications deal with ordered series of information (e.g., time ordered) that need to be processed in a sequential manner. For example tuples from a weather sensor may carry a timestamp that indicates the time they get generated. However, tuple ordering is dependent on the way how the tuple information is interpreted. For example tuples in an input stream may carry a timestamp but may not necessarily require ordering based on that field. Therefore, by default Hirundo does not enforce the ordering of the tuples of the transformed segment of the graph. However, the default behavior can be overridden by adding an Order tag (`#Hirundo-meta:Order`) in the application. If Hirundo's code analyzer identifies the presence of such Order tag, it creates two UDOPs at the beginning and at the end of the parallel region. The UDOP at the beginning of the parallelized region inserts a unique sequence number at the beginning of the stream split. The UDOP at the end of the parallel region observes this sequence number and forwards the tuples in the order (If the last sent tuple's sequence number is P, the UDOP makes sure the next tuple sent will be P+1). When implementing the tuple sequencing strategy we assume that there is no tuple loss in between the two UDOPs.

6.6 Implementation

Hirundo has been implemented using Python programming language. It has been separated into two modules called Main module and Worker module. Architecture of Hirundo is shown in Figure 6.12.

Current version of Hirundo has been developed targeting stream programs written using SPADE language. Hence Hirundo depends on System S and SPADE compiler during its operations. It should be noted that, although System S is dependent on a shared file system such as NFS, Hirundo has been designed not to use such file systems for optimization runs. It uses local hard disks to store the data it handles during optimization runs. Hirundo uses a SQLite database to store its information. Main module has been separated into ten sub modules based on different functionalities they handle. Brief descriptions of the functions of important modules are given below.

6.6.1 Program Structure Analyzer

SPADE program analysis logic has been implemented in Program Structure Analyzer of Hirundo. As pointed out in Section 6.3, Hirundo uses a bespoke grammar written for parsing a SPADE program to identify its structure. Current implementation of Hirundo's grammar supports Source, Functor, Aggregate, Join, Split, Sink BIOPs, and UDOPs. This module uses an LALR parser [295] which has been developed using the GOLD parser generator created by Cook *et al.* [296]. The grammar has been coded

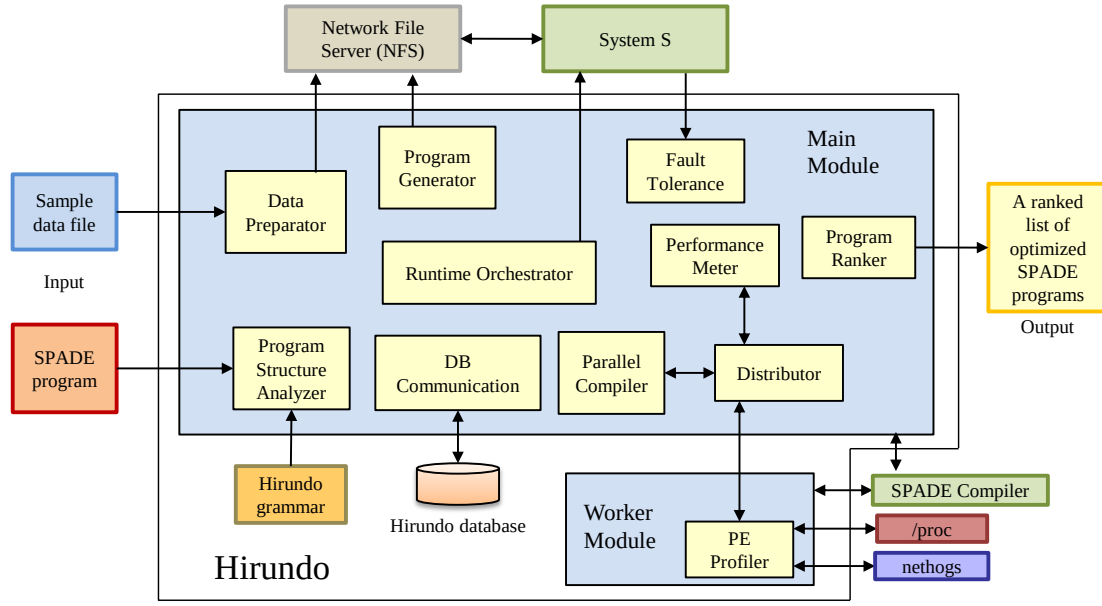


Figure 6.12: System Architecture of Hirundo.

separately from Hirundo (independent of Python programming language) and can be modified easily using the GOLD parser generator [296]. Hirundo's grammar has been written using *Backus-Naur Form (BNF)* notation which is a notation used to describe context free grammars [297]. BNF breaks down the grammar into a series of rules which are used to describe how the programming languages such as SPADE are tokenized in to different logical units [296]. As many other grammars which are written using BNF notation, Hirundo's grammar consists of two main types of symbols called *terminals* and *nonterminals*. In general terminals are delimited by single or double quotes or are left without special formatting [296]. Figure 6.13 shows some of the BNF statements that include terminals such as '[Application]', '[Program]', etc. Note that Figure 6.13 has been divided in to two parts using a vertical line to improve readability.

<pre> MULTI-STREAM = 'stream'{Whitespace}{AlphaNumeric}+'_@i' 'stream'{Whitespace}{AlphaNumeric}+'%@i' STREAM = 'stream'{Whitespace}{AlphaNumeric}+ {PR} = {Printable} - [] - {} + {CR} + {LF} {COMC} = {Printable} - {CR} - {LF} - [#] PREAMBLE = '[Application]'{PR}+'[Program]' BODY = '{PR}+' NUMBER = {Number}+ </pre>	<pre> {VARCHAR} = {AlphaNumeric} + [_]+[@]+[%] VARNAME = {Letter}{VARCHAR}+ PL_LOCAL_VAR = 'my' FILEPATH = ""file://{AlphaNumeric}+'.'{Letter}+"" ""file:/// {AlphaNumeric}+'.'{Letter}+"" ""file:/// {Printable}+'.'{Letter}+"" </pre>
--	--

Figure 6.13: A sample set of expressions of Hirundo's grammar with terminals.

On the other hand nonterminals are structure names and are typically delimited by angle-brackets (See Figure 6.14). A nonterminal is defined by a series of one or more rules which are also known as productions. In BNF, a rule is denoted with a

nonterminal and the rule statement consists of a series of zero or more terminals and nonterminals (i.e., series of productions). To improve readability often the productions are grouped together and separated by a “pipe” symbol (There are 18 pipes in the rule shown in Figure 6.14). Current version of the grammar consists of 38 rules. The rule that defines the structure of a program is shown in Figure 6.14. Each production on Figure 6.14 starts with `<PREAMBLE>` that represent information such as application meta-information (`[Application]`), type definitions (`[Typedefs]`), external libraries (`[Libdefs]`), and node pools (`[Nodepools]`).

```

<Program> ::= <PREAMBLE><SOURCE><FUNCTOR><SINK>
|<PREAMBLE><SOURCE><FUNCTOR><MUX-SINK>
|<PREAMBLE><AUTO-BUNDLE-SINGLE><PARALLEL-SOURCES-CSV><MUX-FUNCTOR-N-TO-M><DEMUX-STRM-M-N-TO-1>
  <SINK>
|<PREAMBLE><AUTO-BUNDLE-MULTI><PARALLEL-SOURCES-CSV><MULTI-FUNCTOR><TRANSFORMER-N-TO-M>
  <MUX-SINK-WITH-STRM-NAME>
|<PREAMBLE><PARALLEL-SOURCES-CSV><MULTI-FUNCTOR><PARALLEL-SINK>
|<PREAMBLE><AUTO-BUNDLE-SINGLE><PARALLEL-SOURCES-CSV><MULTI-FUNCTOR><DEMUX-SINK><SINK>
|<PREAMBLE><AUTO-BUNDLE-MULTI><SOURCE><MUX-SOURCE-CSV><MULTI-FUNCTOR><TRANSFORMER-N-TO-M>
  <PARALLEL-SINK>
|<PREAMBLE><SOURCE><MUX-SOURCE-CSV><FUNCTOR><AGGREGATE><FUNCTOR><AGGREGATE><FUNCTOR>
  <AGGREGATE><FUNCTOR><JOIN><JOIN><SINK>
|<PREAMBLE><SOURCE><MUX-SOURCE-CSV><MULTI-FUNCTOR><TRANSFORMER-N-TO-M><PARALLEL-SINK>
|<PREAMBLE><SOURCE><FOR-START><FUNCTOR><FUNCTOR><AGGREGATE><FUNCTOR><JOIN><FUNCTOR><SINK>
  <FOR-END>
|<PREAMBLE><AUTO-BUNDLE-SINGLE><SOURCE><MUX-SOURCE-CSV><MULTI-FUNCTOR><DEMUX-SINK><SINK>
|<PREAMBLE><AUTO-BUNDLE-SINGLE><DEMUX-SOURCE-CSV><FUNCTOR><MUX-SINK>
|<PREAMBLE><AUTO-BUNDLE-MULTI><PARALLEL-SOURCES-CSV><MUX-FUNCTOR-N-TO-M><TRANSFORMERMODULUS>
  <MUX-SINK-WITH-STRM-NAME>
|<PREAMBLE><SOURCE><STRING-OF-FUNCTOR><SINK>
|<PREAMBLE><SOURCE><AGGREGATE><SINK>
|<PREAMBLE><SOURCE><STRING-OF-FUNCTOR><AGGREGATE><STRING-OF-FUNCTOR><SINK>
|<PREAMBLE><SOURCE><META><AGGREGATE><META><SINK>
|<PREAMBLE><SOURCE><STRING-OF-FUNCTOR><META><AGGREGATE><STRING-OF-FUNCTOR><META><SINK>
|<PREAMBLE><META><UDOP><META><STRING-OF-FUNCTOR><META><AGGREGATE><META><SINK>

```

Figure 6.14: A sample rule from Hirundo’s grammar.

The first production on Figure 6.14 represents a most basic stream application that can be written using SPADE which consists of three operators: Source, Functor, and Sink. Note that there are operator blocks defined for collections of Functors (e.g., `<STRING-OF-FUNCTOR>`) which act as abstractions for lower level operator blocks like Functors. Figure 6.15 shows a concrete example of how the rules in the BNF gets mapped to the data flow graph of the stream application (CDR) shown in Figure 3.1. The operator graph shown on the right-hand side is mapped with the corresponding productions of the Hirundo’s grammar. To maintain brevity, we have omitted the rules that define other operator block categories (e.g., Join) as well as terminal definitions from Figure 6.15. Furthermore, Figure 6.15 shows only the production related to the CDR application with the rule that defines a program.

Program structure analyzer creates a representative graph G for the program it analyses if it can identify its structure. This graph keeps details of all the operators (vertices) identified from the program and the links between them.

<Program> ::= <PREAMBLE><SOURCE><META><UDOP><META><STRING-OF-FUNCTOR><AGGREGATE><SINK>

<SOURCE> ::=

```
STREAM('<SCHEM>') := "Source()"["FILEPATH","nodelays","csvformat"]{}'
| STREAM('<SCHEM>') := "Source()"["FILEPATH","nodelays","csvformat"]{}'
| STREAM('<SCHEM>') := "Source()"["FILEPATH","nodelays","csvformat"]{}'
| STREAM('<SCHEM>') := "Source()"["FILEPATH"]{}'
| STREAM('<SCHEM>') := "Source()"["FILEPATH"]{}'
| MULTI-STREAM('<SCHEM>') := "Source()"["FILEPATH","nodelays","csvformat"]{}'
| MULTI-STREAM('<SCHEM>') := "Source()"["FILEPATH","nodelays","csvformat"]{}'
```

<META> ::= #Hirundo-meta:UnFroze:Start
#Hirundo-meta:UnFroze:End

```
<UDOP> ::= STREAM('<SCHEM>') := "Udop()"[""]{}'
| STREAM('<SCHEM>') := "Udop()"("VARNAME")["""VARNAME"""]{}'
| STREAM('<SCHEM>') := "Udop()"("VARNAME")["""VARNAME"""]{}'
| STREAM('<SCHEM>') := "Udop()"("port_n-streams")["VARNAME"]{}'
```

<STRING-OF-FUNCTOR> ::= <STRING-OF-FUNCTOR><FUNCTOR>
<FUNCTOR>

```
<FUNCTOR> ::= STREAM('<SCHEM>') := "Functor()"("VARNAME")[""]{}'
| STREAM('<SCHEM>') := "Functor()"("VARNAME")["<FILTER-PREDICATE>"]{}'
| STREAM('<SCHEM>') := "Functor()"("VARNAME")["<FILTER-PREDICATE-EXPR>"]{}'
| STREAM('<SCHEM>') := "Functor()"(auto_bundle[:])[""]{}'
| MULTI-STREAM('<SCHEM>') := "Functor()"(auto_split@i)[""]{}'
| MULTI-STREAM('<SCHEM>') := "Functor()"("VARNAME'_@i")[""]{}'
| MULTI-STREAM('<SCHEM>') := "Functor()"("VARNAME'%@i")["<FILTER-PREDICATE-EXPR>"]{}'
| MULTI-STREAM('<SCHEM>') := "Functor()"("VARNAME")[""]{}'
| MULTI-STREAM('<SCHEM>') := "Functor()"("VARNAME")["<FILTER-PREDICATE-EXPR>"]{}'
```

```
<AGGREGATE> ::= STREAM('<SCHEM>') := "Aggregate()"("<WINDOWING-SCHEME>")[""]{}'
| STREAM('<SCHEM>') := "Aggregate()"("<WINDOWING-SCHEME>")["VARNAME"]{}'
| MULTI-STREAM('<SCHEM>') := "Aggregate()"("<WINDOWING-SCHEME>")["VARNAME"]{}'
```

```
<SINK> ::= 'Nil' := "Sink()"("VARNAME")["FILEPATH"]{}'
| 'Nil' := "Sink()"("VARNAME"@i)["FILEPATH"]{}'
| 'Nil' := "Sink()"("VARNAME")["FILEPATH"]{}'
| 'Nil' := "Sink()"("VARNAME'_@i")["FILEPATH"]{}'
| 'Nil' := "Sink()"("VARNAME'_@i")["FILEPATH"]{}'
| 'Nil' := "Sink()"(auto_bundle[:])["FILEPATH"]{}'
| 'Nil' := "Sink()"(auto_bundle[:])["FILEPATH"]{}'
| 'Nil' := "Sink()"(auto_bundle_@i[:])["FILEPATH"]{}'
| 'Nil' := "Sink()"(auto_bundle_@i[:])["FILEPATH"]{}'
| 'Nil' := "Sink()"("VARNAME")["FILEPATH","nodelays"]{}'
| 'Nil' := "Sink()"("VARNAME")["FILEPATH","nodelays"]{}'
| 'Nil' := "Sink()"("VARNAME")["FILEPATH","nodelays","csvformat"]{}'
| 'Nil' := "Sink()"("VARNAME")["FILEPATH","nodelays","csvformat"]{}'
```

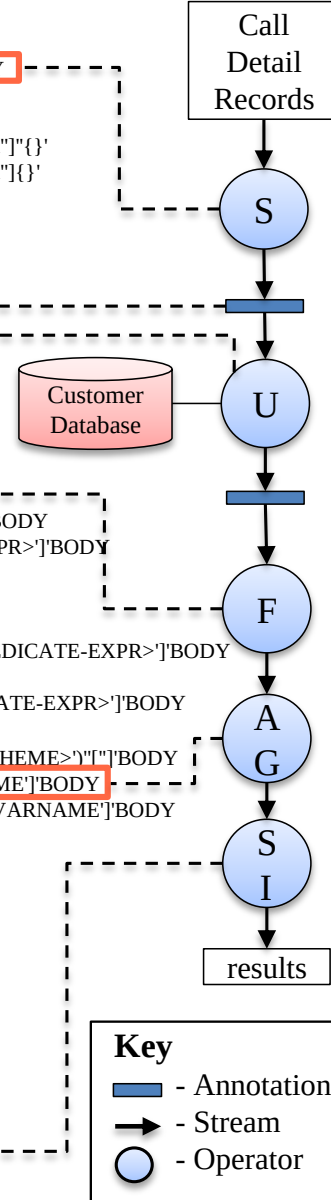


Figure 6.15: An example of how the rules get mapped to the data flow graph of CDR application. The complete source code of CDR application is given in Figure 3.1.

6.6.2 Data Preparator

Unlike most static program optimizers, Hirundo runs selected sample programs for fixed time period in the stream processing environment to identify their performance. The sample programs require representative data during their execution. Data Preparator is the module that creates these required sample data. At the beginning of optimization process, user should provide a sample data file along with input program. This file is splitted in to total $\lfloor d(d+1)/2 - 1 \rfloor$ number of files starting from the groups 2, 3, up to d. The number d is the depth value that is accepted by Tri-OP transformation algorithm. The files are splitted in this way since Tri-OP transformation with depth d may produce d number of maximum transformed source operator blocks for any optimization run. Furthermore, such splitting ensures that all the input data are received by all the source operators of the sample programs homogeneously. User should enter only required size input data file (smaller size file is preferred) to Hirundo since very large files impose unnecessary overhead on Hirundo during its optimization runs. Furthermore, Hirundo uses a file rather than online data source in order to keep the input fixed during all the optimization attempts the input program faces since this could directly affect the end result of program ranking.

6.6.3 Parallel Compiler

Hirundo would have spent substantial portion of its processing time on compilation, if Hirundo were to compile all the sample programs on a single node. E.g., Compiling VWAP (25 programs) and Twitter (27 programs) applications described in this chapter on a single node took approximately 11, 50 minutes respectively. To reduce total time taken for compilation we created a parallel compiler for Hirundo. Current version of Hirundo deploys one worker per each node of the node cluster of which it operates on during its instantiation. Parallel compiler assigns compilation task of a single program to one worker. This technique reduced the percentage of time required for compilation of VWAP and Twitter applications to 5 and 8 minutes resulting 51% and 84% compilation time reduction respectively on 8 nodes of the same cluster.

6.7 Fault Tolerance of Hirundo

Compared to many related work mentioned in Section 2.3 of this thesis, Hirundo emphasizes the importance of fault tolerance during automatic program optimization process. Hirundo strives to eliminate instance failures that might occur in the stream processing environment. An instance failure is just failure of a run-time instance (i.e., a process spawned by stream processing environment). Unexpected failures may occur in stream processing environment when such automatic optimization process has been conducted. While the stream processing run-time could continue with the remaining set of instances, it may not reflect the actual performance that could have been achieved by using a sample program. Ultimately this may lead to an inaccurate ranking of sample programs.

Note that there are no such recording made during the experiments mentioned in this thesis since all the faults were successfully resolved by Hirundo.

It was observed that certain large sample programs overload the run-time instances, and they might run out of memory creating instance failures. Finding the set of sample programs that provide highest performance without breaking the stream processing run-time environment’s stability is a challenging issue. Hirundo uses `streamtool` of System S periodically to obtain the health information of the System S runtime, and compares those information with the runtime snapshot (original health record) obtained at the beginning of the optimization run to detect failures. (Note that at the very beginning of the optimization run, Hirundo displays the original health record to user and gets it confirmed free of faults). If found a failure, Hirundo tries to restart the runtime and compares the health of the newly started runtime with the original health record. If the status of the runtime was restored, it starts running the interrupted job (sample program). It follows the same procedure during three consecutive failures. If it cannot succeed, it marks the sample program as a failure (by recording throughput as -1) and continues the optimization run.

6.8 Sample Program Space Reduction and Ranking

Hirundo’s data flow graph transformation algorithm generates many sample programs for a given input program. Number of sample programs generated per input application for three transformation depth values is shown in Table 6.1. It was observed that out of the five sample program spaces three were growing in an order between d^2 and d^3 while VWAP grew in the orders between d and d^2 .

Transformation depth (d)	Input Application				
	CDR	VWAP	Twitter	Apnoea	Bargain
4	16	8	25	12	12
8	240	24	124	59	59
16	2480	80	728	360	360

Table 6.1: Sample program space sizes.

Twenty four sample programs are generated for VWAP application during an optimization run with $d = 8$. Running all these sample programs for small time period may take time in the order of minutes. E.g., Running all the aforementioned 24 sample programs (+input program) in a System S environment with 8 nodes (cluster Y in Section 6.9.1) took 3 minutes and 22 seconds (an average value calculated over 5 optimization runs). In the case of multipath DAGs the number of sample programs become even greater because of high availability of various operator combinations for input program transformation. To reduce the time taken to find optimized programs we follow two

approaches. First, in the case of multipath DAGs we use operator profile information to identify a particular path in the data flow graph to be transformed. This essentially reduces the number of permutations of the sample programs that could be made by our code generator for multipath DAGs. Second, we employ a performance prediction technique to identify sample programs that could deliver higher performance, and only use them to test in the target environment. Current Hirundo prototype supports performance prediction only on linear DAGs.

Moreover, Hirundo removes some extreme mutations present on the generated sample program set for multipath DAGs using a mode based elimination technique. Put in simple terms, we count the mode (M) of the cardinality values of operator blocks of a generated sample program. Then we check if the program contains any operator blocks with cardinality greater than M. If so, we discard that sample program. For example, the sample program `SCSV_F_AG_F_4F_16AG_4F_F_AG_F_J_J_SI` generated for Apnoea is such application discarded by mode based elimination.

As mentioned before it was observed that performance of a stream program in a certain environment can be repeated. Hence we can predict up to a certain level, what kind of performance could be obtained from a stream program using empirical data. We have implemented an algorithm for linear DAGs (The Algorithms 9 and 10 correspond to this prediction process.) in Hirundo that predicts similarity of optimization runs considering the parameters of structure of data flow graph (G), performance metrics used (e.g., throughput, elapsed time), optimization run depth (d), input data tuple schema (tschema). Hirundo uses a relational database to store its information. Current optimization environment's profile information such as number of hosts, CPU, RAM capacity are stored in the database prior to any optimization attempt. All the important optimization run information (i.e., optimization session information) such as start time, end time, performance metrics used are recorded in the database. Furthermore, performance information (i.e., throughput, elapsed time, etc.) of each sample program which ran during the optimization session are also stored in this database.

6.8.1 Performance Prediction Mode

In this mode of operation Hirundo predicts what kind of performance could be obtained from the input program. Note, that in current Hirundo prototype this mode of operation is only applicable for linear DAGs.

Algorithm 9 first selects a list of optimization runs based on the optimization metric used and the input data tuple schema. Next, it sorts the list based on the transformation depth and the structure of the input data flow graph (i.e., A,B,C values of the graph A_B_C). Finally, the optimization runs are sorted based on node performance values and top m optimization runs are selected as matching optimization runs. Next, these optimization run ids are fed to the `getprogramlabels()` procedure shown in Algorithm 10 to obtain sample program labels to run. For each optimization run, the performance differences of sample programs are gathered and stored in a dictionary called `alldict`. The items in the dictionary are sorted based on their performance difference values in

Algorithm 9 getmatchingoptruns(G , nodePerf, metrics, d , m , tschema)
<pre> 1: optrunList $\leftarrow$ getOptruns(metrics, tschema) 2: optrunList $\leftarrow$ sortUsingTransformationDepth(optrunList, d) 3: optrunList $\leftarrow$ sortUsingMatchingIndices(G, optrunList) 4: optrunList $\leftarrow$ sortUsingMatchingNodes(nodePerf, optrunList) 5: optrunList $\leftarrow$ selectTopN(optrunList, m) 6: return optrunList </pre>
Algorithm 10 getprogramlabels(optrunList, n)
<pre> 1: inproglabels $\leftarrow$ getinputproglabels() 2: alldict[label, avgdifflist] $\leftarrow$ emptyDictionary 3: for all optrunid in optrunList do 4: labelslist $\leftarrow$ getSampleProgLabels(optrunid) 5: filtlist $\leftarrow$ removeInputProgLabels(labelslist, inproglabels) 6: labeldict[label, perfvalue] $\leftarrow$ getperfvals(optrunid, filtlist) 7: perfdict[label, avgdiff] $\leftarrow$ getAvgPerfvalDiffs(labeldict) 8: alldict.append(perfdict) 9: end for 10: rlabels $\leftarrow$ sortUsingAveragePerfDiffAsc(alldict) 11: result_labels $\leftarrow$ selectTopN(rlabels, n) 12: return result_labels </pre>

ascending order. The top n labels of this sorted list are selected as the candidate labels. These n labels correspond to half of the U_1 subset of the sample program labels mentioned in Section 6.3 (Methodology Section), and the remaining steps of the Methodology Section are followed to obtain a ranked list of sample programs (R_2) as the output. Such method of two phase ranking was used in order to increase the accuracy of the end result. Section 6.9 demonstrates results obtained by operating Hirundo in this mode.

6.9 Evaluation

6.9.1 Experiment Setup and Methodology

We ran the experiments on two clusters (From here onwards Cluster X and Cluster Y). Hardware specifications of each cluster is given on Table 6.2. Cluster X was highly provisioned compared to cluster Y as can be observed from the Table 6.2. We used two clusters because we wanted to prove that our mechanism creates optimized sample applications even in different hardware environments. Cluster X ran Linux Cent OS release 5.6 (Linux kernel 2.6.18-238) installed with IBM Infosphere Streams version 1.2 and Python 2.7.1. All the nodes were connected through 1 Gigabit Ethernet. Cluster Y had all the software similar to Cluster X except it ran Linux Cent OS release 5.4 (Linux kernel 2.6.18-164). Since both the OSs had the same major release number we believe that there was no significant effect on the evaluation results. Both Clusters X and Y were reserved only for running these experiments during the optimization sessions.

As mentioned in the Section 6.4.3 we obtain operator profile statistics of input ap-

	Cluster X	Cluster Y
CPU	Intel Xeon X5670 @ 1600MHz, 24 cores	AMD Phenom 9850, 1250MHz, 4 cores
L2 Cache	12MB	512KB
RAM (GB)	47	8
Hard Disk (GB)	280	160
Network	1-Gigabit Ethernet	1-Gigabit Ethernet

Table 6.2: Hardware specifications of experiment environments.

plication to identify expensive paths in the DAG. During the input application profiling step we set the reservoir size (-P) to 1000 (this means 1000 samples will be collected) and sampling fraction (-S) to 0.1 (this means one out of 10 tuples will be used to collect performance data).

We used sample files of sizes 241.3MB for CDR, 181MB for Twitter, 97MB for VWAP and Bargain, and 81MB for Apnoea for all the experiments. In the first half of the experiments we disabled the sample program ranking of Hirundo, and made it to run all the sample programs it generated for a short period of time (each sample program less than 1 minute). In the second half we enabled the sample program ranking of Hirundo. In all of these experiments Hirundo ensured that the experiments are not affected by sudden instance failures by re-instantiating the run-time environment and re-running the sample program that faced an instance failure. Furthermore, the experiments were conducted on two types of clusters. VWAP, Twitter and CDR applications were optimized in both 4 nodes and 8 node environments. Because we wanted to observe how different hardware configurations affect the results of the optimization process. Note that all the results shown in the result graphs are single runs (Opt run) of the experiment since we wanted to avoid the effect of large deviations for the end result. Furthermore, we use only three optimization runs in the graphs because three optimization runs are sufficient to come into a conclusion about a sample stream program’s performance.

6.9.2 Sample Program Naming Convention

Unique labeling format is used to represent each sample program generated for a particular application. This section explains the labeling convention to improve the understanding of the readers about the experiment results presented in the later half of this chapter.

Each label represents the operator blocks in the sample program along with their new cardinality values created because of code transformation. This thesis uses the letter combinations SCSV (Source which accepts tuples of CSV format), F (Functor),

AG (Aggregate), U (UDOP), SI (Sink), J (Join) to denote SPADE operators. For linear DAGs (such as CDR, Twitter, VWAP, etc.) the ordering of the letter combinations on the label represents the relative ordering of the operators on the DAG as well as the ordering of the operators on the SPADE file. For example, Twitter sample application label U_4F_2F_F_AG_SI means the DAG’s root is a UDOP, then there are three Functors, an Aggregate operator, and a Sink at the leaf. Furthermore, the non-zero integers in front of each letter combination represents the cardinality of the operator block. For the above mentioned sample application, 4F means there are four F1 operator blocks and 2F means there are two F2 operator blocks. Label SCSV_4F_8F_4F_4SI means the sample program has one source, four F1 functors, eight F2 functors, four F3 functors, and four sinks.

In the case of multipath DAGs such as Apnoea and Bargain, the label does not indicate network structure of the DAG. But the label indicates the sequential order of the appearance of the operators on the SPADE file and serves as a unique ID. Furthermore, the label indicates each operator block’s cardinality which could be interpreted in the same way as explained above.

6.9.3 Sample Program Performance

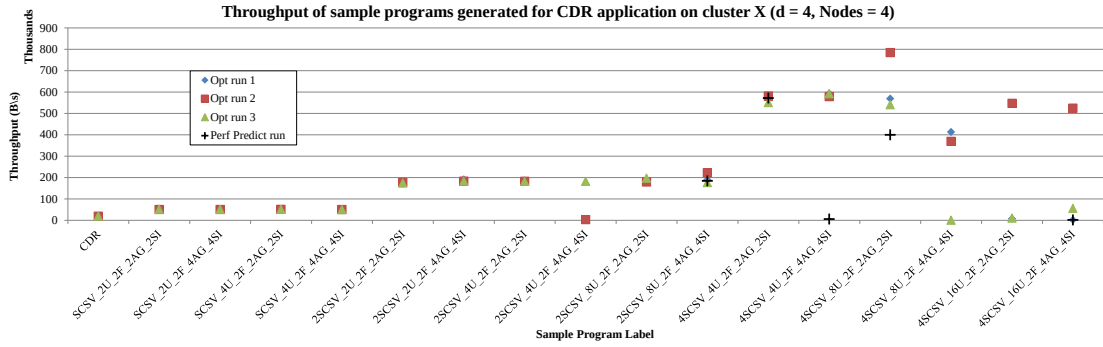
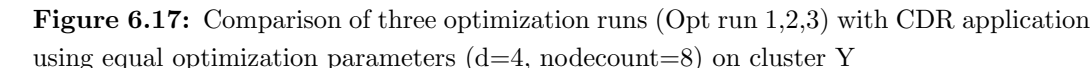


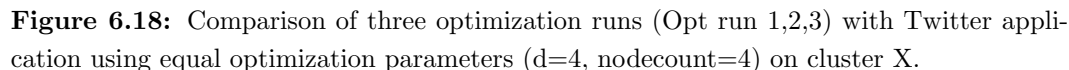
Figure 6.16: Comparison of three optimization runs (Opt run 1,2,3) with CDR application using equal optimization parameters ($d=4$, $nodecount=4$) on cluster X

The main parameter that determined the sample programs present in this section is the transformation depth (d). As mentioned in Section 6.4.3.1, transformation depth was the parameter that determined the amount of parallelism present in sample programs. The X axis of the graphs lists the sample program labels sorted by the number of operators present on the program label from left to right. We used throughput of sample applications as the metric during the evaluations.

The results obtained by running the CDR application in cluster X is shown in Figure 6.16. It can be observed that eleven sample programs out of sixteen provided consistent high performance compared to the input application. The use of term “consistent” here means that there is no significant deviation of the performance results obtained during the three optimization runs. Furthermore, two applications (4SCSV_4U_2F_2AG_2SI and



The results of optimizing the CDR application on cluster Y is shown in Figure 6.17. A direct observation is that the input application as well as all the sample applications produced less throughput performance compared to cluster X because cluster Y is less provisioned. The best performance application on both cluster X and Y was 4SCSV_4U_2F_4AG_4SI which had 1.34 times higher performance on cluster X compared to cluster Y. In the experiment with cluster Y Hirundo was able to identify nine sample programs out of sixteen with optimized performance compared to the CDR application. However, the number of sample programs which provide consistent, high performance compared to the input application is less in the case of cluster Y. Similar to the experiment with 4 nodes, the sample applications 4SCSV_4U_2F_2AG_2SI and 4SCSV_4U_2F_4AG_4SI had the highest consistent performance.



92

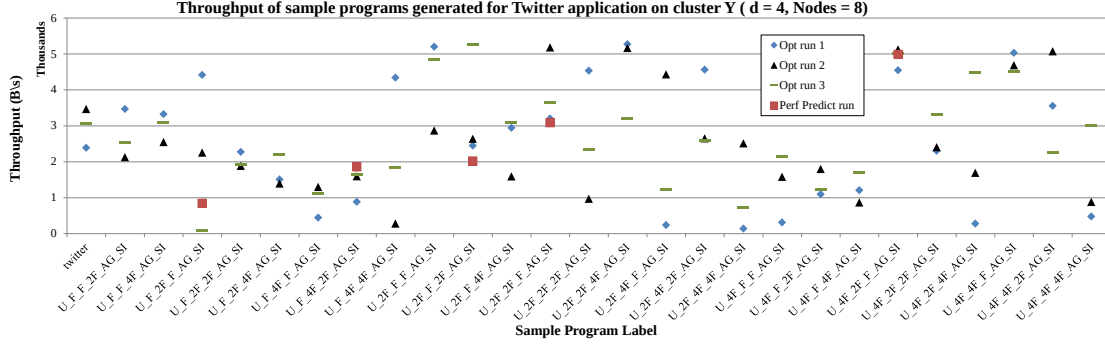


Figure 6.19: Comparison of three optimization runs (Opt run 1,2,3) with Twitter application using equal optimization parameters ($d=4$, $nodecount=8$) on cluster Y.

U_2F_2F_2F_AG_SI, U_4F_2F_F_AG_SI, U_4F_4F_2F_AG_SI, U_4F_4F_4F_AG_SI ran in cluster X and sample programs U_4F_2F_F_AG_SI, U_4F_4F_F_AG_SI, U_2F_2F_4F_AG_SI ran in cluster Y produced higher performance compared to the input application. The sample program U_4F_2F_F_AG_SI produced consistent high performance (even though it showed 1.13 times performance gain in cluster X compared to cluster Y) in both the clusters. Highest performing applications for both CDR and Twitter had the highest performance for both clusters X and Y; which indicates that there can be sample code transformations that could produce higher performance across different hardware configurations.

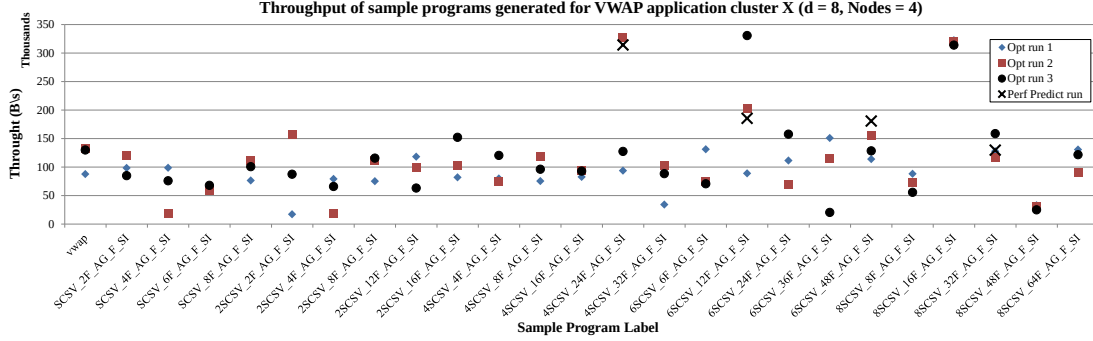


Figure 6.20: Comparison of three optimization runs (Opt run 1,2,3) with VWAP application using equal optimization parameters ($d=8$, $nodecount=4$) on cluster X.

The results of optimizing the VWAP application on cluster X is shown in Figure 6.20. Unlike other optimization runs of this thesis we use a transformation depth of 8 for VWAP application because a transformation depth of 4 produced only 8 sample applications. We wanted to observe performance across much larger sample program space. Sample applications 8SCSV_16F_AG_F_SI, 6SCSV_48F_AG_F_SI had higher performance compared to the input application. When the VWAP application was run in the cluster Y we observed that sample applications 6SCSV_48F_AG_F_SI, 4SCSV_24F_AG

_F_SI and 2SCSV_2F_AG_F_SI produced consistent higher performance compared to the input application.

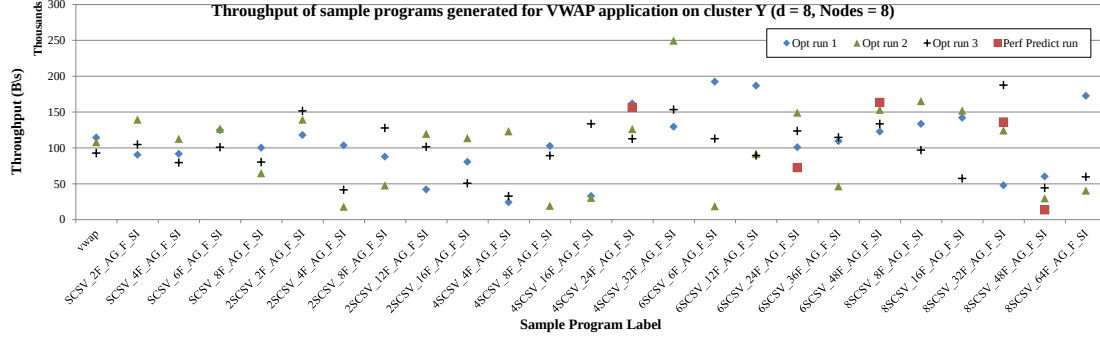


Figure 6.21: Comparison of three optimization runs (Opt run 1,2,3) with VVAP application using equal optimization parameters ($d=8$, nodecount=8) on cluster Y.

Next, we optimized Apnoea and Bargain using Hirundo. Note that we used three portfolio items (IBM, HPQ, and MSFT) during the experiments with Bargain application. For Apnoea (See Figure 6.22), the sample program SCSV_F_AG_F_2F_2AG_4F_F_AG_F_J_J_SI produced higher performance compared to the original Apnoea application. In the case of Bargain application, all the sample programs produced higher performance. SCSV_FOR_S_4F_8AG_4F_F_J_F_SIFOR_E and SCSV_FOR_S_4F_16AG_2F_F_J_F_SIFOR_E produced slightly lesser performance compared to other sample applications (See Figure 6.23).

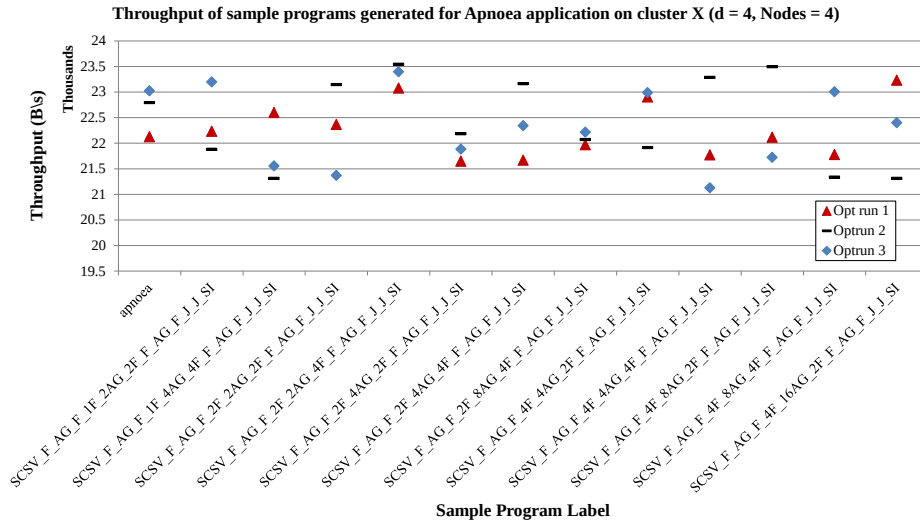


Figure 6.22: Comparison of three optimization runs (Opt run 1,2,3) with Apnoea application using equal optimization parameters ($d=4$, nodecount=4) on cluster X.

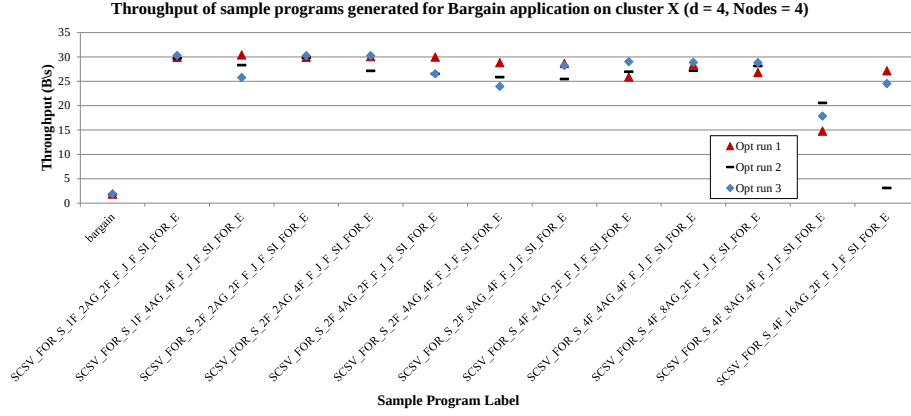


Figure 6.23: Comparison of three optimization runs (Opt run 1,2,3) with Bargain application using equal optimization parameters ($d=4$, $nodecount=4$) on cluster X.

Note that there is virtually no difference of the experiment settings between the three optimization runs. The input data file(s) for each sample application remains the same during three optimization runs and results from previous optimization runs have been used only for selecting sample applications in performance prediction runs. Although these optimal measures have been taken, Hirundo reports that certain stream applications might produce largely different performance figures across different optimization runs because System S sometimes produces different behavior across different optimization runs for each application. This is quite natural and observed behavior in distributed systems that is out of the control of Hirundo’s optimization mechanism. However, we have taken every measure to make sure that the measurements that we make are correct in the Hirundo’s implementation.

6.9.4 Performance Prediction

As described previously, Hirundo’s performance prediction mechanism allows it to reduce the time taken to conduct optimization runs for stream programs with linear DAGs. In this section we demonstrate how the performance prediction mechanism work by using VWAP, CDR, and Twitter applications. Note that the Hirundo database we used during the performance prediction had optimization runs of the previous optimization sessions.

We enabled the performance prediction and sample program ranking feature of Hirundo and ran the experiments with VWAP, CDR, and Twitter applications. In this mode each experiment completed in less than 15 minutes time for cluster Y. This is a 50% reduction of total experiment time compared to without use of performance prediction. The results are shown in the corresponding graphs on Figures 6.16, 6.17, 6.18, 6.19, 6.20, and 6.21. It should be also noted that the total time taken for a performance optimization run completely depends on the capacity of the node cluster. For example, the average experiment time for optimization runs for Twitter application was

about 5 minutes on cluster X. This is about 1.6 times faster compared to the average experiment time of Twitter on cluster Y which requires about 8 minutes. Therefore, we believe that with powerful hardware becoming increasingly prevalent, significance of such performance predictions becomes less.

On cluster X, all the sample programs pointed by our sample program ranking algorithm for VWAP and Twitter showed higher performance compared to their corresponding input applications. On cluster Y, two out of five sample programs had higher performance compared to CDR application. In the case of Twitter application one sample program (U_4F_2F_F_AG_SI) predicted by Hirundo had higher performance compared to original Twitter application.

Input Application	Number of sample applications selected from performance prediction which were also presented in the results of non-predictive runs		Applications with higher performance compared to input application	
	X	Y	X	Y
CDR	2/5	2/5	3/5	4/5
VWAP	2/4	2/5	5/5	2/5
Twitter	3/5	1/5	5/5	2/5

Table 6.3: Summary of performance prediction runs.

A summary of the performance prediction run results is shown in Table 6.3. The second major column shows the number of sample applications selected from performance prediction which were also presented in the results of non-predictive runs (i.e., It compares results of performance prediction results to the optimization that do not use performance prediction). The third major column shows number of predicted sample applications that actually had higher performance compared to the input application.

6.9.5 Performance Clusters

We conducted a cluster analysis on the data sets corresponding to Opt run 1, 2, and 3 of each performance curves of Figures 6.17, 6.19, and 6.21. We clustered throughput values of each sample program recorded during optimization runs. Our intention was to find the characteristics of data flow graphs which lead to higher performance. We used

K-Means clustering [298] for this purpose since we needed to group the data points based on their performance values. The data sets were of convex shape. We set the minimum gap between the clusters as 100B/s. The algorithm was implemented using Python and Scipy [299]. The results are shown in Figures 6.24, 6.25, and 6.26. We calculated average performance values of each clusters, and also recorded program labels if all the three data points corresponding to the three optimization runs fall into a particular cluster.

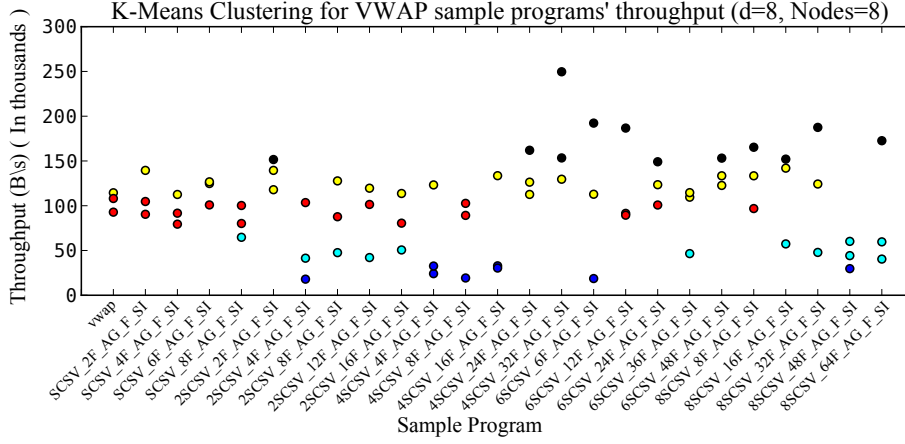
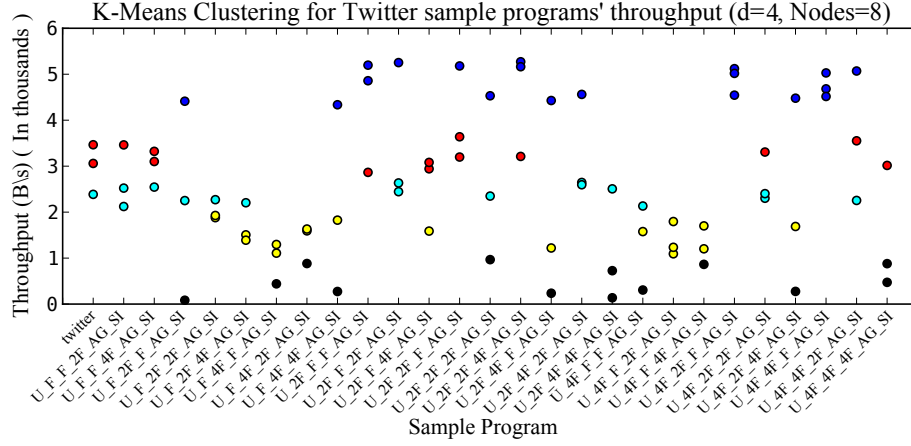


Figure 6.24: Results of applying K-Means clustering for VWAP sample program performance data.

