Runtime Prediction for Scale-Out Data Analytics

THÈSE N^{O} 6629 (2015)

PRÉSENTÉE LE 2 JUILLET 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÉES MASSIVES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Adrian Daniel POPESCU

acceptée sur proposition du jury:

Prof. P. Dillenbourg, président du jury
Prof. A. Ailamaki, directrice de thèse
Prof. S. Babu, rapporteur
Dr A. Balmin, rapporteur
Prof. K. Aberer, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

Your time is limited, so don't waste it living someone else's life.
Don't be trapped by dogma - which is living with the results of
other people's thinking. Don't let the noise of others' opinions
drown out your own inner voice. And most important,
have the courage to follow your heart and intuition.
—Steve Jobs

In the memory of my mother, Ilona, to my beloved father, Dumitru,
to my caring siblings, Gabriela and Cristian,
and to my dearest pals, Daniel and Marina.
I dedicate this thesis to them.

Acknowledgements

First and foremost, I would like to thank my advisor Anastasia Ailamaki for all the guidance and support she has given to me. Without her enthusiasm, energy, and encouragements over the PhD years this work would not be possible. I am profoundly grateful to her for encouraging me to pursue my own research ideas, for boosting my research skills through constructive feedback, and for allowing me to follow my path towards PhD. I consider myself fortunate to have received her excellent advice that shaped me as a more confident person today.

I would like to thank to all the members of my thesis committee that kindly accepted to offer their time and energy to assess my dissertation and to suggest improvements. I would like to thank Vuk Ercegovac for mentoring me during my internship at IBM Almaden, and to all the other members of the group with whom I was fortunate to discuss with. My discussions with Vuk, Andrey Balmin, and Carl-Christian Kanne contributed significantly in shaping the internship project. Special thanks to Andrey Balmin and Vuk Ercegovac for the fruitful collaboration we had after the internship period. Their comments were invaluable in developing PREDIcT, the core of my thesis work.

During my last PhD years I was fortunate to collaborate with Shivnath Babu. I am so grateful for all the discussions we had, for his insightful suggestions, for his enthusiasm and for his guidance. I felt privileged to chat with him multiple times at very late hours during the night. One chapter of this thesis is done in collaboration with him.

I would like to thank to all the members of the DIAS lab for making my PhD journey an enjoyable one. I thank Thomas Heinis for his excellent advice, encouragements, and friendship. He kindly offered his help each time I was seeking an opinion on my work. Debabrata Dash, Verena Kantere mentored me during my first PhD year for which I am grateful. I thank Farhan Tauheed, my sleepless office mate, for tolerating me over the years and for inspiring me through his photographic work. I thank Danica Porobic and Renata Borovica-Gajic for their invaluable feedback on my research work. I would also like to thank Pinar Tözün, Manos Athanassoulis, Radu Stoica, Ioannis Alagiannis, Eleni Tzirita Zacharatou, Mirjana Pavlovic, Erietta Liarou, Manos Karpathiotakis, Miguel Branco, Matt Olma, Cesar Matos, Raja Appuswamy, Satya Valluri, Darius Sidlauskas, Utku Sirin for the invaluable discussions, insightful comments, and the constructive feedback on my ongoing research and on my presentations. I would like to thank Erika Raetz for helping me countlessly over the years in all the administrative

problems I faced, and Dimitra Tsaoussis Melissargos for the administrative help and for her feedback on my paper submissions.

Thank you to Daniel Lupei and to Marina Boia for the precious time spent together outdoors, indoors, or at EPFL, for all the small, very small, and tiny jokes that made me smile, for our countless debates on sports, on stand-up comedy, and on music, just to name a few, for their encouragements and moral support, and foremost, for being my best friends I could have. I have greatly enjoyed the friendship of Immanuel Trummer, Thomas Heinis, Ayush Bhandari, Lada Sycheva, Valentina Sintsova, Eleni Tzirita Zacharatou, Mirjana Pavlovic, Alina Dudeanu, Cristina Ghiurcuta, Danica Porobic, Renata Borovica-Gajic, Pinar Tözün, Denisa Ghita, Mihai Martalogu, Florin Dinu, Andrew Becker, Gregoire Devauchelle, Onur Yürüten. It was my pleasure to spend time together at joyful dinners, pancake parties, barbecues over the lake, biking trips, and hikes in the Alps. Thanks to Immanuel the abstract of this thesis is also available in German.

I would like to thank my family for their unconditional moral support and love.

Finally, I am grateful to NCCR MICS and Hasler Foundation for supporting my research activity during my PhD, and to Amazon for granting me hardware resources on the Amazon Web Services infrastructure.

Lausanne, June 2015 A. D. P.

Abstract

Many analytics applications generate *mixed workloads*, i.e., workloads comprised of analytical tasks with different processing characteristics including data pre-processing, SQL, and iterative machine learning algorithms. Examples of such mixed workloads can be found in web data analysis, social media analysis, and graph analytics, where they are executed *repetitively* on large input datasets (e.g., *Find the average user time spent on the top 10 most popular web pages on the UK domain web graph.*). Scale-out processing engines satisfy the needs of these applications by distributing the data and the processing task efficiently among multiple workers that are first reserved and then used to execute the task in parallel on a cluster of machines. Finding the resource allocation that can complete the workload execution within a given time constraint, and optimizing cluster resource allocations among multiple analytical workloads motivates the need for estimating the runtime of the workload *before* its actual execution.

Predicting runtime of analytical workloads is a challenging problem as runtime depends on a large number of factors that are hard to model a priori execution. These factors can be summarized as *workload characteristics* (data statistics and processing costs) , the *execution configuration* (deployment, resource allocation, and software settings), and the *cost model* that captures the interplay among all of the above parameters. While conventional cost models proposed in the context of query optimization can assess the relative order among alternative SQL query plans, they are not aimed to estimate *absolute* runtime. Additionally, conventional models are ill-equipped to estimate the runtime of *iterative analytics* that are executed repetitively until convergence and that of *user defined* data pre-processing operators which are not "owned" by the underlying data management system.

This thesis demonstrates that runtime for data analytics can be predicted accurately by breaking the analytical tasks into multiple processing phases, collecting key input features during a reference execution on a sample of the dataset, and then using the features to build *per-phase* cost models. We develop prediction models for three categories of data analytics produced by social media applications: iterative machine learning, data pre-processing, and reporting SQL. The prediction framework for iterative analytics, PREDIcT, addresses the challenging problem of estimating the number of iterations, and per-iteration runtime for a class of iterative machine learning algorithms that are run repetitively until convergence. The hybrid prediction models we develop for data pre-processing tasks and for reporting SQL combine the benefits of analytical modeling with that of machine learning-based models. Through a

training methodology and a pruning algorithm we reduce the cost of running training queries to a minimum while maintaining a good level of accuracy for the models.

Key words: database management systems, distributed data management, runtime prediction, data analytics, MapReduce, graph analytics, Bulk Synchronous Parallel, iterative processing, complex analytics, cost model, analytical model, machine learning.

Zusammenfassung

Viele analytische Anwendungen führen zu einem gemischten Workload, bestehend aus Verarbeitungseinheiten völlig unterschiedlichen Typs die zum Beispiel Datenvorverarbeitung, SQL Queries oder iterative Verfahren zum maschinellen Lernen beinhalten können. Die Analyse des Internets, die Analyse von sozialen Medien oder Graph-Analyse sind nur einige wenige Beispiele für Bereiche in denen gemischte Workloads anzutreffen sind und häufig und auf großen Datenmengen ausgeführt werden (zum Beispiel um die durchschnittliche Zeit zu berechnen, die Benutzer auf den 10 populärsten Internetseiten im englischsprachigen Raum verbringen). Horizontal skalierende Systeme genügen den Ansprüchen solcher Anwendungen, indem sie die Daten und Verarbeitungsschritte effizient zwischen mehreren Rechnern aufteilen. Diese Rechner müssen zuerst reserviert werden bevor sie anschliessend die ihnen zugeteilten Aufgaben gleichzeitig ausführen. Um herauszufinden, welche Rechenkapazität benötigt wird um einen gegebenen Workload innerhalb eines vorgegebenen Zeitrahmens auszuführen oder um die beste Aufteilung der vorhandenen Rechenkapazitäten zwischen unterschiedlichen Anwendungen zu ermitteln, ist es notwendig die Laufzeit einer Anwendung zu schätzen bevor die Anwendung gestartet wird.