VWAP data set resulted in 5 clusters. We saw that the cluster with the highest performance (172,947B/s) had two data points corresponding to label 4SCSV_32F_AG_F_SI whereas the second highest cluster (124,082B/s) had two data points corresponding to the labels 4SCSV_24F_AG_F_SI, 6SCSV_36F_AG_F_SI, 6SCSV_48F_AG_F_SI. However, two low performance clusters had all the data points corresponding to the labels SCSV_8F_AG_F_SI and SCSV_2F_AG_F_SI. Moreover, the high performance cluster (black color) is concentrated on the right-hand side of Figure 6.24 whereas medium performance (red color) is concentrated on the left-hand side. When comparing the labels of the sample applications in black cluster with red cluster, it becomes clear that black color cluster's sample applications have more source operators compared to red color cluster. Aforementioned observations indicate that having large number of Source operators has produced high throughput in the case of VWAP application described in this chapter. Furthermore, it indicates that having higher operator density at the beginning of the data flow graph tend to produce higher performance.

The clustering results for Twitter application is somewhat different than the VWAP because Twitter application's UDOP was frozen during program transformation. The highest performing cluster (4824B/s) had all data points corresponding to the label U_4F_2F_F_AG_SI and U_4F_4F_F_AG_SI whereas a medium performance cluster (2nd cluster from the lowest end, having 1514B/s performance) had all the data points corresponding to label U_4F_F_2F_AG_SI. The difference between these programs is that



gram sets it is clear that having higher number of source operators has produced high performance in the case of CDR application. The lowest performing cluster (6722B/s) had all three data points from 4SCSV_16U_2F_4AG_4SI. While this sample program has similar number of source operators to highest performing cluster, large number of UDOPs and relatively larger operator density at the latter part of the flow graph has produced lower performance.

Clustering of the sample applications in this chapter were conducted solely using throughput performance of the sample applications. However, there can be multiple other features that may result in higher performance (E.g., Network usage, memory usage, etc.). We plan to conduct much higher resolution clustering of such features in future to observe important performance characteristics.

6.9.6 Comparison with Operator Fusion

There are three popular techniques for stream program performance optimization at the level of query plan; operator parallelization, pipelining, and fusion. The Tri-OP and Bi-OP transformations used in Hirundo’s program transformer module is based on operator parallelization (i.e., *Parallel Streams* design pattern [233]/Fission [226]). Pipelining is another pattern that segments operator functionality into multiple pipelined stages. Since stream applications usually created by combining small, granular operators they are well divided into stages. All the BIOPs of SPADE language have such characteristics (See Figure 6.27 (a) for an example.). However, if a user includes significant amount of user-defined logic in a single operator (E.g., UDOP) that could act as a bottleneck in the data flow graph. Such user-defined logic can be partitioned into number of lesser overhead operators by following the Pipeline pattern. Pipelining has a larger communication overhead compared to parallelism because in the case of pipelining every tuple is sent to every operator while in parallelization only portion of the tuples need to be sent to parallelized operators. Nonetheless, we consider pipelining as further work for the automated performance optimization techniques presented here. Third approach, fusion achieves optimized performance by grouping several operators into a single process which eliminates communication overhead associated with other approaches. In SPADE applications this is achieved by making the stream connections between the operators within the same PE container reduced to function calls [169]. Note that in current Hirundo’s program transformer, we fuse each individual operator to a separate PE. This section compares our automatic code transformation based optimization technique with operator fusion.

SPADE language provides the key word *partition* which allows for specifying the PE to which an operator should be bounded. Figure 6.27 (a) shows a code snippet of the Apnoea application with operators partitioned to different PEs by using the partition key word. An example operator partitioning of Apnoea is shown in Figure 6.27 (b). We modified Hirundo to use operator fusion to generate sample applications. We used the same Hirundo mechanism for evaluating the alternative approach because it allows us to utilize the same sample application performance measurement infrastructure available

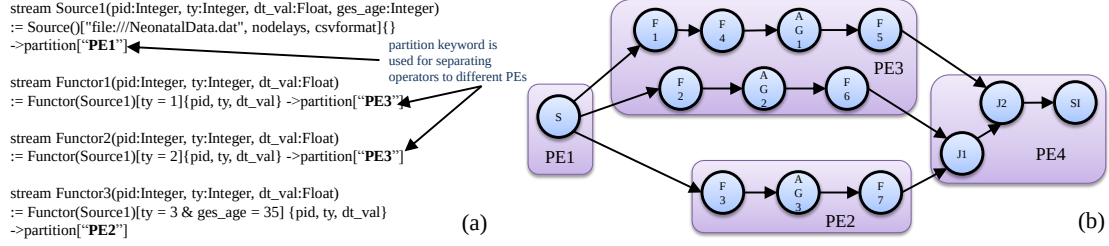


Figure 6.27: An example operator PE partitioning scheme for Apnoea data flow graph.

with Hirundo, which leads for a fair comparison of our code generation algorithm with operator fusion. These applications did not use any code transformations proposed in this chapter, instead had operators grouped into different PE combinations. We employed a set partitioning scheme based on a restricted growth function to generate fusion patterns [39]. For VWAP, CDR, and Twitter applications this resulted in 52, 52, and 203 sample applications respectively (excluding the input application). These sample applications were run for a short period of time, and their throughput information were recorded similar to the evaluations conducted in Section 6.9.3. We plot each sample application's average throughput for three optimization runs in Figures 6.28 and 6.29 with X axis showing the sample program label. Note that we do not do program space reduction in these experiments. However, in System S, PE profile information is used for selecting the optimized fusion to be used (Similar to our code transformation method). The labels follow a naming convention that resembles how the operators are fused to PEs. E.g., The label `vwap.1.1.2.1.3.1.4.2.5.3` indicates first, second, and third operators (from top) in the sample program file is bounded to PE1. Fourth operator is bound to PE2 while fifth operator is bound to PE3. We do not plot the results of Twitter in this thesis because all of the sample applications for Twitter (204 applications) could not be presented on a graph clearly due to the limited space. Furthermore, we do not conduct the experiment with Apnoea and Bargain because they could produce 191 million and 4140 (minimum) large numbers of sample programs.

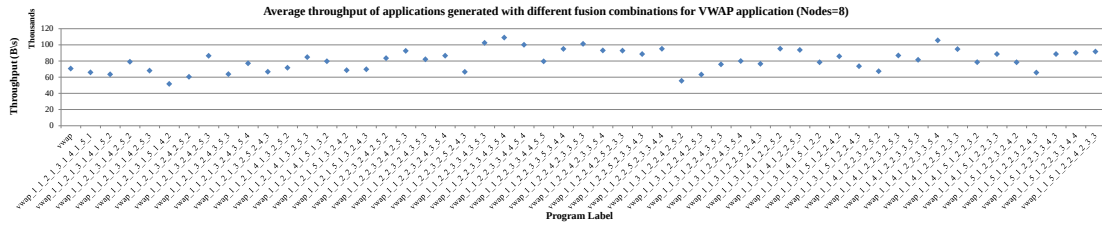
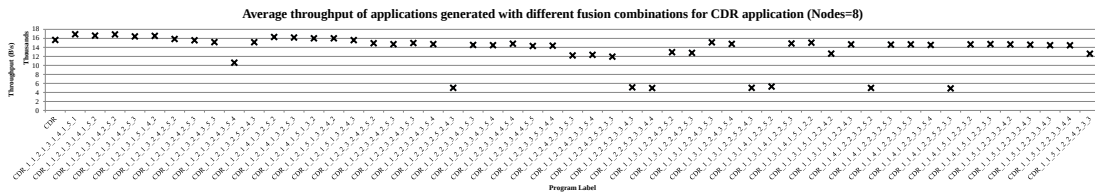


Figure 6.28: Average throughput of applications generated with different fusion combinations of VWAP application on cluster Y.

The evaluation results indicated that operator fusion is capable of producing higher performance applications for all the three applications (VWAP, CDR, and Twitter)



in the middle part of their data flow graphs tend to produce less performance. One can point out the fact that certain topologies of sample programs are favored by certain SPADE optimizations leading to higher performance in execution. However, the way how the operators are grouped during SPADE application compilation is at the discretion of the SPADE compiler [145]. Hence Hirundo can help programmers to identify common characteristics of high performance versions of their programs. Furthermore, we observed that two transformation patterns $1-j-1$ ($j > 1; j \in \mathbb{N}$) and $i-1-k$ ($i > 1; k > 1; i, k \in \mathbb{N}$) appearing at the beginning of the flow graph were effective in improving the performance of the input application.

Theoretically, our program transformation algorithm can generate sample programs with very large number of operators (An analysis of sample program spaces generated by Hirundo was made in Section 6.8 of this chapter.). However, we have observed that an upper limit of such transformations is imposed by the stream computing environment. It should be noted that our program transformation method is capable of exploring the whole set of possible sample program combinations by varying the value of d .

Use of K-Means clustering on sample program database is a promising approach for finding performance characteristics of sample programs. However, in the current version of Hirundo we do not use K-Means clustering for identifying sample labels with high performance. Instead, we use performance difference of sample programs in different optimization runs. We believe this approach is capable of pointing out specific sample programs with consistent high performance. In this thesis we conducted K-Means clustering only on linear DAGs. While a similar performance clustering experiment can be conducted on multipath DAGs as well, we keep it as a future work.

Presently Hirundo's Program Structure Analyzer can identify and transform a class of stream programs which are made out of combinations of Source, Functor, Aggregate, Join, Split, Sink, and UDOP operators. These seven types of operator blocks are sufficient to create many useful real world stream applications such as the ones discussed in this chapter. However, there are other stream applications with different BIOP types such as Sort, Punctor, Barrier, etc. that we hope to integrate in to Hirundo's program analyzer's grammar and transformation logic in future.

Path and mode based filtering has been effective in reducing the sample program space of Apnoea and Bargain by significant sizes. Without any filtering, for a depth value of 4, our code generator produced 1935 and 1583 sample applications for Apnoea and Bargain respectively. Path and mode based filtering techniques reduced these program spaces to 12 programs each. Therefore, these techniques are effective in eliminating a large amount of unnecessary sample applications. However, there are certain exceptions such as the sample program `SCSV_FOR_S_4F_16AG_2F_F_J_F_SLFOR_E` of Bargain application which could not be filtered effectively by mode based elimination. We are investigating different other techniques for filtering such applications.

Current version of Hirundo's program structure analyzer and transformation supports only transformation of stream programs DAGs. Moreover, Hirundo assumes that programmer does not manually allocate node pools in his/her program (Node pool is a SPADE language construct that allows manual allocation of operators to specific hosts.).

While performance prediction mode of operation was able to identify high performance sample applications compared to the input application, performance prediction accuracy need to be improved. This is because we got lesser number of sample applications with consistent higher performance as can be observed in the second major column of Table 6.3. Furthermore, when the complexity of the DAG increases, the accuracy of the performance prediction degrades because the probability of locating a similar performance application becomes less unless the same application has been optimized by Hirundo before.

As mentioned in Section 6.9.1, the decision of use of two different compute clusters X and Y with X being a high performance cluster is motivated by the fact that optimized versions of a program may vary between different compute environments. This was demonstrated through the results of Section 6.9.3.

6.11 Conclusions

This chapter described a mechanism for automatically transforming data flow graph of stream programs to high performance, optimized versions. The stream program automatic optimization methodology uses the notions of Tri-Operator transformation, Bi-Operator transformation, transformer blocks, and operator block fusion for stream program transformation. We implemented Hirundo, an optimization mechanism for SPADE applications using Python.

For a transformation depth value of four (i.e., $d=4$) Hirundo created at least 25 sample programs with different performance levels for the input applications with linear DAGs described in this chapter. With use of the path and mode based filtering, the same transformation depth created twelve sample applications for Apnoea and Bargain. Larger depth values produce even more sample programs. Therefore, we came to conclusion that Hirundo's program transformation algorithm (including Tri-Operator transformation, Bi-Operator Transformation and operator blocks fusion) is able to produce large number of programs with variety of performance levels. The second conclusion that we arrived at is our approach is able to produce SPADE programs with consistent, high throughput gains for SPADE applications having linear and multipath DAG structures. This was demonstrated by optimizing five real world stream applications which include three linear DAGs and two multipath DAGs. Our method creates higher performance gains (31.1, 2.7, 2.2 times for CDR, VWAP, and Twitter applications on cluster X. Also 30.27, 1.3, and 1.65 times for CDR, VWAP, and Twitter applications respectively on cluster Y). Another conclusion we arrived at is that having large number of source operators is likely to produce more throughput. Relatively high operator density in the middle regions of a data stream graph may produce less performance, hence stream programmers should avoid creating such programs. These two conclusions are supported by the observations made using K-Means clustering.

We compared our stream program performance optimization approach with an alternative method called operator fusion, and we observed that our approach is better for improving stream program performance. With CDR application our method was able

to out perform operator fusion by a factor of 26 times on cluster Y with 8 nodes.

6.12 Summary

A source operator graph transformation based technique for automatic stream program performance optimization was described in this chapter. Section 6.1 provided an introduction to the requirement of having auto-tuning techniques for stream job performance optimization. The preliminaries such as the definition of terms for the rest of the chapter's content was presented in Section 6.2. A high level view of the optimization methodology of Hirundo was described in Section 6.3. Next, the program transformation algorithms of the auto-tuning technique were described in Section 6.4. Transformer blocks which are modularizations of operator blocks for automatic tuning process were described in Section 6.5. Implementation details of Hirundo as well as the fault tolerance mechanism of Hirundo were described in Sections 6.6 and 6.7 respectively. Sample program space reduction and ranking techniques were described in Section 6.8. Evaluation results were presented in Section 6.9 while a discussion of the observations made was conducted in Section 6.10. Conclusions of the chapter were provided in Section 6.11. The remaining two chapters of Part I of the thesis present extensions made for the automatic code transformation algorithms described in this chapter. Next chapter describes automated optimization of multiple stream jobs with software auto-tuning.

Automatic Stream Query Optimization in the Presence of Multiple Stream Jobs

Compute clusters used for running large scale data analysis applications often run multiple jobs simultaneously. Most of the data analytics described in Chapter 3 promote the concepts of multitasking. However, the stream job performance autotuning techniques described in the previous chapter were conducted in single application settings. This chapter extends the code transformation algorithms described previously to the domain of multiple stream job performance optimization. Section 7.1 provides an introduction to the domain of multiple stream job performance optimization highlighting the research problem addressed. Methodology followed during multiple stream job optimization is described in Sections 7.2 and 7.3 respectively. The proposed optimization techniques were implemented on a mechanism called Tahitica which is described in Section 7.4. Experimental evaluation details of the multi-query optimization mechanism is given in Section 7.5. A discussion of the experiment results is presented in Section 7.6. The conclusions are presented in Section 7.7.

7.1 Introduction

In this chapter we extend Hirundo to the domain of multiple stream program optimization. In Hirundo, the optimization was conducted considering only a single stream application. However, the stream jobs' performance can be further improved by considering features of several stream jobs. E.g., We may need to allocate some specific operator to a subset of nodes since that operator might need some special hardware resources (such as a multicore processor) that are not present in other hosts. Furthermore, Hirundo's optimization mechanism was centered around parallel streams pattern [233].

This is a very important technique for stream job performance optimization since we are living in the multicore era, where parallelization of serial code is essential for exploiting the hardware architecture's performance potential [288]. However, operator fusion is another technique that leads to higher performance [145] that was not used in Hirundo. Fusion achieves optimized performance by grouping several operators into a single process which eliminates communication overhead associated with operator parallelization, and pipelining. In this chapter we utilize both these techniques for automatic tuning of multiple stream jobs.

We implement the automatic multiple stream query optimization methodology on a mechanism called Tahitica ¹. During the optimization process we first identify the structure of each input stream query and represent it as a Directed Acyclic Graph (DAG) [39]. In this DAG vertices represent query operators while edges represent the streams that flow between the operators. We profile each application to obtain their operators' CPU cost information. If the DAG is a multipath DAG we select the path with highest CPU cost for transformation. Next, for each path we identify the best transformation patterns to be applied based on the path's CPU usage information. The selected code transformations are applied on the input programs to obtain transformed stream applications. The operator blocks in the transformed applications are stitched together to obtain the optimized versions of the stream applications. Once the sample applications are generated, they are profiled in the runtime environment for a short period of time (1 or 2 minutes) to identify high performance sample applications. Each identified high performance sample application is used to create the optimized operator graph fusion.

7.1.1 Input Applications

This chapter uses two input applications during the experiments of Tahitica which represent real-world stream processing applications. The first application is Apnoea [294] described in the previous chapter. Data flow graph of this application is shown in Figure 6.3.

The second application is a Streaming Click Through Rate (SCTR) calculation application (shown in Figure 7.1). Click-Through Rate calculation is very important application in search advertising business. In online advertising the advertisers are charged each time their ads are clicked [19] and this model is called pay-per-click revenue model. Click-Through Rate (CTR) is defined as the ratio between the number of clicks an advertisement receives divided by the number of times the advertisement is shown (i.e., Ad impressions). The CTR calculation application in Figure 7.1 computes the CTR at Functor 3 (F3). However, the application needs to filter out the invalid data tuples as well as some clicks that are generated by invalid user accounts (potentially these accounts can be considered as bots). These activities are conducted at the upper-half of the data flow graph.

¹Tahitica is the species name of the Pacific Swallow.

7.2 Sample Program Generation via Automatic Code Transformation

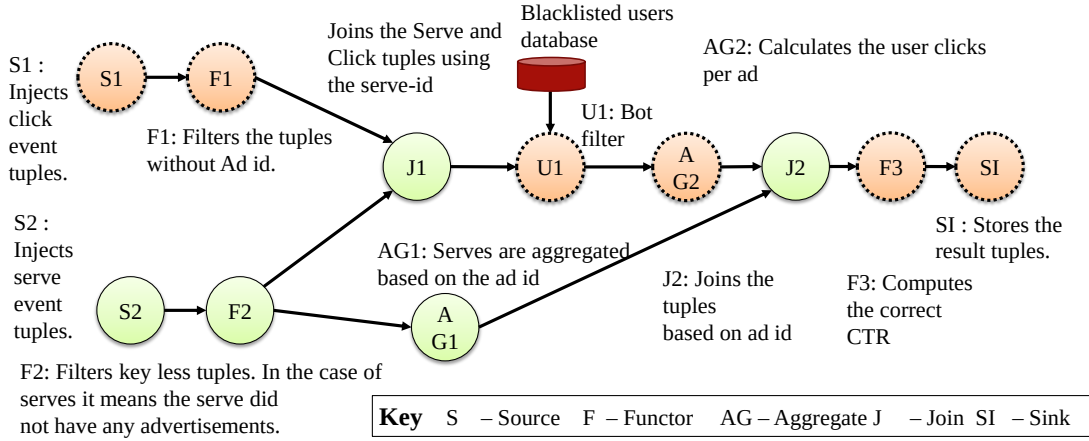


Figure 7.1: Data flow graphs of example input stream applications

7.2 Sample Program Generation via Automatic Code Transformation

As mentioned previously, the code transformation algorithm of Tahitica generates many sample applications using the input application's DAG. The code transformation is conducted focusing its rays. The transformation of the data flow graphs is conducted in two directions, hence we call it 2D Transformation. The methodology followed during the code transformation is shown in Figure 7.2.

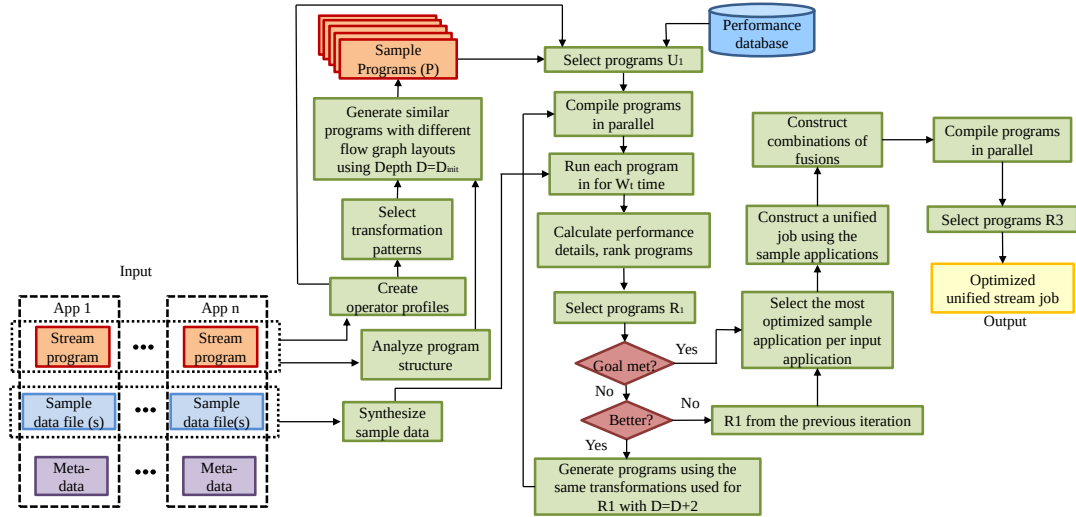


Figure 7.2: Methodology of Tahitica

Input to Tahitica is a collection of stream programs each having sample input data file(s) and meta-data stored in configuration files. Each input program is analyzed using

a custom grammar to identify operator blocks. After Tahitica identifies all the operator blocks present within the input program it constructs a DAG where each operator is a node and streams that connect operators are edges.

The constructed DAGs along with the operator profiling information are used during the optimization process. User can specify the optimization goal before the optimization process begins in terms of expected throughput increase (E.g., 2 times, 3 times, etc.). During run time Tahitica generates a sample program space with initial depth value (d). Also it runs the input application in the environment for a short period and obtains each operator's profile information (CPU usage, memory usage, and network I/O data rates). These profile information along with previous optimization run information are used to select a subset of sample programs (U_1). The selected programs are compiled in parallel in distributed node cluster. Then each sample application in U_1 are run in the intended environment for a time window of W_t . A ranked list of programs R_1 ($R_1: R_1 \subset U_1; |R_1|=m; m < n$) is selected based on the performance results obtained by running the programs. based on the performance results obtained, Tahitica determines whether the optimization goal is achieved or not. If the optimization goal is not met, it increases the transformation depth by 2 and transforms the input application using the same transformations used to generate the R_1 sample programs. The generated sample programs are compiled as previously, run in the environment for W_t time to obtain their performance. Similar to previous run, a ranked list of programs are selected, and is compared whether the optimization goal is achieved or not. This process is continued until the performance optimization goal is met or a less performance was obtained for newly generated sample applications.

Next, Tahitica selects the best performing sample applications for each input application and constructs a unified stream application template. Sample program selection is based on the fact that the program provides high consistent performance. During the selection process we use three latest calibration session's data per input application. The constructed application template is transformed with Fusion Transformation to create different operator graph fusion layouts (We call these secondary sample applications). Each secondary sample application is run in the intended stream processing environment to obtain their performance characteristics and a ranked secondary sample program list is output from Tahitica.

7.2.1 2D Transformation

The code transformation algorithm of Hirundo described in the previous chapter was conducted using only single depth value which determined the extent to which a flow graph is transformed. Furthermore, in the case of multipath DAGs the code transformation was conducted on the most expensive ray on the most expensive path in terms of CPU usage. In Tahitica we extend this code transformation algorithm further by transforming stream data flow graph along the most expensive path. That means in Tahitica it is possible to transform more than a single ray on the most expensive path. We introduced a parameter called "Traversal Depth" that can be provided with the meta-data of

7.2 Sample Program Generation via Automatic Code Transformation

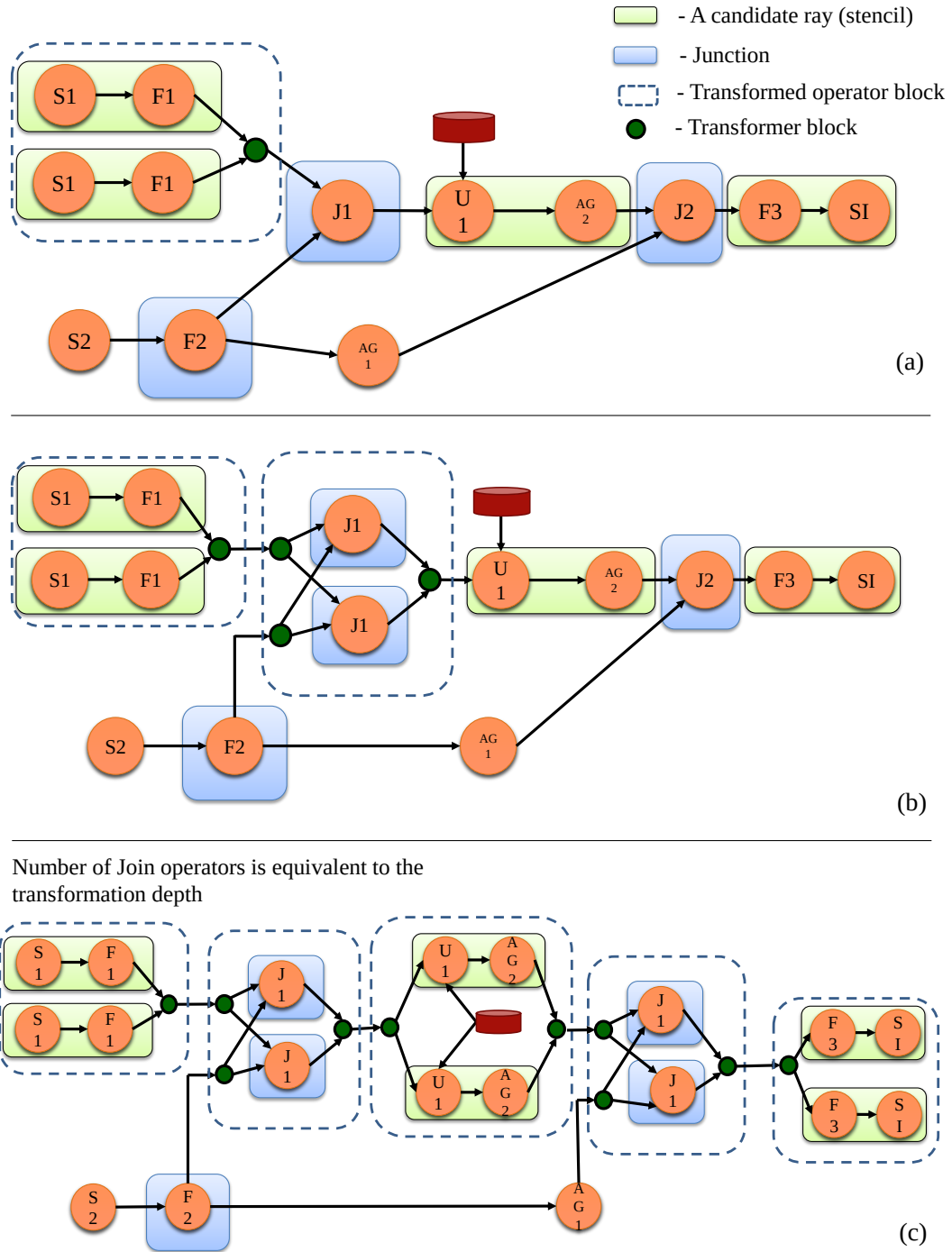


Figure 7.3: 2D Transformation of Tahitica

the input application. Program transformation of Tahitica happens through the costliest path on a multipath DAG until it reaches the Traversal depth. This process is shown in Figure 7.3. The code transformation algorithm of Tahitica is further illustrated in the activity diagram shown on Figure 7.4.

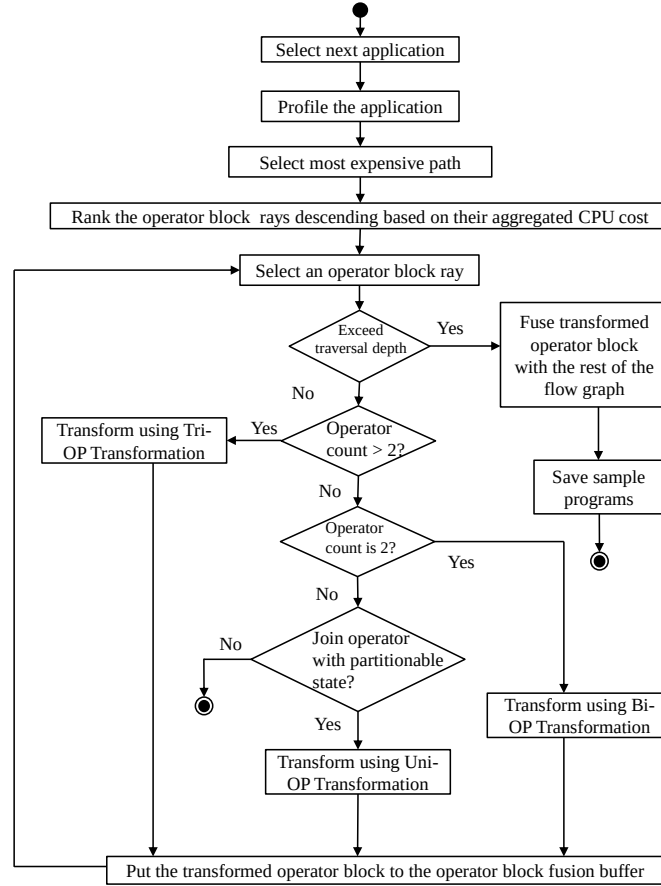


Figure 7.4: Overview of the program transformation of Tahitica.

7.3 Optimization of Multiple Stream Jobs

Once the input application’s corresponding sample applications are generated they are calibrated in the intended runtime environment for a short period to identify the best performing sample applications. The optimization process of multiple stream jobs starts with selection of the best performing sample application for each input application. Next, Tahitica creates a “unified sample application template” by merging the SPADE code of these sample applications. Furthermore, Tahitica counts the number of rays available on the unified sample application template and uses this number for generating a collection of set partitions. These set partitions represent combinations of rays identified

in the previous step. Using these set partition schemes operator partitions are created. The generated operator partitions are used to create secondary sample applications. A secondary sample application essentially has the same structure as the unified sample application template. However, its PE allocation layout is changed based on the set partitioning scheme. In the current implementation of Tahitica we allow user to specify beforehand the number of secondary sample applications (say N) to be generated since the number of set partitions generated can be very large. Then it selects N number of set partitioning schemes randomly and generates the secondary sample applications. The evaluation section of this chapter lists performance results of 21 sample applications that were generated by following the aforementioned method.

7.4 Implementation

Tahitica is implemented using Python programming language by extending the Hirundo mechanism. Similar to Hirundo, Tahitica has two major components called Main module and Worker module. Architecture of Tahitica is shown in Figure 7.5.

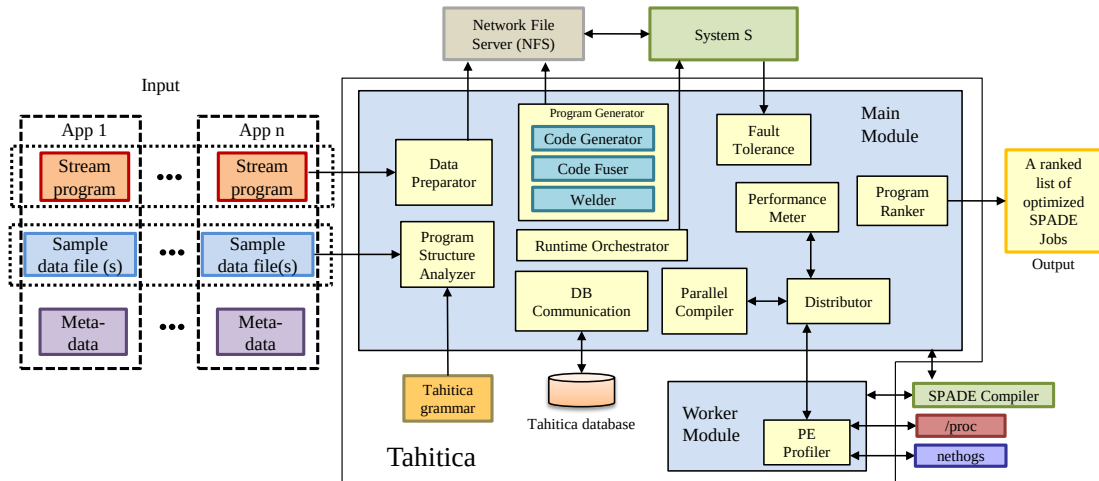


Figure 7.5: Architecture of Tahitica

Tahitica has been developed targeting stream programs written using SPADE language. Therefore, Tahitica depends on System S and SPADE compiler during its operations. Tahitica uses an SQLite database to store its information. Main module has been separated into ten sub modules based on different functionalities they handle.

7.5 Experimental Evaluation

We evaluated the performance of the sample applications generated individually by Tahitica as well as the generated unified single stream job on a cluster of four com-

pute nodes (Cluster X described in Table 6.2).

We first ran Tahitica in the calibration mode of operation. In this mode of operation, Tahitica runs the entire sample applications space (each sample application for a short period of time) generated during the optimization run and records their performance in the performance database. The performance results are shown in Figures 7.6 and 7.7.

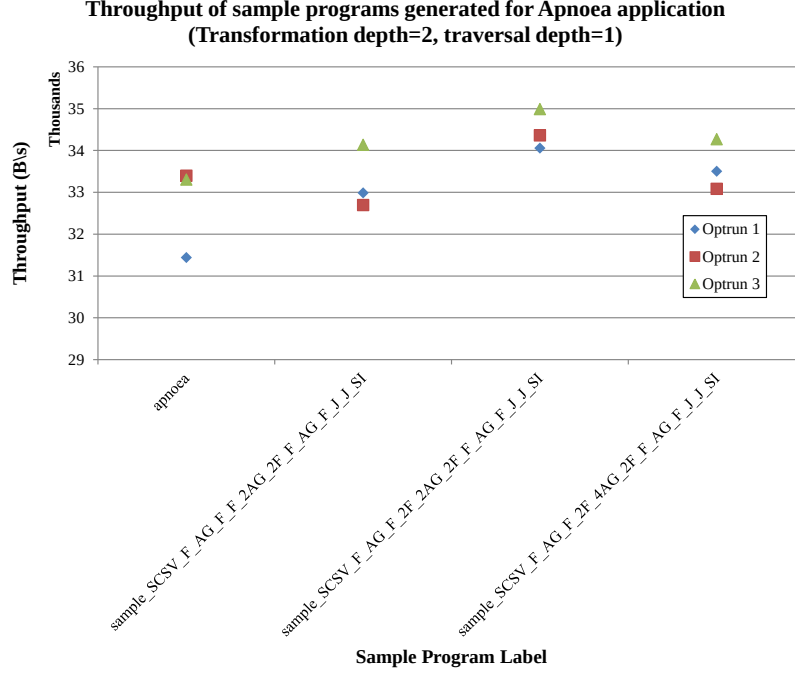


Figure 7.6: Comparison of three optimization runs (Opt run 1,2,3) of the Apnoea application using equal optimization parameters (transformation depth (d) = 2, traversal depth = 1).

Next, we ran Tahitica in the multiple stream optimization calibration mode. In this mode of operation Tahitica selected two sample applications (SCSV_2F_SCSV_F_AG_2J_U_AG_J_F_SI for SCTR and SCSV_F_AG_F_F_2AG_2F_F_AG_F_J_J_SI for Apnoea) which produced high throughput compared to their corresponding input applications (These two sample applications produced consistent high throughput as can be observed from Figures 7.6 and 7.7.). These two applications were used for multiple stream query optimization phase of Tahitica which generated 21 sample applications (We set the value 21 in advance as a parameter for the optimization phase). Since the optimization was run in the calibration mode, Tahitica ran each secondary sample application produced in the environment for a short period of time and gathered their throughput performance information. These results are shown in Figure 7.8.

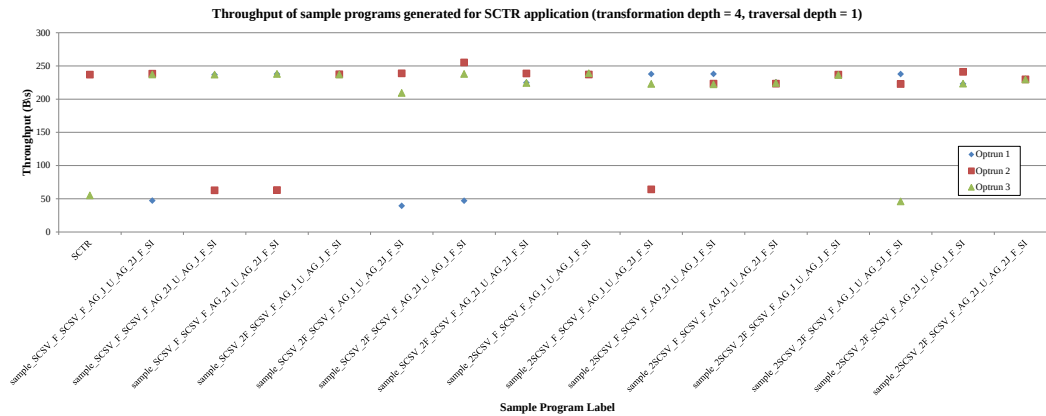


Figure 7.7: Comparison of three optimization runs (Opt run 1,2,3) of the SCTR application using equal optimization parameters (transformation depth (d) = 4, traversal depth = 1).

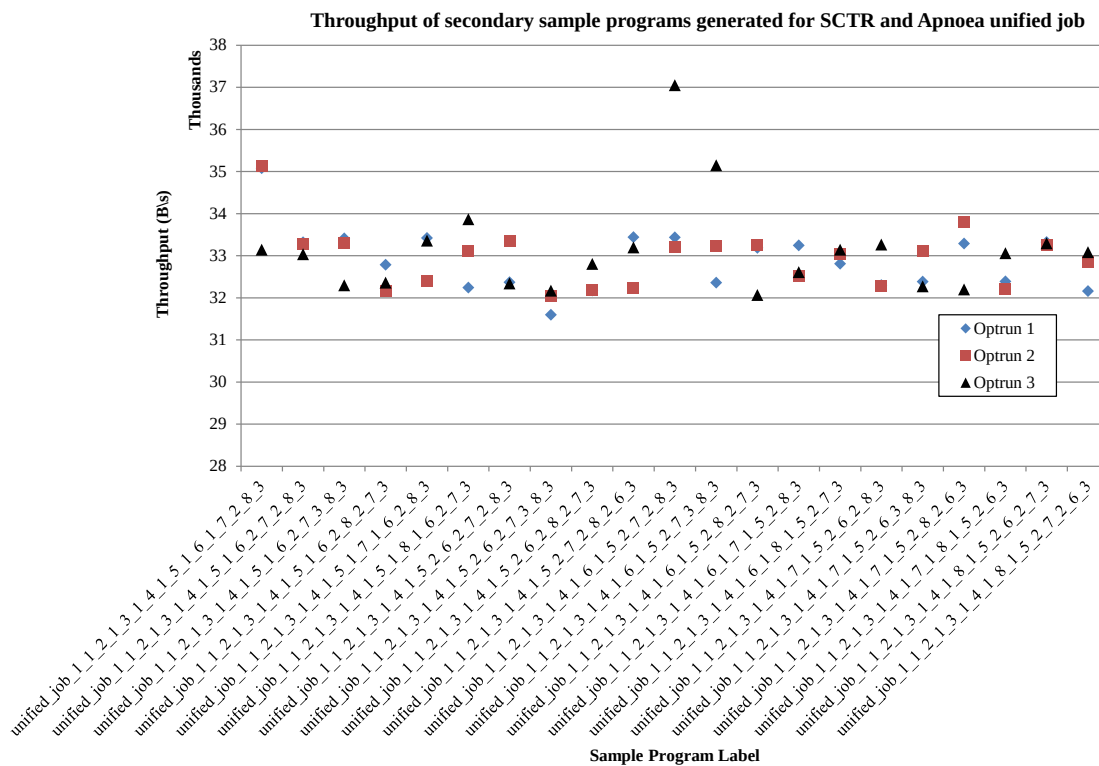


Figure 7.8: Comparison of three optimization runs (Opt run 1,2,3) of the unified job created for Apnoea and SCTR applications. The optimization parameters were the same as in the previous two evaluations.

7.6 Discussion

From the individual sample application calibration sessions we observed that several sample applications were able to produce consistent high performance compared to the input applications. E.g., `SCSV_2F_SCSV_F_AG_2J_U_AG_J_F_SI` produced high throughput compared to the input SCTR application when considering the entire three optimization runs. In the case of Apnoea `SCSV_F_AG_F_F_2AG_2F_F_AG_F_J_J_SI` produced consistent high throughput performance compared to the input application. This shows that our optimization mechanism is able to produce high performance stream applications.

Furthermore, from the multiple stream job optimization session that created a hybrid stream job by merging the top two optimized sample applications from Apnoea and SCTR we observed that the `unified_job_1.1.2.1.3.1.4.1.5.1.6.1.7.2.8.3` produced high throughput performance compared to the other applications. The average throughput measured was 34,448B/s (33.64KB/s). The optimized fused job was able to produce high throughput almost in par with the optimized jobs from Apnoea 34,466B/s (33.65KB/s), and SCTR 179.8 B/s. Note that the notation used in the above unified secondary sample application does not represent the operator arrangement of the secondary sample application. It is just a representation of how the operator blocks are bound to PEs. E.g., The label `unified_job_1.1.2.1.3.1.4.1.5.1.6.1.7.2.8.3` indicates first to sixth operator sets (from top to bottom) in the sample program file is bounded to PE1. Seventh operator is bound to PE2 while eighth operator is bound to PE3.

There are several limitations for our current Tahitica mechanism that we list here. First, the grammar of Tahitica is limited to sub set of SPADE language. Presently Tahitica's Program Structure Analyzer can identify and transform a class of stream programs which are made out of combinations of Source, Functor, Aggregate, Join, Split, Sink, and UDOP operators. These seven types of operator blocks are sufficient to create many useful real world stream applications such as the ones discussed in this chapter. However, there are other stream applications with different BIOP types such as Sort, Puncor, Barrier, etc. that we hope to integrate into Tahitica's program analyzer's grammar and transformation logic in future. In the current prototype we assume that the sample programs with individual highest performance works better even when they are fused into one unified job.

Furthermore, we plan to create a more sophisticated optimization mechanism for Tahitica's multiple stream job optimization algorithm in future based on graph partitioning. Specifically we hope to use the profiling information of each operator to create feature vectors, which can in turn be used to solve a multi-constraint graph partitioning algorithm. Such technique could point out better unified sample jobs than the current random selection technique.

7.7 Conclusions

This chapter described a methodology for automatic production of optimized multiple data stream graphs. Our methodology is based on concepts of operator parallelization

and operator fusion. The code transformation algorithm that we developed is based on a concept called 2D Code Transformation which involves two main transformation parameters (Transformation Depth and Traversal Depth). We used code transformation notions: Uni-operator transformation, Bi-Operator transformation, Tri-Operator transformation, transformation blocks, and operator block fusion for stream program transformation. We implemented the optimization methodology on a mechanism called Tahitica and evaluated its feasibility.

We observed that our multiple stream job performance optimization mechanism is able to create optimized unified stream processing jobs using multiple stream applications. We demonstrated this by using two real world stream applications: SCTR and Apnoea. Tahitica was able to produce a high performance unified stream job with average throughput performance of 34,448B/s (33.64KB/s) using the two input applications.

7.8 Summary

Automatic tuning of large scale data analysis applications in the context of multiple parallel jobs is an important area of data analytics optimization. This chapter provided a technique for conducting such multiple data stream job performance optimization by extending the automatic code generation and transformation algorithms described in the previous chapter. Section 7.1 provided an introduction to the area of multiple stream job optimization. Methodology followed for multiple stream job optimization was described in Sections 7.2 and 7.3 respectively. We implemented the proposed multiple stream job performance optimization on a mechanism called Tahitica. The implementation details of Tahitica was given in Section 7.4. Evaluation of the proposed approach was described in Section 7.5. A discussion of the proposed approach and the experiment results was made in Section 7.6. Conclusions were presented in Section 7.7. The next chapter describes performance maintenance of stream application in the presence of crash failures in a stream processing environment.

Stream Program Performance Recovery in Resource Limited Compute Clusters

System failures are frequent in large scale compute systems used for conducting large data analysis. Crash failures introduce huge challenge for many organizations who conduct data analysis activities in their own compute clusters. Maintaining a decent level of performance is of utmost importance for achieving the Service Level Agreements (SLAs) defined on these systems. This chapter introduces a new technique for performance maintenance of data stream jobs, using the existing compute resources during crash failures. The chapter extends the code transformation techniques described in Chapter 6 to the domain of stream job performance maintenance. Section 8.1 provides an introduction to stream job performance maintenance during crash failures. Methodology followed for performance maintenance is described in Section 8.2. Sample program ranking algorithm used for selecting appropriate sample applications for performance maintenance is described in Section 8.2.1. Program switching model is explained in Section 8.2.2. Load shedding technique used during drastic node failures is described in Section 8.3. Implementation details of the performance maintenance mechanism (Albatross) is described in Section 8.4. Details of how to eliminate single point of failure in Albatross is described in Section 8.5. Experimental evaluation of the performance maintenance approach is described in Section 8.6. A discussion of the evaluation results is presented in Section 8.7 before providing conclusions in Section 8.8.

8.1 Introduction

Highly availability is one of the key challenges faced by large scale data analysis software systems such as stream processing systems in providing continuous services. Operating

system halts, power outages, virtual machine crashes, etc. are some examples for crash faults which may paralyze or take an entire application out of service. Crash faults introduce huge system downtimes and financial losses because those take more time to recover since some of those need direct intervention from system administrators [173]. Physical replication is currently the widely adapted solution for recovering from crash faults in stream computing systems [170]. These techniques require k replicas to tolerate up to $(k - 1)$ simultaneous failures [300]. Maintaining such large number of backup nodes costs a lot in terms of electricity, rack space, cabling, and ventilation.

A possible solution is to balance local stream processing load with public cloud (i.e., use of Hybrid Cloud [301]) which allows for maintaining stream program performance [218][219]. Yet such approaches require access to public clouds which makes it impossible to use such solutions in certain applications. For example, if the stream application deals with sensitive information (e.g., health care [302][303], national defense, etc.) it would be very difficult to follow such solution. Certain stream processing systems have license and software issues [219] which makes it impossible to deploy them in hybrid cloud environments.

In the previous two chapters of this thesis we observed that data flow graphs tend to maintain similar shaped performance curves (i.e., similar performance characteristics) within similar stream environment conditions [165]. This is an indication that relative performance of such data stream graphs is a predictable quantity.