Die Laufzeit einer analytischen Anwendung ist im Vorhinein schwierig einzuschätzen da sie von vielen Faktoren abhängt, die schwer zu modellieren sind. Diese Faktoren können grob in drei Kategorien eingeteilt werden: *Workload Charakteristika* (beispielsweise Statistiken die die zu verarbeiteten Daten beschreiben), die *Ausführungskonfiguration* (Konfiguration der verwendeten Software, Zuteilung der Rechenkapazität) und das *Kostenmodel*, welches das Zusammenspiel aller bisher genannten Faktoren erfasst. Kostenmodelle wie sie zur Optimierung von Datenbank Queries verwendet werden sind darauf zugeschnitten, alternative Verarbeitungspläne miteinander zu vergleichen aber nicht dafür geeignet, die absolute Laufzeit akkurat einzuschätzen. Solche Kostenmodelle sind ebenfalls ungeeignet dafür, die Laufzeit iterativer Prozesse einzuschätzen die bis zur Konvergenz wiederholt werden. Gleichfalls ist es mit solchen Kostenmodellen nicht möglich, die Laufzeit von Benutzer-definierten Funktionen einzuschätzen.

In dieser Doktorarbeit wird der Beweis erbracht, dass die Laufzeit analytischer Prozesse akkurat vorhergesagt werden kann, indem wir den analytischen Prozess in mehrere Phasen unterteilen und während eines Testlaufs Schlüsselstatistiken zu jeder Phase erstellen, welche dann dazu verwendet werden um phasenspezifische Vorhersagemodelle zu kreieren. Wir ent-

wickeln Vorhersagemodelle für drei Kategorien analytischer Anwendungen wie sie im Bereich der Analyse sozialer Medien auftreten: iterative Verfahren zum maschinelles Lernen, Datenvorverarbeitung und SQL Verarbeitung. Unser Vorhersagemodell für iterative Datenanalyse, PREDIcT, stellt sich der schwierigen Aufgabe, sowohl die Anzahl der Wiederholungen als auch die Laufzeit einer einzelnen Wiederholung vorherzusagen für eine Klasse von Algorithmen zum maschinellen Lernen, welche bis zur Konvergenz iteriert werden. Die von uns entwickelten, gemischten Vorhersagemodelle verbinden die Vorteile analytischer Modelle mit denen von auf maschinellem Lernen aufbauenden Modellen. Mithilfe ausgefeilter Trainingsmethoden und eines Algorithmus zur effizienten Aussortierung suboptimaler Lösungen konnten wir die Kosten der Trainingsphase signifikant reduzieren bei weiterhin guter Vorhersagequalität.

Stichwörter: Datenbanksysteme, verteilte Datenbanksysteme, Laufzeitvorhersage, Datenanalyse, MapReduce, Graph-Analyse, bulk synchronous parallel, iterative Verarbeitung, komplexe Datenanalyse, Kostenmodelle, analytische Modelle, maschinelles Lernen.

Contents

Acknowledgements i

Abstract (English/Deutsch) iii

1 Introduction 1
1.1 Motivating Use Cases . . . 2
1.2 Data Analytics Today . . . 2
1.2.1 Example: Pipeline of Analytical Tasks . . . 4
1.3 Prediction Challenges . . . 5
1.3.1 Iterative Analytics on BSP . . . 6
1.3.2 SQL and ETL Analytics on MapReduce . . . 6
1.4 Technical Contributions . . . 7
1.5 Thesis Outline . . . 8

2 Background 11
2.1 Distributed Processing Engines for Scale-Out Analytics . . . 11
2.1.1 MapReduce Execution Model . . . 11
2.1.2 Distributed Graph Processing . . . 13
2.1.3 Spark Processing Engine . . . 15
2.2 Estimating and Optimizing Iterative Processing . . . 16
2.2.1 Approximating and Sampling Large Graphs . . . 17
2.3 Performance Prediction for DBMS . . . 18
2.3.1 Nearest Neighbors-based Prediction . . . 19
2.3.2 Operator Level Models . . . 19
2.3.3 Progress Estimators . . . 20
2.3.4 Performance Modeling for Storage Devices . . . 20
2.4 Performance Prediction for MapReduce . . . 20
2.4.1 Self-tuning and Optimization . . . 20
2.4.2 Nearest neighbors-based prediction . . . 21
2.4.3 Resource Allocation and Scheduling . . . 22
2.5 Existing Prediction Approaches vs. Current Requirements . . . 22
2.6 Runtime Modeling: Background Concepts . . . 23
2.6.1 Runtime Modeling Overview . . . 23
2.6.2 Collecting Input Features . . . 24

2.6.3 Building a Cost Model . . . 25
2.6.4 Prediction . . . 28
2.6.5 Accuracy Metrics of Interest . . . 28
2.7 Summary . . . 29

3 Runtime Prediction for Iterative Analytics 31
3.1 Introduction . . . 31
3.1.1 Sketch of Proposed Approach . . . 32
3.1.2 Contributions . . . 34
3.2 The BSP Processing Model . . . 35
3.3 Modeling Assumptions . . . 36
3.4 PREDIcT's Transformations . . . 36
3.4.1 Sampling Techniques . . . 37
3.4.2 Transform Function . . . 37
3.5 Model Fitting and Prediction . . . 38
3.5.1 Key Input Features . . . 38
3.5.2 Customizable Cost Model . . . 40
3.5.3 Prediction . . . 41
3.6 End-to-end Use Cases . . . 42
3.6.1 PageRank . . . 42
3.6.2 Semi-clustering . . . 43
3.6.3 Top-k Ranking . . . 44
3.6.4 Neighborhood Estimation . . . 45
3.6.5 Labeling Connected Components . . . 46
3.7 Limitations . . . 47
3.8 Experimental Evaluation . . . 47
3.8.1 Setup and Methodology . . . 47
3.8.2 Estimating Key Input Features . . . 49
3.8.3 Upper Bound Estimates . . . 52
3.8.4 Estimating Runtime . . . 53
3.8.5 Sensitivity to Sampling Technique . . . 56
3.8.6 Overhead Analysis . . . 58
3.8.7 Resource Allocation . . . 59
3.9 Summary of Related Work . . . 60
3.10 Conclusion . . . 60

4 Predicting Runtime of Data Pre-processing 61
4.1 Introduction . . . 61
4.2 Jaql . . . 63
4.2.1 Query Example . . . 63
4.2.2 Query Compilation in Jaql . . . 64
4.3 Modeling Assumptions . . . 65
4.4 Model Fitting and Prediction . . . 65

4.4.1 Sketch of Proposed Approach . . . 65
4.4.2 Modeling Segment Performance . . . 66
4.4.3 Modeling Query Runtime . . . 67
4.4.4 Sources of Errors . . . 67
4.5 Experimental Study . . . 68
4.5.1 Experimental Methodology . . . 68
4.5.2 Experimental Setup . . . 69
4.5.3 Job-Level Predictions . . . 69
4.5.4 Query-Level Predictions . . . 71
4.6 Summary of Related Work . . . 72
4.7 Conclusion . . . 72

5 Runtime Prediction for Reporting SQL Analytics **75**
5.1 Introduction . . . 75
5.2 Foundations and Overview . . . 77
5.2.1 Query Execution in HiveQL . . . 77
5.2.2 Problem Definition . . . 77
5.2.3 Starfish's Limitations . . . 78
5.2.4 TITAN Overview . . . 79
5.3 TITAN Prediction Approach . . . 80
5.3.1 Modeling Assumptions . . . 80
5.3.2 Hybrid Prediction Model . . . 81
5.3.3 Localized Training based Models . . . 85
5.3.4 Global Analytical Models . . . 87
5.4 Training Methodology . . . 89
5.4.1 Query Template Pruning . . . 89
5.4.2 Synthetic Query Generation . . . 90
5.5 Translation Models . . . 91
5.5.1 Semantics . . . 91
5.5.2 Operator Phase and Data Mappings . . . 93
5.5.3 Use Cases . . . 94
5.6 Evaluation . . . 95
5.6.1 Setup and Methodology . . . 95
5.6.2 Training Models . . . 99
5.6.3 Testing Models . . . 101
5.6.4 Answering Performance Boost Questions . . . 105
5.7 Summary of Related Work . . . 106
5.8 Conclusions . . . 107

6 Conclusions **109**
6.1 Summary of Contributions . . . 109
6.2 Impact . . . 110
6.2.1 Generality of Techniques to Similar Problems . . . 111

6.3 Predictable vs. Non-Predictable Analytics 112
6.4 Looking Ahead 113
6.4.1 SLA Driven Job Scheduling 113
6.4.2 Cost Models for In-memory Analytical Engines 114
6.4.3 Sharing Cluster Resources Among Analytical Engines 114

List of figures **115**

List of tables **119**

Bibliography **126**

Curriculum Vitae **127**

1 Introduction

Predicting the runtime performance of large scale analytics is motivated by a number of data management tasks including workload optimization ([2]), resource management, and scheduling ([79, 76, 21]). At one end of the spectrum, users and application managers target to optimize the execution of their workloads such that pre-specified time constraints are met (i.e., deadlines). At the other end of the spectrum, resource providers aim to satisfy users' requirements while improving utilization of their resources. In particular, schedulers and resource managers reduce unnecessary over-provisioning of resources through efficient prioritization of resource allocations.

Analytical workloads can be broadly classified into: i) deadline driven workloads with stringent time requirements, ii) best-effort workloads where explicit deadlines are not specified. Ideally, the scheduler interleaves the execution of the workloads such that deadlines are satisfied for the first category, and acceptable latencies are offered for the second category (e.g., Rayon scheduler [21]). Efficient resource planning is however possible only for the cases that the resources required to satisfy a particular optimization goal (e.g., deadline) are known in advance, *before* the workload execution. Therefore, a mechanism to assess the runtime performance of alternative *hypothetical resource allocation configurations* is required.

Query cost modeling has been studied in the context of database optimization, where the end goal is to find the query plan with the smallest execution cost. From a large set of possible query plans, the query optimizer quantifies the cost of each plan by using a set of analytical formulas that approximate the computational requirements of the plan. While query optimizers' cost models were designed to quantify the *relative* order among alternative query plans, they are not very accurate when the goal is to estimate *absolute* time estimates (e.g., [29, 50, 6]). That is due to several factors which can be summarized as: simplification assumptions in the analytical model, inaccurate processing cost estimates, and inaccurate data statistics. For this purpose, *runtime prediction models* that are designed from the ground up to estimate runtime performance have been proposed in the recent years.

1.1 Motivating Use Cases

Runtime prediction models have high practical applicability for answering What-If performance questions when an *execution configuration*[1] that can satisfy user requested deadlines is sought. More recently, with the prevalence of using hardware infrastructure as a service (IaaS) for data management tasks, answering *cluster sizing* questions and *feasibility analysis* questions for hypothetical configurations became crucial ([39]). In this spectrum, questions like: "What hardware configuration and how many machine instances are needed to meet the runtime deadline of my analytical application?" are common, especially when transiting workloads from development clusters into production. Hence, a mechanism for assessing performance of alternative hypothetical configurations is required. We summarize the main use cases for runtime performance prediction bellow.

- **Feasibility analysis**: Given a workload of analytical queries, a plan for each query from the workload, and an execution configuration (i.e., hardware/software configuration and deployment), will the workload complete within a pre-defined deadline? This use case occurs in scheduling and resource allocation where reserving resources over time has to be performed in advance of the workload execution.
- **Workload on-boarding**: Given a workload, a plan for for each query from the workload, and an execution configuration, the runtime execution on the development cluster takes t hours. What execution configuration can reduce the workload execution time to $t/2$ hours?
- **Performance boost**: Given a workload and an execution configuration, the runtime execution on the deployment cluster takes t hours. Is there a new execution configuration that can boost the actual performance of the workload by a factor of $2x$?”

1.2 Data Analytics Today

Today’s analytical requirements are complex, going beyond the traditional analytical operators used to compute statistical summaries over the input datasets [20]. Recent research shows that business intelligence moves towards including *complex analytics* into the business process. In particular, data mining algorithms are often used in analytical pipelines for finding correlations in the ever increasing datasets (e.g., clustering, ranking) [44, 52]. For instance, social networks use machine learning to order stories in the user’s news feed (i.e., ranking). They also use data mining to group users with similar interests together (i.e., clustering).

Independently to the complexity of the analytical operators, data analytics can be categorized based on the query processing characteristics into: *reporting* queries, and *ad-hoc queries*. In contrast to ad-hoc queries which are exploratory, customized to the user’s immediate

[1] The execution configuration consists of a subset of the following parameters: software configuration settings, operator(s) implementation, and an allocation of resources.

needs, reporting queries are mostly static, and are executed periodically on similar datasets or on different portions of the input dataset to answer pre-defined analytical questions about the operational state of a business. Hence, they open up opportunities for *workload re-optimization* ([37, 9]) and *elastic workload deployment*, where the deployment setting can be chosen such that application pre-specified time constraints are met ([39]). Table 1.1 summarizes the conceptual differences between reporting, and ad-hoc query processing.

Workload type	Workload sub-type	Processing characteristics	Data characteristics
reporting	ETL, SQL, complex analytics	repetitive	incremental updates
ad-hoc	SQL, complex analytics	ad-hoc	ad-hoc

Table 1.1 – Conceptual differences between reporting and ad-hoc query processing.

Within the complex analytics category, iterative processing became prevalent in the last few years partly due to the inherent iterative nature of many machine learning tasks used today in web analytics and social media (e.g., clustering, ranking, belief propagation), partly due to the prevalence of distributed graph processing engines that adopted the Bulk Synchronous Parallel (BSP) [55] or the Gather-Scatter [52] execution models. For these processing models abstractions, any algorithm is inherently iterative: it is a succession of processing steps that are executed in parallel on multiple processing nodes. The main difference between iterative processing and traditional query processing is that the iterative task is executed *repetitively* until a convergence condition is met, or a maximum number of iterations is reached. Figure 1.1 illustrates the concept of iterative processing on two input tables S, and T. In DBMS terminology, iterative processing can be interpreted as a join aggregate query among a relation that does not change S and a relation that gets updated in each iteration T. In this thesis we consider iterative tasks that are executed on input datasets that are represented as graphs. Popular examples of iterative analytics used today in social media and web analytics include: *PageRank* [59], Top-K ranking [45], *Semi-clustering* [55], statistics computation in social/web graphs (e.g., *Labeling Connected Components, Neighborhood / diameter estimation* [44]).

Data pre-processing tasks, commonly known as Extract Transform Load (ETL), and SQL analytics at scale are regularly executed on top of the MapReduce [23] processing engine. MapReduce is a distributed processing model that was designed to scale to thousands of commodity nodes by considering availability and fault tolerance as first class concerns. A data processing task executing in MapReduce is decomposed into a direct acyclic graph (DAG) of one or multiple MapReduce jobs. Within a MapReduce job, two processing phases exist: a *map* phase followed by a *reduce* phase. The two processing phases are parallelized by splitting the input data into partitions and by allocating multiple tasks to process the input partitions in parallel. The map tasks (i.e., the tasks executing the map phase) read, process the input partitions, and produce a set of intermediate results as *key-value pairs*. The reduce tasks (i.e., the tasks executing the reduce phase) aggregate all of the intermediate results with the same key generated by the map tasks and produce the final result of the MapReduce job. To tolerate

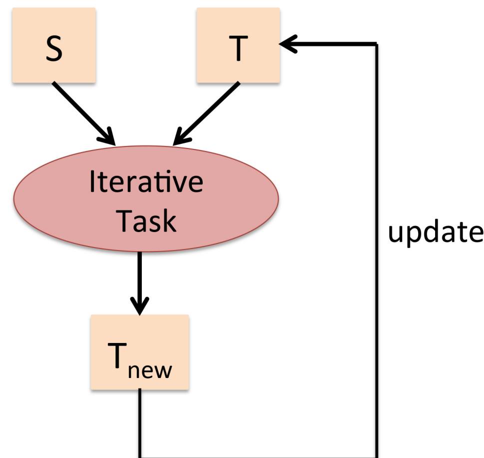


Figure 1.1 – Iterative Processing: S, input dataset that does not change as a result of executing the iterative task (i.e., input graph structure), T, input that gets updated at the end of every iteration.

failures gracefully, MapReduce stores multiple copies of the data in a distributed file system and it checkpoints intermediate results on disk at the end of *each* processing task. In the case of a failure only the failed task is re-executed on another machine that is available.

While MapReduce was originally designed to execute ETL tasks, it is also used to execute SQL-like analytics at scale. Several high-level, SQL-like languages have been introduced to simplify querying in MapReduce (or MapReduce-like frameworks): HiveQL[70] (Facebook), Pig Latin[58] (Yahoo!), Jaql[11] (IBM), DryadLINQ[83] (Microsoft), and others. These languages enable users to express their queries declaratively while their underlying engines automatically translate them into flows of jobs.

Compared with Pig and Jaql, HiveQL resembles the most the ANSI SQL language. HiveQL is extensively used at Facebook to execute data warehousing queries [70]. A recent study that analyses multiple production workloads from Facebook and Cloudera [18] shows that more than 50% of the total tasks execution time of the analytical workloads is spent in running HiveQL queries. Jaql and Pig provide powerful transformations on semi-structured data sets in addition to a large subset of supported SQL constructs. For instance, Jaql is actively used in the context of social media analytics, and machine learning pre-processing (e.g., summarization, cleansing, and statistics computation) [11, 63].