Considering the infrastructure costs associated with replication approach for maintaining performance and the limitations of the aforementioned alternatives, we introduce a different, more efficient solution for maintaining stream program performance which is applicable in resource limited stream computing clusters. Our solution is based on automatic code generation and stream workload shedding [304]. For a given stream program that is to be deployed in the node cluster we generate a variety of data flow graphs (i.e., “sample programs”) with each giving different performance characteristics. Sample programs (one per input program) with consistent high throughput performance compared to the input programs are selected and run in the cluster. During a node failure (e.g., system crash/shutdown) which affects a sub set of nodes in the cluster, our approach evaluates potential of performance maintenance of stream applications. Then it introduces different sample applications which produce better performance in the new environment. Our approach assigns priority to each input program and tries to maintain performance of high prioritized programs. During a drastic failure which reduces system resources considerably Albatross conducts stream workload shedding along with the automatic code generation to maintain the performance of the applications.

We implemented our proposed approach on Albatross¹, which is a middleware that monitors the performance status of each node and the stream processing jobs. Albatross switches between different sample program appropriately to respond to the changes that happen in the node cluster.

¹Albatross is a large seabird which is among the largest flying birds. Albatrosses are highly efficient in the air and they can fly great distances with little exertion.

8.1.1 Resource Limited Stream Compute Clusters

While our approach can be applied in public compute clusters, dynamic resource allocation has become a great problem in resource limited small (typically private) compute clusters. What we mean by use of the term “Resource Limited Stream Compute Clusters” is a compute cluster with fixed set of nodes which cannot be expanded dynamically. Our emphasize is on the number of available nodes rather than the amount of available resources in a particular node of such cluster because we believe that it does not matter how much resources a node has if it crashes suddenly. Such compute clusters are widely run by financial, academic, and healthcare [294] institutions that operate variety of stream processing applications. An important feature of these clusters is that it takes considerable amount of time to replace a defected node in such private clusters compared to public clusters (e.g., Amazon EC2) which can easily provision nodes on demand. We believe that our approach for performance recovery is best applicable to such resource limited clusters.

8.2 Approach for Performance Maintenance

We formerly define our approach below. The notation used in our description is explained in Table 8.1.

Table 8.1: Notation.

Notation	Description
E_N	Stream processing environment with N nodes. ($N > 1$)
S	Set of input stream programs. ($ S = n, n > 0$)
u	Performance margin for stream processing environment E . ($u \in \{0, \dots, 100\}$, $u \in \mathbb{R}^+$). This value is calculated by Albatross using current performance information of E .
m	A performance window set by user. ($m \in \{0, \dots, 100\}$, $m \in \mathbb{R}^+$)
M_i	Input stream program priority margin. ($\forall i, j$ where $i, j \in \mathbb{N}$, $M_i, M_j \in \mathbb{R}^+$, $i, j \in \{0, \dots, (n-1)\}$, $i, j \in \mathbb{N}$, $M_i \neq M_j$, $\sum_{i=0}^{n-1} M_i = 100$). Priority margin is used for ranking input programs based on their importance. This value need to be specified by user prior running Albatross.
r	Calibration run. If $r = 0$ it is a normal mode run. When Albatross is deployed in its usual operation its called normal mode run.
$P_S^{(r)}$	Sample program set generated for S during calibration run r . ($r \in \mathbb{N}$, $r \neq 0$). A calibration is a running of entire sample program space with the intention of obtaining the performance information.
$P_S^{(0)}$	Sample program set generated for S during normal mode run. Here 0 in $P_S^{(0)}$ represents a normal mode run of Albatross.
X_N	Selected sample program set. ($X_{N_i} \in P_{S_i}^{(0)}$, $ X_N = n$, $ X_{N_i} = 1$)
$\text{perf}(x)$	Predicate for performance (e.g., throughput, elapsed time, etc.) of sample program x

Given an E_N which receives a steady flow of input data streams, a set of input programs S each having a priority margin M_i , a set of sample programs generated during

past calibration sessions (each represented as r) of S programs denoted by $P_S^{(r)}$, Albatross generates $P_{S_i}^{(0)}$ for each S_i ($i \in \{0, \dots, (n-1)\}$, $i \in \mathbb{N}$). Then it selects sample program set X_N considering the empirical performance information of each $P_{S_i}^{(r)}$, compiles and runs them in E_N . The algorithm that selects each sample program X_{N_i} is described in the next section.

During a sudden node failure which results in a different environment $E_{N'}$ with N' nodes ($N' < N$; $N' > 0$), Albatross selects $X_{N'}$ ($X_{N'_i} \subset P_{S_i}^{(0)}$, $|X_{N'}| = n$, $|X_{N'_i}| = 1$) sample programs if $M_i \geq (u+m)$ and compiles them. Then Albatross cancels the programs X_N and starts running both X_N and $X_{N'}$ program sets in $E_{N'}$ (with N' nodes) for a time window of W_t . For each program X_{N_i} and $X_{N'_i}$, $\text{perf}(X_{N'_i})$ is compared with $\text{perf}(X_{N_i})$ in the context of $E_{N'}$. Programs with highest performance for each i is kept running and the other jobs are canceled.

When the crashed nodes come back online, $E_{N'}$ changes back to E_N . Therefore, the programs $X_{N'_i}$ are switched back to X_{N_i} .

8.2.1 Program Ranking Algorithm

It should be noted that the sample program sets generated by Albatross are not exponential. E.g., For VWAP for a transformation depth of 4 it produces only 24 sample applications. As described in Chapter 6, Transformation depth is a non-zero integer which denotes the extent to which the input application's data flow graph is expanded [165]. Albatross uses the same program generator from Hirundo which was presented in Chapter 6.

Albatross uses empirical performance information gathered from previous calibration sessions to estimate which sample program version should be the best match for the stream processing environment E . Note that in the case of System S it is assumed that the operator placement decisions will be the same for multiple runs of a particular program on a specific node configuration. The Algorithm 1 describes the selection process. To maintain brevity we describe only the algorithm's inputs and outputs below.

Input to the algorithm is the application's name (this is the name assigned to a SPADE application under its Application meta-information tag [Application]), the structure of the input stream application G (G is a directed graph where each operator is represented as a vertex, and each stream that connects two operators is represented by an edge), transformation depth of the calibration run (i.e., d) and the number of nodes that are currently available on the stream processing environment. As the output, algorithm selects the first sample program which has higher average throughput compared to the input stream application, and has a higher minimum throughput relative to maximum throughput of the input application (lines 24 - 29).

If it could not find a suitable label then it reduces the restrictions, and tries to select the sample program label which has higher average throughput compared to the input application (lines 30 - 37). If this attempt also fails the algorithm returns the input application's name (appname) as the sample application label since the algorithm needs to specify at least one application label (X_{N_i}) that should be run in the environment.

Note that a label is an identification string (The naming convention of such sample programs is described in Section 6.9.2).

Only the latest three calibration run results for sample program selection algorithm are selected since a sample program produces similar performance behavior across multiple runs in the same stream computing environment. Selection of the sample program labels are conducted in two steps (in lines 24 - 29 and in lines 30 - 37). The programs are selected by considering the fact that having higher minimum throughput than the input application's maximum throughput, do have a higher probability of producing higher throughput compared to a decision made considering only the average throughput values (lines 24 - 29).

Algorithm 1: Selection of Replacement Sample Program
<pre> Input : Input application's name (appname), structure of the input stream application (G), transformation depth (d), number of nodes (nodes) Output : Replacement sample program (selectedLabel) Description : 1: optrunDict $\leftarrow$ getLatestThreeOptrunIDs(appname, G, depth, nodes) 2: for all optrun in optrunDict do /* Get sample program performance details */ 3: perfDict[optrun] $\leftarrow$ getPerfInfoForOptrun(optrun) 4: end for 5: labelDiffTable $\leftarrow$ {} 6: for all optrun in perfDict do /* Aggregate throughput information for each label */ 7: for all label in optrun do 8: labelDiffTable[label].append(optrun[label]) 9: end for 10: end for 11: inputStat $\leftarrow$ $\emptyset$ /* Find range, average, */ 12: labelPerfStat $\leftarrow$ {} /* min, max throughput */ 13: for all label in labelDiffTable do /* values for each group */ 14: labelPerfStat[label].range $\leftarrow$ range(labelDiffTable[label]) 15: labelPerfStat[label].average $\leftarrow$ average(labelDiffTable[label]) 16: labelPerfStat[label].min $\leftarrow$ minimum(labelDiffTable[label]) 17: labelPerfStat[label].max $\leftarrow$ maximum(labelDiffTable[label]) 18: if label is Input App then 19: inputStat $\leftarrow$ labelPerfStat[label] 20: end if 21: end for 22: sortByRangeAsc(labelPerfStat) /* Sort labels ascending order using range values */ 23: selectedLabel $\leftarrow$ $\emptyset$ /* Select sample program with higher average */ 24: for all label in labelPerfStat do /* throughput compared to the Input app */ 25: if label.min > inputStat.max then 26: selectedLabel $\leftarrow$ label 27: break 28: end if 29: end for 30: if selectedLabel = $\emptyset$ then /* If could not find a suitable app, */ 31: for all label in labelPerfStat do /* select using average throughput */ 32: if label.average > inputStat.average then 33: selectedLabel $\leftarrow$ label 34: break 35: end if 36: end for 37: end if 38: if selectedLabel = $\emptyset$ then /* If no suitable app found then return the input app */ 39: selectedLabel $\leftarrow$ appname 40: end if 41: return selectedLabel </pre>

8.2.2 Program Switching Model

A program switching model is employed to remove programs (starting from low priority) during drastic node failures that makes the remaining nodes fully or close to fully utilized. The elimination model is formed by Cartesian product of *resource availability functions* (u). If availability of a resource such as total amount of memory, total amount of CPU, etc. is denoted by $p(y)$ and program switching function (binary) is given by $\phi(x)$,

$$p(y) = \frac{(\text{current level of resource } y)}{(\text{initial level of resource } y)} \quad (8.1)$$

$$u = 100 \times \prod p(y) \quad (8.2)$$

$$\phi(x) = \begin{cases} 1 & [M_i - (u + m)] > 0 \\ 0 & \text{otherwise} \end{cases} \quad (8.3)$$

Current implementation of Albatross uses only two resource availability variables: RAM availability ($p(R)$) and Node availability ($p(N)$). Hence Equation 3 can be simplified to,

$$\phi(x) = \begin{cases} 1 & [M_i - ((p(R) \times p(N) \times 100) + m)] > 0 \\ 0 & \text{otherwise} \end{cases} \quad (8.4)$$

Successful operation of stream processing systems largely dependent on main memory availability. Node availability was also introduced to the program switching model because $p(N)$ directly reflects not only $p(R)$, but also other resource availability metrics such as CPU availability, network availability, etc. on a homogeneous cluster. This results in simplified parabola shape u for which users of Albatross can easily specify the priority value m before running Albatross.

8.3 Load Shedding for Drastic Node Failures

Our code generation approach as it is faces issues when large number of applications demand maintenance of higher performance [284]. A solution that we introduce for this issue is incremental load shedding. Load shedding is a well-known technique employed for removing additional load from a system in order to keep query processing up according to the input arrival rates [304]. We implemented random load shedding on Albatross at two levels (called L1 and L2). Load shedding is conducted considering the priority level (M_i) of each application. The Albatross code generator places a special set of operator blocks (we name those as *Valves*) at the starting point of each ray. Such Valve arrangement keeps minimum overhead on the transformed operator graph. Furthermore, we get lesser impact of load shedding on the results when we drop tuples at ray level, rather than dropping the tuples at the gateway from the very beginning.

At the L1 level the load shedding is done on the most costly ray on the DAG. The load shedding decisions are made at the Gateway component. Based on the decisions,

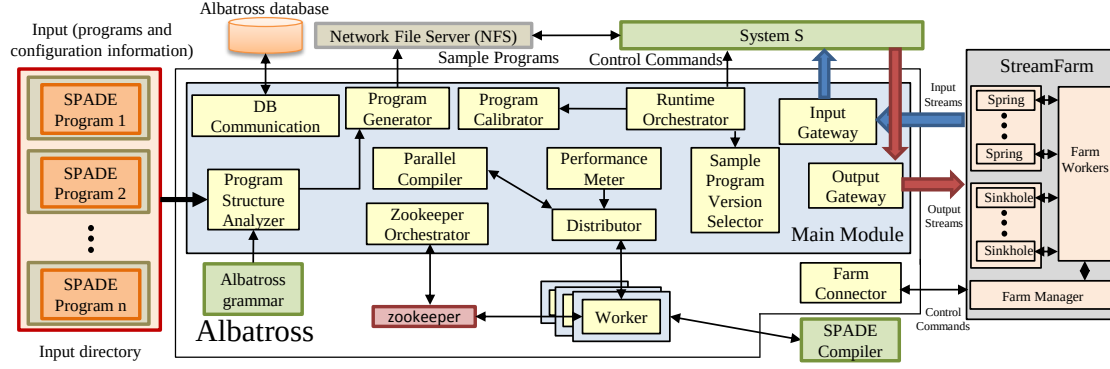


Figure 8.1: System Architecture of Albatross.

the Valves are informed with special data tuples (we call them as *beats*) to drop the tuples at a specific frequency. The Valves send an acknowledgment back to the Gateway module with a time stamp embedded. When the system resources reduces further below a threshold, the Gateway starts shedding tuples of low priority applications. When the system resources plummet further below the level that even load shedding cannot sustain, the applications are removed from the execution stage starting from the low priority applications.

8.4 Implementation

Albatross prototype was developed using Python. Architecture of Albatross is shown in Figure 8.1. The input to Albatross is a collection of directories each with an input stream program. Each program is associated with a configuration file which lists information such as transformation depth (d) of the input program, the priority margin (M_i), etc.

Albatross was developed for performance maintenance of stream programs written in SPADE language. Therefore, Albatross depends on System S and SPADE compiler during its operations. System S is dependent on a shared file system such as Network File System (NFS), General Parallel File System (GPFS), etc. since home directories of install owner and Streams instance owner of System S must be on a such shared file system [232]. However, Albatross does not use NFS as a secondary storage for calibration of stream programs to avoid potential performance bottlenecks. The experiments conducted using Albatross were supported by StreamFarm: the stream workload generator platform described in Chapter 5. Hence Albatross does not depend on any significant file I/O during its operations. An SQLite database is used to store Albatross's information. Brief descriptions of important components of Albatross are described in below subsections.

8.4.1 Runtime Orchestrator

Runtime Orchestrator module monitors health of the cluster using periodic heartbeat messages (i.e., ping messages). If a node crash was detected, the Input Gateway is informed to start the tuple buffering process. Next, the System S runtime is stopped and the defected node is removed from the node list of System S instance configuration. After this System S is restarted on the remaining nodes and the sample programs selected by Albatross’s program ranking algorithm and the original programs are run in the environment for a short period to select the best versions to be deployed. Next, the programs not selected are stopped and removed from the environment. Finally, the buffered data tuples are directed to the chosen applications.

8.4.2 Gateway

All the data streams that go into the sample applications and all the data streams that come out of the sample applications travel through the module called Gateway. The Gateway is used for measuring the data rates of the streams as well as incremental load shedding. The data rate information is used by Program Version Selector to switch between the sample programs that are suitable for a particular environment (E_N). However, in the experiments described in Section 8.6, we utilized the data rate reporting mechanism of StreamFarm to obtain the data rate information.

Furthermore, the Gateway buffers the tuples in memory, while Albatross conducts sample programs switching. After the appropriate sample programs are selected the buffered tuples are released to System S jobs before the incoming tuples are served. When the buffer gets emptied the incoming tuples are directed to System S jobs rather than adding them to the tail of the tuple queue. The gateway of Albatross is located on multiple nodes.

8.5 Elimination of Single Point of Failure

Albatross is controlled by the main module. If the node with the main module crashes the entire Albatross system also crashes since there was no way of re-instantiating the Albatross main module. This issue was solved by interfacing Albatross with Apache Zookeeper [176]. We implemented an algorithm that selects a candidate, live node from the node cluster and instantiates Albatross. The overall process of single point of failure recovery for four node cluster is shown in Figure 8.2.

In this method, a Zookeeper instance is created in all the nodes when Albatross starts its operation. Also all the workers register themselves with Zookeeper service, and keep on monitoring the Zookeeper for changes happen to the main node. When the main node faces a crash failure (Like Node 1 in Figure 8.2 (b)), its Zookeeper subscription gets erased from the Zookeeper’s registry. This event is detected by the Workers of the live nodes. The Workers start bidding on the Zookeeper service to become elected as the next main node. The winner announces itself on the registry (as “recoverycandidate”)

8.5 Elimination of Single Point of Failure

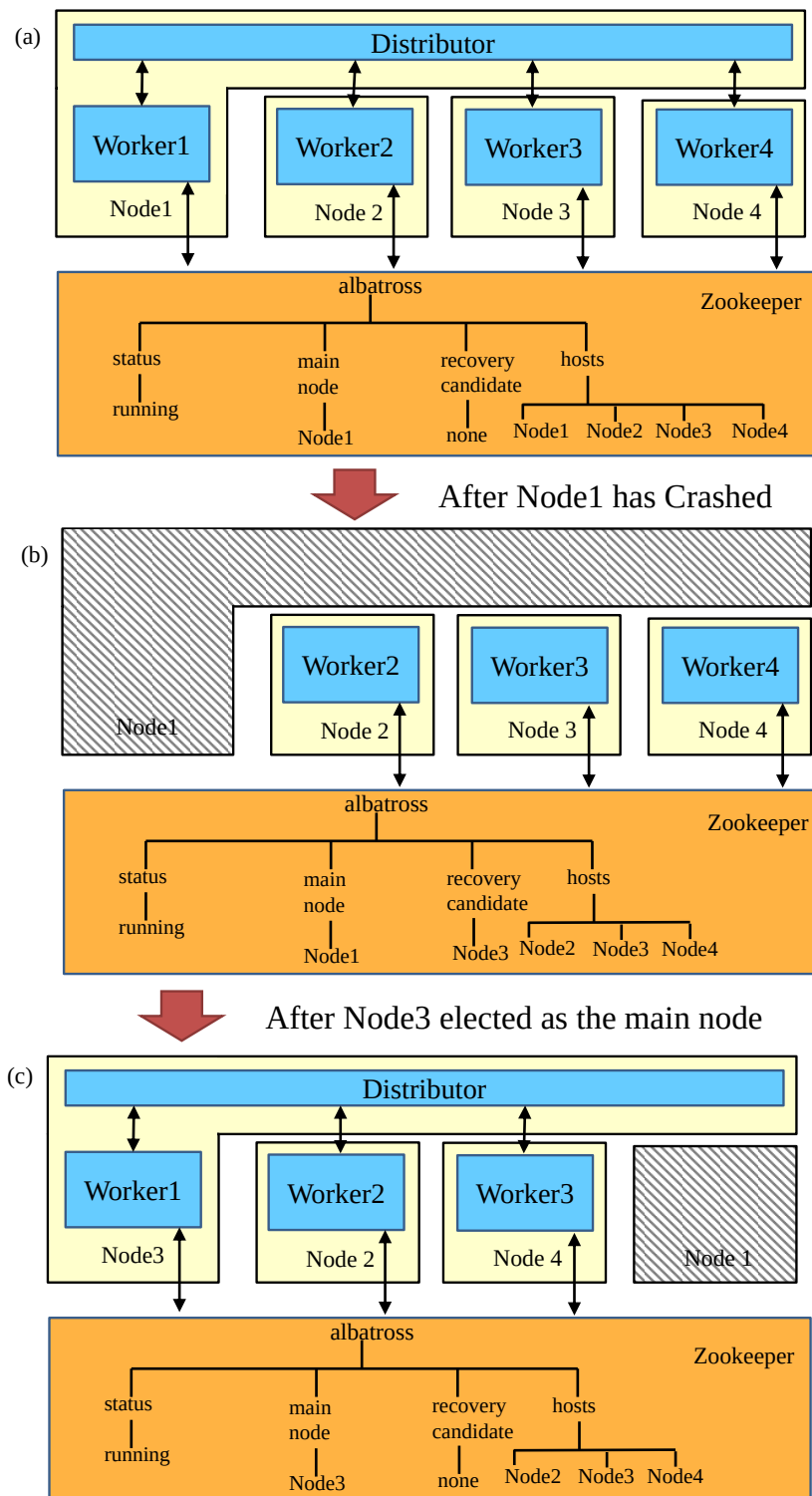


Figure 8.2: Recovery process from single point of failure.

and waits for others to recognize the event. Next, the winner starts new Albatross instance on the node (Node 3) that it runs on. Note that on Figure 8.2 (b) the status on the registry is shown as running, because even though the worker on Node 1 gets crashed, other workers were running so that the new Albatross instance (on Node 3) can acknowledge the fact that it is a single point of failure recovery. Results of evaluation of main node crash are shown in the Section 8.6.4.

8.6 Experimental Evaluation

8.6.1 Experimental Setup

Two compute clusters (lets call them A and B) of Linux Cent OS release 5.4 installed with IBM Infosphere Streams version 1.2 and Python 1.7 were used during the experiments. Each node in cluster A had a Quad-Core AMD PhenomTM9850 processor, 512 KB L2 cache per core, 8GB memory, 160GB hard drive, 1 Gigabit Ethernet (The same node cluster Y described in Table 6.2.). Each node of cluster B had a dual core AMD OpteronTMProcessor 242, 1MB L2 cache per core, 8GB memory, 250GB hard drive, 1Gigabit Ethernet. We used SQLite version 3 as Albatross's database and cluster B had JRE 1.6.0 installed. Both the A and B clusters were reserved for running only these experiments during the experiment period.

8.6.2 Evaluation of Stream Program Performance Variation

In the first half of the evaluations Albatross was modified not to respond to crash failures and ran two sample applications of Regex and VWAP on Cluster A. Then we crashed two nodes which left only 6 operational nodes. The two applications showed different characteristics after the crash (See Figure 8.4). In the case of regex the throughput dropped from an average of 84.16KB/s to an average of 21.38KB/s, a reduction of average data rate by 74.6%. In the case of VWAP application it completely stopped outputting data. Therefore, we observed that for certain applications, crash faults result in no output of data from System S. Some other applications output data at a reduced rate, but might produce different outcome than what is expected. In both the scenarios an explicit intervention like done by Albatross is needed to restore the performance.

In the second half of the experiments Albatross was run in the calibration mode with VWAP application in cluster A with 8 nodes. The results of six calibration runs are shown in Figure 8.3. We observed that certain applications produce similar performance behavior across the six experiments (E.g., SCSV_6F_AG_F_SI) and some applications produce higher performance compared to the input application (E.g., 8SCSV_6F_AG_F_SI). This indicates that our program transformation method generates sample programs with consistent yet different levels of performance.

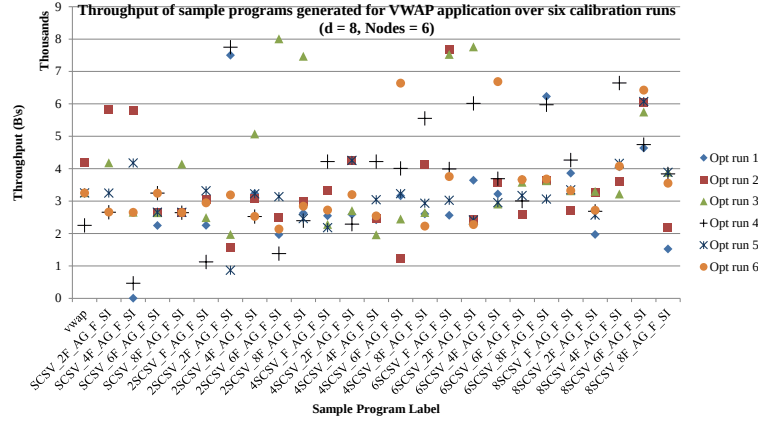


Figure 8.3: Throughput of sample applications produced for VWAP application for six calibration runs.

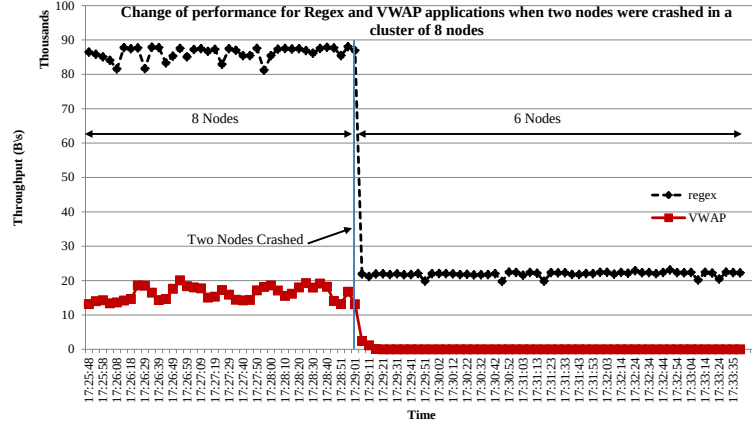


Figure 8.4: How the sample program performance change when two nodes were crashed.

8.6.3 Evaluation of Performance Recovery Process

The VWAP, Regex, and Apnoea applications were calibrated by Albatross for three times prior the experiment for each node configuration (i.e., 8 nodes, 6 nodes, etc.). After the calibrations were complete, the three applications were submitted to Albatross. Albatross and System S were run in the cluster A, while StreamFarm was ran on cluster B to avoid potential interferences. StreamFarm allowed us to maintain a steady, high data rates throughout the experiments which gave us great support compared to file based methods that we used in Chapters 6 and 7 [165].

While in the middle of the experiment run, two nodes were crashed. The resulted throughput curves are shown in Figures 8.5 and 8.6. We observed that for Regex application (See Figure 8.5) the new sample program introduced for 6 nodes environment (SCSV_4F_8F_4F_4SI) was unable to produce a higher throughput compared to ini-

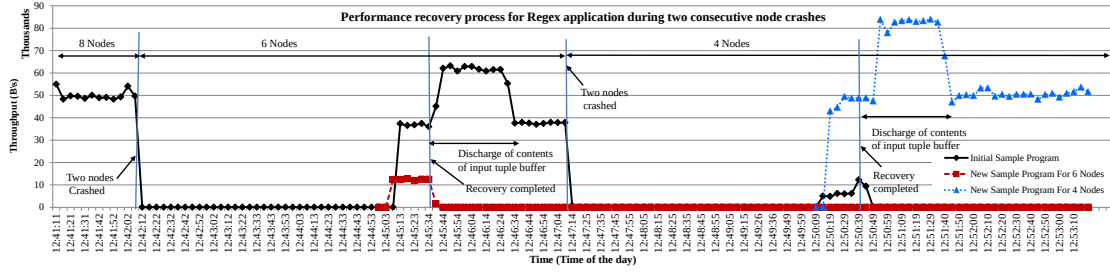


Figure 8.5: Recovering performance of Regex application during two consecutive node crashes.

tial sample application (4SCSV_4F_4F_2F_2SI) hence its execution was canceled. The previous sample application (4SCSV_4F_4F_2F_2SI) was run in the environment since Albatross could not find a better version of the Regex sample applications to run, yet we get only partial degradation of the Regex stream job's performance despite loss of two nodes. However, in the case of VWAP application (see Figure 8.6 (a)) Albatross's choice was much more accurate and performance of the new sample application (8SCSV_48F_AG_F_SI) resembled almost the same performance of initial sample application (8SCSV_64F_AG_F_SI). Note that Regex application consists of a chain of Functors whereas VWAP application has an Aggregate operator that outputs data after gathering a group of tuples. Hence the wavy curve of VWAP application compared to Regex application is formed by the aggregation operation done by VWAP. The temporary high throughput rise in both the graphs are due to the excess tuples during the release of buffered tuples just after completion of recovery process. For Apnoea, the new sample application (SCSV_F_AG_F_F_4AG_4F_F_AG_F_J_J_SI) was able to restore the performance (with a partial degradation) while the initial sample application could not produce any output.

During the rest of the experiment we used only Regex and Apnoea applications. We used the full functionality of Albatross as with the previous experiment. After the first crash we allowed sufficient time for the recovery process and then crashed another two nodes. The results are shown in Figure 8.5 and in Figure 8.6 (b). After the second crash we observed that for the Regex application, the initial sample program (4SCSV_4F_4F_2F_2SI) was unable to restore the performance with 4 nodes and the job was undertaken by (4SCSV_8F_4F_4F_4SI) which was able to produce almost the same throughput of the initial program with 8 nodes. In the case of Apnoea application, the new sample application (SCSV_F_AG_F_2F_2AG_2F_F_AG_F_J_J_SI) could not restore the performance. Hence the previous sample application was deployed to maintain performance with 4 nodes cluster.

8.6.4 Evaluation of Single Point of Failure Recovery Process

In a separate experiment with 6 nodes, we crashed the node with the main module. The resulting data rate behavior of the Apnoea application is shown in Figure 8.7. It

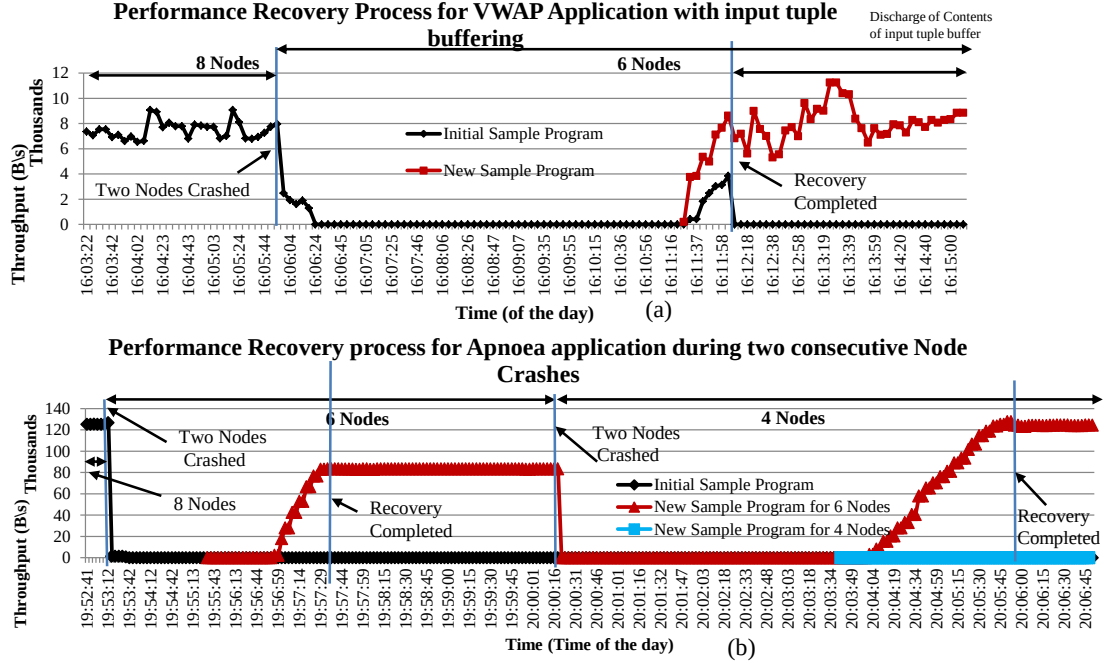


Figure 8.6: Performance recovery process when (a) two nodes were crashed for VWAP (b) two consecutive node crashes for Apnoea

can be seen that the data rate dropped suddenly after the crash and Albatross started rescheduling process. After about 6 minutes the Apnoea application is deployed again in the node cluster and the data rate becomes stable in about 1.5 minutes later. This indicates that Albatross is able to recover from such single point of failures.

8.7 Discussion

From the evaluation results it was clear that Albatross is able to restore the operations back to the normal level (e.g., VWAP on 6 nodes, Regex on 4 nodes, and Apnoea on 4 nodes) or at least run the jobs with relatively less performance yet with a guarantee of the correctness of the execution (e.g., Regex on 6 nodes, and Apnoea on 6 nodes). Furthermore, Albatross was able to restore performance even when half of the nodes in the cluster were crashed (e.g., Apnoea and Regex each on 4 nodes).

This chapter illustrated that automatic code generation is an effective approach for maintaining performance of stream processing jobs. We believe this was successfully demonstrated in the previous section through the evaluation results. However, there are many further work and limitations of the current prototype. One of the limitations is the types of input stream applications that the current prototype can support is limited since Albatross's grammar represents a subset of the constructs of the SPADE

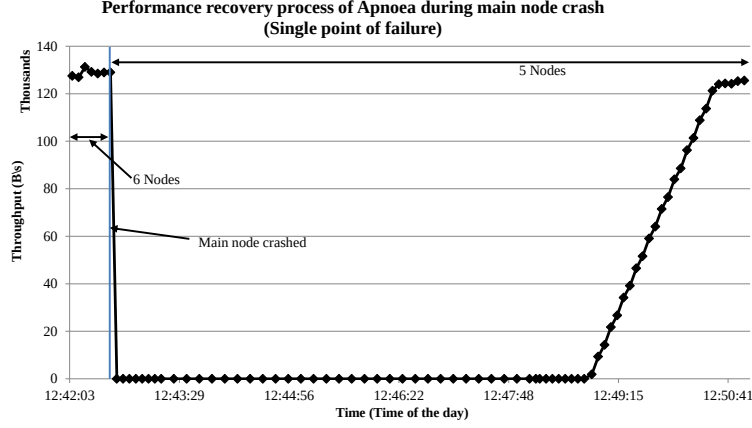


Figure 8.7: Output data rate variation of Apnoea during the recovery process when the main node has been crashed.

language. Current version of Albatross discards the state of the operators during a crash recovery. This is because SPADE language does not support state retrieval/injection of BIOPs. Only UDOPs have such facility. Furthermore, data rate might not produce the correct picture of performance of certain stream applications. E.g., An application that aggregates tuples may emit one tuple per minute irrespective of the number of nodes that serve data for the aggregate operator. Moreover, it takes considerable amount of time (Rounded average recovery times: 6 minutes for VWAP, 3 minutes for Regex application, and 5 minutes for Apnoea) to restore the normal operations of all the stream jobs. Most of this time is spent for orchestrating the System S runtime (start/stop System S runtime, reschedule PEs) which accounts for significant time compared to Albatross's scheduling algorithm. While the time period might be different for some other stream processing system, we hope to investigate on how to reduce this time period. The release of tuples buffered by Albatross temporarily increases the data rate which might not be expected by some applications which receive data from System S jobs.

It should be noted that we do not conduct profiling of the stream applications for each and every node combination. Profiling of the stream applications are conducted only on several node combinations. E.g., If there are 8 nodes in the cluster, we may conduct application profiling for 8 nodes, 6 nodes, and 4 nodes. The profiling results conducted on closely related node configuration is used for selecting the sample application for performance maintenance. E.g., In the case of having only 3 nodes on the node cluster, 4 nodes configuration is used. However, to improve the possibility of performance maintenance using the selected sample application, we profile the selected sample application along with the previous sample application for a short time in the $E_{N'}$ (with N' nodes) for a time window of W_t as described in Section 8.2.

8.8 Conclusions

This chapter described a technique for maintaining performance of data stream applications during crash failures. The approach is widely applicable for resource limited stream processing clusters and is based on automatic code generation. The chapter presented Albatross: a python based middleware that monitors the status of the node clusters, and strives to maintain the performance via swapping the sample programs generated for each input program. It was observed that Albatross can maintain the same performance of the Regex (with 8 nodes, 4 nodes crashed), VWAP (with 8 nodes, 2 nodes crashed), and Apnoea (with 8 nodes, 4 nodes crashed) stream jobs despite loss of nodes from the stream processing environment. Therefore, we came to conclusion that our approach is capable of maintaining stream program performance even if 50% of the nodes in the cluster has been crashed. Furthermore, we implemented a single point of failure recovery algorithm on Albatross by using a Zookeeper service. It was observed that with this extension Albatross was able to recover from single point of failures. Furthermore, we incorporated a stream load shedding mechanism along with automatic code generation to maintain the stream program performance during drastic failures which reduces system resources considerably.

8.9 Summary

Crash failures introduce challenges for maintaining large scale data analysis system performance for resource limited compute clusters. A solution based on the techniques of automatic code generation was presented in this chapter. Background information on performance recovery was described in Section 8.1. The methodology followed for crash recovery was described in Section 8.2. Stream load shedding to recover from drastic node failures was described in Section 8.3. Implementation details of Albatross which implements the proposed approach was given in Section 8.4. Approach followed for recovering from single point of failure vulnerability on Albatross was presented in Section 8.5. Experimental evaluation was presented in Section 8.6 while a discussion of the experiment results was presented in Section 8.7. Conclusions were presented in section 8.8. Next chapter starts Part II of this thesis and the chapter describes XGDBench a benchmark for graph database systems.

Part II

Graph Processing Optimization

Benchmarking Graph Database Systems

Graph databases provide a means of conducting online graph analysis and are specially intended for OLTP style analysis. This chapter describes XGDBench which is a graph database benchmark introduced by this thesis [305]. Section 9.1 provides an introduction to graph database benchmarking. Section 9.2 presents a background for the rest of the chapter. It describes Yahoo! Cloud Serving Benchmark, Synthetic Graph Generators, and Multiplicative Attribute Graphs in subsections 9.2.1 to 9.2.3. Methodology of XGDBench is explained in Section 9.3 and the requirements for the design of the benchmark is provided in Section 9.4. Implementation details of XGDBench is described in Section 9.5. Ensuring reproducibility of the benchmarking sessions is described in Section 9.6. Details on modeling workload spikes in graph database systems is presented in Section 9.7. Evaluation of the benchmark is provided in Section 9.8. The chapter investigates the graph database performance for graph traversal operations using XGDBench. The chapter uses AllegroGraph, Fuseki, Neo4j, OrientDB, and Titan as example databases for this purpose. A discussion of the benchmark as well as its limitations are listed in Section 9.9. Conclusions arrived from the experiment results are listed on Section 9.10.

9.1 Introduction

Cloud computing is an emerging infrastructure paradigm that eliminates the need for maintaining expensive computing hardware and software resources. Cloud computing provides many advantages for the deployment of data-intensive applications such as elasticity of resources, low time to market, pay-per-use cost model, and the perception of unlimited resources and infinite scalability [306][301]. These advantages enable deployment of novel applications which were not economically feasible in traditional enterprise infrastructures. Multiple cloud services which offer data storage in the form of graphs

have started their operations in recent times. NuvolaBase [307], Dydra [308], Cloud-Graph [309], Heruko [270], etc. are some example services that operate on SaaS/PaaS model and provide ability of hosting graph database instances in the cloud.

Current graph database benchmarks such as HPC Scalable Graph Analysis Benchmark focus on some core network analysis features of graph databases. However, the application scenarios modeled by these benchmarks do not realistically model real world applications that involve attribute (i.e., property) graphs. Many benchmarks have been developed by Semantic Web community such as LUBM [116], SP2Bench [119], DBPedia [118] that could be also used to benchmark graph databases. However, these benchmarks are not smoothly scalable since they do not follow a statistical model.

Exascale technology will revolutionize future cloud computing paradigm by creating power efficient cloud computing systems that offer huge performance per watt values. Completely new programming models and programming techniques need to be used when migrating the existing performance modeling and benchmarking infrastructures to exascale clouds because such systems will include heterogeneous hardware such as General-Purpose Graphics Processing Units (GPGPUs), multi-core/many-core CPUs, etc. PGAS languages is a candidate programming technology which could be used to program such systems. As mentioned in Section 3.3.1, X10 [200] is a new programming language that follows Asynchronous Partitioned Global Address Space (APGAS) model. Partitioned Global Address Space (PGAS) languages are potential candidates for programming Exascale clouds. This chapter demonstrates the applicability of X10 programming language for benchmarking activities.

There is little work being done on benchmarking graph data stores on cloud and less effort have been made to investigate the key performance implications and performance characteristics of graph databases on cloud. Especially increasing numbers of cloud computing platforms with different architectures such as public, private, and hybrid clouds offer large amounts of possible choices for deploying graph databases. Understanding the performance of graph database servers is of utmost importance because they are the main contributors for future online transaction processing (OLTP) type graph queries [212]. XGDBench is an extensible benchmarking platform that is aimed for addressing the aforementioned issues [305] which promotes automated testing and benchmarking of cloud graph databases. This essentially leads to efficient characterization of graph database performance that results in a reduction of implementation/deployment costs as well as time to market. XGDBench is based on *Multiplicative Attribute Graph Model* (MAG) [109] which is a synthetic graph model for attribute graphs. Furthermore, a multi-threaded version of XGDBench client was implemented that exploits multi-core systems' capabilities to improve time taken for graph generation as well as to direct larger workloads to the benchmarked databases.

9.2 Background for Graph Database Benchmarking

As mentioned in the previous Section there is an increasing trend in deployment of online graph database services in public cloud computing infrastructures. We describe how to

automate the process of graph database benchmarking in cloud computing systems in a way that it leads to better performance evaluation of graph databases on future exascale clouds. This section describes several technologies and frameworks on which XGDBench builds-up on. The Yahoo! Cloud Serving Benchmark (YCSB) and its intended application scenario are described first. Then some popular synthetic graph models that are currently used for benchmarking graph processing systems are illustrated. Next, the chapter explains Multiplicative Attribute Graph (MAG) model that is used as the synthetic graph data generator model of XGDBench.

9.2.1 Yahoo! Cloud Serving Benchmark

YCSB [310] framework was released by Yahoo! Inc. in 2010, with the motivation of creating a standard benchmark and benchmarking framework to assist evaluation of cloud serving systems (These systems provide online read/write access to data). In their introductory paper on YCSB, Cooper *et al.* defined a core set of benchmarks and reported results for four widely used cloud data serving systems: Cassandra, HBase, Yahoo! PNUTS, and a MySQL system.

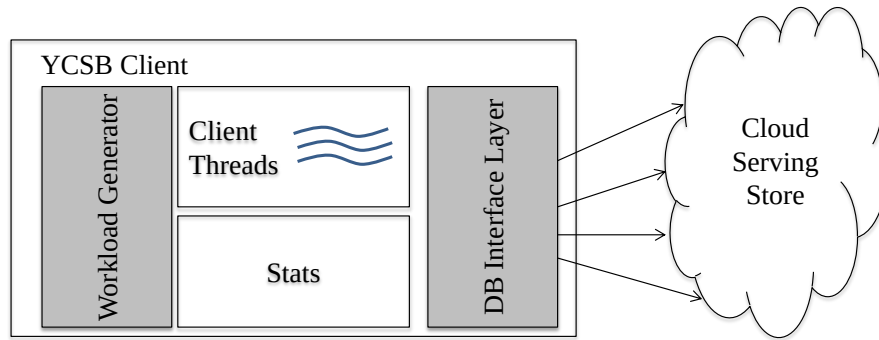


Figure 9.1: Architecture of YCSB Client.

One of the key goals of YCSB is its extensibility. The framework is composed of a workload generator client (the architecture is shown in Figure 9.1) and a package of standard workloads that cover interesting parts of the performance space. The workloads are composed of one of the operations: Insert (Insert a new record), Update (Update a record via replacing the value of one field), Read (read a record either one randomly chosen field or all fields), Scan (scans record in order, starting at a randomly chosen record key.) The workload generator of YCSB supports definition of new workload types which motivated for following YCSB's approach for implementing XGDBench. The random choices made by the workload generator client is controlled by using a random distribution. YCSB has several built-in distributions such as Uniform, Zipfian, Latest, Multinomial, etc. In all the experiments described in this chapter on XGDBench we used Zipfian distribution since it resembles a general characteristics found in real world graph data sets (Power-law characteristics). When choosing records according to Zipfian distribution some records will be extremely popular while most records will be

unpopular.

9.2.2 Synthetic Graph Generators

As mentioned by Chakrabarti *et al.* [311] graph models can be classified into five categories such as, Random graph models (e.g., Erdős Rényi), Preferential attachment models (e.g., Barabasi-Albert model), Optimization-based models (e.g., Highly Optimized Tolerance model), Tensor-based models (R-MAT), and Internet-specific models (e.g., Inet). In random graph models the graphs are generated by a random process. Preferential attachment models are based on the notion of “rich” get “richer” as the network grows. The optimization-based models are based on the concept that power-laws are shown to evolve when risks are minimized using limited resources. Tensor based models generate realistic graphs by utilizing self-similar mechanisms through tensor multiplication. Internet-specific models are special-purpose graph generators that model special features of the Internet graph. Most of these are hybrids of the aforementioned categories.

Each generator model has its own pros/cons and the best generator model depends on the application area. Out of these models R-MAT model has been used as a data generator for Graph 500. R-MAT (Recursive MATrix) generates graphs by recursively traversing the adjacency matrix of a graph without any edges in recursive manner [312]. Graphs generated by R-MAT depends on six parameters $scale(n), a, b, c, d$, and E . The parameters a, b, c , and d are floating point values whose sum is 1. The number of vertices in an R-MAT graph is set to 2^n . The graph generation starts with an empty adjacency matrix, and the process is conducted E times. Each round, one of the four partitions is chosen with probabilities a, b, c , and d . The chosen partition is again divided into four smaller partitions. This procedure is repeated until a simple cell is reached. The nodes corresponding to the cell are linked by an edge in the graph.

9.2.3 An Overview of Multiplicative Attribute Graphs

Multiplicative Attribute Graph (MAG) is an approach for modeling the structure of networks which have node attributes [109]. MAG naturally models the interactions between the network structure and the node attributes. Compared to Kronecker graphs (used in Graph 500) that follow R-MAT model, MAG model creates realistic attribute graphs which are much suited for benchmarking graph databases. It has been proven that MAG generates graphs with both analytically tractable and statistically interesting properties [109].

A schematic representation of general MAG algorithms is shown in Figure 9.2. The Figure 9.2 shows two vertices v , and u each having a vector of n categorical attributes and each attribute have a cardinality d_i for $i = 1, 2, \dots, n$. There are also n matrices denoted by Θ_i , $\Theta_i \in d_i \times d_i$ for $i = 1, 2, \dots, n$. Each entry of Θ_i is the affinity of a real value between 0 and 1. Values α , β , and γ are floating point values between 0 and 1. Given these conditions, probability of an edge (u, v) , $P[u, v]$, is defined as the multiplication of

affinities corresponding to individual attributes as shown in Figure 9.2. Here $a_i(u)$ and $a_i(v)$ represent the value of i -th attribute of nodes u and v .

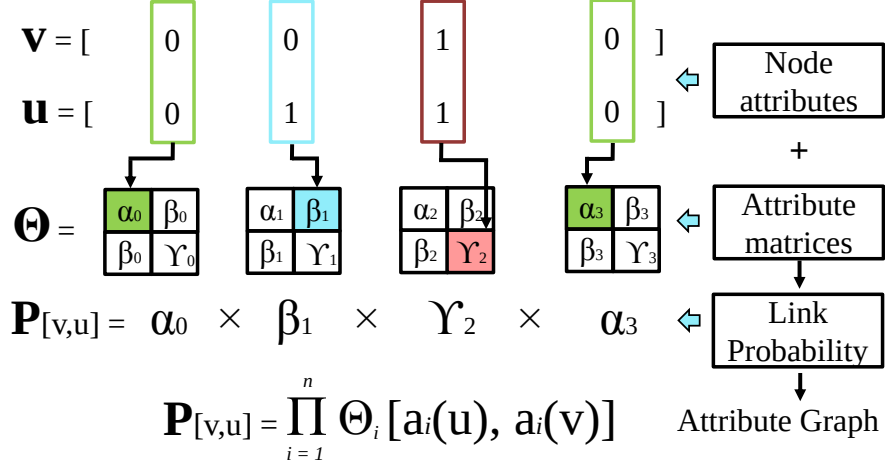


Figure 9.2: Multiplicative Attribute Graphs (MAG) model

However, the MAG algorithm used in XGDBench generator is a simplified version of the model shown in Figure 9.2. The simplification is done by considering undirected version of the model by making each Θ_i symmetric. The node attribute values are made binary (e.g., As shown in Figure 9.3 the attributes may represent yes/no answers received for a question asked by each member of a social network). This makes the Θ_i to be a 2×2 matrix.

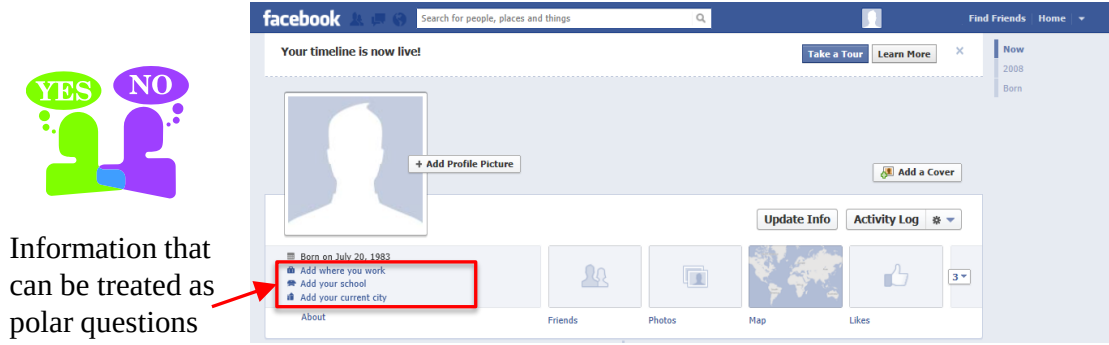


Figure 9.3: An example for attributes with binary values (yes/no) from Facebook online social network.

Also it is assumed that all the affinity matrices are equal (i.e., $\Theta_i = \Theta$) to further reduce the number of parameters [109]. The implementation of the MAG algorithm made by this thesis is shown in Algorithm 1. It accepts number of vertices of the generated graph ($nVertices$), number of attributes per vertex ($nAttributes$), a threshold value for random initialization of attributes ($attribThresh$), an edge affinity threshold value that

determines whether there is an edge between two vertices, and an affinity matrix (*theta*) corresponding to the Θ mentioned above. The `randZeroOrOne()` function in Algorithm 1 constructs a zero initialized matrix of size $nVertices \times nAttributes$. Then the matrix is populated with 1s using randomly generated values if they exceeds the *attribThresh*. This matrix is returned to Algorithm 1's *nAtt* variable. The key feature why MAG model is much suited for graph database benchmarking is that the probability of an edge between pairs of vertices is controlled by the product of individual attribute-attribute affinities. Most of the current graph databases are made to store not only vertices and their relationships, but also attributes of these vertices and relationships. Therefore, MAG is a more natural synthetic graph model suitable for benchmarking graph databases.

Algorithm 1 `mag(nVertices, nAttributes, attribThresh, pThresh, theta)`

```

1: nAtt  $\leftarrow$  randZeroOrOne(nVertices, nAttributes,
                           attribThreshold)
2: result  $\leftarrow$  ones(nVertices, nVertices)
3: for i  $\leftarrow$  0 to nVertices do
4:   for j  $\leftarrow$  0 to nVertices do
5:     for k  $\leftarrow$  0 to nAttribs do
6:       if nAtt[i,k] = nAtt[j,k] then
7:         if nAtt[i,k] = 0 then
8:           result[i,j] = result[i,j] * theta[0]
9:         else
10:          result[i,j] = result[i,j] * theta[3]
11:        end if
12:      else
13:        if nAtt[i,k] = 0 and nAtt[j,k] = 1 then
14:          result[i,j] = result[i,j] * theta[1]
15:        else if nAtt[i,k] = 1 and nAtt[j,k] = 0 then
16:          result[i,j] = result[i,j] * theta[2]
17:        end if
18:      end if
19:    end for
20:  end for
21: end for
22: for i  $\leftarrow$  0 to nVertices do
23:   for j  $\leftarrow$  0 to nVertices do
24:     if result[i,j] > pThresh then
25:       result[i,j] = 1
26:     else
27:       result[i,j] = 0
28:     end if
29:   end for
30: end for
31: return (result)

```

9.3 Methodology of XGDBench

Almost every software benchmark has been developed around a real world application scenario of the software system that it intends for benchmarking [313]. XGDBench has been developed focusing on a graph database application for social networking services. It is because Online Social Networks (OSNs) is one of the rapidly growing areas of data analytics that generates massive graphs. Data storage and analysis of such online social networks are conducted in cloud infrastructures [314]. Therefore OSNs represent a general representative application of graph databases [108].

It is a common phenomenon in social networks, that people with similar interests (i.e., attributes) are more likely to become friends in real world. For example, if person A and person B went to same high school, and both of them graduated in the same year, there is a higher probability that they are friends in the real world, as well as in the social network service than compared to a person C who did not go to the same high school. The fact that people went to the same high school or people graduated in a particular year can be represented as questions with binary answers (yes/no) which can be represented as attribute vectors shown in Figure 9.2.

9.4 Requirements of XGDBench

This section describes the performance aspects that are specifically targeted by XGDBench. These performance aspects are represented by individual operations. These individual operations (which are listed in Table 9.1) gets intermixed according to some predefined proportions to create workloads.

9.4.1 Attribute Read/Update

Graph databases in exascale clouds will have to handle massive graphs online and they will partially load the graph into memory. The workloads will include both read/update operations. However, in most of the future exascale applications the Read operations will dominate the workload [315]. Therefore, it was decided to include a read-heavy (e.g., a workload with 0.95 probability of read operation and 0.05 probability of write operation [310]) and read-only (having only read operations) workloads with XGDBench.

Graphs need to be updated online. In a typical OSN a node represents a user and an edge represents friendship/relationship. Properties of nodes/edges include messages, photos etc. The friendship graph of OSNs change at a slower rate compared to their properties. Therefore, performance of attribute update operation is critical compared to node/edge update. Therefore, it was decided to include an Update Heavy workload with XGDBench.

Moreover, the benchmarking platform needs to be scalable to store data in-memory for update operations. This will eliminate unexpected delays involved in reading large data from secondary storage.

9.4.2 Graph Traversal

Unlike other database types graph databases have the unique property of having data encoded in their graph structures. These information could only be obtained by traversing the graph. Therefore, the benchmark should have support for evaluating the performance of graph traversal operations. While there are a variety of graph traversal techniques, it was decided to use an algorithm that will be most frequently executed against the graph database. This is because it is important to check the performance of frequently used operations than operations that ran infrequently which does not have requirements for real-time execution. A scenario of listing friends of friends was selected which is one of the frequently used traversal operations in OSNs. This includes execution of BFS (breadth-first search) from a particular vertex for detecting the connected component of a graph. Breadth-first search traverses a graph in a level wise manner. Before visiting the vertices at path length $(k+1)$, the traverser first needs to visit all the vertices within path length k [316]. BFS can be also considered as layers or waves growing outwards from a given vertex. The vertices in the first wave are the immediate neighbors of the starting vertex and they have distance 1. The neighbors of those neighbors have distance 2, etc [59].

Note that most real world graphs are irregular data structures [317]. Therefore, it is possible for starting the traversal from a vertex that is heavily connected with the other vertices as well as starting the traversal from an unconnected vertex.

Based on the aforementioned requirements the following set of basic operations can be defined on a graph database (Shown in Table 9.1). We conjecture that these basic operations are sufficient for defining many workloads that are frequently present in graph databases.

Table 9.1: Basic operations of graph databases.

Operation	Description
Read	Reads a vertex and its properties
Insert	Inserts a new vertex
Update	Updates all the attributes of a vertex
Delete	Deletes a vertex from the DB
Scan	Loads the list of neighbors of a vertex
Traverse	Traverses the Graph from a given vertex using BFS. This represents finding friends of a person in social networks.

9.5 Implementation

Implementation details of XGDBench is shown in Figure 9.4. XGDBench Client is the software application that is used to execute the benchmark's workloads. Its main components are Graph Generator, Graph Data Structure, Workload Executor, Graph DB Workload, and Graph DB Interface Layer. The XGDBench client is written in managed X10. The entire XGDBench client was compiled using X10 compiler (x10c) and the benchmarking sessions were run using the X10 interpreter (x10). XGDBench Client accepts a collection of input parameters that are used during the benchmarking process. Each of these parameters are described in the below sub sections.

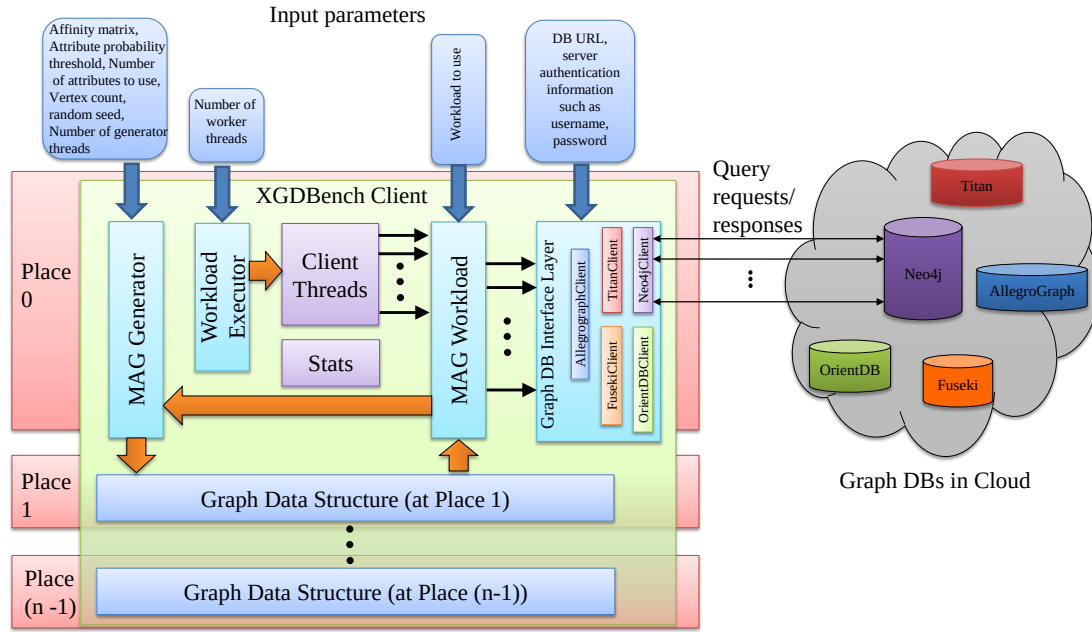


Figure 9.4: Architecture of XGDBench Client

XGDBench has two phases of execution called *loading phase* and *transaction phase*. The loading phase generates an attribute graph by using the MAG algorithm shown in Algorithm 1. The transaction phase of XGDBench calls a method in *CoreWorkload* called *doTransaction()*, which invokes the basic operations such as database read, update, insert, scan, and traverse. We have implemented the workloads that satisfies the requirements stated in Section 9.4 on XGDBench and these workloads are listed in Table 9.2. We use these workloads on five graph database servers and report performance in Section 9.8.

We use throughput (operations per second), latency (milliseconds), runtime (milliseconds) as the performance metrics in XGDBench. Furthermore, XGDBench can be configured to output a histogram of latencies for each operation.

Table 9.2: Core workloads of XGDBench.

A : Update heavy
Workload A is a mix of 50/50 read/update workload. Read operations query a vertex V and reads all the attributes of V . Update operation changes the last login time attribute of the vertices. Attributes related to vertex affinity are not changed.
B : Read mostly
A mix of 95/5 read/update workload. Read/update operations are similar to A.
C : Read only
Consists of 100% read operations. The read operations are similar to A.
D : Read latest
This workload inserts new vertices to the graph. The inserts are made in such a way that the power-law relations of the original graph are preserved.
E : Short Range Scan
This workload reads all the neighbor vertices and their attributes of a Vertex A. This represents the scenario of loading the friend list of person A onto an application.
F : Traverse heavy
Consists of 45/55 mix of traverse/read operations.
G : Traverse Only
Consists of 100% traverse operations.

9.5.1 Graph Generator

XGDBench Client consists of a graph data generator (MAG Generator shown in Figure 9.4) for generating the data to be loaded to the database. The workload generator is implemented using Multiplicative Attribute Graphs (MAG) model [109] as described in the previous section. As can be observed in the line 1 of Algorithm 1, the graph generator accepts an attribute matrix that is initialized with random attribute values (either 0 or 1). To ensure the repeatability of the benchmarking experiments, the attribute matrix needs to be initialized with the same attribute values across different benchmarking experiments that contain the same set of input parameters. The details of ensuring this property is given in Section 9.6.

The initial graph generator implementation [305] was a single threaded one. Although the graph generation time is not reflected on the benchmarking results, it acted as a bottleneck when working with large graphs. To solve this issue we implemented a multi-threaded version of the graph generator that generates large graphs on multi-core systems faster. In the improved implementation, user can specify the number of threads to be used in the generator. Then each generator thread is assigned a portion of the attribute matrix to operate on which reduced the total time taken during the graph generation process.

We use the DistArray of X10 to implement the distributed graph data structure of XGDBench Client. This data structure is useful for storing very large graphs that cannot be stored on a single node's memory. By default the vertex, edge information are stored in Place 0 (Place 0 runs on the node that invoked the XGDBench client.) and when the graph grows exceeding the pre-specified vertex count per place, the excess vertices are transferred to the next place. In Figure 9.4 the data structure is shown distributed across n places.

9.5.2 Workload Executor and Graph DB Workload

The Workload Executor initializes multiple client threads which invoke operation sequences according to the workloads it handles. A sequential series of operations are executed by each client thread. Graph database interface layer translates these simple requests from client threads into calls against the graph database. Unlike its predecessor (YCSB), XGDBench faces a problem when implementing the multi-threaded workload execution. This is because each thread needs to access the same generator object to get its next vertex/edge information. However, in YCSB there was no such requirement for querying a single object for information because the operations invoked did not have relationships like edges in graphs. Currently only the code that obtains the next vertex/edge information is synchronized from the generator which solves this problem.

Graph DB Workload (MAG Workload in Figure 9.4) is a component that represents a workload that can be invoked on the Graph DB Interface Layer. It wraps-up the workload's properties that are specified in the property files as well as command line arguments. Furthermore, it acts as the bridge between the client threads and the graph generator. The Graph DB Workload component also forwards each operations invoked

by the client threads to the Graph DB Interface Layer.

9.5.3 Graph DB Interface Layer

Graph DB interface layer consists of interfaces for different graph databases. Most of the current graph database servers have their own optimized query interfaces. E.g., RexPro [318] is a binary protocol that can be utilized to send Gremlin scripts to remote Rexster instances. However, it was decided to use common protocols such as HTTP/REST for implementing the Graph DB Layer because it enables for doing fair comparison of different systems. Furthermore, there were limitations of the HTTP/REST interfaces of certain graph database servers that made us to use some alternatives in combination in HTTP/REST interfaces to implement the required functions. E.g., The Rexster server 2.1.0 used for Titan graph database server threw an error when POSTing edges through HTTP/REST interface which made us to use the Rexster's Gremlin interface to conduct edge insertion.

In current XGDBench implementation the focus is on benchmarking graph database servers. The reason is XGDBench is a benchmarking framework rather than a benchmark specification. If XGDBench was a benchmark specification the specified benchmark operations have to be implemented in the target graph database using its query language which could be either an embedded graph database or a standalone graph database server. Such approach can be categorized as rather a white box approach because the developer can implement the benchmark specification in the way he/she wants. However, XGDBench treats the graph database server completely as a black box which makes the benchmarking process and the workloads executed on the graph databases work more similar to a graph based application communicating with a standalone graph database server. Nevertheless, if a user wants to benchmark an embedded graph database with XGDBench that is completely doable with the current implementation. In such scenario the graph database client will create an embedded graph database server instance within the same JVM instance (Note that managed X10 is interpreted by the System's JVM). But the benchmarking result will be interfered by the benchmarking software itself. This is another reason for why not to use XGDBench for benchmarking embedded graph databases.

9.5.4 Implementation of Traversal Operation

Traversal operation of the XGDBench was implemented by creating BFS traversal for finding friends of friends scenario for each graph database client. The BFS traversal operation is conducted up to only two hops from a randomly chosen starting point. This is because many of the social network users are interested of finding their friends' information as well as the friends of friends' information. It is rare that users go beyond this two hop traversal (Some social network services even do not allow such operation). Here "two hop traversal" means visiting a vertice's all neighbors and neighbors of neighbors by traversing the graph. Social networking happens among peers not among strangers [319].

9.6 Ensuring Reproducibility of Results Across Benchmarking Sessions

The code for implementation of traverse operation on Titan Client is shown in Figure 9.5. The `traverse()` method on each graph database client accepts a starting vertex and the number of levels to be traversed. The output from the traversal is a Vector object that contains all the discovered vertices.

9.5.5 Implementation of Insert and Update Operations

The update operations on the graph data preserves the power-law distribution that is present in the original graph created by MAG because the update operations are conducted only on attributes that are not related to calculation of probability of an edge. Furthermore, the insert operations of the vertices are done during the workload executions in such a way that it preserves the power-law structure. While it is rare to find graph database applications that execute 100% traverse operations, the workload G was created as a complete traverse only workload because traverse operation is one of the inherent key features of a graph database server, and it is important to compare graph traverse operation performance of graph databases.

9.6 Ensuring Reproducibility of Results Across Benchmarking Sessions

The workload generator of XGDBench is based on Multiplicative Attribute Graphs. Since this model is a stochastic probabilistic model there is a tendency for production of graphs with different numbers of edges (Even though there is a fixed vertex count) if a true random number generator was employed. This is because the edge probabilities that are determined by MAG algorithm are random due to the random initialization of the vertex attributes at the beginning of the graph generation process [305] (The first line of Algorithm 1 shows how the random initialization of the node attributes happen.). However, this issue can be easily overcome through use of Pseudo-random number generators present with X10 (The same feature is available with Java using `java.util.Random` class). Such number generators need an initial integer seed value which determines the sequence of integers generated by the random number generator subsequently. We have implemented the facility of specifying an integer seed at the XGDBench client's commandline interface so that users can repeatedly generate the same property graph. We ran several experiments with specifying a seed value of 100 and found that the MAG generator produces the same property graph in all of the experiments.

9.6 Ensuring Reproducibility of Results Across Benchmarking Sessions

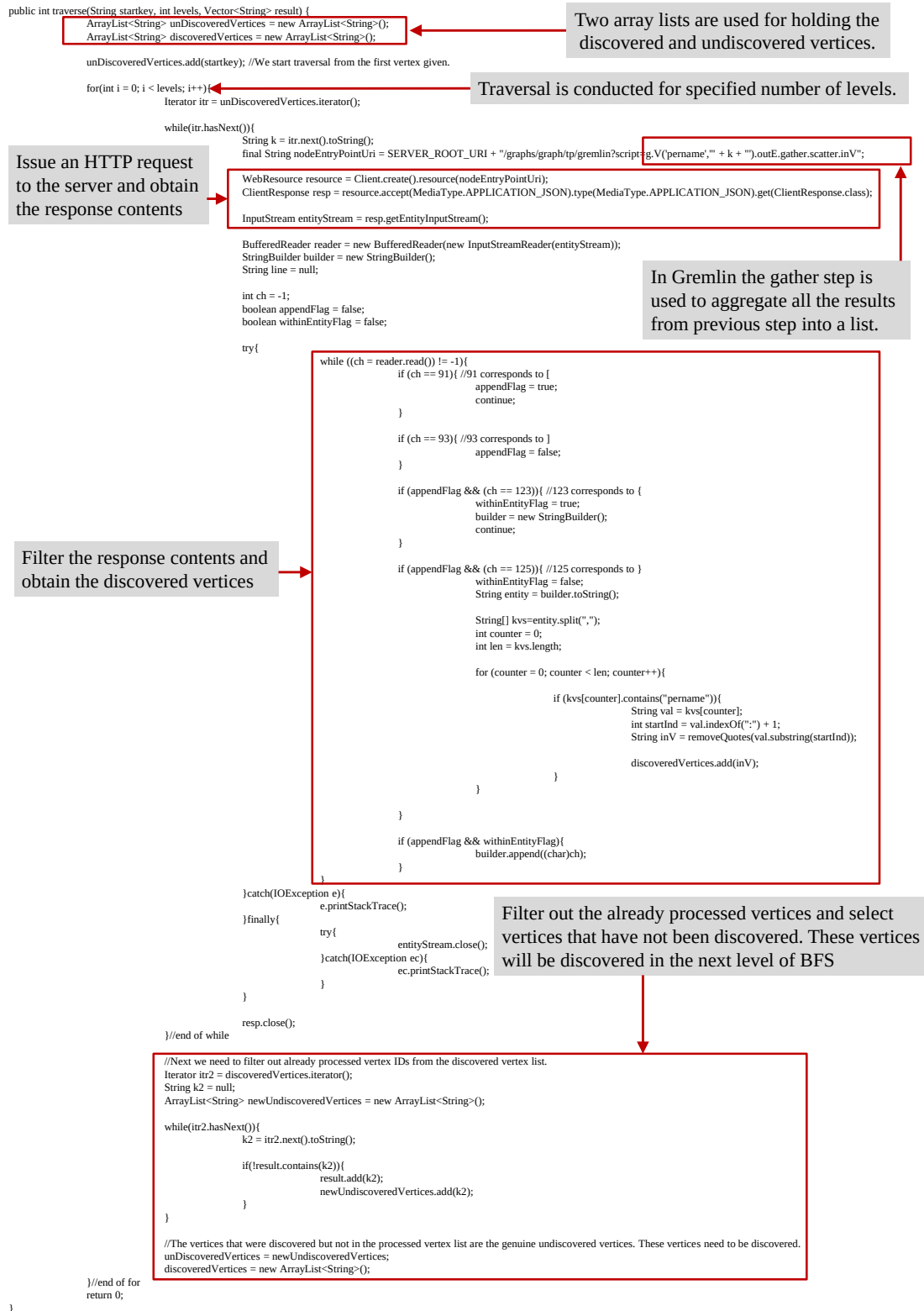


Figure 9.5: Implementation of traverse operation on Titan client

9.7 Modeling Workload Spikes in Graph Database Systems

Large scale data analysis systems need to be optimized for facing workload spikes because sudden spikes could occur any time during a system's life cycle. The system designers need to have a means of testing their data analysis systems beforehand to face spikeful situations. Modeling of spikeful situations is often complex for benchmarks because of the need for generating a huge aggregated volume of requests to mimic a real world spikeful situation is very challenging task. This section explains how spikeful workloads are modeled in XGDBench.

As mentioned in the Section 9.1 the field of graph databases is still relatively young and it lacks standards for benchmarking and performance comparisons [110]. The popular graph database benchmarks such as HPC Scalable Graph Analysis Benchmark are intended for exploring the graph structural properties of graph databases. However, most of the current graph databases implement property graph data model (i.e., attribute graphs model). Furthermore, the properties of vertices and edges of the graph data generally reflects the relationships between the vertices. Therefore, better synthetic graph data models are required for modeling graphs on such graph database systems.

9.7.1 Spike Generation Methodology

As other compute systems such as web servers, relational database servers, etc.; graph database systems are susceptible for workload spikes. Such spikes are becoming quite frequent in famous websites such as Facebook, Twitter, Wikipedia, etc. Out of the different spike types, "Volume Spike" refers to an unexpected sustained increase in aggregate workload volume [320]. Another important category of spikes is called a "Data Spike" which is an unexpected increase in demand for certain data objects.

The original design of XGDBench did not have support for modeling workload spikes [305]. This is because the XGDBench client just accepted the number of threads (e.g., N) to be used during a benchmarking session and then each thread executed portion of the total number of operations that need to be executed on the database (i.e., $(\text{Total number of operations})/N$). By changing the number of client threads dynamically, a volume spike can be emulated during a benchmarking session. We dynamically orchestrate places in different nodes to achieve this aim. The methodology used for generating workload spikes on XGDBench is shown in Figure 9.6.

The Figure 9.6 (a) shows how a volume spike is generated. The client activities running at each place generates the same graph structure (shown in blue, black, and red colors) but with different properties (i.e., Vertex attribute labels that do not correspond to graph generation such as vertex name). During a volume spike, each activity directs requests to its corresponding subgraph located on the graph database server. When XGDBench is run with a large number of places it will stress all the components of the graph database server (especially in the case of distributed servers such as Titan).

How a data spike is generated is shown in Figure 9.6 (b). In this case all the activities

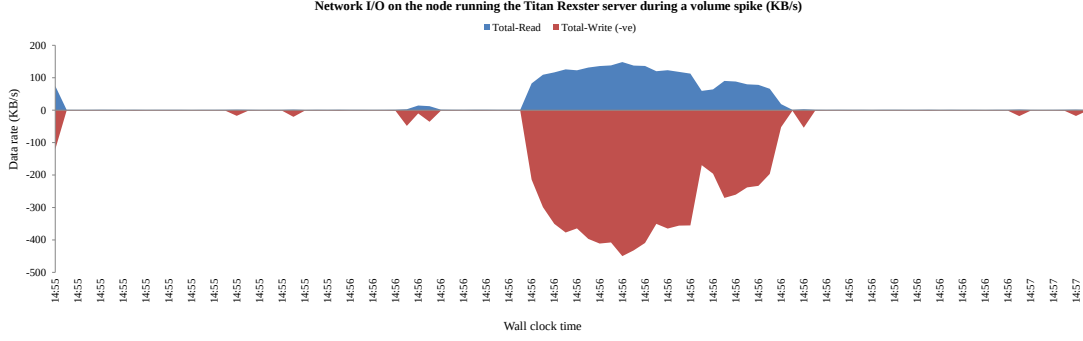


Figure 9.6: Methodology used for workload spike generation on XGDBench. (a) Scenario of volume spike generation (b) Scenario of data spike generation.

running in different places generated the same graph structures with same property values. However, during simulation of a data spike, the query requests for the graph database server will be focused only onto a particular subgraph of the entire graph database server. This is because each activity has the same content for formulating queries; hence the graph database server needs to use the same section of its data storage to answer the queries. This is similar to a scenario that requests information of a famous person on a social graph by multiple applications simultaneously.

The spike generation workload model implemented in XGDBench is shown in Figure 9.7. During execution of a spiky workload, few places start execution of the requests (e.g., places 1, and 2). Once an initial time period T_i has elapsed, the remaining activities start issuing requests to the database server. All the activities will be issuing queries for a plateau time period T_p . Next, each place completes execution of its workload with time steps T_s as shown in Figure 9.7 which results in a workload spike.

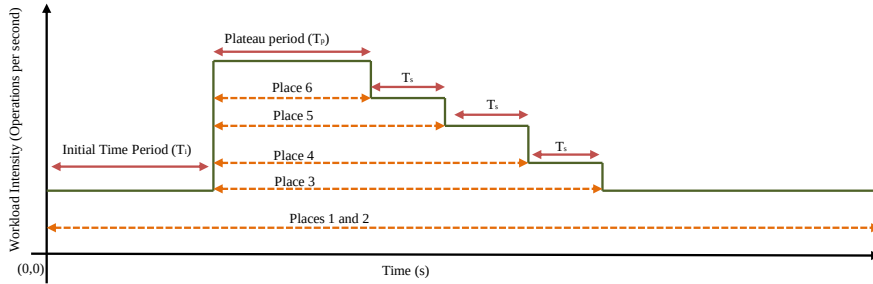


Figure 9.7: Model of the workload spike.

9.8 Evaluation

First the evaluations are conducted on the XGDBench's data generator model's (MAG) properties. We compare the degree distribution of MAG with a real world social net-

work graph. Next, MAG is compared with the popular R-MAT model focusing on the properties of the graphs generated by them. In the second half of the experiments we evaluate the performance characteristics drawn out by XGDBench on five graph DBs. Finally, evaluation details of the workload spike generation is provided.

9.8.1 Properties of the MAG Data Generator

Degree distribution and graph community structure of MAG model were used to evaluate its properties as follows,

9.8.1.1 Power-law Distribution

Degree distributions of many real world graphs that arise in many applications such as the web and social networks satisfy power-law distribution [321]. Two variables x and y are related to each other by a power-law when,

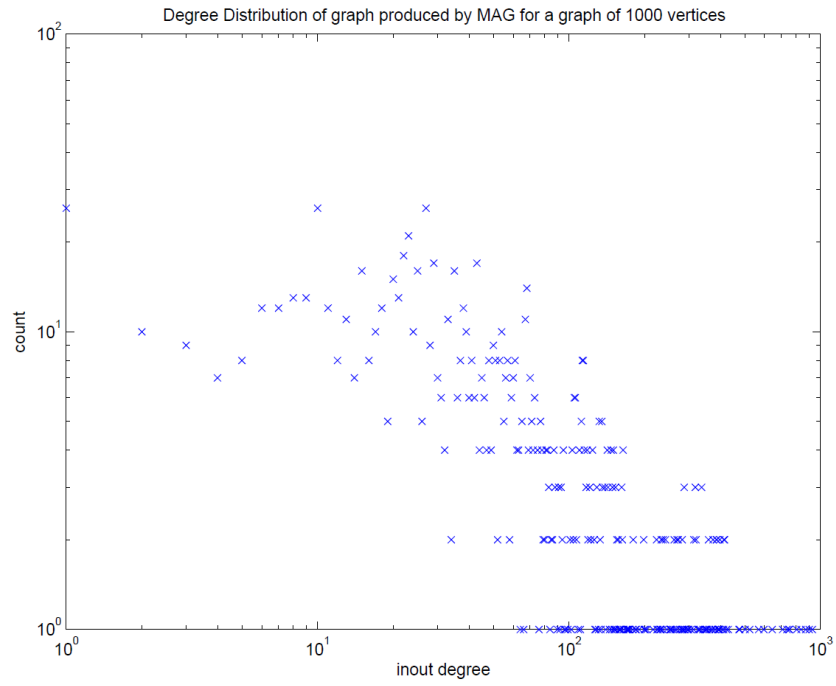
$$y(x) = Ax^{-\gamma} \quad (9.1)$$

where A and γ are positive constants [311]. The degree distribution of a graph with 1000 vertices produced by XGDBench generator was plotted to observe degree distribution characteristics (shown in Figure 9.8(a)). Note that the “inout degree” in Figure 9.8 means the total number of edges connected to a vertex without considering the edges’ directions. It was observed that it creates a degree distribution similar to a power-law distribution. Degree distribution of an online social network called Epinions social network (described in [322]) was also plotted to compare and confirm that MAG creates power-law degree distributions similar to real world graphs.

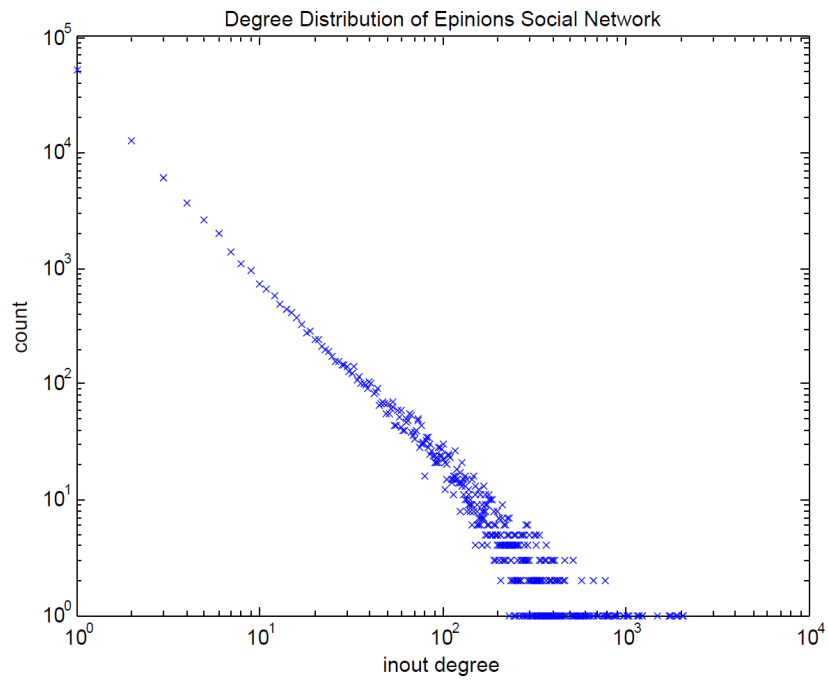
9.8.1.2 Categorical Prominence Within Communities

The data generator module of XGDBench need to model real life graphs that reside on graph databases. As mentioned before, graph databases are designed to store not just plain graphs (with only vertices and edges), but more colorful graphs with vertex and edge attributes. The graph generator should generate such realistic attribute graphs in order for XGDBench to generate realistic workload scenarios. The chapter compares famous non-attribute graph generator model R-MAT which is used by famous benchmarks such as Graph 500 with MAG data generator to observe which model best fits for graph database benchmarking scenarios.

For comparison purpose an R-MAT version of XGDBench was implemented by replacing the data generator algorithm with R-MAT algorithm. Furthermore, after generating the R-MAT synthetic graph the vertex attributes were randomly populated to mimic the attribute graphs produced by MAG model. Five graphs from each model were used with R-MAT scale (n) 10 to 14 for this purpose. R-MAT graph was generated with parameters $a=0.6$, $b=0.15$, $c=0.15$, $d=0.1$ (these values closely resemble the Graph 500 R-MAT generator’s initiator parameters). For MAG, a probability threshold of 0.01



(a)

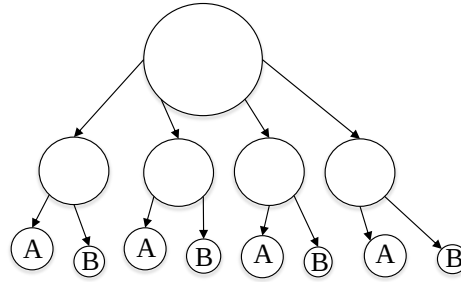


(b)

Figure 9.8: Comparison of the degree distribution produced by MAG generator of XGDBench with a real world social network

was used with an attribute threshold of 0.25. Each graph had 4 attributes per vertex. Details of the generated graphs are listed on Table 9.3.

Next, a community cluster analysis was conducted on each graph. Cytoscape [323] which is a platform for complex network analysis and visualization was used for this purpose. The vertices in the top 3 resulted clusters were further clustered based on the vertex attributes. Next, the percentages of vertex counts in each sub cluster were taken and were ranked based on their percentage values. A cluster prominence metric called *Categorical Prominence* (C_p) was defined as the percentage difference between the largest sub cluster and the second largest sub cluster (See Figure 9.9). The measured C_p values are reported on the Table 9.3.



Categorical prominence (C_p) = Average of the vertex count differences between A and B

Figure 9.9: An illustration of categorical prominence in a graph data set. In the graph data set the vertices are classified into two sets called A and B.

Table 9.3: Categorical Prominence

Vertices (Scale)	MAG		R-MAT	
	Edges	Cluster prominence (C_p)	Edges	Cluster prominence (C_p)
1024 (10)	23077	24.00	2704	6.33
2048 (11)	121298	23.33	3912	3.33
4096 (12)	413281	29.33	1218	1.33
8192 (13)	1634377	26.67	8782	3.33
16384 (14)	6363791	36.67	15974	3.67

From the results it was observed that the graphs generated by MAG model had sub clusters with higher prominence indicating that the communities created by MAG represented the phenomenon of social affinity that is present in real social networks.

9.8.2 Graph DB Performance Comparison

Next, performance evaluation experiments of OrientDB, Allegrograph, Neo4j, Fuseki, and Titan were conducted using XGDBench. The experiments were done on Tsubame 2.0 [324] cloud computing environment (A single node's specifications are given on Table 9.4).

Table 9.4: Specification of a single compute node

CPU	Two Intel Xeon X5670 @2.93GHz, each CPU has 6 cores (total 12 cores)
RAM (GB)	54
HDD (GB)	-
Network	SDR Infiniband $\times$ 2
SSD (GB)	120
OS	SUSE Linux Enterprise Server 11 SP1
File System	Lustre

Table 9.5: Specifications of Graph DBMSs

Name	Data Model	Programming Language	Version	JVM Heap Size (GB)	Distributed?
OrientDB	Network	Java	v1.0rc9	2	No
Neo4j	Network	Java	community v1.6.1	4	No
Fuseki	RDF	Java	v0.2.1	2	No
AllegroGraph	RDF	LISP	v4.6	-	No
Rexster with Titan	Network	Java	2.1.0	4	Yes

Two nodes were used in each experiment for OrientDB, Allegrograph, Neo4j, and Fuseki. In the case of Titan six nodes were used with Rexster [275] server (configured to use a Titan database) running on one node, Cassandra running on four nodes and XGDBench running on the remaining node (Figure 9.11 shows how each software is deployed in Tsubame 2.0 in this configuration.). Rexster is a graph server that exposes any Blueprints enabled graph through REST and a binary protocol called Rexprow [275][318].

The XGDBench Client was run on the remaining node. In the case of OrientDB, AllegroGraph, Neo4j, and Fuseki the graph database server were set up in one node and XGDBench was run on the next node. The JVM heap sizes were set as mentioned in the Table 9.5. Since AllegroGraph is LISP based server it was need not to set JVM heap size.

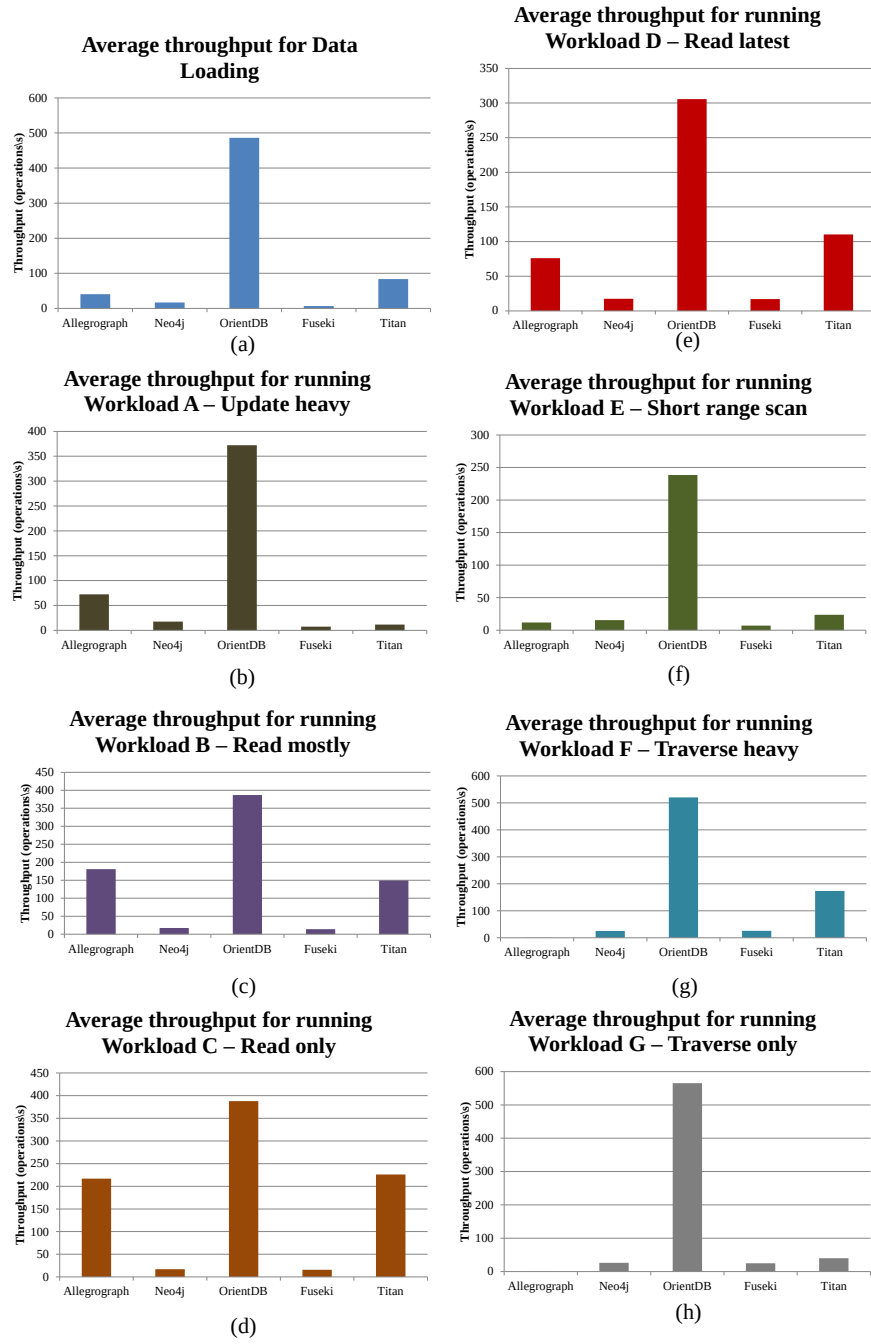


Figure 9.10: Data loading phase and transaction phases' throughput of graph database servers on Tsubame cloud

For XGDBench a heap of 8GB was setup for X10 runtime. Large maximum heap sizes drive garbage collection less frequently which reduces the consumption of CPU cycles for garbage collection during the benchmark execution. This is the main motivation behind choosing 8GB heap size for X10 runtime. X10 runtime 2.3.1 was used which is build with fully optimized settings. XGDBench generated graphs with 1024 vertices during these evaluations. For Titan the evaluations used Cassandra back-end with Rexster [275] REST graph server. The decision to use Cassandra back-end for Titan was motivated by the fact that Cassandra’s ability to provide scalable and highly available service. The results are shown in Figure 9.10.

9.8.3 Performance Evaluation of Titan

The next half of the evaluations used Titan (through Rexster 2.1.0) configured on Tsubame 2.0. Four Tsubame 2.0 nodes were used to setup Apache Cassandra cluster while Titan Rexster server was run in another node. XGDBench client was run in a separate node. The arrangement of the experiment node cluster is shown in Figure 9.11.

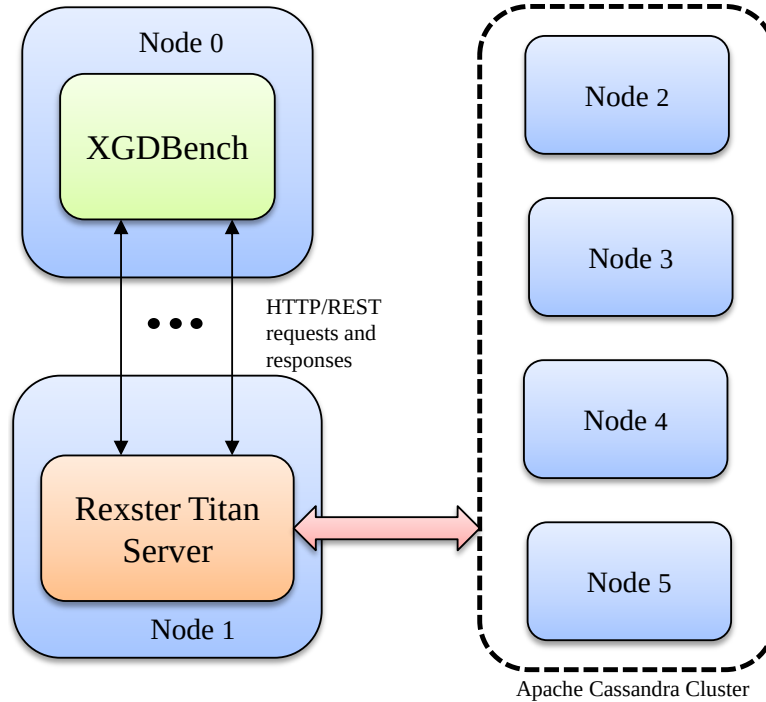


Figure 9.11: How XGDBench, Titan Rexster server, and Cassandra are deployed in the experiment node cluster on Tsubame.

The data loading phase as well as transaction phase workloads were executed on the Titan Rexster server for different vertex counts. Twenty four threads were used for XGDBench during all the experiments because a single node on Tsubame contains

24 hardware threads. Before each experiment round the Titan and Cassandra instances were truncated to make sure each experiment is started with clean graph database server. The initial random seed value of 100 was used with XGDBench. The results of the data loading phase are shown in Figure 9.12.

The results of executing transaction phase are shown in Figures 9.13 and 9.14. Note that the experiments results shown in Figures 9.12, 9.13, and 9.14 are single experiment runs.

9.8.4 Evaluation of Graph Generation Time

The time taken for generating large graphs was evaluated with XGDBench’s graph generator. The purpose of this evaluation was to identify to what extent the generator can generate large graphs. The results are shown in Figure 9.15. It was observed that XGDBench’s generator is able to generate a graph with 250 thousand vertices and 622 million edges in about 315 seconds with using a JVM heap size of 32GB on a single node of Tsubame. While such large scale graphs can be generated with XGDBench, benchmarking graph database servers with such gigantic graphs cannot be achieved easily because most of the current graph database servers are not capable of handling such large graphs efficiently.

9.8.5 Evaluation of Workload Spike Generation

Workload spike generation performance was evaluated using a graph data set generated by XGDBench’s MAG graph generator with 3206 vertices and 18895 edges. Titan [214] distributed graph database server was used as the representative graph database server. Similar to previous experiments Titan server was exposed as a HTTP/REST server through REXSTER 2.1.0 [275]. Rexster was configured to use Apache Cassandra 1.1.6 [325] backend. The Cassandra cluster used in this chapter consisted of eight nodes each with AMD PhenomTM Processor 9850, 8GB RAM, 4 cores, 1600MHz 512KB L2 cache per core, 160GB hard drive. These nodes are denoted as Nodes $B_i (i \in \{0 \dots 7\})$ in Figure 9.16. XGDBench client (on three nodes) and Titan server (on one node) were deployed on a cluster of nodes denoted as $A_j (j \in \{0 \dots 3\})$ in Figure 9.16. each with Intel CoreTMi7-2600K CPU @ 3.40GHz, 8 cores, 16GB RAM, 8MB L2 cache, 200GB hard drive. All the nodes were installed with Linux CentOS (kernel 2.6.18), Oracle JDK 1.6.0.43. A JVM heap size of 4GB was used for Titan Rexster server. XGDBench was configured to use a heap of 8GB and a stack of 1GB.