1.2.1 Example: Pipeline of Analytical Tasks

Figure 1.2 shows a motivating pipeline of analytical tasks in the context of web data analysis. The analytical pipeline finds the top ten best ranked pages within a web domain using a ranking algorithm, then, for each of these pages, it computes the average time the users spent

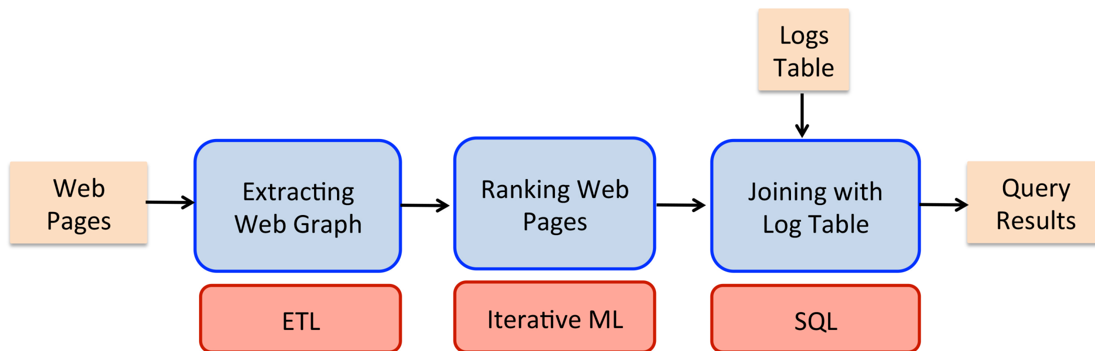


Figure 1.2 – Pipeline of Analytical Tasks in Web Data Analysis

on it during the last week. The first stage of the analytical pipeline consists of an ETL task that from a list of web pages crawled from the web extracts the hyperlinks with all the other pages (to build the web graph). Then, it filters out the pages outside of the targeted web domain. The second stage runs an iterative ranking algorithm on the input graph corresponding to the web domain. The third stage joins the output produced by the ranking algorithm with the log table that keeps statistics about page visits. Such pipelines of mixed analytical tasks (i.e., ETL, iterative ML, and SQL) are common in web analysis, blog analysis, social media analytics [82], and are executed *repetitively* (e.g., every week, every month) on updated input datasets.

To estimate performance of mixed analytical tasks, mechanisms for estimating the runtime of each task are required. In the following section we discuss the prediction challenges associated with each analytical task sub-category.

1.3 Prediction Challenges

Runtime prediction in a distributed setting is inherently a hard problem as runtime depends on a large number of factors that are hard to model a priori execution. Such factors include *workload characteristics* represented by: data statistics that determine the input processed by each database operator, and processing costs that measure the cost of executing each operator per data unit (e.g., per input tuple cost). Additional factors include the execution configuration (i.e., the resources that are allocated, the software configuration settings, the level of parallelism) and the current system state. Building highly accurate analytical models that can account for all these factors is very challenging taking into consideration the complexity of the modern hardware components and the complexity of the multi-layered software stack. Depending on the workload category (i.e., iterative ML, ETL, SQL), and the execution configuration used for running the workload there are different challenges for predicting the runtime. We discuss them in turn.

1.3.1 Iterative Analytics on BSP

Predicting the runtime of iterative analytics poses two main challenges that are not addressed by conventional prediction approaches proposed in the context of DBMS: i) predicting the number of iterations, and ii) predicting the processing time of each iteration. As both parameters depend on the characteristics of the dataset and on the convergence function, estimating their values before execution is difficult.

On one hand, the number of iterations depends on how fast the algorithm converges. Convergence is typically given by a *distance metric* that measures incremental updates between consecutive iterations. Unfortunately, an accurate closed-form formula cannot be built in advance, before materializing *all* intermediate results. On the other hand, the runtime of any given iteration may vary widely compared with the subsequent iterations according to the algorithm's semantics and as a function of the iteration's current *working set* [26]: Conceptually, different code paths are executed from one iteration to the next according to the working set.

Figure 1.3 shows the accuracy limitations of analytical upper bounds when estimating the number of iterations for algorithms with constant resource requirements per iteration (e.g., PageRank), and for algorithms with variable resource requirements per iteration (e.g., connected components). For PageRank the analytical upper bounds over-estimate the number of iterations by a factor of 2.5 on average on multiple input datasets. For connected components, analytical upper bounds over-estimate per iteration resource requirements (here, message bytes transferred) by two orders of magnitude starting from the fifth iteration.

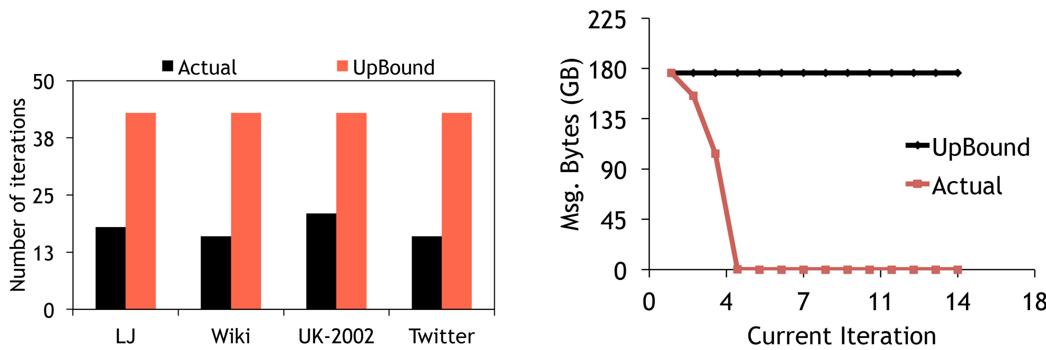


Figure 1.3 – Using analytical upper bounds to approximate the number of iterations for PageRank algorithm (left), and per iteration resource requirements (i.e., message bytes) for connected components (right).

1.3.2 SQL and ETL Analytics on MapReduce

Cost modeling and prediction of more traditional SQL analytics is also a very challenging task as it can be seen in the large body of work carried over the past decades. While there are many analytical models proposed in the context of query optimization (e.g., [67, 54]), recent

research showed that the cost models used by query optimizers are not very accurate when the goal is to predict *runtime performance* metrics [29, 50, 6]. As analytical models are aimed to assess the relative order among alternative query plans, they make simplifying assumptions about the processing costs of the operators such as: i) using a *constant* per tuple cost metric, independent of the workload characteristics, and ii) disregarding the current *system execution state*.

As a result, machine learning based approaches were proposed as an alternative approach for estimating the runtime of analytical queries (e.g., [4, 25, 29, 50, 6, 28, 63]). When using training datasets that cover the space of input queries such models are more accurate than pure analytical models as they can capture a wide range of runtime execution effects that are hard to model otherwise (e.g., interplay among workload, DBMS and underlying hardware, impact of batching). While training based prediction models can alleviate the inaccuracies introduced by simplifying modeling assumption of conventional analytical models, they have two main limitations: *high re-training* cost, which is required each time the testing workload or the execution setting changes, and *reduced accuracy* outside of the training boundaries.

In contrast to SQL-like analytics, modeling query runtime performance for ETL analytics on MapReduce using pure analytical models (as in traditional *query optimization*) is still an open problem. One of the main differences, is that MapReduce does not always "own" the data or the query's operators. The input data is *in-situ* files whose structure may be opaque to the system. Queries, even if written in a high-level language, often contain user defined functions (UDFs) typically written in Java. In this context, modeling the query runtime using machine learning techniques based on historical executions is more feasible.

1.4 Technical Contributions

Runtime prediction of data analytics produced by social media applications is key to facilitate feasibility analysis and cluster resource allocation as presented in Section 1.1. This thesis shows that runtime can be predicted accurately through *hybrid prediction models* that combine the benefits of analytical modeling with the advantages of machine learning-based models and simulation. The generic prediction methodology we develop breaks the analytical task into multiple processing phases (based on semantics), collects key input features during a reference execution on a sample of the dataset, and then uses the collected features to build *per-phase* cost models. We customize this methodology for three categories of workloads: iterative machine learning, data pre-processing, and reporting SQL. We elaborate the technical contributions of this thesis bellow:

- **Sampling and Input Transformations for Iterative Processing on Graphs**: We develop the set of transformations applied to the input graph dataset and to the input parameters that altogether can preserve the processing characteristics of the iterative task during a short run on sample dataset. We propose *Biased Random Jump*, a sampling technique

that exploits the connectivity among highly connected nodes in scale-free graphs, and can be successfully used for prediction for a number of iterative algorithms widely used in analyzing social media data. We empirically show that the judicious choice of the sampling technique, that preserves key properties of the input dataset, altogether with input parameter transformations can be effectively used in prediction.

- **Hybrid Prediction Models for Data Analytics**: We design hybrid prediction models that combine the generality of analytical models , with the power of machine learning models to exploit prior workload executions. Such a modeling design allows to estimate the runtime of both conventional SQL operators, but also the runtime of *user defined* pre-processing tasks, and that of *iterative analytics* that are not “owned” by the data management layer. Additionally, our contribution includes the pool of key features that we identified for each workload category.
- **Training Methodology and Translation Models for Reporting SQL**: We propose a methodology for generating training queries and a pruning algorithm that limit the number of queries used in the training workload to a minimum. The training methodology reduces the time of running the training workload from days to hours while maintaining a good level of prediction accuracy for the models. Translation models, i.e., relative prediction models that exploit prior reference executions of the query that is predicted, improve the prediction accuracy of conventional prediction models beyond the training boundaries.

1.5 Thesis Outline

In this thesis we propose performance prediction techniques for *reporting* queries that include a class of iterative machine learning algorithms executing on BSP, ETL tasks expressed in Jaql, and SQL-like queries expressed in HiveQL.

The following section summarizes the contributions of the thesis and gives an overview of the next chapters.

- **Background**: Chapter 2 introduces general related work in the context of runtime performance prediction for data analytics. We start with background concepts on distributed processing engines in Section 2.1. Estimation techniques and sampling for iterative processing are presented in Section 2.2. Prediction approaches for DBMS are presented in Section 2.3, while prediction approaches for MapReduce are presented in Section 2.4. Section 2.5 summarizes recent prediction approaches while illustrating their limitations. Section 2.6 presents background modeling concepts that we later use for designing prediction models for iterative analytics, data pre-processing, and reporting SQL.
- **Runtime Prediction Methodology for Iterative Analytics**: Chapter 3 considers the problem of estimating the runtime of a class of iterative analytics operating on graph

datasets. Our main contribution for this problem is PREDIcT, an experimental methodology that proposes a set of transformations that can be used to estimate the number of iterations and the key input features of the iterative task using a sample-run on a small sample of the input dataset. The other contribution of this chapter is the design of a framework for building customized cost models for iterative analytics executing on top of Bulk Synchronous Parallel execution model (in particular, the Apache Giraph implementation). Finally, we present a thorough performance evaluation of PREDIcT on real datasets. For a 10% sample, the relative errors for estimating key input features range in between 5%-20%, while the errors for estimating the runtime range in between 10%-30% for all the scale-free graphs analyzed.

- **Runtime Prediction for Data Pre-processing (ETL Tasks)**: Chapter 4 tackles the problem of predicting the runtime of data pre-processing tasks. Examples of data pre-processing tasks include machine learning pre-processing (e.g., data cleaning) and ETL (Extract Transform Load). In this chapter, we propose a technique that predicts the runtime performance of a class of *fixed queries* running over varying input data sets. Our approach uses minimal statistics about the input data sets (e.g., input size, tuple size, cardinality), which are complemented with historical information about prior query executions (e.g., execution time). Our experiments on real workloads show the feasibility of the approach: we obtain less than 25% relative prediction error for 90% of predictions.

- **Runtime Prediction Methodology for SQL Analytics**: Chapter 5 addresses the problem of estimating the runtime of reporting SQL analytics. Starting from a prior execution of a reporting query we propose an approach for estimating the runtime of the query for other hypothetical configurations consisting of: i) query plan re-writes in terms of different operator implementation and possibly different packings of operators within one or several MapReduce jobs, and ii) a pool of potential hardware deployments. For this purpose we develop TITAN: i.e., Training Methodology and Translation Models for runtime prediction. Our contributions include a hybrid prediction approach and a training methodology that altogether reduce the training cost of state of the art prediction approaches while maintaining a good level of accuracy for the models. Our experiments show the feasibility of the prediction approach both on private and on public clusters. The 95-percentile average relative error is less than 25% on the testing benchmarks.

- **Conclusions**: Chapter 6 summarizes this thesis and outlines future avenues of research in query runtime prediction. We discuss a number of interesting topics that are worth pursuing in the context of scheduling, in-memory analytical engines, and shared infrastructures.

2 Background

In the first sections of this chapter we present related work on distributed processing engines and runtime prediction techniques applied for data analytics in general. Specific differences with respect to the prediction techniques proposed in this thesis are also summarized later in the respective chapters. In the last section of this chapter we introduce runtime modeling concepts that we use for runtime prediction in all of the following chapters.

2.1 Distributed Processing Engines for Scale-Out Analytics

2.1.1 MapReduce Execution Model

MapReduce is a programming model and a framework for processing large sets of raw data. A MapReduce program consists of two functions: map and reduce. The map function process the input data and produces a set of intermediate results as key-value pairs, while the reduce function aggregates all the intermediate results with the same key to produce the final result. MapReduce framework operates in conjunction with a distributed file system, where it stores the input and output data. Input data is represented as text or key/value pairs. Hence, the burden of data parsing is passed to the user's code. The data model allows for more flexibility as compared with state-of-the-art DBMS where the data has predefined structure, but comes at the cost of parsing the data each time a task is being executed.

Figure 2.1 illustrates the processing phases when running an analytical job on MapReduce. The job has three map tasks that read the input from the distributed file system, then apply the user defined transformations defined in the map function. Map tasks produce intermediate results as key-value pairs which are spilled to the local file system of each of the map tasks. Two reduce tasks copy the key-value pairs assigned to them based on a partitioning strategy on key, merge key-value pairs having the same key, then apply the user defined reduce function that produces the final result.

Some of the features that made MapReduce execution model popular among practitioners of

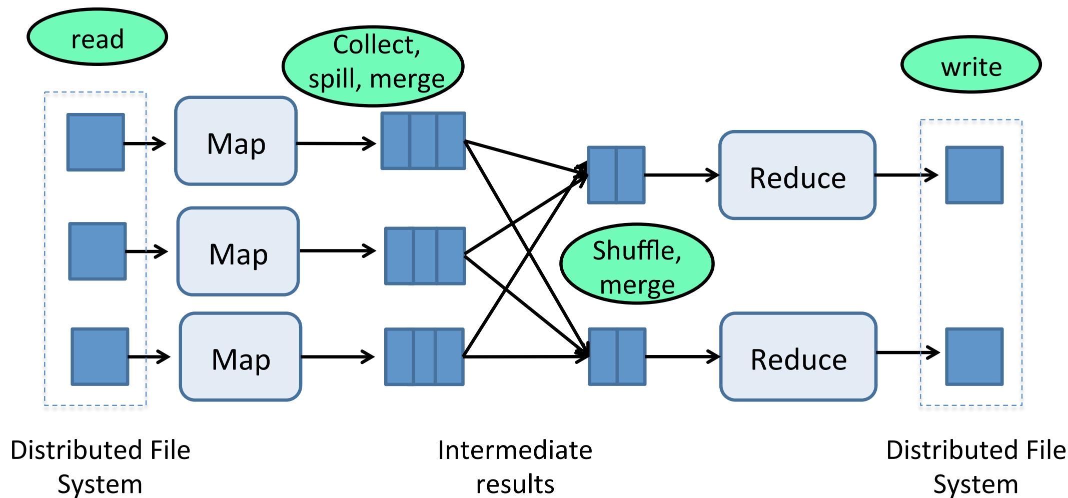


Figure 2.1 – MapReduce Execution Model: Map tasks transform the input data and output intermediate results as key-value pairs. The reduce tasks copy and merge all the values corresponding to the same key, then apply the reduce function to produce the final result.

data analytics:

- **Fault Tolerance and Scalability**: Designed to process large amounts of data using thousands of commodity machines, MapReduce has mechanisms to tolerate failures gracefully. MapReduce is resilient to large scale worker node failures by re-scheduling tasks on live nodes and by checkpointing intermediate results of the MapReduce job. Hence, in the case of a node failure the framework re-executes only the failed task of the MapReduce job.

- **Elasticity**: Worker nodes can be easily added in or removed. The underlying filesystem takes care of balancing the load among all the data nodes, while computation is uniformly balanced across all the worker nodes. Extending deployment of a traditional parallel database requires significant efforts in tuning it before actually exploiting the new hardware resources efficiently.