The experiments used workload C which is a read-only workload of XGDBench (Described in Table 9.2) for this evaluation. The plateau period (T_p) of the spike was set to 5 seconds and activities terminated their workload execution within a time gap (i.e., spike decremental interval T_s) of 2 seconds. The non-spike activity count was set to two, hence at the beginning of the spikeful workload execution (i.e., Time T_i) only two activities executed their workloads. A total of 16 places were employed during the experiment in such a way that each place having one activity. Hence 13 activities ignited

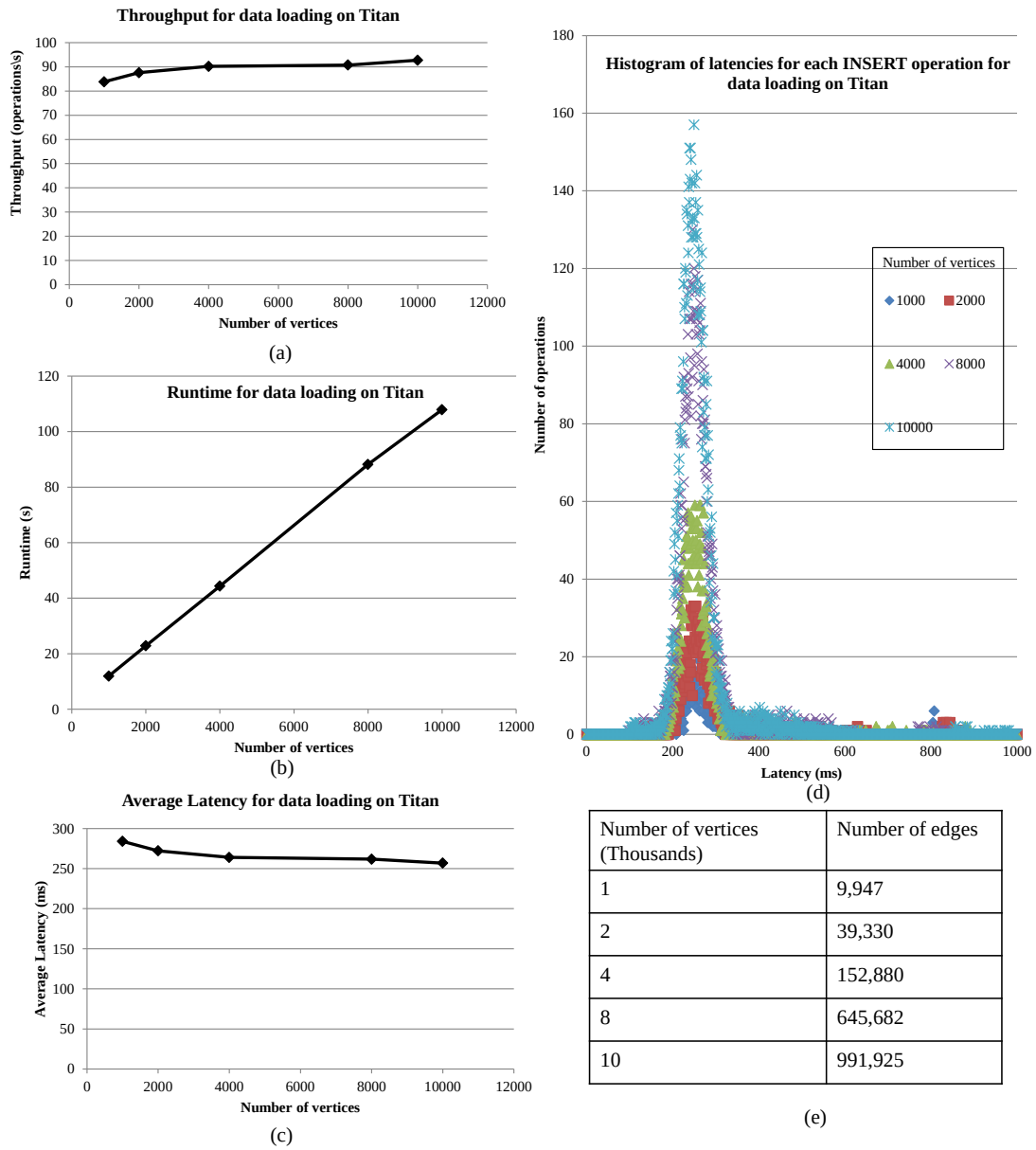


Figure 9.12: Data loading phase of Titan on Tsubame cloud

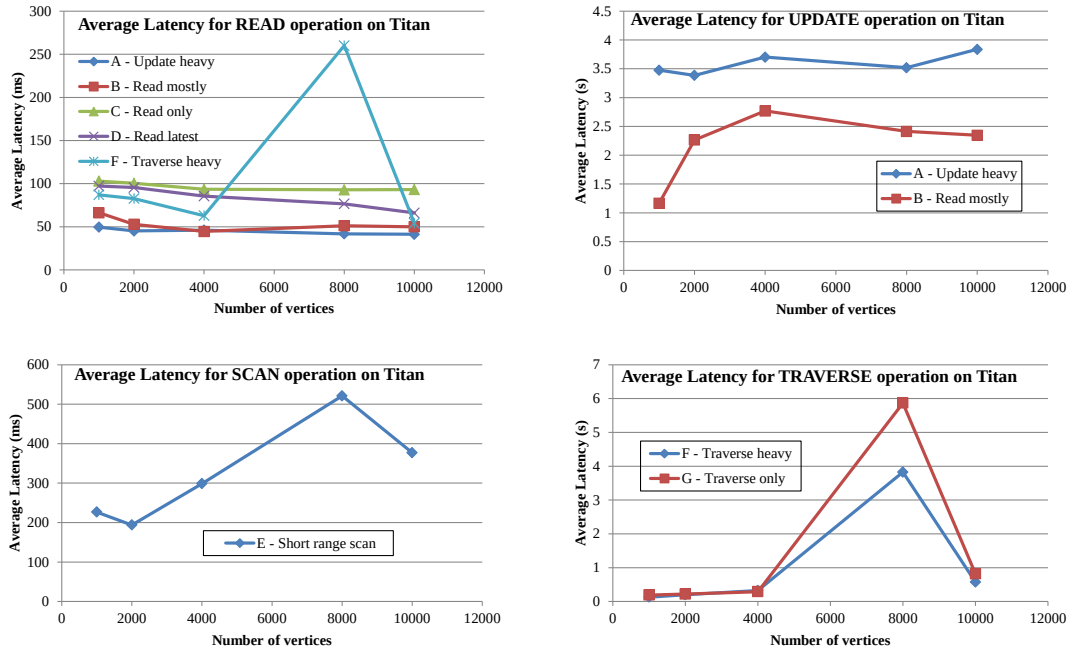


Figure 9.13: Average latency for transaction phase of Titan on Tsubame cloud

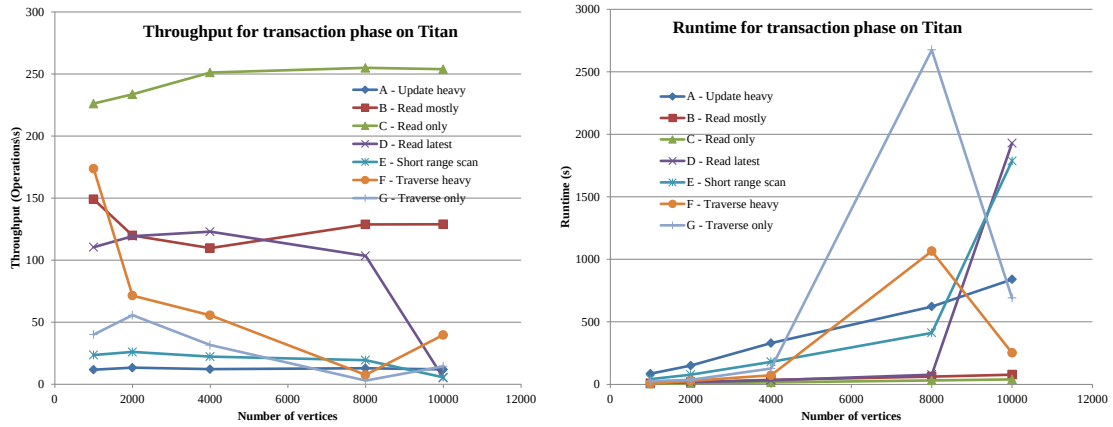


Figure 9.14: Throughput and runtime for transaction phase of Titan on Tsubame cloud

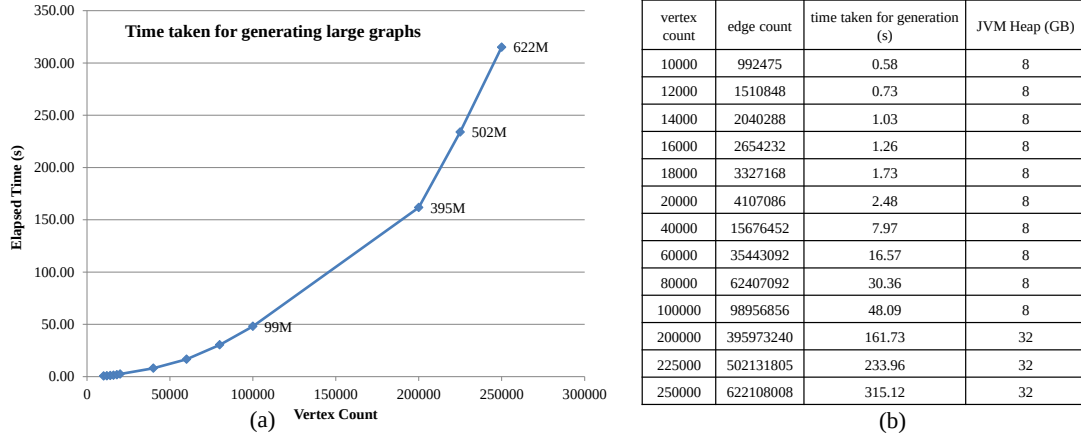


Figure 9.15: Evaluation of graph generation time on XGDBench graph generator on Tsubame cloud. The numbers on the curve in (a) indicate the number of edges generated in millions (M).

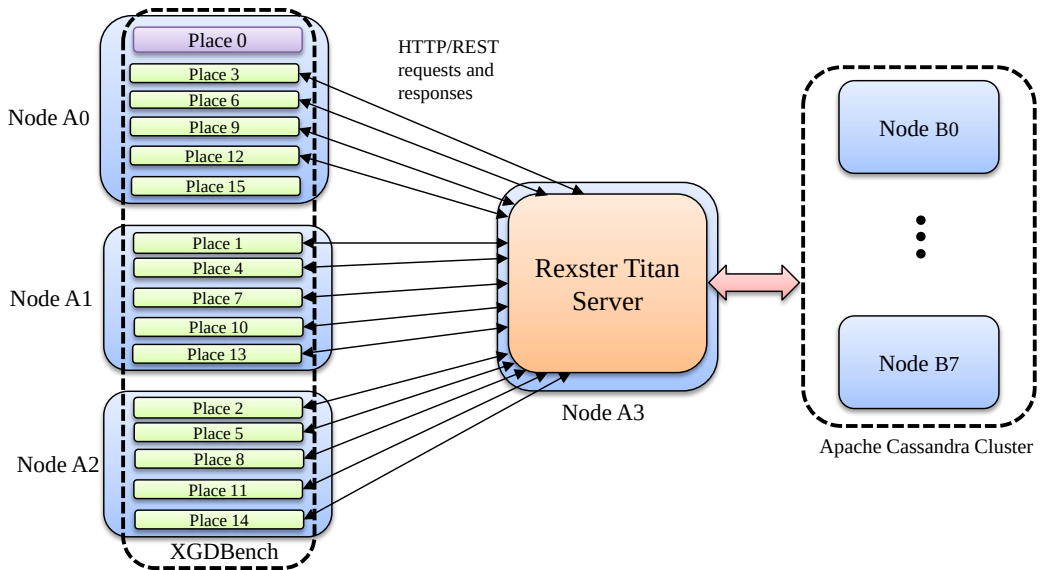


Figure 9.16: How XGDBench, Titan Rexster server, and Apache Cassandra are deployed during the experiment on workload spike generation.

the workload spike after the time period T_i (Activity running at place 0 did not run workloads hence it was kept for workload coordination).

Nmon [280] which is a famous profiling tool was utilized to observe the behavior of the Titan Rexster server during the workload spike. The results are shown in Figures 9.17, 9.18, 9.19.

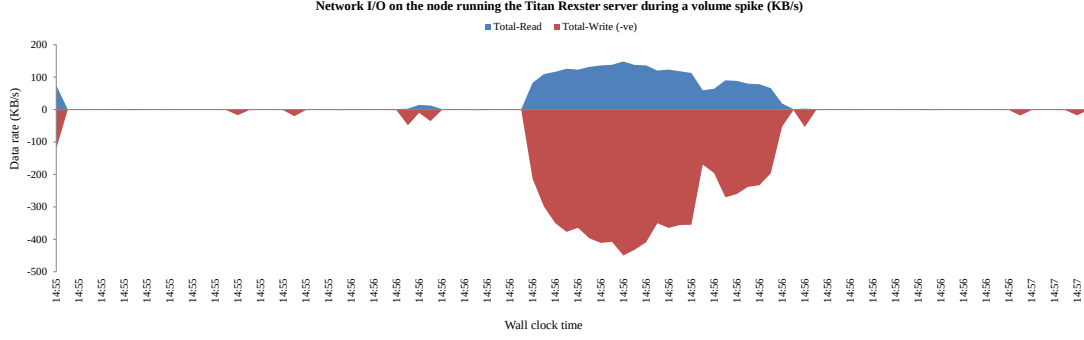


Figure 9.17: Network I/O

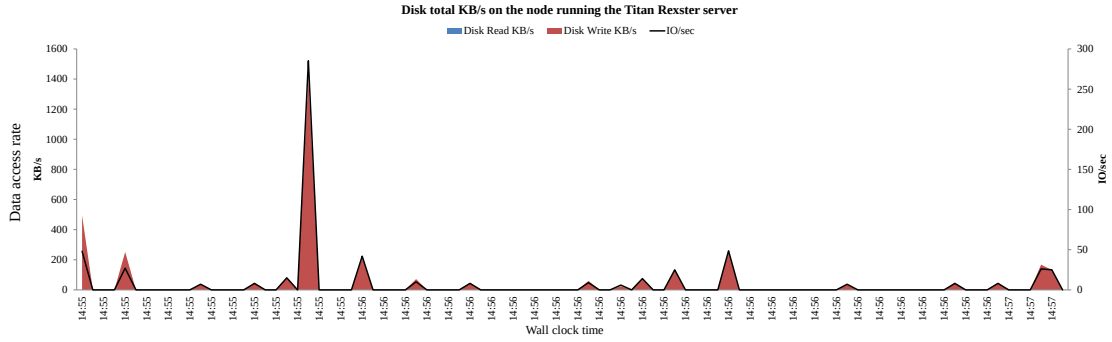


Figure 9.18: Disk access

From the results on Figures 9.17, 9.18, 9.19 it was observed that XGDBench is able to generate realistic workload spikes for graph database benchmarking. Also it was observed that during a workload spike the Titan Rexster server was busy accessing the disks.

9.9 Discussion and Limitations

From the results obtained in Section 9.8.1 it was observed that XGDBench's graph generator model is much suited for evaluating performance of graph databases. Furthermore, the attribute graphs produced by MAG follow the power-law distributions which enables realistic benchmarking scenarios. The chapter also demonstrated the XGDBench's

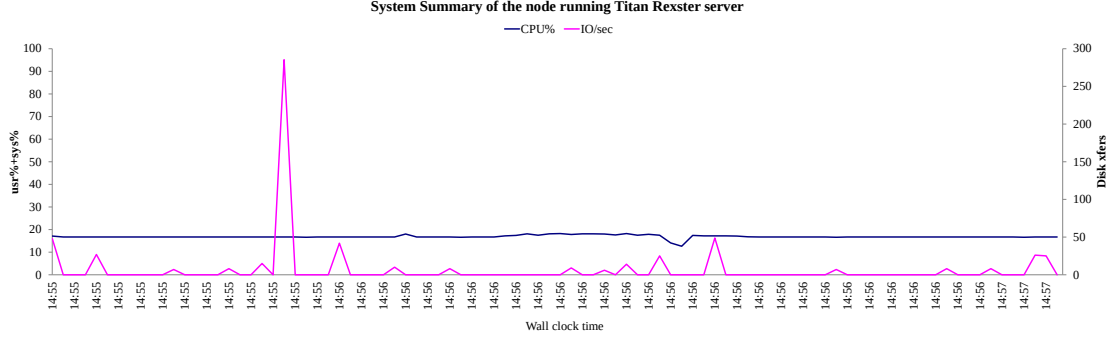


Figure 9.19: System summary

graph generator’s ability for generating gigantic graphs with hundreds of millions of edges within short periods of time.

From the performance evaluation of the graph DBs conducted in Section 9.8.2 it was observed that OrientDB performed well during the data loading phase and with all the workloads having 1024 vertices. AllegroGraph showed better performance compared to Neo4j and Fuseki during the transaction phases even though its performance was not as higher as OrientDB. Titan had almost similar performance to AllegroGraph with workloads B, C, and D. AllegroGraph performed poorly for workloads that include traversal operations (Workloads F and G).

One of the key reasons for why these different databases perform differently is because their external interface. Most of the graph databases studied in this Thesis other than the OrientDB utilized an HTTP/REST web service to communicate with external parties. This acts as a major bottleneck for database scalability even though REST interfaces provide ubiquity. OrientDB used its own Java based communication interface which made its faster performance. For read only workload (Workload C) AllegroGraph and Titan indicated significant performance gain compared to read mostly workload (Workload B). In the case of traverse workloads it can be observed that traverse only workload (Workload G) has less throughput for Titan compared to traverse heavy workload (Workload F). AllegroGraph reported very low throughput for traverse workloads. This indicates that Titan and AllegroGraph are optimized for read only operations.

The experiment results in Section 9.8.3 on performance evaluation of Titan indicated scalability of Rexster Titan server. The details of the graphs generated during the evaluation are shown in the Table on Figure 9.12 (e). Throughput of data loading phase increased with larger graph sizes (See Figure 9.12 (a)). However, the data insert latency increased with increase of the number of vertices (See Figure 9.12 (d)). Furthermore, the running time of the entire experiment showed almost linear increase. This indicates that the Titan server takes more time to answer queries when the size of the graph handled is large. From the results obtained for transaction phase of Titan in Figures 9.13 and 9.14 it was observed that the average latency of scan and traverse operations increase with the vertex count. Furthermore, for a graph with 8000 vertices it was observed a

sudden increase in the average latency, runtime parameters for workloads that involve traverse (F and G) operations. Such increase was not observed for read only (A), read latest (B) workloads. This indicates that the traversal operations conducted on the graph with 8000 vertices had more connected components compared to the other graphs which made the Rexster Titan server take more time in answering the traversal queries. This is an example scenario of irregularity that is commonly found in real world graphs. Read only workload (C) showed scalable throughput performance on the Rexster Titan server.

Most of the current graph databases (including four of the databases used in this study) are not distributed which is one of the limitations of this work. Even the distributed graph database servers that are available such as FlockDB [216] (An open source graph DBMS from Twitter) do not provide full breadth of functionalities that are available in their non-distributed counter parts. For example the current implementation of FlockDB does not support attribute graphs. Furthermore, although 1024 vertices is not enough to thoroughly characterize performance of graph databases we had to use 1024 vertices because some database servers used in this evaluation performed poorly which made the experiments restricted to 1024 vertices.

9.10 Conclusions

This chapter described XGDBench, a graph database benchmarking framework that allows for realistic modeling and evaluation of graph database workloads. The chapter provided a discussion on the architecture of XGDBench as well as how the traversal based workloads are implemented. The data generator of XGDBench is based on MAG Model which enables realistic modeling of attribute graphs. XGDBench has been implemented using X10 which enables XGDBench to be extended easily to support benchmarking future Exascale graph databases. The applicability of MAG model for graph database benchmarking was evaluated through this work. The chapter also conducted a performance comparison of graph databases to demonstrate the capabilities of XGDBench for graph database benchmarking. From the cluster analysis it was observed that MAG model creates more realistic attribute graphs compared to the popular R-MAT model which is used in several graph based benchmarks. The performance comparison of the graph databases revealed that OrientDB had better performance for both data loading and transaction phases. Performance evaluation of the Titan database server indicated its scalability.

9.11 Summary

Benchmarking of graph databases: a category of data analysis software that represent a key area of large scale graph data analytics was explained in this chapter. An introduction to benchmarking graph database systems was given in Section 9.1. Next, Section 9.2 provided background information on YCSB, synthetic graph generators, and MAG

model respectively. Methodology followed in designing XGDBench was explained in Section 9.3 while Section 9.4 described the features that are expected from XGDBench. XGDBench has been implemented using X10, and the implementation details of the benchmark was presented in Section 9.5. How the reproducibility of the benchmarking results across different benchmarking sessions was described in Section 9.6. Modeling spikes in graph database systems was described in Section 9.7. Evaluation of the XGDBench was described in Section 9.8. A discussion of the benchmark’s experiment results made in Section 9.9 before concluding the chapter in Section 9.10. Next chapter describes ScaleGraph 1.0 which is a graph processing library developed for parallel and distributed graph processing.

An X10 Library for Billion Scale Graph Analytics

Achieving optimized performance in large scale data analytics is a great challenge because of rapidly increasing complexity of hardware systems. Data analysis applications need to be fine tuned for the target environment in a productive and efficient manner. For improved productivity of data analysis applications, programming systems need to provide simple execution model for the programmer. As mentioned in Chapter 1 simple execution models reduce the application programming complexity required to achieve the goals of exposing all parallelism and locality. The software autotuning techniques described in the previous part of the thesis exploits the programming abstractions for constructing optimized data analysis scenarios. However, for application of such automated techniques for performance optimization of large scale data analysis systems we need to have well-defined software modularizations. SPADE is an example for such software modularization for stream applications. However, in the case of data analytics involving graph data processing there is very limited number of such software modularizations exist. This chapter describes the design and implementation of ScaleGraph 1.0 (here onwards referred to as ScaleGraph), a graph processing library made to address such requirements [48][326]. Section 10.1 of this chapter provides an introduction to the research problem addressed. Architecture of ScaleGraph is described in Section 10.2. Implementation details of ScaleGraph is provided in Section 10.3. Performance evaluations conducted using ScaleGraph is described in Section 10.4. A discussion of the evaluation results of ScaleGraph as well as its limitations are described in Section 10.5. Conclusions are provided in Section 10.6.

10.1 Introduction

Present high performance computing landscape is dominated by message passing computers with distributed memory. For example, the K computer which is the top ranked

super computer in Top 500 list of June 2011 is a distributed memory machine [327]. It is expected that this trend will continue at the dawn of the Exascale era as well [83].

There are well known shortcomings of distributed memory machines that makes them harder to program [73]. One of the key issues is latency of accessing non-local data is relatively high compared with local data accesses. Another issue is load balancing for applications with adaptive or dynamic computational requirements is challenging. Moreover, writing message-passing software is very complex task that is limited to a small community of HPC programmers from the software programmers at large. Creating scalable large graph analysis activities on distributed memory machines remains a key challenge for non-HPC programmers in Exascale era which is a key research question addressed in this chapter.

PGAS (Partitioned Global Address Space) [328] languages such as UPC [329], X10 [246], Chapel [245], etc. provide a means of programming such distributed memory machines in Exascale environments. Development of comprehensive libraries for large graph analysis in such PGAS languages is still in its infancy. We have developed ScaleGraph which is an X10 library for massive graph analytics targeting large scale graph analysis scenarios [48] to show the applicability of such PGAS languages for graph analysis.

10.2 ScaleGraph Architecture

ScaleGraph library has been designed from ground-up with the aim of defining solid abstractions for large scale graph processing. Architecture of ScaleGraph is shown in Figure 10.1. X10 application programmers can utilize ScaleGraph to write graph applications for Native X10. ScaleGraph library depends on third party C++ libraries such as Xerces-C++ XML Parser [330], numerical packages such as SCALAPACK [331], etc.

X10 applications which use ScaleGraph can be written to operate in three different scales called SMALL, MEDIUM, and LARGE. The SMALL scale represents a graph application that runs on a single Place (Lets take the maximum supported graph size as 2^n ($n : n > 0, n \in \mathbb{N}$)). We created this configuration to support complex network analysis community at large, who might be interested of using ScaleGraph in single machine settings. If an application which uses the library in SMALL scale is run in multiple places, the graph will be stored in the place designated by *home* (i.e., Place 0).

The second configuration type is MEDIUM scale in which the number of vertices stored in one place is 2^m ($m : m > n, m \in \mathbb{N}$). However, the total graph size equals to $(2^m * \text{number of Places})$. For example, when the application is developed for MEDIUM scale size with $m=25$ and is run on 32 places, the application can handle graphs up to 2^{30} (i.e., ≈ 1 billion) vertices (As shown in Figure 10.2 (a)).

The third category of applications is the LARGE scale (shown in Figure 10.2 (b)). This category has been created to support scenarios where the end-user does not have enough compute resources to instantiate sufficient amount of places to hold billion scale graphs. This type of application scenarios will be frequent for users with small compute clusters with limited RAM or even in resource full compute clusters such as supercomputers when the processed graph need to be persisted on disks.

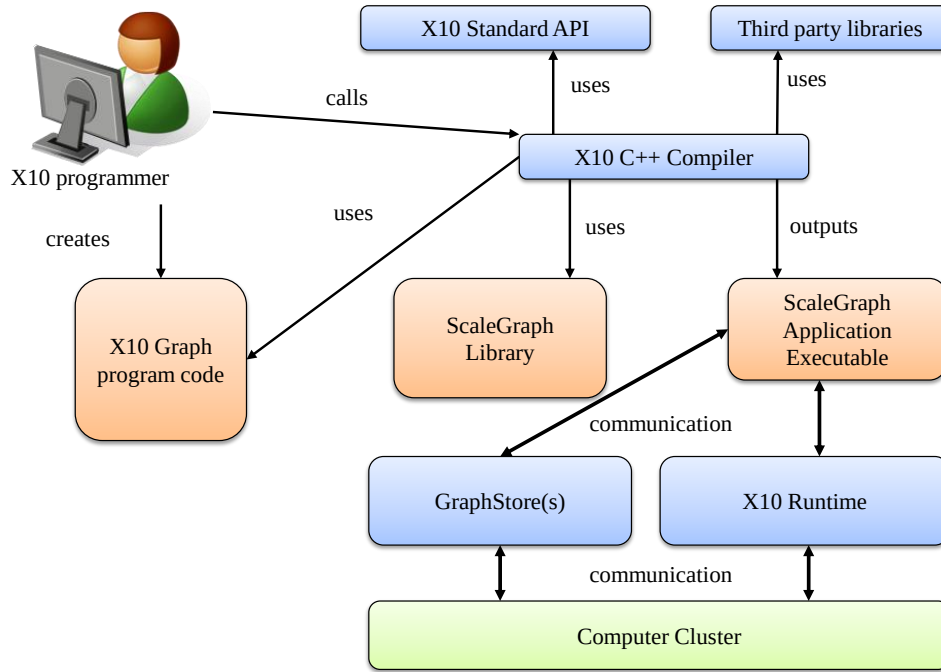


Figure 10.1: ScaleGraph Architecture.

We have introduced such three scales of operations due to resource availability and performance trade-offs present in many graph analysis applications. While the library scales well with increasing numbers of machines, one cannot expect it to process a very large graph that could not be kept on a single laptop's memory. We believe the three scales of operation modes lead to a more simpler yet robust architecture of ScaleGraph.

The library has been modeled entirely using object oriented software design techniques. The design of the library contains six main categories; graph, I/O, generators, metrics (graph structural properties), clustering, and communities. Package structure of ScaleGraph is shown in Figure 10.3.

The graph package holds all the classes related to graph representation. All the graphs of ScaleGraph implement a single interface called *Graph*. ScaleGraph separates graph representation from rest of the algorithms. A Graph in ScaleGraph is just a data structure and it has no associated operations implementing specific analysis algorithms (E.g., degree, pagerank, centrality, etc.). Graph algorithms are coded in separate classes. ScaleGraph has two types of Graph classes named **PlainGraph** and **AttributedGraph**. The **PlainGraph** is used to store non-attributed graphs (i.e., Graphs without attributes for both vertices and edges) while **AttributedGraphs** can store attributes on both vertices and edges.

ScaleGraph uses an adjacency list representation of graph data in its Graph interface. Most of the real world graphs are sparse graphs which can be efficiently represented using an adjacency list compared to an adjacency matrix. While adjacency matrices provide

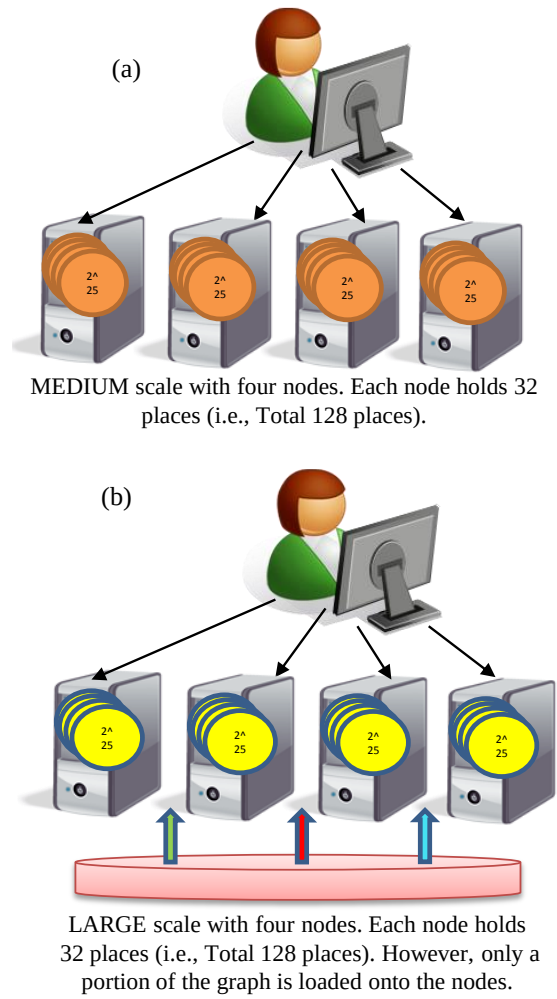


Figure 10.2: Medium scale and large scale configurations of ScaleGraph.

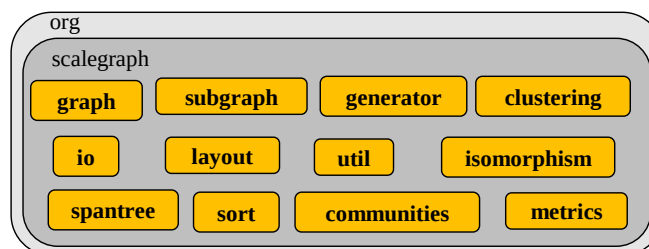


Figure 10.3: Package structure of ScaleGraph.

a marginal advantage over adjacency lists for memory utilization for representing big graphs, and less time for edge insertion and deletion, it is well recognized fact that adjacency lists are better for most applications of graphs [39].

ScaleGraph contains a set of classes for reading and writing graph files located under `org.scalegraph.io`. All the readers implement *Reader* interface while all the writers implement *Writer* interface both of which are located on `org.scalegraph.io`. There are many different types of graph file formats used by complex network research community. Out of them we support some frequently used file formats for attributed graphs such as GML, GEXF, and CSV. For non-attributed graphs we support popular formats such as edgelist and CSV. Certain file formats have more than single file reader/writer classes. An example is `ScatteredEdgeListReader` which reads a collection of files created by partitioning an edgelist file in small pieces.

The generators package includes a collection of graph generators. We have implemented an R-MAT [312] generator which is located in the generators package. ScaleGraph contains a set of classes for obtaining the structural properties of graphs. ScaleGraph has implemented betweenness centrality and degree distribution structural property calculation. The planned other graph structural properties include diameter, pagerank, density, complexity, cliques, KCores, Mincut, connected component, etc. The main interfaces of ScaleGraph include *Graph*, *Reader*, and *Writer* interfaces which are described above.

10.3 Implementation

This section describes the structural properties used for scalability evaluation study of ScaleGraph.

10.3.1 Graph Structural Properties

10.3.1.1 Degree Distribution Calculation

Degree distribution is one of the widely studied properties of a graph [332]. Degree of a vertex in a graph is the number of edges connected to it [333]. If one denotes degree by k , then the degree distribution can be represented by p_k . Two types of degree distributions can be calculated for directed graphs such as world wide web graph, citation networks called in-degree and out-degree distributions. In the context of a web graph, in-degree of a vertex V is the number of vertices that link to V . Out-degree of V is the number vertices that V links to [333]. ScaleGraph supports calculation of both in-degree and out-degree for directed graphs. In ScaleGraph a boolean flag has been used to determine the directedness of a graph. If the flag is set to true, the graph is treated as a directed graph. In this chapter we only evaluate performance of in-degree calculation since both in-degree and out-degree calculation algorithms are similar algorithms.

10.3.1.2 Betweenness Centrality

Betweenness centrality (BC) [334][335] is a graph structural property which measures the extent to which a vertex lies on paths between other vertices [59]. It is one of the most frequently employed metrics in social network analysis [101]. We can define BC of a general network as follows. Let n_{st}^i be the number of geodesic paths (i.e., shortest paths [336][337]) from s to t that pass through i (s, t , and i are vertices of the graph, $s \neq t \neq i$). Let's denote the total number of geodesic paths from s to t as g_{st} . Then the BC of vertex i (i.e., x_i) is given by,

$$x_i = \sum_{st} n_{st}^i / g_{st} \quad (10.1)$$

We implement a more efficient version of BC introduced by Brandes [101]. For a graph with n vertices and m edges this algorithm requires $O(n+m)$ space. The algorithm runs in $O(nm)$ and $O(nm+n^2 \log n)$ time on unweighted and weighted graphs respectively [101]. Brandes' algorithm traverses the vertices in non-increasing order of their distance from source vertex (Brandes does not mandate use of a specific traversal algorithm for this purpose [101]). Once this is done it backtracks through the frontiers to update sum of importance values of each vertex [186]. However, it should be noted that in the case of **AttributedGraph** we use Dijkstra's algorithm instead of BFS in order to account for edge weights. We do not do any approximation of BC rather calculate exact BC scores on large graphs.

In our BC algorithm at the beginning, place 0 instantiates **BetweennessCentrality** class objects in all the places. After construction of each object, it invokes the method for constructing neighbor map that includes information of the neighbor connectivity. Once each object constructs their own neighbor map, each object runs Brandes on assigned vertices on them and calculates BC in parallel. Finally, betweenness scores are scattered among each place via a distributed all reduce operation, which are then reported as an Array object from place 0. A code snippet of our BC implementation on **PlainGraph** is shown in Figure 10.4. Note that important X10 language constructs are highlighted in bold italics font in Figures 10.4 and 10.5.

10.3.1.3 Spectral Clustering

Graph clustering is the act of grouping vertices of a graph into clusters considering the edge structure of the graph in such a way that there should be many edges within each cluster and relatively few edges between the clusters [338]. Graph clustering algorithms can be divided into two categories called "Node Clustering Algorithms" and "Graph Clustering Algorithms" [65]. Spectral clustering is a node clustering algorithm.

If there are n objects labeled $x_1, x_1, x_2, \dots, x_n$ with a pairwise similarity function F defined between them (F is symmetric and non-negative), spectral clustering includes all methods and techniques that partition the set into clusters by using eigenvectors of matrices, like F itself or other matrices derived using it [339]. Spectral clustering algorithm includes two main steps as shown in Figure 10.5. First, spectral clustering algorithm

```

val distVertexList:DistArray[Long] = this.plainGraph.getVertexList();
val localVertices = distVertexList.getLocalPortion();
val numParallelBfsTasks = Runtime.NTHREADS;

finish {
  for(taskId in 0..(numParallelBfsTasks -1 )) {
    async doBfsOnPlainGraph(taskId, numParallelBfsTasks,
                          this.numVertex, localVertices);
  }
}

// If undirected graph divide by 2
if(this.plainGraph.isDirected() == false) {
  if(this.isNormalize) {
    // Undirected and normalize
    betweennessScore.map(betweennessScore, (a: Double) => a /
                        (((numVertex - 1) * (numVertex - 2)))));
  } else {
    // Undirected only
    betweennessScore.map(betweennessScore, (a: Double) => a / 2 );
  }
} else {
  if(this.isNormalize) {
    // Directed and normalize
    betweennessScore.map(betweennessScore, (a: Double) => a /
                        ((numVertex -1) * (numVertex - 2)) );
  }
}

if(Place.ALL_PLACES > 1) {
  Team.WORLD.allreduce(here.id, betweennessScore, 0,
betweennessScore, 0, betweennessScore.size, Team.ADD);
}

```

Figure 10.4: A code snippet of BC calculation on PlainGraph.

```

makeCorrespondenceBetweenIDandIDX();

//Step 1: Make a degree matrix and a Laplacian matrix and solve a
generalized eigenvalue problem
val dm:DenseMatrix = getEigenvectors();
if(dm == null){
    return null;
}

//copy eigenvectors to DistArray
val nPoints = dm.M;
val points = DistArray.make[Vector](Dist.makeBlock(0..(nPoints-1)),
(Point) => Vector.make(nClusters));
finish for(p in points.dist.places()) async at(p) {
    for([i] in points.dist.get(p)){
        for(var j:Int = 0; j < nClusters; j++){
            points(i)(j) = dm(i, dm.N - j - 1);
        }
    }
}

//Step 2: Apply K-Means algorithm to eigenvectors
val resultArray:DistArray[Int] = kmeans(nClusters, points);
val result:ClusteringResult = makeClusteringResult(nClusters,
resultArray);

```

Figure 10.5: A code snippet of Spectral Clustering algorithm of ScaleGraph.

transforms the initial set of objects into a set of points in space, whose coordinates are elements of eigenvectors. In spectral clustering an eigenvector or a combination of several eigenvectors is used as the vertex similarity measure for computing the clusters. Next, the set of points are clustered via standard techniques such as K-Means clustering [298]. Spectral clustering has the ability of separating data points that could not be resolved by applying K-Means clustering directly which is a key advantage compared to other techniques. Spectral clustering has been applied for analysis of the network of the Internet autonomous-system domains, graph partitioning, etc. [338]. A code snippet depicting the Spectral Clustering implementation of ScaleGraph is shown in Figure 10.5. Our spectral clustering code utilizes SCALAPACK [331] for solving eigenvalue problem [340].

10.4 Evaluation

This section lists the evaluation results of the execution performance of degree distribution calculation, betweenness centrality calculation, and spectral clustering algorithms on graphs stored in `PlainGraph` structure. All the evaluations of ScaleGraph has been conducted on Tsubame 2.0 [324] on 8 nodes with the library set to operate in MEDIUM scale. Each node has 2 Intel®Xeon®X5670 @2.93GHz CPUs each with 6 cores (total 12 cores per node/24 hardware threads). Each node has 54GB memory and 120GB SSD and were connected with a Lustre file system for data storage. Each node runs on SUSE

Linux Enterprise Server 11 SP1. We used X10 release version, X10.2.2.2. The X10 distribution was built to use MPI runtime with MVAPICH2-1.8 [341] and was built with maximumly optimized versions of the class libraries by providing `-DNO_CHECKS=true -Doptimize=true squeakyclean` as arguments. We set `X10_STATIC_THREADS` environment variable to 22. This allows us to avoid test applications generating excessive amounts of threads. We used SCALAPACK version 2.0.0.

We used a Kronecker graph data set obtained from [342]. The graph contains 19,683 vertices and ≈ 40.3 million edges of the form $A \rightarrow B$. We scattered the data file (193MB on Lustre) in to 97 files each of size 2MB and used the `ScatteredEdgeListReader` class to load the scattered data files. This allowed us to load the graph data into the `PlainGraph`'s data structures faster than using the `EdgeListReader` class of `ScaleGraph` which reads only a single file. However, a drawback of this method is that, since we used Linux's `split` command with specific file size (2MB), it cuts the original file exactly into 2MB size which resulted in loss of few edges from the loaded graph since we had to discard beginning/ending lines of certain small files. We hope to eliminate this problem in future versions of `ScaleGraph`.

We also used a real data set obtained from LiveJournal with ≈ 69 million edges and 4.8 million vertices (file size 965MB) [343][344]. The dataset is available at Stanford Network data sets repository which can be accessed from [345].

First, we obtained the performance information of running in-degree calculation on `PlainGraph`. We used all the 8 nodes for this purpose and in each experiment we changed the number of places as 8, 12, 16, 20, and 24. For each place combination we ran the algorithm three times and got the average value. The results of this experiment for Kronecker graph and for LiveJournal graph are shown in Figure 10.6 and Figure 10.7 respectively.

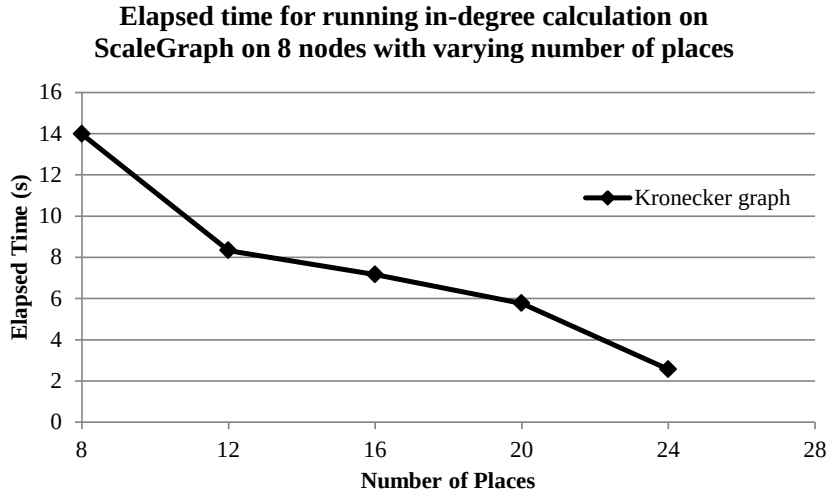


Figure 10.6: Elapsed time for calculating in-degree distribution of Kronecker graph.

We saw the degree distribution calculation in multiple places scales well with the

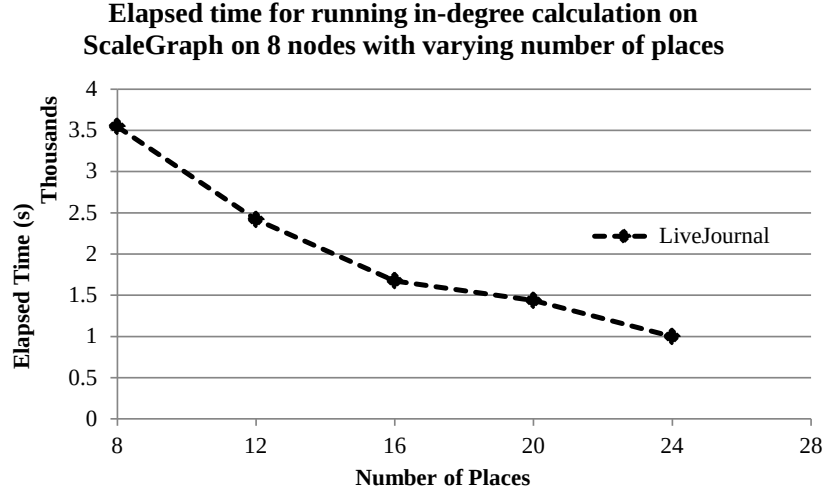


Figure 10.7: Elapsed time for calculating in-degree distribution of LiveJournal graph.

increase of the number of places. However, from 26 places onwards (not shown in Figures 10.6 and 10.7) we saw reduced scalability due to saturation of X10 runtime. It should be noted that we could not observe strong scaling across nodes (like we did in BC) in the case of in-degree calculation because communication overhead between nodes becomes a significant factor that determines elapsed time compared to the performance gain obtained from added CPU/memory resources.

Next, we ran betweenness centrality calculation on **PlainGraph**. We followed a strong scaling approach in BC experiment because BC is computation intensive algorithm. Since, we get only 24 hardware threads per place, we believe this approach allows us to observe scalability of our BC implementation. We ran BC on 1, 2, 4, and 8 nodes each node having single place per experiment. Note that computations on single place were parallelized across all the cores of a node using X10's `async` statement. Each experiment was conducted three times and we plot the average elapsed times on Figure 10.8. We observed that our BC implementation is scalable across multiple places in all except with two nodes. The reason why two nodes takes little more time compared to single node experiment is that the Kronecker graph we used during this study had big connected component which was processed at place 0 which made place 0 (on first node) to take more time to complete compared to place 1 on second node.

Finally, we ran spectral clustering on **PlainGraph** (each experiment 3 times) and obtained average elapsed time. We used the same node patterns similar to BC because spectral clustering is also a computation heavy algorithm. We saw that the speed up gain we achieved becomes less with increase of nodes. The results are shown in Figure 10.9. The spectral clustering algorithm identified two clusters on the Kronecker graph.

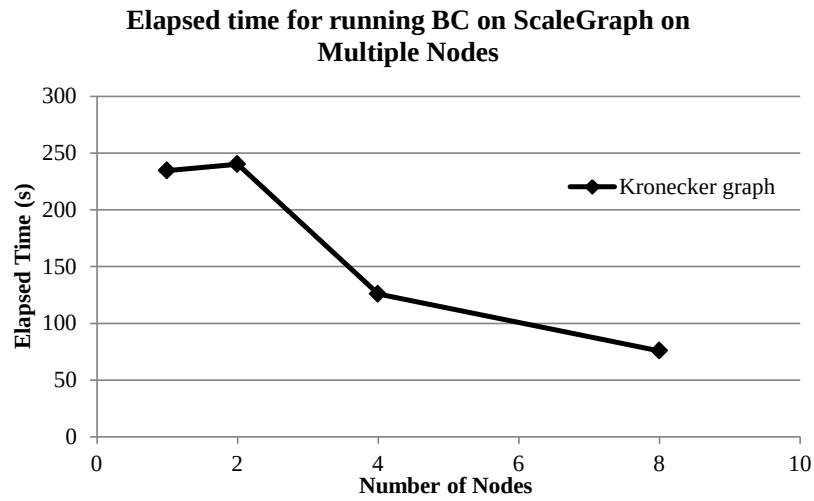


Figure 10.8: Elapsed time for obtaining BC up to 8 nodes.

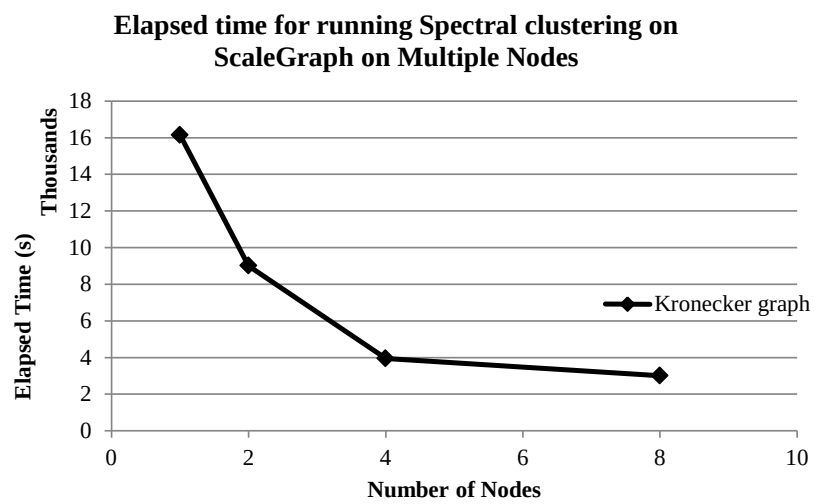


Figure 10.9: Elapsed time for running Spectral Clustering up to 8 nodes.

10.5 Discussion and Limitations

Unlike most contemporary graph libraries, ScaleGraph tries to introduce concrete abstractions for representing graph data on distributed environments while providing a simple programming interface for X10 application developer community [48][326]. One of the key features that distinguishes ScaleGraph library from other contemporary libraries is that ScaleGraph creates a distributed index of graph data (i.e., the vertices are indexed by their vertex ID) which makes it difficult to load the graph data in different places asynchronously. Current solution for this issue is to split the large graph into pieces and then load them via our library. A future enhancement would be creating a cyclic version of the distributed index of graph data to enable scalable loading of graph data on ScaleGraph. Since current X10 DistArray does not implement block cyclic distribution we need to implement our own cyclic distributions. We also study other different data distribution techniques that could evenly distribute computations across all nodes during a distributed graph computation.

The scalability evaluations on this chapter have been presented up to 8 nodes. It was observed that from 8 nodes onwards the scalability advantage reduce increasingly. We believe that ScaleGraph can scale better with larger data sets (compared to the ones presented in this chapter) on more than 8 nodes.

In the case of BC, limited concurrency and main memory in distributed memory machines are the main bottleneck for scalability. It should be noted that we are calculating the exact BC rather than making BC approximations for large graphs in such constrained environments different from some of the works mentioned in the Related Work (Section 2.6). The bottlenecks are same for Spectral clustering as well since it is computation intensive algorithm. However, in the case of degree distribution calculation which is less CPU intensive, its better to use lesser number of nodes because adding more nodes increases the communication overhead which acts as the main bottleneck.

10.6 Conclusion

This chapter described ScaleGraph which is an X10 library for large scale graph data analysis. ScaleGraph library has been designed ground up following the object-oriented programming concepts emphasizing programmer productivity of creating scalable graph analysis applications in distributed memory and heterogeneous environments. We evaluated the performance of degree distribution calculation, betweenness centrality, and spectral clustering on **PlainGraph** in Tsubame 2.0 distributed memory computing environment and observed the scalability of the library. We were able to calculate in-degree distributions of Kronecker and LiveJournal graphs in 2.5 seconds and in 998 seconds respectively on 24 places in 8 nodes. Running BC on Kronecker graph took 75 seconds to complete on 8 nodes. Spectral clustering took 3006 seconds to complete on 8 nodes. We observed that scalability of X10 graph algorithms in such distributed memory settings is determined by communication overhead, number of available hardware threads, and the amount of main memory.

10.7 Summary

Well-defined software modularizations are required for improving productivity of large scale data analysis software systems. Furthermore, such modularizations can act as the input for automated performance tuning mechanisms (Such as the data analysis software auto-tuning techniques described in Part II). Focusing on the research problem of lack of scalable well-defined software abstractions for large graph processing, this chapter described the architecture of ScaleGraph which is an X10 library for large graph processing. Section 10.1 provided an introduction to the area of development of modularizations for large graph analysis. The software design and architecture of ScaleGraph was described in Section 10.2. Implementation details of ScaleGraph was provided in Section 10.3. Evaluation details of ScaleGraph was given in Section 10.4. A discussion on the performance characteristics of ScaleGraph was provided in Section 10.5. Conclusions of the chapter were presented in Section 10.6. Next chapter presents Exedra which is a domain specific language for large graph analysis.

A Domain-Specific Language for Large Graph Analytics

Many of the large scale data analysis activities are organized as workflows because data analysis tasks involve application of multiple analysis algorithms on the target data. Typically data analysis workflows are constructed using multiple data analysis software using a general purpose programming language. Therefore, the workflow optimizations need to be conducted at each different software library or middleware level which could reduce the productivity of the workflow optimization due to the same reasons pointed out in Section 6.1. This chapter describes a solution for this issue through use of Domain Specific Languages (DSL). The chapter describes a domain specific language for large graph analysis that provides a unified interface for the graph analysis workflow. Section 11.1 provides an introduction to the area of domain specific languages. Section 11.2 describes Exedra which is a DSL proposed for large scale graph analysis. The graph analysis activities written in Exedra are executed in a middleware called Dipper. Design and implementation of Dipper middleware is described in Section 11.3. Evaluation details of Exedra and Dipper are provided in Section 11.4. A discussion of the evaluation results is presented in Section 11.5. Conclusions are presented in Section 11.6.

11.1 Introduction

As mentioned in Chapter 1 special classes of software have emerged to process graph data on distributed computing infrastructures such as clusters and super computers. Pegasus [346], Pregel [77], GBase [347], Trinity [212], Hama [348], Giraph [349], HipG [350], PBGL [70], ScaleGraph [48], KDT [186], GraphLab [64], GraphBuilder [351], GPS [352], GraphX [353] etc. are few examples for the large body of research on design and implementation of graph analysis middleware and software platforms targeted for large graph analysis in high performance computing systems. These software provide means for modeling large graphs and allow conducting analysis on them.

Exascale systems [79] which are expected to appear in the time frame 2018~2020 will revolutionize data analysis technologies. However, programming on such Exascale systems will have to operate on a diverse range of hardware such as multi-core processors, hardware accelerators, etc. Furthermore, massive levels of parallelism, increase in system faults, and complex, multi-component software environment are major challenges the Exascale applications will face. There are different path ways for implementing data processing systems in Exascale environments. First such method is use of a standard general purpose programming paradigm and adapt it to different architectures. E.g., MPI is widely used programming model for programming parallel systems. Therefore, we can either emulate MPI and different devices or combine MPI with device specific languages to create graph data processing applications. In the same way we can use novel general purpose parallel programming languages such as PGAS languages (UPC [354], X10 [200], Chapel [245], etc.) to program data analysis tasks on Exascale systems.

However, a major drawback associated with the aforementioned approaches is that each one of them have their own solution. User must be veteran of the general purpose language rather than the application he/she is interested of. We need unified means of doing large graph processing in future large scale systems that will enable broad scientific community to leverage the benefits of multiple systems. This chapter proposes an approach for programming large graph analysis on large scale environments (which is currently less studied) with Domain-specific languages [355]. Domain-specific languages (DSLs) are programming languages tailored to a specific application domain, and they represent majority of the programming languages. They exploit domain knowledge for productivity and efficiency. Use of DSLs offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages used for graph analysis.

We present Exedra which is a domain-specific language for conducting large graph analysis on future large scale systems. Unlike current DSLs for graph analysis our focus is to create a complete DSL for specifying graph analysis workflows. Through Exedra we define a language for specifying graph representation and analytics. We apply this language to a graph processing platform that we developed called Dipper. Dipper allows for specifying large graph analysis tasks using R [356] which is a programming language for statistical computing. Dipper extends the famous Igraph [60] graph analysis package for specifying the large graph analysis tasks. Users can run graph analysis tasks using their familiar Igraph/R [193][60] interface which gets translated to Exedra statements by Dipper. These statements are interpreted by the Exedra interpreter and they get mapped into the underlying high performance libraries.

11.2 Exedra Domain-Specific Language

As mentioned earlier, Exedra is a domain-specific language for large graph analysis. We designed the language following data flow paradigm. Exedra is an interpreted language similar to several related domain-specific languages such as R, Gremlin, etc. Exedra also follows the design decision to use the Igraph/R syntax as mentioned previously. Exedra has language constructs for defining computation compartments. Computations are

conducted on different compartments that makes a data flow path. A compartment is an independent block of computation. It can either receive data from another compartment or instantiate data from itself. After processing the data the compartment hands over the data to another compartment in the flow graph.

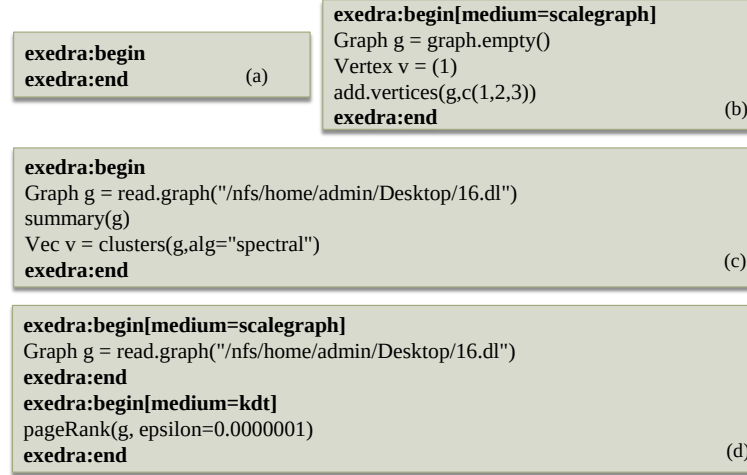


Figure 11.1: Example Exedra code segments.

Figure 11.1 shows some examples for codes written in Exedra depicting the language's core language constructs. An Exedra program consists of one or more compartments. Every Exedra compartment starts with **exedra:begin**, and ends with **exedra:end**, thus the simplest program is of the form Figure 11.1(a). User can pass parameters to a compartment. One of the most important parameters is *medium*. The medium is the library/middleware that the Exedra compartment runs on. Currently we implemented support for ScaleGraph, KDT, and R mediums. While ScaleGraph and KDT are large graph processing libraries, we provide R language also as a medium in Exedra since non-graph computations can be specified in R. Furthermore, if a compartment has been marked with execution medium as R, its contents are not interpreted by Exedra interpreter, rather directed to R interpreter directly.

If any parameter has not been provided (as in the case with compartments in Figure 11.1 (a) and (c)) the default medium's language configured with Exedra interpreter is used as the language of interpreted output.

Furthermore, variables in Exedra have an associated type. For example, the variable *g* in Figure 11.1 (c) is of type **Graph**. The language has support for Graph, Arr (Array), Vec (Vector), String, Int (Integer), Float, Double, Long types. Note that type system is only valid on non-R Exedra compartments. Comments on Exedra begins with **#** and Exedra supports only single line comments (see Figure 11.5).

If an Exedra program specifies more than one compartment (E.g., Figure 11.1 (d)) they are executed in sequential order. In Figure 11.1 (d), first the graph loading happens

```

exedra:begin[medium=scalegraph]
Graph g = read.graph("/nfs/home/admin/Desktop/16.dl")
Vec d = degree(g,mode=c("total"))
exedra:end

exedra:begin[medium=R]
len = length(d) - 1
from = c()
to = c()

repeat{ flg = 0
    flg2 = 0
    for(i in 1:len){
        if(d[i] < 0){
            flg = 1
            break
        } else if(d[i] != 0){
            flg2 = 1
        }
    }
    if((flg == 1) || (flg2 == 0)){
        break
    }
    randval = sample(1:len, 1)
    d[randval] = 0

    largest = 0
    counter = 0
    len = length(d)
    for(i in 1:len){
        if(randval != i){
            if(d[i] > largest){
                largest = i
            }
        }
    }

    d[largest] = d[largest] - 1
    from = append(from, c(randval))
    to = append(to, c(largest))
}
relation = data.frame(from,to)

exedra:end

exedra:begin[medium=scalegraph]
Graph result = graph.data.frame(relation, directed=TRUE)
exedra:end

```

Figure 11.2: Example Exedra code for scenario 1.

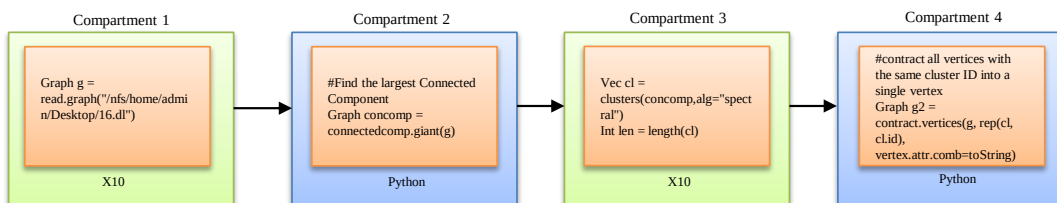


Figure 11.3: Arrangement of compartments of Scenario 2.

in ScaleGraph compartment. Once the ScaleGraph compartment finishes its execution, PageRank is calculated on KDT compartment. Such type of multiple compartments can be used to leverage several libraries/middleware to conduct a computation that cannot be conducted single handed on any of those software. For example in the scenario shown in Figure 11.1 (d), ScaleGraph is capable of loading edgelist files while present version of KDT does not support such feature.

In this chapter we describe two scenario workflows for graph analysis and show how they can be represented in Exedra. The first scenario is privacy preservation of graphs and social networks [357]. In a social network an adversary can know any subgraph around a certain individual A [358]. If such a subgraph has been identified in the anonymized graph with high probability, user A has high identity disclosure risk. Liu *et al.* described a technique for minimally modifying the graph to protect the identity of each individual involved. In brief their technique involves two steps. First, they construct an anonymous degree sequence $\mathbf{d}'$ for a given degree sequence $\mathbf{d}$ of a graph G . Next, they use $\mathbf{d}'$ to construct an anonymized graph G' . More details on their technique is available from [358]. The second step of this process is shown in Figure 11.2.

This involves three Exedra compartments. The first compartment loads the graph G and calculates its degree sequence $\mathbf{d}$. Next, the R compartment creates the anonymized graph using the degree sequence $\mathbf{d}$ (Note that for simplicity we choose the same degree sequence $\mathbf{d}$ as the $\mathbf{d}'$ anonymized sequence). Next, the data frame named `relation` constructed by the R compartment is fed to the third compartment which constructs a ScaleGraph Graph instance. This is an example for an Exedra code that involves multiple compartments. Also, the second compartment does not involve processing any graph instances.

The second example scenario we have chosen is a Mini-workflow that was introduced by Lugowski *et al.* while introducing the Knowledge Discovery Toolbox (KDT) library [186]. The workflow is illustrated in Figure 11.4 which conducts spectral clustering on the giant component of a graph. How this workflow gets executed on Dipper is shown in Figure 11.3. The workflow first loads the graph of edge list format from disk. This is done in a ScaleGraph compartment because KDT does not support this feature currently. Next, the graph's largest connected component is calculated on a KDT compartment (Currently the largest connected component calculation is not supported in ScaleGraph). The largest connected component is sent to a ScaleGraph compartment to conduct spectral clustering which is not implemented in KDT yet. Finally, all the vertices of each cluster are contracted in a KDT compartment. The Exedra code for this workflow is shown in Figure 11.5.

11.3 Design and Implementation of Dipper

We have implemented a large graph processing platform called Dipper which uses Exedra as the language for programming graph analysis tasks. This infrastructure is written in C/C++. System architecture of Dipper is shown in Figure 11.6. It should be noted that even though Dipper accepts instructions given in Exedra, the end-user of the system

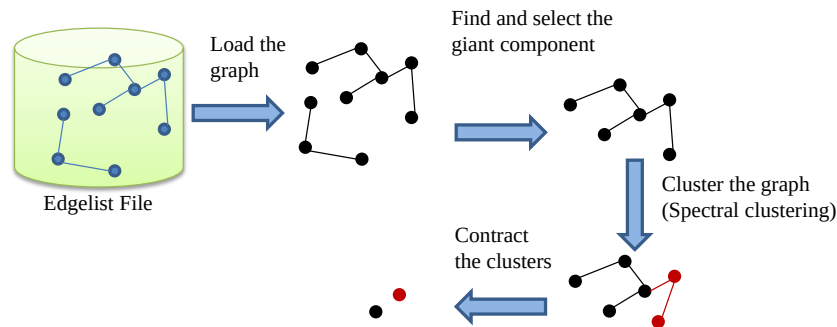


Figure 11.4: Example workflow of Scenario 2.

```

exedra:begin[medium=scalegraph]
Graph g = read.graph("/nfs/home/admin/Desktop/16.dl")
exedra:end

exedra:begin[medium=kdt]
#Find the largest Connected Component
Graph concomp = connectedcomp.giant(g)
exedra:end

exedra:begin[medium=scalegraph]
Vec cl = clusters(concomp,alg="spectral")
exedra:end

exedra:begin[medium=kdt]
#contract all vertices with the same cluster ID into a single vertex
Graph g2 = contract.vertices(g, rep(cl, cl.id), vertex.attr.comb=toString)
exedra:end

```

Figure 11.5: Example Exedra code for scenario 2.

writes the graph analysis computations in Igraph/R syntax.

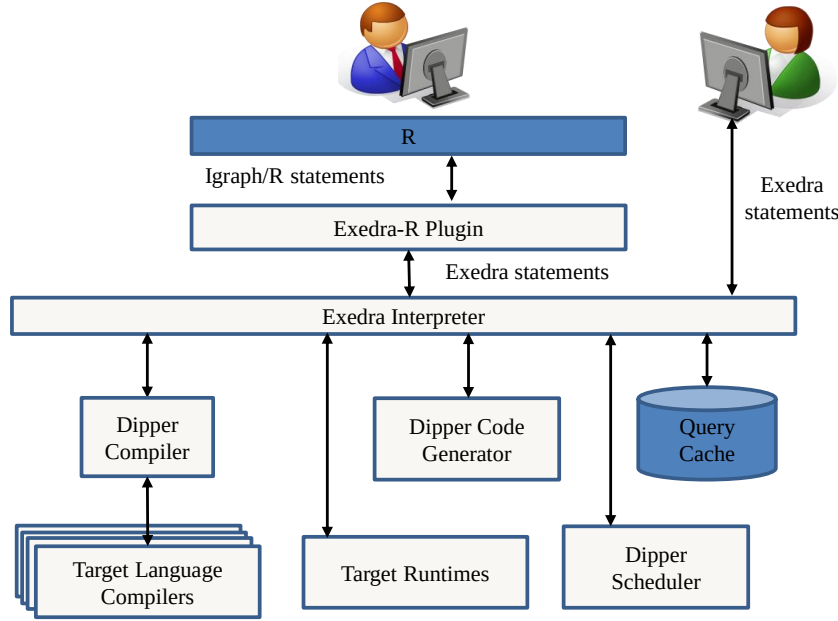


Figure 11.6: System Architecture of Dipper.

We have developed an R extension (Exedra-R Plugin) by extending the Igraph/R package. Currently this extension converts a subset of the Igraph/R instructions into Exedra instructions (The remaining Igraph/R instructions are interpreted by the Igraph/R extension itself). The generated Exedra statements are fed to the Exedra Interpreter component. It consists of an Exedra parser written using popular ANTLR [286] parser generator. The Exedra parser identifies each available compartment on the Exedra program. It also checks in its Query Cache whether same query (i.e., Exedra program) has been executed recently. If the same query had been executed recently, it will just reuse the program binaries available on Query Cache rather than moving to Dipper’s code generation and compilation phases. If the query was not available in query cache the Exedra Interpreter proceeds to the code generation phase. In this phase, once the target medium is identified each statement in the program are translated to target medium’s language (E.g., X10, Python, etc.) or operating instructions. Next, these instructions are added to predefined compartments by the Dipper Code Generator.

A compartment is a standalone software module that runs as an application of its target language. For example an X10 compartment is an X10 application. Compartment has server/client interface; and each compartment communicates with its parent (i.e., predecessor in a compartment chain), and its child (i.e., successor) using a custom protocol written over TCP/IP. The location of a compartment is determined by the Dipper Code Generator. The location where a compartment gets executed is determined during the code generation time.

Once the Dipper Code Generator’s code generation phase completes we get a complete graph analysis workflow that is made out of different graph analysis software components. Next, the Dipper Compiler compiles this workflow using each compartment’s target programming language. This step is needed only for compiled languages such as X10, C/C++, Java, etc. Once the complete workflow is ready to be executed; the Dipper Scheduler instantiates the compartments and launches the workflow.

11.4 Evaluation

We evaluated the performance of running a Dipper workflow and also the performance of running a similar workflow in a single library. The second scenario described in the Section 11.2 has been chosen as the example workflow. Specifically we report the execution time for execution of the workflow shown on Figure 11.5, and also its KDT implementation. The KDT implementation’s code is shown in Figure 11.7.

```
import sys
import kdt

bigG = kdt.DiGraph.load("/tmp/data/qh768.mtx")
bigG.spOnes()

#Finding the largest component
comp = bigG.connComp()
giantCompRoot = comp.hist().argmax()
G = bigG.subgraph(mask=(comp==giantCompRoot))

#Clustering
clus, markovG = G.cluster('Markov', addSelfLoops=True, expansion=2,
inflation=2, prunelimit=0.00001)

#Contracting
smallG = G.contract(clusterParents=clus)
```

Figure 11.7: KDT version of the example scenario 2.

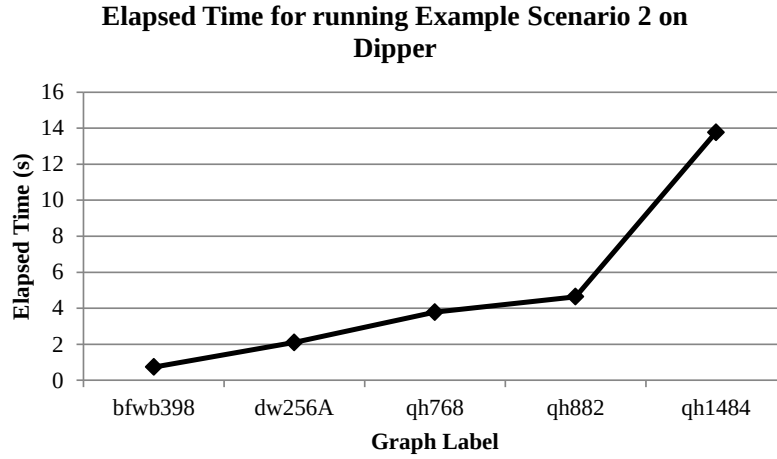
We use a compute cluster of 4 nodes each with Intel®Core™i7-2600K CPU @ 3.40GHz, 24 cores, 15GB RAM, 8KB L2 cache, 200GB hard drive to run both the Dipper workflow and the KDT workflow. Each node was installed with Linux CentOS kernel 2.6.18, X10 version 2.2.2, and Python 2.7.1. We used KDT version 0.2 during the evaluation. All the four nodes were connected through 1 Gigabit Ethernet. We used the University of Florida’s sparse matrices during this evaluation [359]. The details of the graphs that were used are listed on Table 11.1. While we could have used larger graphs during this initial evaluation, we used the graphs mentioned in the Table 11.1 because the KDT implementation that we used has been evaluated using graphs of such scale.

The first half of the experiments were run on a single node. Our objective was to get an idea of how the performance varies for each experiment setup with different graph characteristics. The performance results are shown in Figures 11.8 and 11.9. Next, in

Table 11.1: Experiment graph data sets

graph	vertices	edges
bfbw398	398	2910
dw256A	512	2,480
qh768	768	2,934
qh882	882	3354
qh1484	1484	6100

the second step we distributed the Dipper compartments on to 4 different nodes (of the same specification described above). Each node had one Dipper compartment. However, this step could not be performed on KDT since the application was made to run on single node. The results of the second step are shown in Figure 11.10. All the results shown in Figures 11.8, 11.9, and 11.10 are three times average values.

**Figure 11.8:** Experiment results of running Dipper.

11.5 Results and Discussion

From the results in Figures 11.8 and 11.9 we saw that increase of the graph size increases the elapsed time of the analysis process. However, Dipper spent more time in completing the task because it had the overhead of format conversion. Furthermore, the workflow used Spectral clustering rather than the Markov clustering (used in KDT). The distributed version of the experiment indicated that Dipper has the ability of reducing the

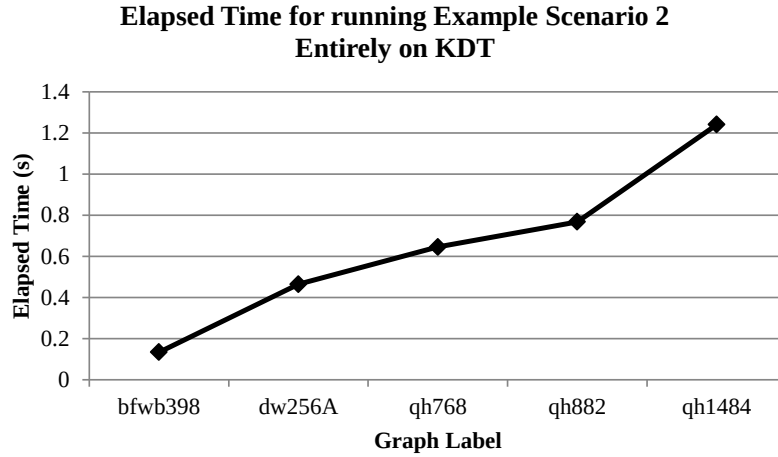


Figure 11.9: Experiment results of running KDT workflow.

time consumed for completing the process when distributed (Even though this involves additional network overhead during communication between compartments on different nodes). Note that we were not able to get results for `dw256A` graph in distributed environment due to an error thrown in KDT compartment.

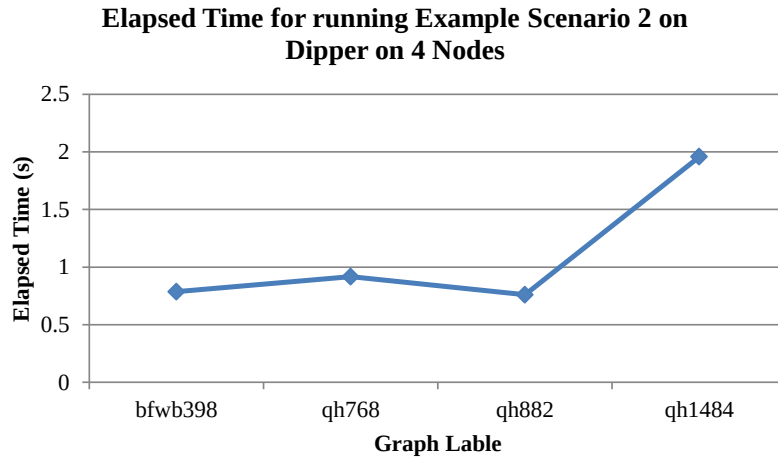


Figure 11.10: Experiment results of running Dipper distributed on 4 nodes.

There are several limitations for current Exedra Language and the Dipper framework that we list down here. Current version of Exedra grammar is limited to graph reading and graph analysis algorithms such as degree distribution calculation, clustering, largest component, etc. Current version of Exedra grammar does not support graph traversal operations. Furthermore, we have implemented compartments for ScaleGraph and

KDT libraries only which restricts us using other mediums. Despite the benefit that Exedra provides for constructing large graph analysis workflows and distributing graph computations; communication between different compartments introduces an additional overhead of data format conversion. We are investigating on different approaches to minimize the overhead involved with format conversion. Furthermore, the current Dipper framework is not able to withstand the changes made to the libraries/middleware on which the compartments run on.

11.6 Conclusion

Mining valuable information from large graphs is essential in the era of Exascale which is predicted to appear in next five or seven years time period. However, most of the analysis, and mining of such large graph data organized as workflows need coordinated effort of several software components since it is hard to find a single library/middleware that provides all the features expected by users. This is because the rapid evolution of the field of large graph analysis and the plethora of software written for this purpose have different applications. In this chapter we described Exedra, a domain-specific language that we believe will provide a unique way of specifying large graph analysis workflows. The workflows get translated to the most appropriate middleware/library that it should get executed. This provides ability of constructing scalable graph analysis workflows that is not possible using single library or middleware.

11.7 Summary

This chapter described Exedra which is a domain specific language for large graph analysis. Section 11.1 provided an introduction to use of domain specific languages for large graph analysis. Details of the Exedra language was presented in Section 11.2. The design and implementation of Dipper graph analysis platform was provided in Section 11.3. Evaluation details of Exedra was provided in Section 11.4. Evaluation results were discussed in Section 11.5 and the conclusions were provided in Section 11.6. The next chapter provided the overall conclusions arrived by this thesis and also lists some future work.

Part III

Conclusion and Future Directions

Conclusion & Further Work

This chapter provides overall conclusions arrived by the research conducted in this thesis. A summary of the research problem being addressed and the proposed solution are described at the beginning of this chapter. The specific contributions made by this thesis are listed on Section 12.1. A discussion and limitations of the overall approach is presented in Section 12.2. Further work for this thesis are listed in Section 12.3.

Data analytics refers to application of computer systems to analyze large data sets with the main motivation of providing support for decision making. As described in the first chapter of this thesis, stream processing and batch processing are two computing paradigms used for conducting large scale data analysis. Large scale computing systems (such as Exascale systems) that are predicted to appear within the next decade will offer huge computing power (i.e., performance per watt) for conducting large scale data analysis in online within real-time constraints. However, the complexity involved in programming such large scale systems are increasing rapidly due to the distributed nature of these systems. Therefore, the thesis pointed out the research problem “Current HPC software architectures as it is for large scale data analysis is less optimal in terms of execution performance and programmability”.

The thesis proposed software automatic tuning that is supported by well-defined software modularizations as a solution for this issue. As shown in Figure 1.2 the proposed solution comprises of optimization of three closely related areas of large scale data analytics: Performance Analysis, Software Autotuning, and Software Modularization. The thesis conducted workload characterization of stream processing systems, benchmarking of graph database systems to identify important performance features present in large data analysis systems. Based on the observations made through performance analysis as well as based on the empirical research results the thesis proposed a new technique for optimizing stream processing system performance which uses software automatic tuning. Furthermore, proposed software automatic tuning techniques depend on well-defined software modularizations and software abstractions [360]. To address this issue the thesis proposed two levels of graph data analytics abstractions which are intended to fulfill this requirement. The overall view of the proposed solution and categorization

of contributions is shown in Figure 12.1.

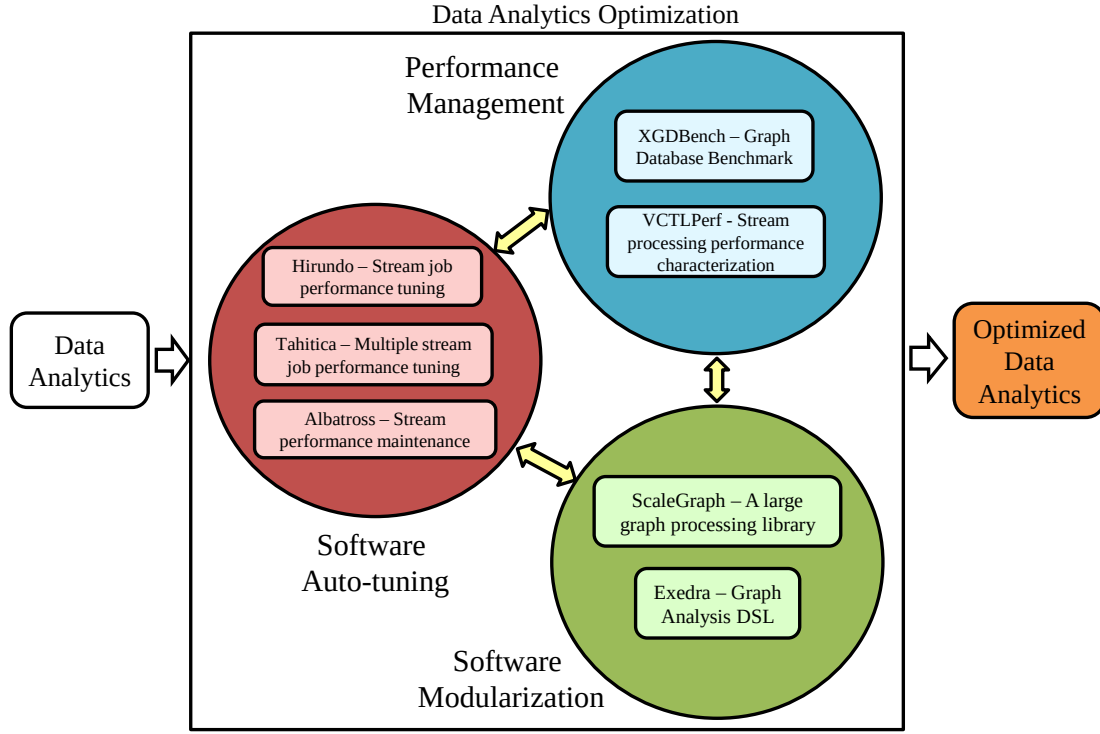


Figure 12.1: Summary of the contributions.

12.1 Contributions

This thesis made contributions to the following areas of large scale data analytics.

- **Stream Processing Performance Characterization:** The thesis conducted performance analysis of three latest, popular stream processing systems. We identified that having higher operator density at the starting portion of an operator graph tend to produce better performance. Furthermore, the thesis created StreamFarm which is a stream workload generator platform that can be used for performance characterizations. StreamFarm provides flexible interface for third party applications.
- **Graph Database Benchmarking:** Chapter 9 of the thesis described XGDBench, which is a scalable graph database benchmarking platform. Furthermore, the thesis conducted benchmarking and workload characterization of five popular graph database systems: Neo4j, OrientDB, AllegroGraph, Fuseki, and Titan using XGDBench. The thesis also described how to model different graph database workload scenarios such as workload spikes using XGDBench.

- **Stream Processing Performance Optimization:** The thesis proposed source code level operator graph transformation and auto-tuning as a technique to improve performance of stream processing performance. In Chapter 6 thesis described Tri-Operator transformation and Bi-Operator transformation techniques for automatic transformation of stream applications. Hirundo which is a stream performance optimization mechanism based on these techniques was described in Chapter 6. The automatic code transformation techniques were applied for multiple stream job performance optimization in Chapter 7 and a multiple stream job performance optimization mechanism was presented.
- **Performance maintenance and Fault tolerance of stream processing:** The automatic code transformation techniques described in Chapter 6 were applied in the context of performance maintenance. A mechanism called Albatross which maintains performance during crash failures was introduced in Chapter 8. With the experiments made using Albatross we demonstrated that software automatic transformation is capable for maintaining performance of stream applications during consecutive node failures.
- **Large graph processing:** The thesis contributed to the area of large graph processing by creating the architecture of ScaleGraph which is an X10 library for large graph analysis. Furthermore, the thesis introduced Exedra which is a domain-specific language for large graph analysis in Chapter 11.

12.2 Discussion and Limitations

The thesis proposed software automatic tuning and software modularization based optimizations for constructing optimized large scale data analytics. While discussions were made on each research approach and their limitations in different chapters, this section provides an overall discussion and limitations of this thesis.

One of the main objectives of this thesis was to conduct performance analysis of large data analysis software systems with the aim of identifying their important performance features. This objective was achieved by conducting performance evaluation of different stream processing systems, as well as benchmarking graph database systems.

Implementation of automatic performance tuning mechanisms for large scale data analysis applications is the main contribution of this thesis. This objective was achieved by creating source operator graph level optimization mechanisms for stream processing systems. The thesis described the architecture of Hirundo, Tahitica and Albatross mechanisms that are developed based on these concepts. An abstract view of the relationship between these three software systems is shown in Figure 12.2.

The objective of creation of well-defined software modularizations for large scale distributed data analysis systems was achieved by designing and implementing a scalable graph processing library called ScaleGraph. Furthermore, the thesis demonstrated the ability for use of a domain specific language for efficient specification of graph analysis tasks.

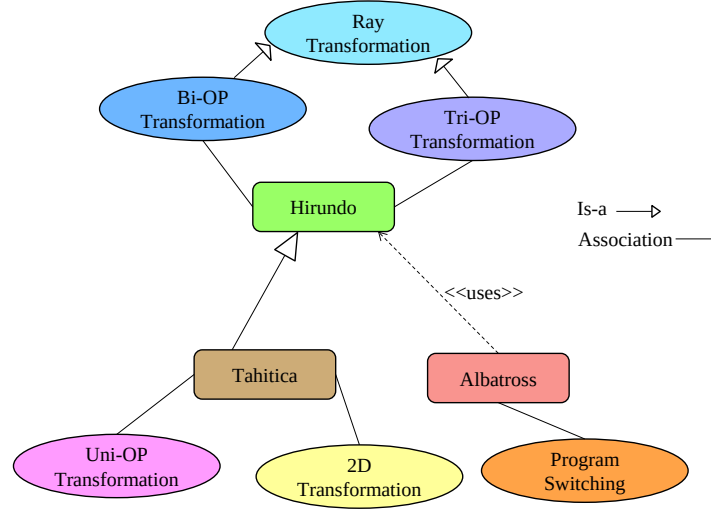


Figure 12.2: High-level view of the relationships between the stream program optimization mechanisms and techniques described in this thesis.

The techniques described in this thesis were implemented on software performance optimization mechanisms such as Hirundo, Albatross, and Tahitica. Furthermore, Scale-Graph 1.0 was implemented in native X10 and was evaluated in the Tsubame 2.0 environment. XGDBench benchmarking framework was implemented in X10 language and it was used for evaluating performance of different graph database servers in Tsubame 2.0 environment. These activities have contributed to achievement of the last two objectives mentioned in Section 1.6.

Current implementations of the automatic software tuning techniques are limited to System S environment, which made it impossible to be deployed and evaluated in Tsubame 2.0 like environments due to the licensing issues of System S.

12.3 Further Work

12.3.1 Stream Processing Optimization

From the performance analysis conducted on stream processing systems (specially with S4) it was observed that the number of processing elements (PEs) spawned by the stream processing system plays an important role in scalability of the application. A stream processing system architecture that scales in terms of number of PEs is a further work that is inspired by that observation. Another future work is to conduct a code level performance study on S4, specially to identify which components, code segments are most resource intensive.

As mentioned in Section 6.10 currently Hirundo’s program structure analyzer grammar is limited to a subset of SPADE language. We hope to improve the program struc-

ture analyzer and the program generator modules of Hirundo in future to support different other stream operator categories. We also plan to study different techniques for performance prediction of multipath DAGs. Another planned future activity is to identify important semantics of input programs and incorporate those during Hirundo's performance optimization process.

Multiple stream job performance optimization described in Chapter 7 is entirely based on ray fusion. In future we hope to implement a more sophisticated fusion algorithm that uses a multi-constraint graph partitioning algorithm.

Albatross mechanism indicated its ability for maintaining performance during multiple node crashes. In future we hope to create a sophisticated scheduling algorithm for Albatross's stream job control process to reduce the time taken for recovery process. Experimenting Albatross with more complex stream applications is another future work. We hope to conduct performance recovery experiments with multiple stream jobs. We also plan to study techniques for predicting sample program performance to improve Albatross's calibration process.

Performance prediction on multipath DAGs is another area that we hope to investigate in future. Pipelining based automatic tuning of UDOPs, automatic optimization of stream applications by changing the order of operators on the data flow graph are some other future research directions. Automatic identification of a parallelizable code segment of a stream data flow graph is another future work for the work described in this thesis.

12.3.2 Graph Processing Optimization

Performance comparisons of the graph database servers described in this thesis were limited to 1024 vertex graphs. A future work is to conduct much deeper evaluation of graph databases using XGDBench. During this process we hope to investigate the reasons for why graph databases perform poorly and find path ways to improve their performance. In future we hope to investigate more deeply on the aspects of graph database workload spikes and implement them on XGDBench. Another future direction that we hope to take this research is random evolution of property graphs which will enable creation of benchmarking sessions with reduced memory usage [361]. We also plan to study more on different other types of realistic workload generators and evaluate their feasibility for graph database benchmarking with XGDBench.

Current Exedra grammar has limited support for a variety of graph analysis tasks. We hope to implement such analysis processes in future versions of Exedra grammar. Currently Exedra provides support for specifying graph analysis workflows in sequential manner. In a future version of Exedra we hope to support multipath workflows as well. Furthermore, we hope to support different other types of mediums such as Pegasus, Apache Giraph, GraphLab [362], PBGL, etc. in future. Another future work is to conduct performance evaluation of Dipper framework with graph sizes of billions of scale.

Exedra provides an abstraction of a unified graph analysis workflow. In future we

hope to autotune Exedra applications by extending the code transformation and optimization techniques described in Part I of this thesis [363]. Furthermore, we hope to improve Exedra in such a way that it allows for implementing multipath DAGs (i.e., more complex graph analysis workflows with concurrent and cyclic execution). Since Exedra will have to face huge serialization and deserialization overhead for billion-scale graphs, in future we hope to investigate approaches for avoiding heavy communication overhead.

Variable scope of the current Exedra Language has a single global scope. In future we hope to implement more sophisticated levels of scope for Exedra. Furthermore, libraries such as ScaleGraph changes rapidly. Therefore, Exedra interpreter's and dipper's internal representation of the libraries (specially their interfaces) need to be robust enough to with stand such changes. This is another future work we plan to conduct on Exedra.

In future we hope to implement robust fault tolerance mechanism for Exedra and Dipper to avoid unexpected crashes happen in Exedra workflows.

12.3.3 Distributed Graph Database Engine for Large Scale Elastic Graph Data Management and Mining in Hybrid Clouds

The graph database benchmarking research presented in Chapter 9 indicated the scalability issues of current graph database servers. This section presents a future work for creating a scalable graph database server targeted for operations in hybrid clouds.

Public and private clouds represent the two established ends of the cloud computing spectrum based on the ownership and efficiency of shared resources. Private clouds are operated by organizations solely for their compute needs and such clouds provide cheap access to compute resources with more control over aspects such as security, privacy, etc. Public clouds allow for purchasing on demand compute resources and provide virtually infinite amounts of compute resources. However, public clouds have associated issues such as cost of resource provisioning and data communication costs between public cloud and local compute clusters becomes challenging when required to transfer huge amounts of data [364]. A third deployment model called hybrid cloud that incorporates characteristics of public and private clouds is currently emerging as a solution. Hybrid clouds provide the ability of extending an organization's private cloud infrastructure by providing elastic cloud computing resources in a cost-effective manner. There have been previous work on hybrid operation of different data analysis applications in the domains of stream computing [217][218][219], MapReduce [365][366], etc. However, there are not much work been done on graph database operation in hybrid cloud settings. In this future work we hope to address this important issue.

Operations in a hybrid cloud in the context of a graph database is challenging because graphs by their nature are linked, irregular data collections. Therefore, we face the common dilemma of cloud data management: data partitioning. There is a trade-off between the partitioning decisions, and efficiency of query execution on graph databases. The key research question that we try to answer is what will be the optimal method of operating a hybrid graph database service in such a way that we get minimum economic

cost and reduced latency.

In achieving our aim we plan to create an operation model for hybrid graph databases. Also we plan to implement this method on a hybrid graph database service that we develop called Acacia. Acacia is intended for managing attribute graphs and will be developed using X10 [200]. Furthermore, Acacia will be using HSQLDB and Neo4j as its storage back ends.

12.3.3.1 Proposed System Design

The design of Acacia is based on graph partitioning. While graph partitioning requires sophisticated partitioning algorithms and considerable processing overhead, it provides us the benefit of reducing the costly communication between different nodes among the nodes in both public and private clusters. We plan Acacia to store attribute graphs which has become a de facto standard in the industry. The overview the proposed hybrid graph database model is shown in Figure 12.3.

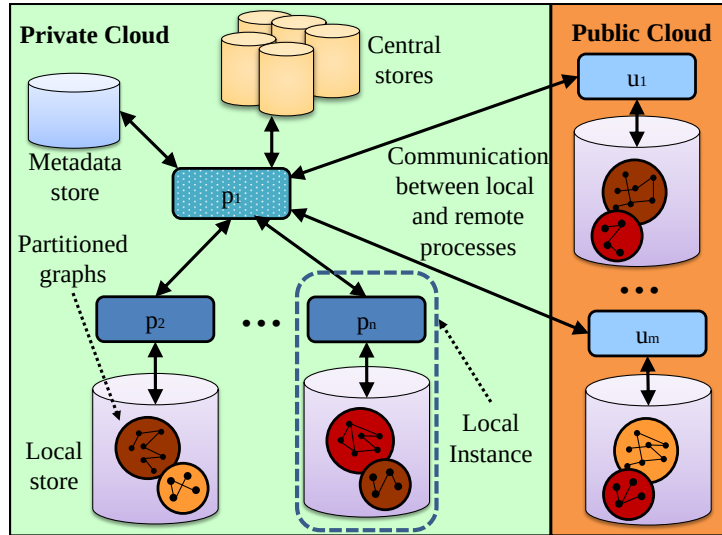


Figure 12.3: Overview of the proposed Acacia hybrid graph database system

Acacia comprises of a set of processes and associated data stores. When a new graph is added to Acacia it first partitions the graph into number of subgraphs which is equal to the number of nodes that it currently run. Initially, the partitioned graphs are distributed to *Local Instances* (A local instance is a standalone, non-distributed collection of graph databases) based on their available capacity. Acacia keeps on monitoring the local stores' remaining capacities as well as the CPU utilization of each node on the local cluster. When it finds the utilization exceeds the SLA values agreed before the execution, it considers of migrating portion of sub graphs in to public cloud. Acacia calculates the required number of nodes to be allocated in the public cloud. Then Acacia starts the VMs on public cloud and instantiates new *Remote Instances*. As soon as the Public

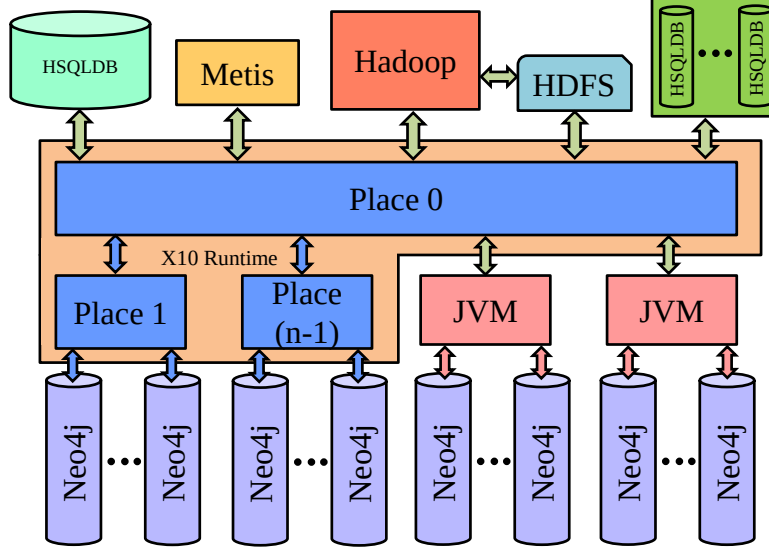


Figure 12.4: Proposed system architecture of Acacia

VMs are available, Acacia starts migrating the selected graph data to the public cloud.