- **Load Balancing**: The MapReduce framework divides the Map and Reduce phases into a number of pieces much larger than the number of machines such that each machine executes many different tasks. This improves dynamic load balancing and recovery. If a node fails, tasks executed on the failed node can be spread out across all the other machines. Additionally, to deal with straggler nodes (i.e., machine that is performing poorly), MapReduce schedules backup tasks. Specifically, when the job is close to completion the master node schedules backup executions for the remaining tasks. A task finishes when either the primary or backup task completes its execution.

- **Open Implementation**: A big advantage of MapReduce based systems is that there is

an open source implementation that is available for free. Hadoop[1] is an open source implementation of the MapReduce framework.

Traditional applications of MapReduce include: ETL systems (Extract, Transform, Load) that transform data into different formats that are further consumed by other storage systems, complex analytics that cannot be expressed in SQL (multiple passes over the data: e.g., data mining, data clustering) or data analytics on unstructured data (grep, URL access frequency, inverted index).

Iterative Processing on MapReduce

Mahout [7] is a library for MapReduce that aims to scale the execution of machine learning algorithms on large input datasets. The library includes both iterative and non-iterative algorithms. The iterative category includes: clustering (e.g., spectral, k-means, canopy) and algorithms used by recommender systems (e.g., matrix factorization, collaborative filtering). The non-iterative category includes: classification (e.g., Naive Bayes, Logistic Regression), dimensionality reduction (e.g., PCA), etc.

Pegasus [44] is a similar effort as Mahout with the difference that the project is targeting to mine large input graphs. Pegasus proposes efficient *matrix-vector multiplication* abstractions that can be used to implement a class of graph mining algorithms on top of MapReduce engine. Examples of algorithms that benefit from such an abstraction: connected components, diameter/radius estimation, random walks, PageRank.

HaLoop [14] proposes optimization techniques when executing iterative computation on MapReduce. As explained in Section 1.2 and illustrated in Figure 1.1, iterative processing is executed among one relation that does not change and one relation that gets updated in every iteration. The MapReduce execution model is inherently inappropriate to execute iterative processing efficiently as it requires to read both inputs at the beginning of every iteration, and additionally it shuffles and then spills intermediate results on disk. For algorithms with a large number of iterations such an execution model is very inefficient. In this context, HaLoop [14] caches invariant input datasets in memory when executing iterative algorithms on MapReduce.

2.1.2 Distributed Graph Processing

An important class of analytics today is executed on graphs. Social media analysis and blog analysis require graph processing engines that can process large data sets efficiently. Recent graph processing engines use a *vertex centric approach*, that is, each vertex executes an instance of a *vertex program* towards implementing the graph algorithm. Concretely, each vertex executes local computation on its local data structures (state information can be associated

[1] https://hadoop.apache.org/

with the vertex and the neighboring edges), then it communicates with other vertices of the graph as needed to implement the semantics of the algorithm. The distributed algorithm implementation is in fact a succession of *iterations* that are composed of *local computation* and *communication*. Based on the communication and synchronization models (whether *synchronization* among all vertex programs is enforced or not at the end of each iteration), we can classify engines into: *synchronous* and *asynchronous*. In the following, we describe Pregel [55], a synchronous graph processing engine, and GraphLab [52] an asynchronous variant.

Pregel

Pregel follows a bulk synchronous parallel (BSP) [73] processing model that uses message passing for communication. All vertices run their vertex programs in parallel for a number of iterations (aka, super-steps). Before starting a new iteration, each vertex receives all its designated messages from the other vertices that were sent in the previous iteration. At the end of an iteration, each vertex send messages to its neighbors as needed. A barrier synchronization point is enforced at the end of each iteration to ensure that all messages send in one iteration are received before starting the next iteration. Vertices that completed running their vertex program can vote to halt. By doing so they remove themselves from the list of vertices that will execute computation in the next iteration. A halted vertex that receives a message it is automatically re-started in the next superstep. The algorithm completes when there are no more messages to be sent among vertices and all vertices voted to halt. Pregel introduces the concept of *combiners* which are user defined function that merge messages destined to the same vertex. Combiners are required to be associative and commutative operators.

GraphLab

GraphLab builds on an *asynchronous distributed shared memory* abstraction. In this processing model, vertex programs have shared access to a distributed graph. Concretely, each vertex program can access the data of its current vertex, of the neighboring vertices, and of the neighboring edges. A vertex program can be scheduled to be re-executed again (i.e., a new iteration) by itself or by its neighboring vertices through a signaling mechanism. We observe that GraphLab removes message passing interface and the synchronization point at the end of each iteration. Instead of using a barrier among all vertex programs, the underlying processing engine ensures serializability by preventing neighboring vertices (that access the same shared state) to be executed at the same time.

GraphLab introduces the Gather-Apply-Scatter (GAS) model to represent the conceptual phases of a vertex program. During the gather phase, state about adjacent vertices and edges is collected and aggregated into aggregator object. The aggregate operator can be a generalized *sum* over the neighborhood of the current vertex, and it must be associative and commutative. During the apply phase the current vertex value and the aggregator value are used to update

the new vertex value. Finally, during the scatter phase the new vertex value is used to update the data on adjacent edges and vertices. We note that the scatter and gather phases control the fan-in and the fan-out of the vertex program.

PowerGraph [31] was introduced to address the challenges of power-law graphs where there is a lot of work imbalance among vertices of the graph. In particular, power-law graphs have a small number of vertices that have much larger neighborhoods than most of the vertices of the graph. For such graphs, processing abstractions that distribute work symmetrically among vertices suffer from per iteration processing time imbalance. PowerGraph is a hybrid approach that adds parallelization within a vertex program. That is, vertices with large neighborhoods distribute the vertex program among multiple workers. PowerGraph is in fact a hybrid of GraphLab and Pregel abstractions, inheriting from GraphLab the asynchronous engine and the distributed shared memory model for data access, and from Pregel the combiner concept. PowerGraph parallelizes the gather and scatter phases of the GAS abstraction, uses a combiner to aggregate all the results created during the gather phase, then executes the apply function to update the vertex value with the aggregated result.

Other Graph Processing Engines

Other graph processing engines that optimize the runtime performance when processing large scale graphs in parallel were proposed in the recent years. Mizan [45] is a BSP graph processing engine that balances the load dynamically among workers based on performance characteristics collected at runtime. XStream [66] is a system for processing both in-memory and out-of-core graphs using a shared memory machine. GreenMarl [42] is a domain specific language abstraction and a runtime system that allows users to express their graph processing algorithms declaratively while not trading-off on the performance. The optimizer takes full control of translating the high level code into optimized code that is then executed in parallel using a distributed shared memory abstraction.

2.1.3 Spark Processing Engine

Spark [84] is an alternative MapReduce paradigm implementation that was designed for very fast computation. Spark is compatible with Hadoop's storage API, and is on average 40x faster than Hadoop [82]. Spark develops an in-memory storage structure called *resilient distributed datasets (RDDs)* for storing intermediate results. As a result of saving intermediate results on RDDs instead of saving them on disks, Spark is very efficient for iterative computation with a large number of iterations.

RDDs are a restricted form of distributed shared memory. They were designed to offer high throughput (close to the maximum given by the memory bandwidth) for coarse granularity updates. Concretely, RDDs are distributed collection of records that are immutable and cached in the memory of the cluster. They can only be built through coarse-grained deterministic

parallel transformations (e.g., map, filter, join, etc). In the case of failures, RDDs are re-built using lineage.

Shark Processing Engine for Mixed Analytics: Shark [82] is an execution engine for *mixed analytics* that supports both efficient SQL and machine learning computation at scale with *fine grain* fault tolerance. Shark is essentially HiveQL on Spark and is compatible with Hive's interfaces. Overall, Shark is in the range of (10x, 100x) faster than HiveQL on Hadoop. Besides the performance improvements due to using Spark's resilient distributed datasets abstraction, query planning in Shark benefits from additional optimizations such as: *partial DAG execution*, *map pruning*, *efficient memory storage*.

Partial dag execution: collects statistics at the runtime that are later used to select a particular join implementation (e.g., map join or shuffle join), and the degree of parallelism of the following jobs in the query DAG. For instance, based on the partition sizes fine grain partitions can be coalesced into coarse partitions. Other statistics that are collected: record counts, approximate histograms.

Map pruning: is a mechanism of pruning partitions that do not contain query results based on statistics that are collected during the data loading phase of each partition.

Efficient memory storage: Shark employs column-oriented storage using arrays of primitive types instead of storing rows as Java objects. Shark reduces the book-keeping overhead and at the same time it reduces the access time.