12.3.3.2 Proposed Scheduling Methodology

We plan to model the operation of Acacia to a scenario of solving a Knapsack problem [39][367]. Input for a Knapsack problem is a set of items $S = 1, \dots, n$, where item i has size s_i and value v_i and a knapsack capacity of C . We find the subset $S' \subset S$ that maximizes the value of $\sum_{i \in S'} v_i$, given that $\sum_{i \in S'} s_i \leq C$; [39]. We will use the most common formulation of the Knapsack problem called *0/1 Knapsack problem* where each item must be added completely in the knapsack or not included at all [39]. This is because in Acacia once a graph has been partitioned in to pieces we do not attempt to repartition the same graph in to different number again since it requires lot of computational resources. However, this 0/1 property makes the solving of the Knapsack problem hard since we cannot use simple greedy algorithms for this purpose.

12.3.3.3 Proposed Implementation Technology

We plan to implemented Acacia using X10. The choice of X10 is motivated by the facts such as X10 is open source and its ability to withstand architectural challenges posed by heterogeneous systems. The planned system architecture of Acacia is shown in Figure 12.4. Graph partitioning component of Acacia can be plugged in to the system separately. In the planned architecture we use Metis [368] as the graph partitioner software. Graph partitioning is supported by a series of MapReduce jobs written in Hadoop which conducts format conversion of graph data to suite for Metis partitioner. Furthermore, the MapReduce jobs also contribute for construction of the graph partitions.

The constructed partitions are transferred to Acacia Instances through a gzip based protocol. File gzipping allows for sending large text data in a more efficient manner over the network. Each subgraph partition is stored in a separate Neo4j instance. This design decision was taken because having multiple subgraphs loaded into a single Neo4j instance may lead to performance bottlenecks on the single Neo4j instance which might make the entire system down. Furthermore, having each graph in separate embedded neo4j instance allows the subgraph to be easily transferred to remote nodes by gzipping the entire Neo4j instance folder. The preliminary evaluation results of the data loading phase of Acacia made at the time of writing this thesis indicates that such mechanism allows for scalable distributed operation of the system.

12.4 Summary

This chapter provided overall conclusions for this thesis. The research problem addressed by this thesis as well as the proposed solution were presented at the beginning of this chapter. Contributions of the thesis were summarized in Section 12.1. A discussion was made on the overall achievement of the goals of the thesis as well as the limitations of the thesis's approach in Section 12.2. Next, the chapter provided future work for this thesis in Section 12.3.

References

- [1] T. A. RUNKLER. **Introduction.** In *Data Analytics*, pages 1–3. Vieweg+Teubner Verlag, 2012.
- [2] A. GHOTING. **Data Mining.** In D. PADUA, editor, *Encyclopedia of Parallel Computing*, pages 518–523. Springer US, 2011.
- [3] A. RAJARAMAN, J. LESKOVEC, AND J. D. ULLMAN. **Mining of Massive Datasets.**
- [4] S. MORISHITA. **Big Data Parallel Processing of Personal Genomes.** 2013.
- [5] ASCAC. **Synergistic Challenges in Data-Intensive Science and Exascale Computing.** 2013.
- [6] MP VAN, H. **Lofar: The low frequency array.** *European Astronomical Society Publications Series*, **15**:431–444, 2005.
- [7] J. HAN. **Data Mining.** In L. LIU AND M. T. ZSU, editors, *Encyclopedia of Database Systems*, pages 595–598. Springer US, 2009.
- [8] C. C. AGGARWAL. **An Introduction to Data Streams.** In C. C. AGGARWAL, editor, *Data Streams*, **31** of *Advances in Database Systems*, pages 1–8. Springer US, 2007.
- [9] J. HAN AND B. DING. **Stream Mining.** In L. LIU AND M. T. ZSU, editors, *Encyclopedia of Database Systems*, pages 2831–2834. Springer US, 2009.
- [10] O. ETZION. *Event processing in action.* Manning, Greenwich, Conn., United States, 2011.
- [11] G. CUGOLA AND A. MARGARA. **Processing flows of information: From data stream to complex event processing.** *ACM Comput. Surv.*, **44**(3):15:1–15:62, June 2012.
- [12] M. STONEBRAKER. **Stream Processing.** In L. LIU AND M. T. ZSU, editors, *Encyclopedia of Database Systems*, pages 2837–2838. Springer US, 2009.

-
- [13] H. ANDRADE, B. GEDIK, KUN-LUNG WU, AND P.S. YU. **Scale-Up Strategies for Processing High-Rate Data Streams in System S**. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1375–1378, 2009.
- [14] M. BLOUNT, M.R. EBLING, J.M. EKLUND, A.G. JAMES, C. MCGREGOR, N. PERCIVAL, K.P. SMITH, AND D. SOW. **Real-Time Analysis for Intensive Care: Development and Deployment of the Artemis Analytic System**. *Engineering in Medicine and Biology Magazine, IEEE*, **29**(2):110–118, march-april 2010.
- [15] A. ARASU, M. CHERNIACK, E. GALVEZ, D. MAIER, A. S. MASKEY, E. RYVKINA, M. STONEBRAKER, AND R. TIBBETTS. **Linear road: a stream data management benchmark**. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04*, pages 480–491. VLDB Endowment, 2004.
- [16] N. JAIN, L. AMINI, H. ANDRADE, R. KING, Y. PARK, P. SELO, AND C. VENKATRAMANI. **Design, implementation, and evaluation of the linear road bnchmark on the stream processing core**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06*, pages 431–442, New York, NY, USA, 2006. ACM.
- [17] A. BIEM, E. BOUILLET, H. FENG, A. RANGANATHAN, A. RIABOV, O. VERSCHEURE, H. N. KOUTSOPOULOS, M. RAHMANI, AND B. GÜÇ. **Real-Time Traffic Information Management using Stream Computing**. *IEEE Data Eng. Bull.*, **33**(2):64–68, 2010.
- [18] M. HILL, M. CAMPBELL, Y. C. CHANG, AND V. IYENGAR. **Event detection in sensor networks for modern oil fields**. In *Proceedings of the second international conference on Distributed event-based systems, DEBS '08*, pages 95–102, New York, NY, USA, 2008. ACM.
- [19] L. NEUMEYER, B. ROBBINS, A. NAIR, AND A. KESARI. **S4: Distributed Stream Computing Platform**. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops, ICDMW '10*, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.
- [20] E. BOUILLET, R. KOTHARI, V. KUMAR, L. MIGNET, S. NATHAN, A. RANGANATHAN, D. S. TURAGA, O. UDREA, AND O. VERSCHEURE. **Processing 6 billion CDRs/day: from research to production (experience report)**. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS '12*, pages 264–267, New York, NY, USA, 2012. ACM.
- [21] A. BIEM, B. ELMEGREEN, O. VERSCHEURE, D. TURAGA, H. ANDRADE, AND T. CORNWELL. **A streaming approach to radio astronomy imaging**. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 1654–1657, march 2010.

-
- [22] M. S. MAHMOUD, A. ENSOR, A. BIEM, B. G. ELMEGREEN, AND S. GULYAEV. **Data Provenance and Management in Radio Astronomy: A Stream Computing Approach.** *CoRR*, abs/1112.2584, 2011.
 - [23] J.P. HAYES, H.R. KOLAR, A. AKHRIEV, M.G. BARRY, M.E. PURCELL, AND E.P. MCKEOWN. **A real-time stream storage and analysis platform for underwater acoustic monitoring.** *IBM Journal of Research and Development*, **57**(3/4):15:1–15:10, 2013.
 - [24] M. LEHNING, N. DAWES, M. BAVAY, M. PARLANGE, S. NATH, AND F. ZHAO. **Instrumenting the earth: next-generation sensor networks and environmental science.** In *The Fourth Paradigm*, pages 45–51. 2009.
 - [25] J. GRAY. *Transaction processing : concepts and techniques*. Morgan Kaufmann Publishers, San Mateo, Calif., United States, c1993.
 - [26] R. ELMASRI AND S. B. NAVATHE. *Fundamentals of Database Systems*. Addison Wesley, 4th edition, 2003.
 - [27] C. J. DATE. *Database in Depth*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2005.
 - [28] R. RAMAKRISHNAN AND J. GEHRKE. *Database Management Systems*. McGraw-Hill Science/Engineering/Math., 1221 Ave. of the Americas New York, NY 10020-1095, 3rd edition, 2003.
 - [29] J. DEAN AND S. GHEMAWAT. **MapReduce: a flexible data processing tool.** *Commun. ACM*, **53**(1):72–77, January 2010.
 - [30] J. DEAN AND S. GHEMAWAT. **MapReduce: simplified data processing on large clusters.** *Commun. ACM*, **51**(1):107–113, January 2008.
 - [31] T. WHITE. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2010.
 - [32] C. OLSTON, B. REED, U. SRIVASTAVA, R. KUMAR, AND A. TOMKINS. **Pig latin: a not-so-foreign language for data processing.** In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.
 - [33] R. CHAIKEN, B. JENKINS, P. LARSON, B. RAMSEY, D. SHAKIB, S. WEAVER, AND J. ZHOU. **SCOPE: easy and efficient parallel processing of massive data sets.** *Proc. VLDB Endow.*, **1**(2):1265–1276, August 2008.
 - [34] W. JIANG, V.T. RAVI, AND G. AGRAWAL. **A Map-Reduce System with an Alternate API for Multi-core Environments.** In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 84–93, 2010.

-
- [35] R. PIKE, S. DORWARD, R. GRIESEMER, AND S. QUINLAN. **Interpreting the data: Parallel analysis with Sawzall.** *Sci. Program.*, **13**(4):277–298, October 2005.
- [36] M. ISARD, M. BUDIU, Y. YU, A. BIRRELL, AND D. FETTERLY. **Dryad: distributed data-parallel programs from sequential building blocks.** In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.
- [37] Y. YU, M. ISARD, D. FETTERLY, M. BUDIU, U. ERLINGSSON, P. K. GUNDA, AND J. CURREY. **DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language.** In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.
- [38] A. GUPTA. **Graph Data Management in Scientific Applications.** In L. LIU AND M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 1261–1263. Springer US, 2009.
- [39] S. S. SKIENA. *The Algorithm Design Manual*. Springer, 2 edition, 2008.
- [40] P. M. KIM, J. O. KORBEL, AND M. B. GERSTEIN. **Positive selection at the protein network periphery: evaluation in terms of structural constraints and cellular context.** *Proceedings of the National Academy of Sciences*, **104**(51):20274–20279, 2007.
- [41] J. DUDLEY, Y. POULIOT, R. CHEN, A. MORGAN, AND A. BUTTE. **Translational bioinformatics in the cloud: an affordable alternative.** *Genome Medicine*, **2**(8):51, 2010.
- [42] L. A. ADAMIC AND N. GLANCE. **The political blogosphere and the 2004 U.S. election: divided they blog.** In *Proceedings of the 3rd international workshop on Link discovery*, LinkKDD '05, pages 36–43, New York, NY, USA, 2005. ACM.
- [43] S. EKINS, R. GUPTA, E. GIFFORD, B. BUNIN, AND C. WALLER. **Chemical Space: Missing Pieces in Cheminformatics.** *Pharmaceutical Research*, **27**:2035–2039, 2010. 10.1007/s11095-010-0229-0.
- [44] K. E. JOYCE, P. J. LAURIENTI, J. H. BURDETTE, AND S. HAYASAKA. **A New Measure of Centrality for Brain Networks.** *PLoS ONE*, **5**(8):e12200, 08 2010.
- [45] M. GUYE, G. BETTUS, F. BARTOLOMEI, AND P. J. COZZONE. **Graph theoretical analysis of structural and functional connectivity MRI in normal and pathological brain networks.** *Magnetic Resonance Materials in Physics, Biology and Medicine*, **23**(5-6):409–421, 2010.

-
- [46] Z. J. CHEN, Y. HE, P. ROSA-NETO, J. GERMANN, AND A. C. EVANS. **Revealing modular architecture of human brain structural networks by using cortical thickness from MRI.** *Cerebral cortex*, **18**(10):2374–2381, 2008.
- [47] D. S. BASSETT, E. BULLMORE, B. A. VERCHINSKI, V. S. MATTAY, D. R. WEINBERGER, AND A. MEYER-LINDENBERG. **Hierarchical organization of human cortical networks in health and schizophrenia.** *The Journal of Neuroscience*, **28**(37):9239–9248, 2008.
- [48] M. DAYARATHNA, C. HOUNGKAEW, AND T. SUZUMURA. **Introducing Scale-Graph: an X10 library for billion scale graph analytics.** In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 6:1–6:9, New York, NY, USA, 2012. ACM.
- [49] D. PROUNTZOS, R. MANEVICH, AND K. PINGALI. **Elixir: a system for synthesizing concurrent graph programs.** *SIGPLAN Not.*, **47**(10):375–394, October 2012.
- [50] FACEBOOK INC. **Facebook Annual Report.** Annual report, 2012.
- [51] B. BABCOCK, S. BABU, M. DATAR, R. MOTWANI, AND J. WIDOM. **Models and issues in data stream systems.** In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.
- [52] D. CARNEY, U. ÇETINTEMEL, M. CHERNIACK, C. CONVEY, S. LEE, G. SEIDMAN, M. STONEBRAKER, N. TATBUL, AND S. ZDONIK. **Monitoring streams: a new class of data management applications.** In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.
- [53] D. J. ABADI, Y. AHMAD, M. BALAZINSKA, M. CHERNIACK, J. H. HWANG, W. LINDNER, A. S. MASKEY, E. RASIN, E. RYVKINA, N. TATBUL, Y. X., AND S. ZDONIK. **The design of the borealis stream processing engine.** In *In CIDR*, pages 277–289, 2005.
- [54] A. ARASU, B. BABCOCK, S. BABU, J. CIESLEWICZ, M. DATAR, K. ITO, R. MOTWANI, U. SRIVASTAVA, AND J. WIDOM. **STREAM: The Stanford Data Stream Management System.** Technical Report 2004-20, Stanford InfoLab, 2004.
- [55] B. GEDIK, H. ANDRADE, K. L. WU, P. S. YU, AND M. DOO. **SPADE: the system s declarative stream processing engine.** In *SIGMOD '08*, pages 1123–1134, 2008.
- [56] D. JUNGnickel. *Graphs, networks and algorithms.* Springer, Berlin ; Tokyo, Germany, 1999.

-
- [57] E. LEONARD. **Solutio problematis ad geometriam situs pertinentis**. *Commentarii academiae scientiarum Petropolitanae* 8, 8:128–140, 1736.
- [58] N. L. BIGGS, E. K. LLOYD, AND R. J. WILSON. *Graph Theory 1736-1936*. Oxford University Press, 1998.
- [59] M. NEWMANN. *Networks: An Introduction*. Oxford University Press, 2010.
- [60] G. CSARDI AND T. NEPUSZ. **The igraph software package for complex network research**. *InterJournal, Complex Systems*:1695, 2006.
- [61] SOURCEFORGE. **JUNG - Java Universal Network/Graph Framework**. URL: <http://jung.sourceforge.net/index.html>, April 2013.
- [62] Y. PIGNÉ, A. DUTOT, F. GUINAND, AND D. OLIVIER. **GraphStream: A Tool for bridging the gap between Complex Systems and Dynamic Graphs**. *CoRR*, abs/0803.2093, 2008.
- [63] J. G. SIEK, L. Q. LEE, AND A. LUMSDAINE. *Boost Graph Library: User Guide and Reference Manual, The*. Addison-Wesley Professional, 2001.
- [64] Y. LOW, J. GONZALEZ, A. KYROLA, D. BICKSON, C. GUESTRIN, AND J. M. HELLERSTEIN. **Distributed GraphLab: A Framework for Machine Learning in the Cloud**. *PVLDB*, 5(8):716–727, 2012.
- [65] C. C. AGGARWAL AND H. WANG. **A Survey of Clustering Algorithms for Graph Data**. In C. C. AGGARWAL, H. WANG, AND A. K. ELMAGARMID, editors, *Managing and Mining Graph Data*, 40 of *The Kluwer International Series on Advances in Database Systems*, pages 275–301. Springer US, 2010.
- [66] H. KWAK, C. LEE, H. PARK, AND S. MOON. **What is Twitter, a social network or a news media?** In *Proceedings of the 19th international conference on World wide web*, WWW ’10, pages 591–600, New York, NY, USA, 2010. ACM.
- [67] P. S. PACHECO. *An introduction to parallel programming*. Morgan Kaufmann, Burlington, Mass., United States, c2011.
- [68] U KANG, C. E. TSOURAKAKIS, AND C. FALOUTSOS. **PEGASUS: mining peta-scale graphs**. *Knowledge and Information Systems*, 27(2):303–325, 2011.
- [69] W. JIANG AND G. AGRAWAL. **Ex-MATE: Data Intensive Computing with Large Reduction Objects and Its Application to Graph Mining**. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 475–484, 2011.
- [70] D. GREGOR AND A. LUMSDAINE. **Lifting sequential graph algorithms for distributed-memory parallel computation**. *SIGPLAN Not.*, 40:423–437, October 2005.

-
- [71] F. HIELSCHER AND P. GOTTSCHLING. **ParGraph**. URL: <http://pargraph.sourceforge.net/>, January 2012.
- [72] A. BULUÇ AND J. R. GILBERT. **The Combinatorial BLAS: design, implementation, and applications**. *International Journal of High Performance Computing Applications*, **25**(4):496–509, 2011.
- [73] K. MADDURI, B. HENDRICKSON, J. BERRY, D. BADER, AND J. CROBAK. *Multithreaded Algorithms for Processing Massive Graphs*. 2008.
- [74] D. A. BADER, G. CONG, AND J. FEO. **On the architectural requirements for efficient execution of graph algorithms**. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 547 – 556, june 2005.
- [75] D. EDIGER, K. JIANG, J. RIEDY, D. A. BADER, AND C. CORLEY. **Massive Social Network Analysis: Mining Twitter for Social Good**. In *Proceedings of the 2010 39th International Conference on Parallel Processing, ICPP '10*, pages 583–593, Washington, DC, USA, 2010. IEEE Computer Society.
- [76] J.W. BERRY, B. HENDRICKSON, S. KAHAN, AND P. KONECNY. **Software and Algorithms for Graph Queries on Multithreaded Architectures**. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –14, march 2007.
- [77] G. MALEWICZ, M. H. AUSTERN, A. J. C. BIK, J. C. DEHNERT, I. HORN, N. LEISER, AND G. CZAJKOWSKI. **Pregel: a system for large-scale graph processing**. In *Proceedings of the 2010 international conference on Management of data, SIGMOD '10*, pages 135–146, New York, NY, USA, 2010. ACM.
- [78] B. W. BARRETT, J. W. BERRY, R. C. MURPHY, AND K. B. WHEELER. **Implementing a portable Multi-threaded Graph Library: The MTGL on Qthreads**. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –8, may 2009.
- [79] J. DONGARRA AND ET AL. **The International Exascale Software Project roadmap**. *International Journal of High Performance Computing Applications*, **25**(1):3–60, 2011.
- [80] J. SHALF, S. DOSANJH, AND J. MORRISON. **Exascale computing technology challenges**. In *Proceedings of the 9th international conference on High performance computing for computational science, VECPAR'10*, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.
- [81] M. A. HEROUX. **Software Challenges for Extreme Scale Computing: Going From Petascale to Exascale Systems**. *Int. J. High Perform. Comput. Appl.*, **23**(4):437–439, November 2009.

-
- [82] P. M. PHOTHILIMTHANA, J. ANSEL, J. RAGAN-KELLEY, AND S. AMARASINGHE. **Portable performance on heterogeneous architectures.** *SIGPLAN Not.*, **48**(4):431–444, March 2013.
- [83] V. SARKAR, W. HARROD, AND A. E. SNAVELY. **Software challenges in extreme scale systems.** *Journal of Physics: Conference Series*, **180**(1):012045, 2009.
- [84] S. AMARASINGHE, D. CAMPBELL, W. CARLSON, A. CHIEN, W. DALLY, E. ELNOHAZY, M. HALL, R. HARRISON, W. HARROD, K. HILL, J. HILLER, S. KARP, C. KOELBEL, D. KOESTER, P. KOGGE, J. LEVESQUE, D. REED, V. SARKAR, R. SCHREIBER, M. RICHARDS, A. SCARPELLI, J. SHALF, A. SNAVELY, AND T. STERLING. **ExaScale Software Study: Software Challenges in Extreme Scale Systems.** September 2009.
- [85] J. SHIN, M. W. HALL, J. CHAME, C. CHEN, AND P. D. HOVLAND. **Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology.** In K. NAONO, K. TERANISHI, J. CAVAZOS, AND R. SUDA, editors, *Software Automatic Tuning*, pages 353–370. Springer New York, 2010.
- [86] M. MENDES, P. BIZARRO, AND P. MARQUES. **A Performance Study of Event Processing Systems.** In *Performance Evaluation and Benchmarking*, **5895** of *Lecture Notes in Computer Science*, pages 221–236. 2009.
- [87] B. SNYDER, D. BOSANAC, AND R. DAVIES. *ActiveMQ in Action*. 2011.
- [88] T. SUZUMURA, T. YASUE, AND T. ONODERA. **Scalable performance of system S for extract-transform-load processing.** In *SYSTOR '10*, pages 7:1–7:14, 2010.
- [89] X. J. ZHANG, S. PAREKH, B. GEDIK, H. ANDRADE, AND K. L. WU. **Workload characterization for operator-based distributed stream processing applications.** In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 235–247, New York, NY, USA, 2010. ACM.
- [90] S. PAREKH, K. W. HILDRUM, D. RAJAN, J. L. WOLF, AND K. L. WU. **Characterizing, constructing and managing resource usage profiles of system S applications: challenges and experience.** In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1177–1186, New York, NY, USA, 2009. ACM.
- [91] D. J. ABADI, D. CARNEY, U. ÇETINTEMEL, M. CHERNIACK, C. CONVEY, S. LEE, M. STONEBRAKER, N. TATBUL, AND S. ZDONIK. **Aurora: a new model and architecture for data stream management.** *The VLDB Journal*, **12**(2):120–139, August 2003.

-
- [92] THE STREAM GROUP. **STREAM: The Stanford Stream Data Manager**. Technical Report 2003-21, 2003.
 - [93] E. ZEITLER AND T. RISCH. **Scalable Splitting of Massive Data Streams**. In *DASFAA '10*, pages 184–198, 2010.
 - [94] M. R. N. MENDES AND ET AL. **A framework for performance evaluation of complex event processing systems**. DEBS '08, pages 313–316, 2008.
 - [95] M. R. N. MENDES, P. BIZARRO, AND P. MARQUES. **Towards a standard event processing benchmark**. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 307–310, New York, NY, USA, 2013. ACM.
 - [96] M. DAYARATHNA, Y. TSUKAHARA, AND K. SUGIURA. **TelescopeVisualizer: a real-time internet information visualizer with a flexible user interface**. In *Proceedings of the Sixth Asian Internet Engineering Conference*, AINTEC '10, pages 16–23, New York, NY, USA, 2010. ACM.
 - [97] W. THIES, M. KARZMAREK, AND S. AMARASINGHE. **StreamIt: A Language for Streaming Applications**. In R. N. HORSPOOL, editor, *Compiler Construction*, **2304** of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002.
 - [98] W. THIES AND S. AMARASINGHE. **An empirical characterization of stream programs and its implications for language and compiler design**. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.
 - [99] R. NAMBIAR, N. WAKOU, F. CARMAN, AND M. MAJDALANY. **Transaction Processing Performance Council (TPC): State of the Council 2010**. In RAGHUNATH NAMBIAR AND MEIKEL POESS, editors, *Performance Evaluation, Measurement and Characterization of Complex Systems*, **6417** of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2011.
 - [100] R. ANGLES, A. PRAT-PÉREZ, D. DOMINGUEZ-SAL, AND J. L. LARRIBA-PEY. **Benchmarking database systems for social network applications**. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM.
 - [101] U. BRANDES. **A Faster Algorithm for Betweenness Centrality**. *Journal of Mathematical Sociology*, **25**:163–177, 2001.
 - [102] D. A. BADER, J. FEO, J. GILBERT, J. KEPNER, D. KOESTER, E. LOH, K. MADURI, B. MANN, T. MEUSE, AND E. ROBINSON. **HPC Scalable Graph Analysis Benchmark**. Feb 2009.

-
- [103] D. DOMINGUEZ-SAL, P. URBÓN-BAYES, A. GIMÉNEZ-VAÑÓ, S. GÓMEZ-VILLAMOR, N. MARTÍNEZ-BAZÁN, AND J. L. LARRIBA-PEY. **Survey of graph database performance on the HPC scalable graph analysis benchmark.** In *Proceedings of the 2010 international conference on Web-age information management*, WAIM'10, pages 37–48, Berlin, Heidelberg, 2010. Springer-Verlag.
 - [104] J. PARTNER, A. VUKOTIC, AND N. WATT. *Neo4j in Action*. Manning Publications Company, 2013.
 - [105] THE APACHE SOFTWARE FOUNDATION. **Apache Jena**. URL: <http://jena.apache.org/>, 2013.
 - [106] B. IORDANOV. **HyperGraphDB: A Generalized Graph Database**. In H. T. SHEN, J. PEI, M. T. ZSU, L. ZOU, J. LU, T. W. LING, G. YU, Y. ZHUANG, AND J. SHAO, editors, *Web-Age Information Management*, **6185** of *Lecture Notes in Computer Science*, pages 25–36. Springer Berlin Heidelberg, 2010.
 - [107] N. MARTINEZ-BAZAN, S. GOMEZ-VILLAMOR, AND F. ESCALE-CLAVERAS. **DEX: A high-performance graph database management system**. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 124–127, april 2011.
 - [108] D. DOMINGUEZ-SAL, N. MARTINEZ-BAZAN, V. MUNTES-MULERO, P. BALETA, AND J. L. LARRIBA-PAY. **A discussion on the design of graph database benchmarks**. In *Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems*, TPCTC'10, pages 25–40, Berlin, Heidelberg, 2011. Springer-Verlag.
 - [109] K. MYUNGHWAN AND J. LESKOVEC. **Multiplicative Attribute Graph Model of Real-World Networks**. *Internet Mathematics*, **8**(1-2):113–160, 2012.
 - [110] M. CIGLAN, A. AVERBUCH, AND L. HLUCHY. **Benchmarking Traversal Operations over Graph Databases**. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 186–189, 2012.
 - [111] P. T. WOOD. **Query languages for graph databases**. *SIGMOD Rec.*, **41**(1):50–60, April 2012.
 - [112] J. PARTNER, A. VUKOTIC, AND N. WATT. In *Neo4j in Action*. Manning Publications Co., 2012.
 - [113] GREMLIN. **Gremlin**. URL: <https://github.com/tinkerpop/gremlin/wiki/>, April 2013.
 - [114] F. HOLZSCHUHER AND R. PEINL. **Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j**. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 195–204, New York, NY, USA, 2013. ACM.

-
- [115] THE APACHE SOFTWARE FOUNDATION. **Shindig - Welcome To Apache Shindig**. URL: <http://shindig.apache.org/>, April 2013.
- [116] Y. GUO, Z. PAN, AND J. HEFLIN. **LUBM: A benchmark for OWL knowledge base systems**. *Web Semantics: Science, Services and Agents on the World Wide Web*, **3**(23):158 – 182, 2005.
- [117] C. BIZER AND A. SCHULTZ. **The berlin sparql benchmark**. *International Journal On Semantic Web and Information Systems*, 2009.
- [118] M. MORSEY, J. LEHMANN, S. AUER, AND A. C. N. NGOMO. **DBpedia SPARQL Benchmark - Performance Assessment with Real Queries on Real Data**. In *International Semantic Web Conference (1)’11*, pages 454–469, 2011.
- [119] M. SCHMIDT, T. HORNING, G. LAUSEN, AND C. PINKEL. **SP2Bench: A SPARQL Performance Benchmark**. *CoRR*, abs/0806.4627, 2008.
- [120] C. VICKNAIR, M. MACIAS, Z. ZHAO, X. NAN, Y. CHEN, AND D. WILKINS. **A comparison of a graph database and a relational database: a data provenance perspective**. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE ’10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.
- [121] K. ROHLOFF, M. DEAN, I. EMMONS, D. RYDER, AND J. SUMNER. **An Evaluation of Triple-Store Technologies for Large Data Stores**. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, **4806** of *Lecture Notes in Computer Science*, pages 1105–1114. Springer Berlin / Heidelberg, 2007.
- [122] D. THAKKER, T. OSMAN, S. GOHIL, AND P. LAKIN. **A Pragmatic Approach to Semantic Repositories Benchmarking**. In L. AROYO, G. ANTONIOU, E. HYVNEN, A. TEIJE, H. STUCKENSCHMIDT, L. CABRAL, AND T. TUDORACHE, editors, *The Semantic Web: Research and Applications*, **6088** of *Lecture Notes in Computer Science*, pages 379–393. Springer Berlin / Heidelberg, 2010.
- [123] L. MA, Y. YANG, Z. QIU, G. XIE, Y. PAN, AND S. LIU. **Towards a Complete OWL Ontology Benchmark**. In Y. SURE AND J. DOMINGUE, editors, *The Semantic Web: Research and Applications*, **4011** of *Lecture Notes in Computer Science*, pages 125–139. Springer Berlin / Heidelberg, 2006.
- [124] R. MURPHY, J. BERRY, W. MCLENDON, B. HENDRICKSON, D. GREGOR, AND A. LUMSDAINE. **DFS: A Simple to Write Yet Difficult to Execute Benchmark**. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 175 –177, oct. 2006.
- [125] B.M. CHAPMAN, H. HERBECK, AND H.P. ZIMA. **Automatic Support for Data Distribution**. In *DMCC*, pages 51 –58, May 1991.

-
- [126] D. J. PALERMO, E. W. HODGES IV, AND P. BANERJEE. **Compiler Optimization of Dynamic Data Distributions for Distributed-Memory Multicomputers**. In S. PANDE AND D. P. AGRAWAL, editors, *Compiler Optimizations for Scalable Parallel Systems*, **1808** of *Lecture Notes in Computer Science*, pages 445–484. Springer Berlin Heidelberg, 2001.
- [127] P. BANERJEE, J. A. CHANDY, M. GUPTA, E. W. HODGES IV, J. G. HOLM, A. LAIN, D. J. PALERMO, S. RAMASWAMY, AND E. SU. **The Paradigm Compiler for Distributed-Memory Multicomputers**. *Computer*, **28**:37–47, Oct 1995.
- [128] M. GUPTA AND P. BANERJEE. **PARADIGM: a compiler for automatic data distribution on multicomputers**. In *Proceedings of the 7th international conference on Supercomputing*, ICS '93, pages 87–96, New York, NY, USA, 1993. ACM.
- [129] J. QIN, T. FAHRINGER, AND R. PRODAN. **A novel graph based approach for automatic composition of high quality grid workflows**. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 167–176, New York, NY, USA, 2009. ACM.
- [130] C. S. LIEW, M. P. ATKINSON, J. I. V. HEMERT, AND L. HAN. **Towards optimising distributed data streaming graphs using parallel streams**. In *HPDC '10*, pages 725–736, New York, NY, USA, 2010. ACM.
- [131] S. PELLEGRINI, T. FAHRINGER, H. JORDAN, AND H. MORITSCH. **Automatic tuning of MPI runtime parameter settings by using machine learning**. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 115–116, New York, NY, USA, 2010. ACM.
- [132] A. FARAJ AND X. YUAN. **Automatic generation and tuning of MPI collective communication routines**. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM.
- [133] A. FARAJ, X. YUAN, AND D. LOWENTHAL. **STAR-MPI: self tuned adaptive routines for MPI collective operations**. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 199–208, New York, NY, USA, 2006. ACM.
- [134] Y. CHEN, H. SONG, R. THAKUR, AND X. H. SUN. **A layout-aware optimization strategy for collective I/O**. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 360–363, New York, NY, USA, 2010. ACM.
- [135] G. YAIKHOM, C. LIEW, L. HAN, H. VAN, M. ATKINSON, AND A. KRAUSE. **Federated Enactment of Workflow Patterns**. In *Euro-Par 2010 - Parallel Processing*, **6271**, pages 317–328. 2010.

-
- [136] M. HALL, J. CHAME, C. CHEN, J. SHIN, G. RUDY, AND M. M. KHAN. **Loop transformation recipes for code generation and auto-tuning.** In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 50–64, Berlin, Heidelberg, 2010. Springer-Verlag.
- [137] L. T. YANG, X. MA, AND F. MUELLER. **Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution.** In *SC '05*, Washington, DC, USA, 2005.
- [138] S. SODHI, J. SUBHLOK, AND Q. XU. **Performance prediction with skeletons.** *Cluster Computing*, **11**:151–165, 2008.
- [139] R. SUDA. **A Bayesian Method of Online Automatic Tuning.** In K. NAONO, K. TERANISHI, J. CAVAZOS, AND R. SUDA, editors, *Software Automatic Tuning*, pages 275–293. Springer New York, 2010.
- [140] P. BALAPRAKASH, S. M. WILD, AND B. NORRIS. **SPAPT: Search Problems in Automatic Performance Tuning.** *Procedia Computer Science*, **9**(0):1959 – 1968, 2012. Proceedings of the International Conference on Computational Science, ICCS'2012.
- [141] S. BELLAMKONDA, R. AHMED, A. WITKOWSKI, A. AMOR, M. ZAIT, AND C. C. LIN. **Enhanced subquery optimizations in Oracle.** *Proc. VLDB Endow.*, **2**:1366–1377, 2009.
- [142] R. AHMED, A. LEE, A. WITKOWSKI, D. DAS, H. SU, M. ZAIT, AND T. CRUANES. **Cost-based query transformation in Oracle.** *VLDB '06*, pages 1026–1036, 2006.
- [143] N. KABRA AND D. J. DEWITT. **Efficient mid-query re-optimization of sub-optimal query execution plans.** *SIGMOD '98*, pages 106–117, 1998.
- [144] H. HERODOTOU, N. BORISOV, AND S. BABU. **Query optimization techniques for partitioned tables.** In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 49–60, New York, NY, USA, 2011. ACM.
- [145] R. KHANDEKAR, K. HILDRUM, S. PAREKH, D. RAJAN, J. WOLF, K. L. WU, H. ANDRADE, AND B. GEDIK. **COLA: Optimizing Stream Processing Applications via Graph Partitioning.** In J. M. BACON AND B. F. COOPER, editors, *Middleware 2009*, **5896** of *Lecture Notes in Computer Science*, pages 308–327. Springer Berlin Heidelberg, 2009.
- [146] Z. WANG AND M. F. P. O'BOYLE. **Partitioning streaming parallelism for multi-cores: a machine learning based approach.** In *PACT '10*, pages 307–318, 2010.

-
- [147] S. BABU. **Towards automatic optimization of MapReduce programs.** SoCC '10, pages 137–142, 2010.
- [148] F. HUESKE, M. PETERS, A. KRETTEK, M. RINGWALD, K. TZOUMAS, V. MARKL, AND J. FREYTAG. **Peeking into the optimization of data flow programs with MapReduce-style UDFs.** In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1292–1295, 2013.
- [149] H. LIM, H. HERODOTOU, AND S. BABU. **Stubby: a transformation-based optimizer for MapReduce workflows.** *Proc. VLDB Endow.*, **5**(11):1196–1207, July 2012.
- [150] D. FETTERLY Y. YU, M. ISARD AND M. BUDIU. **DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.** OSDI '08, pages 1–14, 2008.
- [151] A. A. FREITAS. *Data Mining and Knowledge Discovery with Evolutionary Algorithms.* Springer-Verlag New York, Inc., 2002.
- [152] W. B. LANGDON AND R. POLI. *Foundations of Genetic Programming.* Springer-Verlag New York, Inc., 2002.
- [153] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHALIAN, M. HOUSTON, AND P. HANRAHAN. **Brook for GPUs: stream computing on graphics hardware.** *ACM Trans. Graph.*, **23**(3):777–786, August 2004.
- [154] M. DRAKE, H. HOFFMANN, R. RABBAH, AND S. AMARASINGHE. **MPEG-2 decoding in a stream programming language.** In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 2006.
- [155] S. AMARASINGHE, M. GORDON, M. KARCZMAREK, J. LIN, D. MAZE, R. M. RABBAH, AND W. THIES. **Language and Compiler Design for Streaming Applications.** *International Journal of Parallel Programming*, **33**(2-3):261–278, 2005.
- [156] C. CONSEL, H. HAMDI, L. RVEILLRE, L. SINGARAVELU, H. YU, AND C. PU. **Spidle: A DSL Approach to Specifying Streaming Applications.** In F. PFENNING AND Y. SMARAGDAKIS, editors, *Generative Programming and Component Engineering*, **2830** of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2003.
- [157] W. R. MARK, R. S. GLANVILLE, K. AKELEY, AND M. J. KILGARD. **Cg: a system for programming graphics hardware in a C-like language.** *ACM Trans. Graph.*, **22**(3):896–907, July 2003.
- [158] M. I. GORDON, W. THIES, M. KARCZMAREK, J. LIN, A. S. MELI, A. A. LAMB, C. LEGER, J. WONG, H. HOFFMANN, D. MAZE, AND S. AMARASINGHE.

- A stream compiler for communication-exposed architectures.** In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS X, pages 291–303, New York, NY, USA, 2002. ACM.
- [159] M. I. GORDON, W. THIES, AND S. AMARASINGHE. **Exploiting coarse-grained task, data, and pipeline parallelism in stream programs.** *SIGARCH Comput. Archit. News*, **34**(5):151–162, October 2006.
- [160] A. A. LAMB, W. THIES, AND S. AMARASINGHE. **Linear analysis and optimization of stream programs.** In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 12–25, New York, NY, USA, 2003. ACM.
- [161] P. CASTRO, O., S. LOUISE, AND D. BARTHOU. **Reducing memory requirements of stream programs by graph transformations.** In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 171–180, 2010.
- [162] G. BILSEN, M. ENGELS, R. LAUWEREINS, AND J. PEPPERSTRAETE. **Cycle-static dataflow.** *Signal Processing, IEEE Transactions on*, **44**(2):397–408, 1996.
- [163] S. SESHADRI, B. BAMBA, B. F. COOPER, V. KUMAR, L. LIU, K. SCHWAN, AND G. ZHANG. **Grouping Distributed Stream Query Services by Operator Similarity and Network Locality.** In *Proceedings of the 2008 IEEE Congress on Services - Part I*, SERVICES '08, pages 11–18, Washington, DC, USA, 2008. IEEE Computer Society.
- [164] S. SESHADRI, V. KUMAR, B. COOPER, AND L. LIU. **A Distributed Stream Query Optimization Framework through Integrated Planning and Deployment.** *Parallel and Distributed Systems, IEEE Transactions on*, **20**(10):1439–1453, 2009.
- [165] M. DAYARATHNA AND T. SUZUMURA. **Hirundo: a mechanism for automated production of optimized data stream graphs.** In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 335–346, New York, NY, USA, 2012. ACM.
- [166] S. SCHNEIDER, M. HIRZEL, B. GEDIK, AND K. L. WU. **Auto-parallelizing stateful distributed streaming applications.** In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 53–64, New York, NY, USA, 2012. ACM.
- [167] G. LOSA, V. KUMAR, H. ANDRADE, B. GEDIK, M. HIRZEL, R. SOULÉ, AND K. L. WU. **CAPSULE: language and system support for efficient state sharing in distributed stream processing systems.** In *Proceedings of the 6th*

- ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 268–277, New York, NY, USA, 2012. ACM.
- [168] J. WOLF, N. BANSAL, K. HILDRUM, S. PAREKH, D. RAJAN, R. WAGLE, K. L. WU, AND L. FLEISCHER. **SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems**. In V. ISSARNY AND R. SCHANTZ, editors, *Middleware 2008*, **5346** of *Lecture Notes in Computer Science*, pages 306–325. Springer Berlin Heidelberg, 2008.
- [169] B. GEDIK, H. ANDRADE, AND K. L. WU. **A code generation approach to optimizing high-performance distributed data stream processing**. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 847–856, New York, NY, USA, 2009. ACM.
- [170] J. H. HWANG, U. CETINTEMEL, AND S. ZDONIK. **Fast and Highly-Available Stream Processing over Wide Area Networks**. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 804–813, april 2008.
- [171] J. H. HWANG, Y. XING, U. CETINTEMEL, AND S. ZDONIK. **A Cooperative, Self-Configuring High-Availability Solution for Stream Processing**. pages 176–185, april 2007.
- [172] Y. GU, Z. ZHANG, F. YE, H. YANG, M. KIM, H. LEI, AND Z. LIU. **An empirical study of high availability in stream processing systems**. *Middleware '09*, pages 23:1–23:9, New York, NY, USA, 2009. Springer-Verlag New York, Inc.
- [173] Z. ZHANG, Y. GU, F. YE, H. YANG, M. KIM, H. LEI, AND Z. LIU. **A Hybrid Approach to High Availability in Stream Processing Systems**. In *Distributed Computing Systems (ICDCS)*, pages 138–148, june 2010.
- [174] D. LOGOTHETIS, C. TREZZO, K. C. WEBB, AND K. YOCUM. **In-situ MapReduce for log processing**. *USENIXATC'11*, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.
- [175] Y. WANG, W. LU, B. YU, AND B. WEI. **HAaaS: Towards Highly Available Distributed Systems**. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 618–621, sept. 2012.
- [176] P. HUNT, M. KONAR, F. P. JUNQUEIRA, AND B. REED. **ZooKeeper: wait-free coordination for internet-scale systems**. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, *USENIXATC'10*, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [177] L. Q. LEE, J. G. SIEK, AND A. LUMSDAINE. **The generic graph component library**. *SIGPLAN Not.*, **34**:399–414, October 1999.

-
- [178] J. LESKOVEC. **SNAP: Stanford Network Analysis Project**. URL: <http://snap.stanford.edu/>, January 2012.
- [179] R. GARCIA, J. JARVI, A. LUMSDAINE, J. G. SIEK, AND J. WILLCOCK. **A comparative study of language support for generic programming**. OOPSLA '03, pages 115–134, New York, NY, USA, 2003. ACM.
- [180] D. BATENKOV. **Boosting productivity with the Boost Graph Library**. *XRDS*, 17:31–32, March 2011.
- [181] J. LAW. **Review of "The boost graph library: user guide and reference manual by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine."** Addison-Wesley 2002. *ACM SIGSOFT Software Engineering Notes*, 28(2):35–36, 2003.
- [182] V. SARKAR K. EBCIOGLU, V. SARASWAT. **X10: Programming for Hierarchical Parallelism and Non-uniform data access**. In *3rd International Workshop on Language Runtimes, Impact of Next Generation Processor Architectures on Virtual Machine Technologies*, 2004.
- [183] P. AN, A. JULA, S. RUS, S. SAUNDERS, T. SMITH, G. TANASE, N. THOMAS, N. AMATO, AND L. RAUCHWERGER. **STAPL: an adaptive, generic parallel C++ library**. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing, LCPC'01*, pages 193–208, Berlin, Heidelberg, 2003. Springer-Verlag.
- [184] K. PINGALI, D. NGUYEN, M. KULKARNI, M. BURTSCHER, M. A. HASSAAN, R. KALEEM, T. H. LEE, A. LENHARTH, R. MANEVICH, M. MÉNDEZ-LOJO, D. PROUNTZOS, AND X. SUI. **The tao of parallelism in algorithms**. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 12–25, New York, NY, USA, 2011. ACM.
- [185] J. O'MADADHAIN, D. FISHER, S. WHITE, AND Y. BOEY. **The JUNG (Java Universal Network/Graph) Framework**. Technical report, UCI-ICS, October 2003.
- [186] A. LUGOWSKI, D. M. ALBER, A. BULUÇ, J. R. GILBERT, S. REINHARDT, Y. TENG, AND A. WARANIS. **A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis**. In *SDM*, pages 930–941, 2012.
- [187] G. CONG, G. ALMASI, AND V. SARASWAT. **Fast PGAS Implementation of Distributed Graph Algorithms**. SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [188] G. CONG, G. ALMASI, AND V. SARASWAT. **Fast PGAS connected components algorithms**. PGAS '09, pages 13:1–13:6, New York, NY, USA, 2009. ACM.