2.2 Estimating and Optimizing Iterative Processing

Prior work on iterative algorithms mainly focuses on providing theoretical bounds for the number of iterations an algorithm requires to converge (e.g., [46, 44, 34]) or worst case time complexity (e.g., [8]). These parameters, however, are not sufficient for providing wall time estimates due to the following two reasons: i) As simplifying assumptions about the characteristics of the input dataset are made, theoretical bounds on the number of iterations are typically *loose* [46, 8]. This problem is further exacerbated for a category of iterative algorithms executing *sparse computation*, where the processing requirements of any arbitrary iteration vary a lot as compared with subsequent/prior iterations [26, 52]. For such algorithms, per iteration worst case time complexities are impractical when the goal is to estimate *actual runtime*. ii) Per iteration processing runtime cannot be captured solely by a complexity formula. System level resource requirements (i.e., CPU, networking, I/O), critical path modeling and a cost model are additionally required for modeling runtime.

Iterative execution was also analyzed in the context of recursive query processing. In particular, multiple research efforts [10, 3, 13] discuss execution strategies (i.e., top-down versus bottom-up) with the goal of performance optimization. Ewen et al. [26] optimize execution of *incremental iterations* that are characterized by few localized updates, in contrast with

bulk iterations, that always update the complete dataset. Although performance optimization has an immediate impact on the runtime of the queries, the aforementioned techniques are complementary to the runtime prediction problem we study in this thesis.

2.2.1 Approximating and Sampling Large Graphs

With the goal of reducing the processing time of ever increasing input graphs, sampling and sketching techniques that can *approximate* some of the properties of the complete graph have been studied over the past recent years (e.g., [47, 48, 33, 43]). In these works, the main goal is to take a sample that can be used to approximate the result of the graph processing task. For instance, evaluating whether the input graph is connected, approximating the in/out node degree distributions, the effective diameter (i.e., 90-th percentile longest distance).

Sampling graphs had been analyzed in the context of social networks. Leskovec et al. propose sampling techniques based on *random walks* [47] with the goal of maintaining certain properties on the sample such as the in/out node degree distributions, clustering coefficient, and effective diameter.

A random walk on a graph starting at a vertex v corresponds to randomly picking an edge that starts at v and ends at one of v's neighboring vertices. A sampling technique based on random walk takes multiple random walks on the input graph until a certain percentage of vertices (or edges) have been sampled. There are multiple variants of sampling algorithms based on random walks. An excellent survey is that of Hu et al. [43]. In the following we summarize the best performing sampling techniques in the context of preserving connectivity, node in/out degree proportionality, and the effective diameter of the sampled graph.

- **Random Walk** [47]: Random Walk picks a starting seed vertex uniformly at random from all the input vertices. Then, at each sampling step an outgoing edge of the current vertex is picked uniformly at random and the current vertex is updated with the destination vertex of the picked edge. With a probability p the current walk is ended and a new random walk is started from the original seed vertex. The process continues until the number of vertices picked reaches the sampling ratio. With this sampling strategy there is a risk of getting stuck, if the starting vertex is a sink, or if it belongs to a small isolated component. If after a long number of sampling steps there is no progress in the number of picked vertices, random walk re-initializes the starting node to a new arbitrary vertex of the graph.

- **Random Jump** [47]: Random Jump is very similar with Random Walk. The difference is that Random Jump re-initializes the starting node to an arbitrary vertex of the graph *each time* a new random walk is started. Hence, this sampling scheme has no risk in getting stuck during the sampling process.

- **Metropolis-Hastings Random Walk** [30]: Sampling techniques based on random walk

are known to have bias towards high degree nodes in the input graph. That is vertices with high out degree are likely to be visited more often during the sampling process than vertices with low out degree. With the goal of sampling vertices uniformly at random (i.e., with a probability of $\frac{1}{|V|}$, where $|V|$ the total number of vertices in the graph), Metropolis-Hastings Random Walk adjusts the transition probability within a random walk as follows:

$$P_{v,w} = \begin{cases} \frac{1}{k_v} \times min(1, \frac{k_v}{k_w}), & \text{if w is neighbor of v} \\ (1 - \sum_{y!=v} P_{v,y}), & \text{if w = v} \\ 0, & \text{for any other vertex of the graph} \end{cases}$$

, where k_v, and k_w the out degrees of vertices v, and w. In summary, MHRW always accepts a walk towards a vertex with a lower degree and rejects some of the moves to vertices with higher degree. Thus, it eliminates the bias towards vertices with high degree.

Sampling vs. Sketching

In the context of data streaming model, McGregor et al. proposes *sketching* techniques with the goal of reducing the cost of processing large input graphs [33, 5]. Concretely, sketches reduce the algorithm processing space complexity from $O(n^2)$ to $O(n \times polylog(n))$. Sketching techniques use multiple linear projections of the input graph so that they can preserve a certain property of the original graph (such as connectivity, k-connectivity, bipartiteness) in the sketch space with high probability. Once a sketch is constructed, the algorithm is executed in the sketch space to approximate results: Given a graph processing task T, an input graph G, and a corresponding sketch S, the result of executing T on G is approximated with the result obtained by executing T on S. The main differences with sampling approaches based on random walks can be summarized as follows: i) random walk-based sampling approaches aim to preserve multiple properties of the input graph while sketching is customized to preserve only one input property with high probability. ii) random walk samples are used to summarize some of the characteristics of the complete graph whereas sketches aim to reduce the memory (space) requirements of processing large input graphs in the context of *data streams*.

2.3 Performance Prediction for DBMS

Estimating the runtime execution of analytical workloads was heavily studied in the DBMS context from multiple angles: *initial runtime predictors* [29, 4, 25], *progress estimators* [16, 53, 57], and *self-tuning systems* [40, 38]. Prediction approaches proposed in the DBMS context account for system level resource requirements (i.e., CPU, IO, network, memory) and use a

cost model (either analytical, based on black box modeling or a hybrid) for translating them into runtime. For instance, [29] proposes a technique to predict query performance using a cost model based on machine learning that clusters queries with similar performance based on query's input parameters as known as *input features* in the machine learning community.

2.3.1 Nearest Neighbors-based Prediction

Ganapathi et al. propose an approach for predicting the runtime execution of HP Neoview queries using statistical methods based on Kernel Canonical Correlation Analysis [29]. The proposed model predicts the runtime of a query based on the runtime of *m* nearest neighbor queries for which performance was tracked during a training phase. For finding the nearest neighbors an n-dimensional distance metric is used (i.e., kernel), the dimension being given by the size of the feature vector. The key input features include operator types, operator counts, and input data statistics as returned from the query optimizer.

A related approach that uses linear regression to model the runtime of analytical queries was proposed by Zhu et al. in the context of Multi Database Systems [86]. The proposed method separates queries into classes according to their access methods so that the cost of the queries in each query class can be approximated by the same formula. The set of features considered are the input/output cardinalities, the size of intermediate results, the tuple length and the physical size of the input/output tables. The approach uses multi-variate linear regression to model the cost of queries.

2.3.2 Operator Level Models

Recent research introduced the idea of using *operator level* machine learning models [6, 50]. In particular, Akdere et al. [6] propose multiple granularity prediction models: "plan level" models, "operator level" models or hybrids of the above two with the goal of predicting the runtime of analytical queries for both static and ad-hoc workloads. Their work stresses the idea of *model re-usability* through operator level models. While operator models are more accurate for computing the runtime of operators, they require additional modeling mechanisms for computing query level estimates (i.e., modeling the critical paths, taking into account operator pipelining).

Li et al. propose an approach for improving the prediction accuracy on testing sets outside of the training boundaries [50]. In particular, each operator is modeled through a hybrid of models with a fixed functional form and decision trees which are discrete (i.e., in particular, Multiple Additive Regression Trees). Based on the observation that models with a fixed functional form are more powerful on testing sets outside of the training boundaries, and that decision trees are more accurate when the training sets have a good coverage of the testing set, they propose a hybrid of the two.

2.3.3 Progress Estimators

A sub-class of performance estimators focuses on estimating the progress of queries at runtime (e.g., [16, 53, 57]) rather than on predicting their runtime before execution. In contrast with prediction mechanisms, progress estimators benefit from a feedback loop mechanism which can correct wrong estimates at the runtime. Similar adaptive techniques that are calibrating statistics at runtime were also proposed in the literature [69, 24]. Progress estimators do not replace the requirement for runtime predictors, as for many use cases (i.e., scheduling, resource allocation) runtime estimates are required *before* the query starts execution. In fact, runtime predictors can be used in conjunction with progress estimators to estimate the runtime of forthcoming query pipelines for which dynamic statistics (collected at runtime) are not yet available.