-
- [189] M. KULKARNI, K. PINGALI, B. WALTER, G. RAMANARAYANAN, K. BALA, AND L. P. CHEW. **Optimistic parallelism requires abstractions.** In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.
- [190] M. W. HALL, R. M. KIRBY, F. LI, M. D. MEYER, V. PASCUCCHI, J. M. PHILLIPS, R. RICCI, J. E. VAN DER MERWE, AND S. VENKATASUBRAMANIAN. **Rethinking Abstractions for Big Data: Why, Where, How, and What.** *CoRR*, abs/1306.3295, 2013.
- [191] S. HONG, H. CHAFI, E. SEDLAR, AND K. OLUKOTUN. **Green-Marl: a DSL for easy and efficient graph analysis.** In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, New York, NY, USA, 2012. ACM.
- [192] S. HONG, J. VAN DER LUGT, A. WELC, R. RAMAN, AND H. CHAFI. **Early experiences in using a domain-specific language for large-scale graph analysis.** In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.
- [193] THE IGRAPH PROJECT. **The igraph library.** URL: <http://igraph.sourceforge.net/>, March 2013.
- [194] M. A. RODRIGUEZ AND P. NEUBAUER. **The Graph Traversal Pattern.** *CoRR*, abs/1004.1001, 2010.
- [195] GREMLIN. **Gremlin.** URL: <https://github.com/tinkerepop/gremlin/wiki/>, May 2012.
- [196] M. A. RODRIGUEZ AND P. NEUBAUER. **Constructions from dots and lines.** *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.
- [197] W3C. **SPARQL Query Language for RDF.** URL: <http://www.w3.org/TR/rdf-sparql-query/>, June 2012.
- [198] C. BUNCH, B. DRAWERT, N. CHOCHAN, C. KRINTZ, L. PETZOLD, AND K. SHAMS. **Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics.** *Journal of Grid Computing*, 10:23–46, 2012.
- [199] UPC CONSORTIUM. *UPC Language Specifications, v1.2.* Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005., 2011.
- [200] P. CHARLES, C. GROTHOFF, V. SARASWAT, C. DONAWA, A. KIELSTRA, K. EBCIOGLU, P. C. VON, AND V. SARKAR. **X10: an object-oriented approach to non-uniform cluster computing.** In *Proceedings of the 20th annual*

- ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.
- [201] S. DAS, Y. SISMANIS, K. S. BEYER, R. GEMULLA, P. J. HAAS, AND J. MCPHERSON. **Ricardo: integrating R and Hadoop**. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 987–998, New York, NY, USA, 2010. ACM.
- [202] J. ANSEL, C. CHAN, Y. L. WONG, M. OLSZEWSKI, Q. ZHAO, A. EDELMAN, AND S. AMARASINGHE. **PetaBricks: a language and compiler for algorithmic choice**. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.
- [203] C. CHAN, J. ANSEL, Y. L. WONG, S. AMARASINGHE, AND A. EDELMAN. **Autotuning multigrid with PetaBricks**. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 5:1–5:12, New York, NY, USA, 2009. ACM.
- [204] K. KENNEDY, B. BROOM, A. CHAUHAN, R.J. FOWLER, J. GARVIN, C. KOELBEL, C. MCCOSH, AND J. MELLOR-CRUMMEY. **Telescoping Languages: A System for Automatic Generation of Domain Languages**. *Proceedings of the IEEE*, **93**(2):387–408, feb. 2005.
- [205] C. JOSLYN, S. CHOUDHURY, D. HAGLIN, B. HOWE, B. NICKLESS, AND B. OLSEN. **Massive scale cyber traffic analysis: a driver for graph database research**. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 3:1–3:6, New York, NY, USA, 2013. ACM.
- [206] NEO4J. **Learn, Develop, Participate - Neo4j: The World's Leading Graph Database**. URL: <http://www.neo4j.org/>, February 2013.
- [207] ORIENT TECHNOLOGIES. **OrientDB Graph-Document NoSQL dbms**. URL: <http://www.orientdb.org/>, February 2013.
- [208] ALLEGROGRAPH. **AllegroGraph RDFStore Web 3.0's Database**. URL: <http://www.franz.com/agraph/allegrograph/>, February 2013.
- [209] L. Y. HO, J. J. WU, AND P. LIU. **Distributed Graph Database for Large-Scale Social Computing**. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 455–462, june 2012.
- [210] L. BARGUÑO, V. MUNTÉS-MULERO, D. DOMINGUEZ-SAL, AND P. VALDURIEZ. **ParallelGDB: a parallel graph database based on cache specialization**. In *Proceedings of the 15th Symposium on International Database Engineering and Applications*, IDEAS '11, pages 162–169, New York, NY, USA, 2011. ACM.

-
- [211] Y. LI B. SHAO, H. WANG. **The Trinity Graph Engine**. *Technical Report 161291, Microsoft Research*, 2012.
- [212] B. SHAO, H. WANG, AND Y. XIAO. **Managing and mining large graphs: systems and implementations**. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 589–592, New York, NY, USA, 2012. ACM.
- [213] B. SHAO, H. WANG, AND Y. LI. **Trinity: a distributed graph engine on a memory cloud**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.
- [214] AURELIUS. **TITAN: Distributed Graph Database**. URL: <http://thinkaurelius.github.com/titan/>, December 2012.
- [215] OBJECTIVITY. **Infinite Graph Overview**. URL: <http://objectivity.com/INFINITEGRAPH>, February 2013.
- [216] FLOCKDB. **FlockDB**. URL: <https://github.com/twitter/flockdb>, April 2013.
- [217] V. GULISANO, R. JIMENEZ-PERIS, M. PATIO-MARTINEZ, C. SORIENTE, AND P. VALDURIEZ. **StreamCloud: An Elastic and Scalable Data Streaming System**. *Parallel and Distributed Systems, IEEE Transactions on*, **23**(12):2351–2365, dec. 2012.
- [218] W. KLEIMINGER, E. KALYVIANAKI, AND P. PIETZUCH. **Balancing load in stream processing with the cloud**. In *Data Engineering Workshops (ICDEW)*, pages 16–21, april 2011.
- [219] A. ISHII AND T. SUZUMURA. **Elastic Stream Computing with Clouds**. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 195–202, july 2011.
- [220] Z. YAN-HUA, F. LEI, AND Y. ZHI. **Optimization of Cloud Database Route Scheduling Based on Combination of Genetic Algorithm and Ant Colony Algorithm**. *Procedia Engineering*, **15**(0):3341–3345, 2011.
- [221] S. DAS, S. AGARWAL, D. AGRAWAL, AND A. E. ABBADI. **ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud**. *UCSB Computer Science Technical Report 2010-04.*, 2010.
- [222] T. YU, J. QIU, B. REINWALD, L. ZHI, Q. WANG, AND N. WANG. **Intelligent Database Placement in Cloud Environment**. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 544–551, june 2012.

-
- [223] U. F. MINHAS, S. RAJAGOPALAN, B. CULLY, A. ABOULNAGA, K. SALEM, AND A. WARFIELD. **RemusDB: transparent high availability for database systems.** *The VLDB Journal*, **22**:29–45, 2013.
- [224] E. ANGELOU, N. PAPAILIOU, I. KONSTANTINOY, D. TSOUMAKOS, AND N. KOZIRIS. **Automatic scaling of selective SPARQL joins using the TIRAMOLA system.** In *Proceedings of the 4th International Workshop on Semantic Web Information Management, SWIM '12*, pages 1:1–1:8, New York, NY, USA, 2012. ACM.
- [225] R. CHEN, M. YANG, X. WENG, B. CHOI, B. HE, AND X. LI. **Improving large graph processing on partitioned graphs in the cloud.** In *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12*, pages 3:1–3:13, New York, NY, USA, 2012. ACM.
- [226] S. SCHNEIDER, M. HIRZEL, AND B. GEDIK. **Tutorial: Stream Processing Optimizations.** In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13*, New York, NY, USA, 2013. ACM.
- [227] R. SUDA, K. NAONO, K. TERANISHI, AND J. CAVAZOS. **Software Automatic Tuning: Concepts and State-of-the-Art Results.** In K. NAONO, K. TERANISHI, J. CAVAZOS, AND R. SUDA, editors, *Software Automatic Tuning*, pages 3–15. Springer New York, 2010.
- [228] P.G. WHITING AND R.S.V. PASCOE. **A history of data-flow languages.** *Annals of the History of Computing, IEEE*, **16**(4):38–59, 1994.
- [229] J. B. DENNIS. **Data Flow Graphs.** In D. PADUA, editor, *Encyclopedia of Parallel Computing*, pages 512–518. Springer US, 2011.
- [230] F. HUESKE, M. PETERS, M. J. SAX, A. RHEINLÄNDER, R. BERGMANN, A. KRETTEK, AND K. TZOUMAS. **Opening the black boxes in data flow optimization.** *Proc. VLDB Endow.*, **5**(11):1256–1267, July 2012.
- [231] IBM. **IBM InfoSphere Streams Version 1.2: Programming Model and Language Reference.** February 2010.
- [232] IBM. **IBM InfoSphere Streams Version 1.2.1: Installation and Administration Guide.** October 2010.
- [233] C. BALLARD AND ET AL. *IBM Infosphere Streams: Harnessing Data in Motion.* IBM, 2010.
- [234] M. HIRZEL, H. ANDRADE, B. GEDIK, V. KUMAR, G. LOSA, M. MENDELL, H. NASGAARD, R. SOULE, AND K. L. WU. **SPL Stream Processing Language Specification.** November 2009.

-
- [235] M. HIRZEL, H. ANDRADE, B. GEDIK, G. JACQUES-SILVA, R. KHANDEKAR, V. KUMAR, M. MENDELL, H. NASGAARD, S. SCHNEIDER, R. SOULE, AND K.-L. WU. **IBM Streams Processing Language: Analyzing Big Data in motion.** *IBM Journal of Research and Development*, **57**(3/4):7:1–7:11, 2013.
 - [236] A. BIEM, E. BOUILLET, H. FENG, A. RANGANATHAN, A. RIABOV, O. VERSCHEURE, H. KOUTSOPOULOS, AND C. MORAN. **IBM infosphere streams for scalable, real-time, intelligent transportation services.** In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1093–1104, New York, NY, USA, 2010. ACM.
 - [237] H. ANDRADE, B. GEDIK, K.-L. WU, AND P.S. YU. **Processing high data rate streams in System S.** *Journal of Parallel and Distributed Computing*, **71**(2):145 – 156, 2011. Data Intensive Computing.
 - [238] H. BALAKRISHNAN, M. BALAZINSKA, D. CARNEY, U. CETINTEMEL, M. CHERNIACK, C. CONVEY, E. GALVEZ, J. SALZ, M. STONEBRAKER, N. TATBUL, R. TIBBETTS, AND S. ZDONIK. **Retrospective on Aurora.** *The VLDB Journal*, **13**(4):370–383, December 2004.
 - [239] A. ARASU, B. BABCOCK, S. BABU, M. DATAR, K. ITO, R. MOTWANI, I. NISHIZAWA, U. SRIVASTAVA, D. THOMAS, R. VARMA, AND J. WIDOM. **STREAM: The Stanford Stream Data Manager.** *IEEE Data Eng. Bull.*, **26**(1):19–26, 2003.
 - [240] S. CHANDRASEKARAN, O. COOPER, A. DESHPANDE, M. J. FRANKLIN, J. M. HELLERSTEIN, W. HONG, S. KRISHNAMURTHY, S. R. MADDEN, F. REISS, AND M. A. SHAH. **TelegraphCQ: continuous dataflow processing.** In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.
 - [241] ESPERTECH. **Esper - Complex Event Processing.** URL: <http://esper.codehaus.org/>, February 2012.
 - [242] J. DONGARRA, R. GRAYBILL, W. HARROD, R. LUCAS, E. LUSK, P. LUSZCZEK, J. MCMAHON, A. SNAVELY, J. VETTER, K. YELICK, S. ALAM, R. CAMPBELL, L. CARRINGTON, T. Y. CHEN, O. KHALILI, J. MEREDITH, AND M. TIKIR. **DARPA's {HPCS} Program: History, Models, Tools, Languages.** In M. V. ZELKOWITZ, editor, *Advances in COMPUTERS High Performance Computing*, **72** of *Advances in Computers*, pages 1 – 100. Elsevier, 2008.
 - [243] UNITED STATES. PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITTEE. *Computational Science: Ensuring America's Competitiveness.* National Coordination Office for Information Technology Research & Development, 2005.
 - [244] D. CALLAHAN, B.L. CHAMBERLAIN, AND H.P. ZIMA. **The cascade high productivity language.** In *High-Level Parallel Programming Models and Supportive*

- Environments, 2004. Proceedings. Ninth International Workshop on*, pages 52–60, 2004.
- [245] B.L. CHAMBERLAIN, D. CALLAHAN, AND H.P. ZIMA. **Parallel Programmability and the Chapel Language**. *Int. J. High Perform. Comput. Appl.*, **21**(3):291–312, August 2007.
- [246] IBM. **X10 Language Specification (Version 2.2)**. January 2012.
- [247] E. ALLEN, D. CHASE, J. HALLETT, V. LUCHANGCO, J. W. MAESSEN, S. RYU, STEELE G. L., AND S. TOBIN-HOCHSTADT. **The Fortress Language Specification**. Technical report, Sun Microsystems, Inc., 2007.
- [248] G. ALMASI. **PGAS (Partitioned Global Address Space) Languages**. In D. PADUA, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer US, 2011.
- [249] E. TEJEDOR, M. FARRERAS, D. GROVE, R. M. BADIA, G. ALMASI, AND J. LABARTA. **A high-productivity task-based programming model for clusters**. *Concurr. Comput. : Pract. Exper.*, **24**(18):2421–2448, December 2012.
- [250] G. STEELE. **Fortress Wrapping Up**. URL: https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up, 2013.
- [251] A. PRANTL, T. EPPERLY, S. IMAM, AND V. SARKAR. **Interfacing Chapel with Traditional HPC Programming Languages**. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS ’11, 2011.
- [252] IBM. **X10: Performance and Productivity at Scale**. URL: <http://x10-lang.org/>, January 2012.
- [253] S. AGARWAL, R. BARIK, V. SARKAR, AND R. K. SHYAMASUNDAR. **May-happen-in-parallel analysis of X10 programs**. PPOPP ’07, pages 183–193, 2007.
- [254] D. GROVE, O. TARDIEU, D. CUNNINGHAM, B. HERTA, I. PESHANSKY, AND V. SARASWAT. **A Performance Model for X10 Applications: What’s going on under the hood?** 2011.
- [255] O. TARDIEU, D. GROVE, B. BLOOM, D. CUNNINGHAM, B. HERTA, P. KAMBADUR, V. SARASWAT, A. SHINNAR, M. TAKEUCHI, AND M. VAZIRI. **X10 for productivity and performance at scale**. *A Submission to the 2012 HPC Class II Challenge*, 2012.
- [256] D. CUNNINGHAM, R. BORDAWEKAR, AND V. SARASWAT. **GPU Programming in a High Level Language: Compiling X10 to CUDA**. 2011.

-
- [257] J. MILTHORPE, V. GANESH, A.P. RENDELL, AND D. GROVE. **X10 as a Parallel Language for Scientific Computation: Practice and Experience**. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1080–1088, may 2011.
- [258] I. ROBINSON, J. WEBBER, AND E. EIFREM. *Graph Databases*. O'Reilly Media, Incorporated, 2013.
- [259] E. F. CODD. **A relational model of data for large shared data banks**. *Commun. ACM*, **13**(6):377–387, June 1970.
- [260] R. ANGLES AND C. GUTIERREZ. **Survey of graph database models**. *ACM Comput. Surv.*, **40**(1):1:1–1:39, February 2008.
- [261] V. CHRISTOPHIDES. **Resource Description Framework (RDF) Schema (RDFS)**. In L. LIU AND M. T. ZSU, editors, *Encyclopedia of Database Systems*, pages 2425–2428. Springer US, 2009.
- [262] W3C. **RDF Primer**. URL: <http://www.w3.org/TR/rdf-primer/>, June 2013.
- [263] D. C. FAYE, O. CURE, AND G. BLIN. **A survey of RDF storage approaches**. *ARIMA Journal*, **15**:11–35, 2012.
- [264] FRANZ INC. **AllegroGraph RDFStore Web 3.0's Database**. URL: <http://www.franz.com/agraph/allegrograph/>, 2013.
- [265] J. AASMAN. **Allegro Graph: RDF Triple Database**. Technical report, Technical Report 1, Franz Incorporated, 2006.
- [266] B. HASLHOFER, E. MOMENI, B. SCHANDL, AND S. ZANDER. **Europeana RDF Store Report**. Technical report, University of Vienna, Vienna, March 2011.
- [267] FRANZ INC. **AllegroGraph 4.11 Introduction**. URL: <http://www.franz.com/agraph/support/documentation/v4/agraph-introduction.html>, 2013.
- [268] B. HASLHOFER, E. MOMENI, B. SCHANDL, AND S. ZANDER. **Europeana RDF store report**. 2011.
- [269] THE APACHE SOFTWARE FOUNDATION. **Fuseki: serving RDF data over HTTP**. URL: http://jena.apache.org/documentation/serving_data/, 2013.
- [270] NEO4J. **Neo4j Heroku Add-on**. URL: <http://www.neo4j.org/develop/heroku>, December 2012.
- [271] R. ANGLES. **A Comparison of Current Graph Database Models**. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177, 2012.

-
- [272] AURELIUS. **Titan: Distributed Graph Database**. URL: <http://thinkaurelius.github.io/titan>, 2013.
- [273] ORACLE. **Oracle Berkeley DB Java Edition**. URL: <http://www.oracle.com/technetwork/database/berkeleydb>, 2013.
- [274] FRAMES. **Frames**. URL: <https://github.com/tinkerpop/frames>, June 2013.
- [275] AURELIUS. **Rexster**. URL: <https://github.com/tinkerpop/rexster/wiki>, December 2012.
- [276] BLUEPRINTS. **Blueprints**. URL: <https://github.com/tinkerpop/blueprints>, June 2013.
- [277] M. DAYARATHNA, S. TAKENO, AND T. SUZUMURA. **A performance study on operator-based stream processing systems**. *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 0:79, 2011.
- [278] M. DAYARATHNA AND T. SUZUMURA. **A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking**. In K. JOSHI, M. SIEGLE, M. STOELINGA, AND P. R. DARGENIO, editors, *Quantitative Evaluation of Systems*, **8054** of *Lecture Notes in Computer Science*, pages 225–240. Springer Berlin Heidelberg, 2013.
- [279] SOURCEFORGE. *OProfile - A System Profiler for Linux*. June 2011. URL: <http://oprofile.sourceforge.net>.
- [280] NMON. *nmon for Linux*. June 2011. URL: <http://nmon.sourceforge.net>.
- [281] V. CORTELLESA, A. D. MARCO, AND P. INVERARDI. *Model-Based Software Performance Analysis*. 2011.
- [282] INFINIBAND TRADE ASSOCIATION. **Infiniband Architecture Specification Volume 1**. Technical report, November 2007.
- [283] D. C. LUCKHAM. *The power of events : an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley, Boston, United States, c2002.
- [284] M. DAYARATHNA AND T. SUZUMURA. **A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters**. In W. MENG, L. FENG, S. BRESSAN, W. WINIWARTER, AND W. SONG, editors, *Database Systems for Advanced Applications*, **7826** of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin Heidelberg, 2013.
- [285] THE HSQL DEVELOPMENT GROUP. **HSQLDB**. URL: <http://hsqldb.org/>, 2013.

-
- [286] T. PARR. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2011.
- [287] S. AKRAM, M. MARAZAKIS, AND A. BILAS. **Understanding and improving the cost of scaling distributed event processing**. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS '12*, pages 290–301, New York, NY, USA, 2012. ACM.
- [288] C. DAVE AND R. EIGENMANN. **Automatically tuning parallel and parallelized programs**. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing, LCPC'09*, pages 126–139, Berlin, Heidelberg, 2010. Springer-Verlag.
- [289] T. KARCHER AND V. PANKRATIUS. **Run-time automatic performance tuning for multicore applications**. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I, Euro-Par'11*, pages 3–14, Berlin, Heidelberg, 2011. Springer-Verlag.
- [290] S. M. FARHAD, Y. KO, B. BURGSTALLER, AND B. SCHOLZ. **Profile-guided deployment of stream programs on multicores**. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES '12*, pages 79–88, New York, NY, USA, 2012. ACM.
- [291] M. DAYARATHNA AND T. SUZUMURA. **Automatic optimization of stream programs via source program operator graph transformations**. *Distributed and Parallel Databases*, **31**(4):543–599, 2013.
- [292] N. BACKMAN, R. FONSECA, AND U. ÇETINTEMEL. **Managing parallelism for stream processing in the cloud**. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing, HotCDP '12*, pages 1:1–1:5, New York, NY, USA, 2012. ACM.
- [293] TWITTER. **#numbers**. URL: <http://blog.twitter.com/2011/03/numbers.html>, March 2011.
- [294] C. CATLEY, K. SMITH, C. MCGREGOR, A. JAMES, AND J. M. EKLUND. **A framework to model and translate clinical rules to support complex real-time analysis of physiological and clinical data**. In *Proceedings of the 1st ACM International Health Informatics Symposium, IHI '10*, pages 307–315, New York, NY, USA, 2010. ACM.
- [295] A. W. APPEL. *Modern compiler implementation in Java*. Cambridge University Press, 2002.
- [296] DEVIN COOK. **GOLD Parsing System**. URL: <http://www.goldparser.org/>, December 2011.

-
- [297] J. LEVINE. *flex & bison*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2009.
- [298] S. MARSLAND. *Machine Learning : An Algorithmic Perspective*. Chapman & Hall/CRC, 2009.
- [299] SCIPY. **Scientific Tools for Python**. URL: <http://www.scipy.org/>, July 2011.
- [300] A. S. TANENBAUM AND M. V. STEEN. *Distributed Systems*. Pearson Education, Inc., 2007.
- [301] B. FURHT AND A. ESCALANTE. *Handbook of Cloud Computing*. Springer-Verlag New York, Inc., 2010.
- [302] E. J. SCHWEITZER. **Reconciliation of the cloud computing model with US federal electronic health record regulations**. *Journal of the American Medical Informatics Association*, **19**(2):161–165, 2012.
- [303] A. M. H. KUO. **Opportunities and challenges of cloud computing to improve health care services**. *Journal of medical Internet research*, **13**(3), 2011.
- [304] N. TATBUL. **Load Shedding**. In L. LIU AND M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 1632–1636. Springer US, 2009.
- [305] M. DAYARATHNA AND T. SUZUMURA. **XGDBench: A benchmarking platform for graph stores in exascale clouds**. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 363–370, 2012.
- [306] S. SAKR AND A. LIU. **SLA-Based and Consumer-centric Dynamic Provisioning for Cloud Databases**. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 360–367, june 2012.
- [307] NUVOLABASE. **NuvolaBase: cloudize your data - Commercial support, training and services about OrientDB**. URL: <http://www.nuvolabase.com/site/>, April 2012.
- [308] DYDRA. **Dydra: Networks Made Friendly**. URL: <http://dydra.com/>, April 2012.
- [309] CLOUDGRAPH. **CloudGraph .NET graph database**. URL: <http://www.cloudgraph.com/>, April 2012.
- [310] B. F. COOPER, A. SILBERSTEIN, E. TAM, R. RAMAKRISHNAN, AND R. SEARS. **Benchmarking cloud serving systems with YCSB**. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

-
- [311] D. CHAKRABARTI, C. FALOUTSOS, AND M. MCGLOHON. **Graph Mining: Laws and Generators**. In C. C. AGGARWAL, H. WANG, AND A. K. ELMAGARMID, editors, *Managing and Mining Graph Data*, 40 of *The Kluwer International Series on Advances in Database Systems*, pages 69–123. Springer US, 2010.
- [312] D. CHAKRABARTI, Y. ZHAN, AND C. FALOUTSOS. **R-MAT: A Recursive Model for Graph Mining**. In *Fourth SIAM International Conference on Data Mining*, April 2004.
- [313] K. HUPPLER. **Performance Evaluation and Benchmarking**. chapter The Art of Building a Good Benchmark, pages 18–30. Springer-Verlag, Berlin, Heidelberg, 2009.
- [314] M. SARWAT, S. ELNIKETY, Y. HE, AND G. KLIOT. **Horton: Online Query Execution Engine for Large Distributed Graphs**. In *ICDE*, pages 1289–1292, 2012.
- [315] P. CUDRÉ-MAUROUX AND S. ELNIKETY. **Graph Data Management Systems for New Application Domains**. *PVLDB*, 4(12):1510–1511, 2011.
- [316] J. WANG. **Sequential Patterns**. In L. LIU AND M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2621–2625. Springer US, 2009.
- [317] F. VERSACI AND K. PINGALI. **Processor allocation for optimistic parallelization of irregular programs**. In *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part I, ICCSA’12*, pages 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.
- [318] AURELIUS. **RexPro**. URL: <https://github.com/tinkerpop/rexster/wiki/RexPro>, April 2013.
- [319] Z. ZHAO, J. LIU, AND N. CRESPI. **The design of activity-oriented social networking: Dig-Event**. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, iiWAS ’11*, pages 420–425, New York, NY, USA, 2011. ACM.
- [320] P. BODIK, A. FOX, M. J. FRANKLIN, M. I. JORDAN, AND D. A. PATTERSON. **Characterizing, modeling, and generating workload spikes for stateful services**. In *Proceedings of the 1st ACM symposium on Cloud computing, SoCC ’10*, pages 241–252, New York, NY, USA, 2010. ACM.
- [321] M. FALOUTSOS, P. FALOUTSOS, AND C. FALOUTSOS. **On power-law relationships of the Internet topology**. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, August 1999.
- [322] J. LESKOVEC, D. HUTTENLOCHER, AND J. KLEINBERG. **Signed networks in social media**. In *Proceedings of the 28th international conference on Human*

- factors in computing systems*, CHI '10, pages 1361–1370, New York, NY, USA, 2010. ACM.
- [323] P. SHANNON, A. MARKIEL, O. OZIER, N. S. BALIGA, J. T. WANG, D. RAMAGE, N. AMIN, B. SCHWIKOWSKI, AND T. IDEKER. **Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks**. *Genome Research*, **13**(11):2498–2504, November 2003.
- [324] T. ENDO, A. NUKADA, S. MATSUOKA, AND N. MARUYAMA. **Linpack evaluation on a supercomputer with heterogeneous accelerators**. In *IPDPS*, pages 1–8, 2010.
- [325] APACHE. **The Apache Cassandra Project**. URL: <http://cassandra.apache.org/>, 2013.
- [326] M. DAYARATHNA, C. HOUNGKAEW, H. OGATA, AND T. SUZUMURA. **Scalable performance of ScaleGraph for large scale graph analysis**. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–9, 2012.
- [327] A. YONEZAWA, T. WATANABE, M. YOKOKAWA, M. SATO, AND K. HIRAO. **Advanced Institute for Computational Science (AICS): Japanese National High-Performance Computing Research Institute and its 10-petaflops supercomputer "K"**. In *State of the Practice Reports*, SC '11, pages 13:1–13:8, New York, NY, USA, 2011. ACM.
- [328] A. VAJDA. *Programming Many-Core Chips*. Springer-Verlag, 2011.
- [329] UPC CONSORTIUM. **UPC Language Specifications V1.2**. May 2005.
- [330] APACHE XML PROJECT. **Xerces-C++ XML Parser**. URL: <http://xerces.apache.org/xerces-c/>, January 2012.
- [331] L. S. BLACKFORD, J. CHOI, A. CLEARY, E. D'AZEVEDO, J. DEMMEL, I. DHILLON, J. DONGARRA, S. HAMMARLING, G. HENRY, A. PETITET, K. STANLEY, D. WALKER, AND R. C. WHALEY. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
- [332] M. HAY, C. LI, G. MIKLAU, AND D. JENSEN. **Accurate Estimation of the Degree Distribution of Private Networks**. In *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, pages 169–178, dec. 2009.
- [333] M. NEWMANN, A. L. BARABASI, AND D. J. WATTS. *The Structure and Dynamics of Networks*. Princeton University Press, 2006.
- [334] J. M. ANTHONISSE. **The rush in a directed graph**. *Technical Report BN*, **9/71**, 1971.

-
- [335] L. C. FREEMAN. **A Set of Measures of Centrality Based on Betweenness.** *Sociometry*, **40**(1):35–41, March 1977.
- [336] B. H. KORTE. *Combinatorial optimization : theory and algorithms*. Springer-Verlag, Berlin, Germany, 5th ed edition, c2012.
- [337] G. L. NEMHAUSER. *Optimization*. North-Holland, Amsterdam ; Tokyo, Netherlands, 1989 1989.
- [338] S. E. SCHAEFFER. **Graph clustering.** *Computer Science Review*, **1**(1):27 – 64, 2007.
- [339] S. FORTUNATO. **Community detection in graphs.** *CoRR*, abs/0906.0612, 2009.
- [340] H. OGATA, M. DAYARATHNA, AND T. SUZUMURA. **Towards highly scalable X10 based spectral clustering.** In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–5, 2012.
- [341] NETWORK-BASED COMPUTING LABORATORY. **MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE.** URL: <http://mvapich.cse.ohio-state.edu/>, May 2012.
- [342] U. KANG. **Dataset.** URL: <http://www.cs.cmu.edu/~ukang/dataset/>, May 2012.
- [343] L. BACKSTROM, D. HUTTENLOCHER, J. KLEINBERG, AND X. LAN. **Group formation in large social networks: membership, growth, and evolution.** In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.
- [344] J. LESKOVEC, K. J. LANG, A. DASGUPTA, AND M. W. MAHONEY. **Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters.** *Internet Mathematics*, **6**(1):29–123, 2009.
- [345] J. LESKOVEC. **SNAP: Network datasets: LiveJournal social network.** URL: <http://snap.stanford.edu/data/soc-LiveJournal1.html>, May 2012.
- [346] U. KANG, C. E. TSOURAKAKIS, AND C. FALOUTSOS. **PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations.** In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.
- [347] U. KANG, H. TONG, J. SUN, C. Y. LIN, AND C. FALOUTSOS. **GBASE: a scalable and general graph management system.** In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 1091–1099, New York, NY, USA, 2011. ACM.

-
- [348] S. SEO, E. J. YOON, J. KIM, S. JIN, J. S. KIM, AND S. MAENG. **HAMA: An Efficient Matrix Computation with the MapReduce Framework**. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726, 30 2010-dec. 3 2010.
- [349] THE APACHE SOFTWARE FOUNDATION. **Apache Incubator Giraph**. URL: <http://incubator.apache.org/giraph/>, June 2012.
- [350] E. KREPSKA, T. KIELMANN, W. FOKKINK, AND H. BAL. **HipG: parallel processing of large-scale graphs**. *SIGOPS Oper. Syst. Rev.*, **45**(2):3–13, July 2011.
- [351] N. JAIN, G. LIAO, AND T. L. WILLKE. **GraphBuilder: scalable graph ETL framework**. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES '13*, pages 4:1–4:6, New York, NY, USA, 2013. ACM.
- [352] S. SALIHOGLU AND J. WIDOM. **GPS: a graph processing system**. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management, SSDBM*, pages 22:1–22:12, New York, NY, USA, 2013. ACM.
- [353] R. S. XIN, JOSEPH E. GONZALEZ, MICHAEL J. FRANKLIN, AND ION STOICA. **GraphX: a resilient distributed graph system on Spark**. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES '13*, pages 2:1–2:6, New York, NY, USA, 2013. ACM.
- [354] T. EL-GHAZAWI AND L. SMITH. **UPC: unified parallel C**. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06*, New York, NY, USA, 2006. ACM.
- [355] M. MERNIK, J. HEERING, AND A. M. SLOANE. **When and how to develop domain-specific languages**. *ACM Comput. Surv.*, **37**(4):316–344, December 2005.
- [356] R CORE TEAM. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.
- [357] X. WU, X. YING, K. LIU, AND L. CHEN. **A Survey of Privacy-Preservation of Graphs and Social Networks**. In C. C. AGGARWAL, H. WANG, AND A. K. ELMAGARMID, editors, *Managing and Mining Graph Data*, **40** of *The Kluwer International Series on Advances in Database Systems*, pages 421–453. Springer US, 2010.
- [358] K. LIU AND E. TERZI. **Towards identity anonymization on graphs**. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08*, pages 93–106, New York, NY, USA, 2008. ACM.

-
- [359] T. DAVIS AND Y. HU. **The University of Florida Sparse Matrix Collection.** URL: <http://www.cise.ufl.edu/research/sparse/matrices/>, July 2012.
- [360] G. RUDY, M. KHAN, M. HALL, C. CHEN, AND J. CHAME. **A Programming Language Interface to Describe Transformations and Code Generation.** In K. COOPER, J. MELLOR-CRUMMEY, AND V. SARKAR, editors, *Languages and Compilers for Parallel Computing*, **6548** of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2011.
- [361] M. DEIJFEN AND M. LINDHOLM. **Growing networks with preferential deletion and addition of edges.** *Physica A: Statistical Mechanics and its Applications*, **388**(19):4297 – 4303, 2009.
- [362] Y. LOW ET AL. **Distributed GraphLab: a framework for machine learning and data mining in the cloud.** *Proc. VLDB Endow.*, **5**(8):716–727, April 2012.
- [363] S. A. KAMIL. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages.* PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.
- [364] M. STONEBRAKER AND J. HONG. **Researchers’ big data crisis; understanding design and functionality.** *Commun. ACM*, **55**(2):10–11, February 2012.
- [365] T. BICER, D. CHIU, AND G. AGRAWAL. **Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds.** In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 636–643, May.
- [366] K. KAMBATLA, A. PATHAK, AND H. PUCHA. **Towards optimizing hadoop provisioning in the cloud.** In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud’09, Berkeley, CA, USA, 2009. USENIX Association.
- [367] S. MARTELLO. *Surveys in combinatorial optimization.* North-Holland, Amsterdam ; Tokyo, Netherlands, 1987 1987.
- [368] G. KARYPIS AND V. KUMAR. **A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.** *SIAM J. Sci. Comput.*, **20**(1):359–392, December 1998.

Appendix A

Data Flow Graphs of the High Performance Sample Applications

This appendix lists the optimized data flow graphs of input applications that are produced by Hirundo.

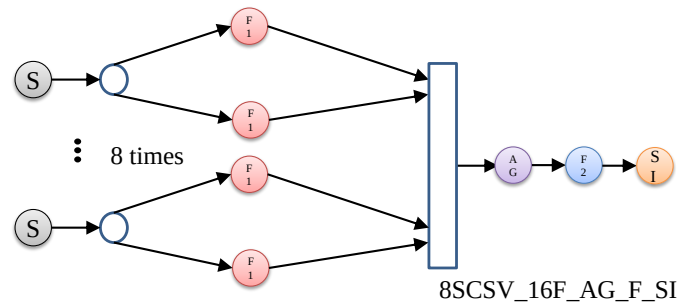


Figure A.1: DAG of 8SCSV_16F_AG_F_SI sample application of VWAP which produced highest consistent performance compared to the original VWAP application on cluster X.

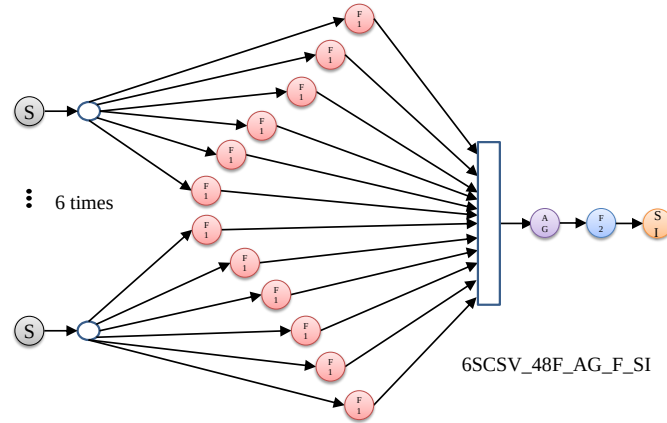


Figure A.2: DAG of 6SCSV_48F_AG_F_SI sample application of VWAP which produced highest consistent performance compared to the original VWAP application on cluster Y.

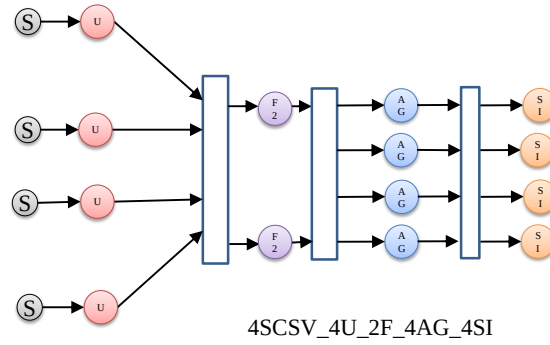


Figure A.3: DAG of 4SCSV_4U_2F_4AG_4SI sample application of CDR which produced highest consistent performance compared to the original CDR application on clusters X and Y.

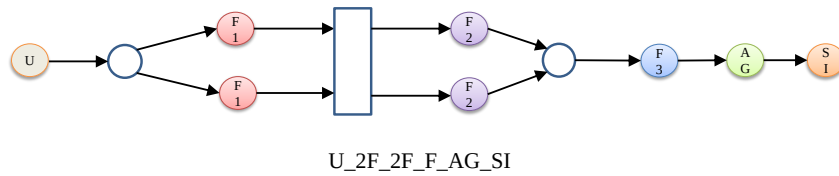


Figure A.4: DAG of U_2F_2F_F_AG_SI sample application of Twitter which produced highest consistent performance compared to the original Twitter application on cluster X.

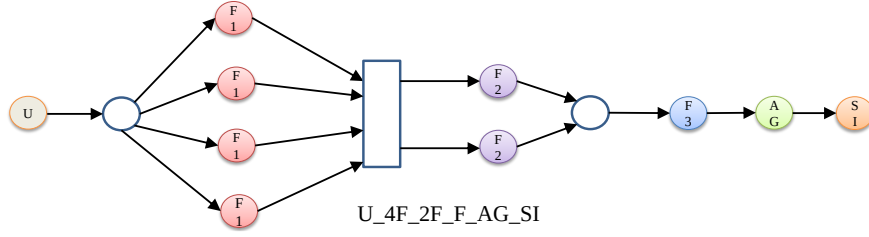


Figure A.5: DAG of U_4F_2F_F_AG_SI sample application of Twitter which produced highest consistent performance compared to the original Twitter application on cluster Y.

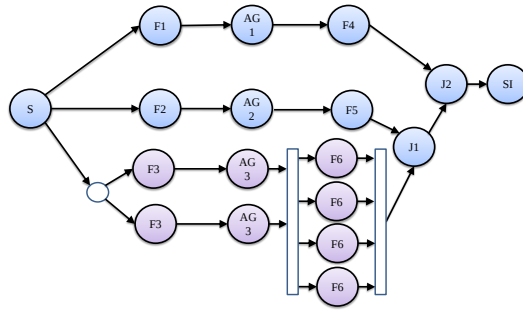


Figure A.6: DAG of SCSV_F_AG_F_2F_2AG_4F_F_AG_F_J_J_SI sample application of Apnoea which produced highest consistent performance compared to the original Apnoea application on cluster X.

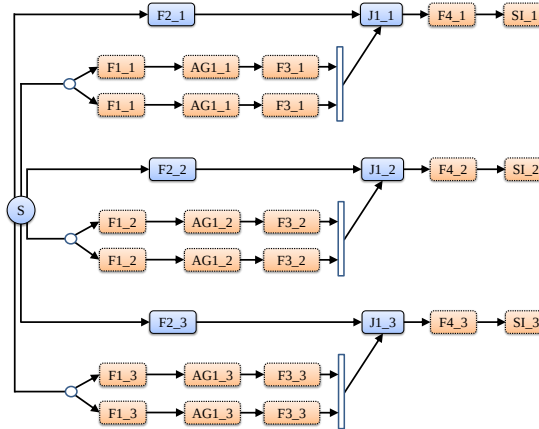


Figure A.7: DAG of SCSV_FOR_S_2F_2AG_2F_F_J_F_SI_FOR_E sample application of Bargain which produced highest consistent performance compared to the original Bargain application on cluster X.

Appendix B

Some Example Transformations for Quadri-Operator Transformation

An example collection of transformed operator blocks possible using Quadri-Operator Transformation (Quadri-OP transformation) are shown in Figure B.1. Note that these Quadri-OP transformations are listed only for illustrating the complexity involved in transforming four operator blocks (A_B_C_D) at a time. We list few possible transformations only for two corresponding transformations produced by Tri-OP Transformation. Transformed operator block from Tri-OP transformation shown in Figure B.1 (a) has been produced by applying 1-2-1 transformer pattern. Similar possible transformed operator blocks from Quadri-OP transformation are shown in Figures B.1 (a-1) to (a-4). Similar analogous transformed operator blocks for Tri-OP transformed operator block of 2-2-2 transformation (shown in Figure B.1 (b)) are shown in Figures B.1 (b-1) to (b-7). These transformed operator blocks are produced only for a transformation depth of 2. Hence, transformations even at smaller transformation depths produce significantly larger numbers of transformed operator blocks for Quadri-OP transformation which makes Quadri-OP transformation not suitable for Hirundo's code transformer. Furthermore, as can be observed from these figures such as Figures B.1 (a), (a-3) and Figures B.1 (b), (b-1); many of the Quadri-OP transformed operator blocks can be produced by using Tri-OP transformation along with operator fusion. Due to these reasons we conduct only up to Tri-OP transformation in Hirundo's program transformer.

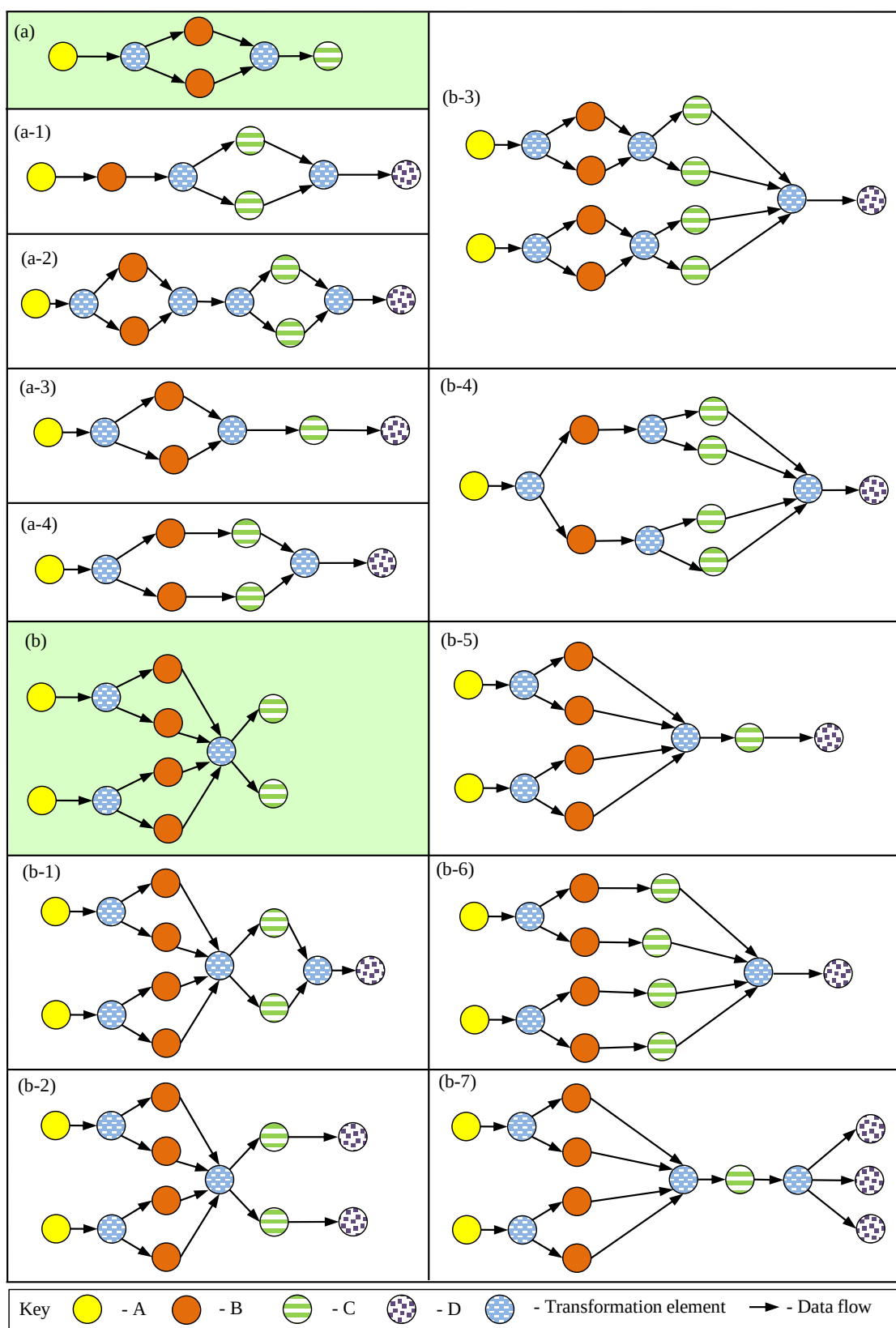


Figure B.1: Some transformations possible using Quadri-Operator Transformation.

Appendix C

Representation of a Communication Network with ScaleGraph 1.0

This appendix lists an example X10 code for representing a communication network with ScaleGraph. Note that the line numbers are included in the Figure C.1 as a support for explanation. ScaleGraph 1.0's `AttributedGraph` class is used for this purpose. `AttributedGraph` is composed of `Vertex`, `Edge`, and `Attribute` objects and it has predefined schemas for vertices and edges. Use of the term “schema” is analogous to relation schema in Relational Databases. The vertex and edge schemas are fixed for the rest of the life cycle of the graph. Defining of such fixed schema allows for eliminating subsequent requests to add elements with different attribute schemas. As can be observed from Figure C.1, each vertex and edge of the graph can be represented as objects which is an advantage in modeling real world scenarios. However, the information of these objects are not stored as objects to avoid the performance penalties such as excessive memory use.

The X10 code in lines 2 to 5 define the vertex attribute schema. Lines 7 to 10 create the first vertex corresponding to Alice in the communication network. Lines 12 to 15 create the vertex corresponding to Bob. In line 17 an empty `AttributedGraph` is created and the two vertex objects are added to this new graph object in subsequent lines. Next, an `AttributeSchema` object is created for edges and set in the graph object (In line 22 to 25). An `Edge` object that represents an email message exchanged between Alice and Bob is created in line 30 and is added to the graph in line 32.

```

1:  val attrArray:ArrayList[Attribute] = null;
2:  schema: AttributeSchema = new AttributeSchema();
3:  schema.add("fname", AttributeSchema.StringAttribute);
4:  schema.add("email_add", AttributeSchema.StringAttribute);
5:  schema.add("age", AttributeSchema.IntAttribute);
6:
7:  attrArray = new ArrayList[Attribute]();
8:  attrArray.add(new StringAttribute("fname", "Alice"));
9:  attrArray.add(new StringAttribute("email_add", "alice@gmail.com"));
10: v0:Vertex = new Vertex(attrArray);
11:
12: attrArray = new ArrayList[Attribute]();
13: attrArray.add(new StringAttribute("fname", "Bob"));
14: attrArray.add(new StringAttribute("email_add", "bob@gmail.com"));
15: v1:Vertex = new Vertex(attrArray);
16:
17: g: AttributedGraph = AttributedGraph.make();
18: g.setVertexAttributeSchema(schema);
19: g.addVertex(v0);
20: g.addVertex(v1);
21:
22: schema: AttributeSchema = new AttributeSchema();
23: schema.add("title", AttributeSchema.DateAttribute);
24: schema.add("dtime", AttributeSchema.DateAttribute);
25: g.setEdgeAttributeSchema(schema);
26:
27: attrArray = new ArrayList[Attribute]();
28: attrArray.add(new StringAttribute("title", "Meeting"));
29: attrArray.add(new DateAttribute(2012,2,10));
30: e0: Edge = new Edge(v0,v1, attrArray);
31:
32 : g.addEdge(e0);

```

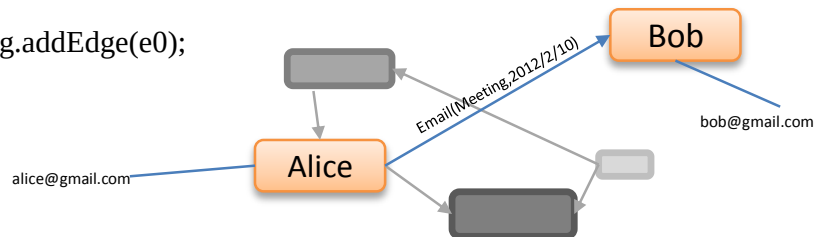


Figure C.1: An example X10 code for representing a communication network with Scale-Graph 1.0.