2.3.4 Performance Modeling for Storage Devices

Mesnier et al. [56] propose relative fitness models for storage devices that estimate performance of a workload on device D1 based on a set of features that include performance and resource utilization counters (in addition to the workload characteristics) corresponding to the workload execution on another device D2. The training phase consists of: i) running synthetic benchmarks on all storage devices that are representative for a large spectrum of real workloads, and ii) building pairwise models among pairs of devices that exploit correlations among performance metrics of a given workload executing on different devices. Classification and Regression Tree (CART) models are used in the model fitting phase due to their simplicity and flexibility.

2.4 Performance Prediction for MapReduce

2.4.1 Self-tuning and Optimization

Herodotou et al. propose Starfish [38], a self-tuning system for Extract Transform Load (ETL) workloads that uses performance models with the goal of workload tuning, in particular, finding the best set of configuration settings for a given workload and a cluster deployment. Starfish was designed to help practitioners in data analytics getting the best job performance without requiring them to know the tuning knobs of the underlying MapReduce infrastructure. The key building block for finding the best configuration settings is the *job profile*, which models the processing characteristics of an input job. The processing characteristics are grouped into: processing cost factors, and data statistics. Given a MapReduce job, a job profile is taken by executing the job on a *sample* of the input dataset. Then, the approach uses the job's processing characteristics from the job profile, a set of analytical models, and simulation to predict the job's runtime for a range of input configuration settings. Starfish's What-If engine is called for each potential input configuration, and the best configuration setting is returned to the user.

Elastisizer [39] extends Starfish's approach for the *cluster sizing problem*, which stands for finding the cluster size and the type of machine instances (in terms of resource characteristics) that best meet the requirements of the workload. Hence, in addition to configuration settings, the search space includes cluster resources in terms of instance types on Amazon EC2. Elastisizer uses controlled black box models for estimating the processing cost factors on the target deployment. A set of synthetic workloads are generated and executed on each instance type to generate the data that is used for training. In terms of model fitting algorithm, M5 tree models [65] are used. M5 tree is a decision tree that instead of using the average of all training examples falling within a leaf node, uses a second level modeling phase among all observation falling within each leaf node. In particular, linear regression models are used to fit the observations within each node. We make several observations: Both Starfish and Elastisizer *average* processing cost factors among the job's task profiles to produce a *reference profile* that is later used in prediction. As we later show averaging processing cost factors among tasks with very different input data properties is one source of modeling error that may cause important inaccuracies when estimating the runtime with Starfish's analytical models. The task profiles are collected at MapReduce phase granularity and thus, do not track more specific information about the processing tasks executed within the map, and reduce functions (e.g., scan, join, project operators).

Wu et al. [80] propose analytical cost models for HiveQL operators with the underlying goal of query optimization. Their work is tailored towards reducing the size of intermediate results by adaptively grouping join operators that can be processed in one single MapReduce job. Unlike conventional optimization, the proposed optimization approach is tailored to the characteristics of MapReduce processing model such as materialization of intermediate results and data shuffling. Similarly with PostgreSQL cost model, or the query cost calibration approaches proposed for PostgreSQL [81, 68], processing cost factors are assumed *constant* for a given hardware infrastructure.

2.4.2 Nearest neighbors-based prediction

Ganapathi et al. extend their method proposed in the context of database queries [29] for HiveQL queries executing on top of MapReduce [28]. The input features used include a mix of MapReduce specific configuration settings and query features taken from the query plan. The models are built at coarse granularity (job/query granularity). The set of input features considered include configuration parameters and input data characteristics: i.e., number and location of map/reduce slots, input bytes, bytes read from local disk, bytes read from the distributed file system (i.e., HDFS in Hadoop). A large number of training queries is used for fitting the models (in the order of 1000s).

2.4.3 Resource Allocation and Scheduling

FLEX [79], ARIA [76], and Rayon [21] optimize resource allocation for large scale analytical workloads in accordance with deadlines and user contracted Service Level Agreements (SLAs). For finding an optimal allocation of resources all of the above schedulers require as input query runtime estimates corresponding to each potential resource allocation configuration. Thus, mechanisms for estimating the *runtime requirements* associated with each resource allocation configuration are of paramount importance.

ARIA is an SLA aware scheduler based on the Earliest Deadline First scheduling policy. Given a MapReduce job and a deadline, ARIA estimates the resources required to execute the job while satisfying the deadline. The prediction component of ARIA uses a job profile to summarize the performance characteristics of the job. Their workloads include ETL tasks such as: word count, sort, classification, term frequency - inverse document frequency (TF-IDF).

Rayon system [21] introduces the concept of resource reservations into YARN [75] aiming to provide *predictable resource allocations* for mixed analytical workloads. Rayon is built on top of the capacity scheduler configured with one dedicated queue where it accepts reservation requests for production jobs (that have strict deadline constraints), and one default queue where it accepts requests for best effort jobs (with no time constraints, but sensitive to latency). Given this workload mix, Rayon targets to improve the resource utilization of the cluster as much as possible while satisfying deadline constraints for production jobs, and minimizing latency for best effort jobs.

2.5 Existing Prediction Approaches vs. Current Requirements

Table 2.1 summarizes recent prediction approaches proposed in the context of DBMS and MapReduce along multiple dimensions: workload type, modeling approach, modeling granularity, and prediction goal. Prediction approaches proposed in the DBMS context that use machine learning as their underlying technique: *KCCA*, *Hybrid QPP*, and *MART* show that machine learning models are more accurate than pure analytical models when the training datasets cover the input query space well. The reason is that they can capture a wide range of runtime execution effects (e.g., interplay among workload, DBMS and underlying hardware) implicitly rather than explicitly by exploiting prior query executions. To improve the applicability of training based approaches to a broader set of testing queries, not covered by the queries from the training set, *operator granularity* models were proposed by *Hybrid QPP* and *MART* as a replacement to the query granularity models proposed by *KCCA*. *Enhanced PostgreSQL* proposes mechanisms to calibrate PostgreSQL's cost models and use a pure analytical approach.

Prediction approaches proposed in the MapReduce context target workload optimization (i.e., *AQUA*) and tuning, i.e., finding the set of configuration settings that give the best performance for a given workload (i.e., *Starfish* and *Elastisizer*). With the exception of *Elastisizer* and *KCCA*

for MapReduce, they use an analytical approach for building the cost models at the granularity of MapReduce phases.

We observe that neither of these approaches addresses the problem of estimating the runtime of *iterative analytics* which are prevalent in today's analytical workflows (as presented in Section 1.2). Additionally, all of the prediction approaches that use machine learning as their underlying modeling technique are targeted for fixed deployments and use a very large number of queries for training their models. While for *fixed deployments* training incurs an one time cost, in the context of *elastic* deployments, where a large number of potential hardware deployments can be provided to applications on-demand, high training cost is unacceptable. Finally, neither approaches analyzes the trade-offs among analytical modeling versus hybrid modeling for reporting SQL queries executed at scale. This thesis aims to fill in these gaps.

Name	Workload type	Workload sub-type	Context	Modeling approach	Model granularity	Training cost	Deploy-ment	Goal
KCCA [29, 28]	ad-hoc	SQL	DBMS MapReduce	ML	query	1000 training queries (days)	fixed	prediction
Hybrid QPP [6]	ad-hoc	SQL	DBMS single node	hybrid	operator	800 training queries (days)	fixed	prediction
MART [50]	ad-hoc	SQL	DBMS single node	hybrid	operator	2500 training queries (days)	fixed	prediction
Enhanced Postgre-SQL [81]	ad-hoc	SQL	DBMS single node	analytical	optimizer cost model	sample run, calibration queries (an hour)	fixed	prediction
AQUA [80]	ad-hoc	HiveQL	MapReduce distributed	analytical	operator	calibration queries (NA)	fixed	optimi-zation
Starfish [38]	reporting	ETL	MapReduce distributed	analytical	average task phase	sample run (p% of actual run, e.g., 10%)	fixed	tuning
Elasti-sizer [39]	reporting	ETL	MapReduce distributed	hybrid	average task phase phase	sample run, calibration queries (hours)	elastic	tuning, resource allocation

Table 2.1 – Recent prediction approaches for analytical workloads.

2.6 Runtime Modeling: Background Concepts

In this section we describe the core concepts used for building hybrid prediction models in the following chapters. We start with an overview on the modeling steps required to build cost models using a machine learning approach. We detail each step in turn, then, give examples of fitting algorithms used for building the models. In the end, we present the metrics of interest for assessing the accuracy performance of the models.

2.6.1 Runtime Modeling Overview

Building a cost model using a learning approach involves multiple phases: i) generating training data sets; ii) collecting and identifying key input features (i.e., input parameters) that have a high impact on the query runtime (i.e., candidate input features extraction); iii)

building the cost model that maps key input feature values into runtime (i.e., model fitting). After the model is built we can use it for prediction. For estimating the runtime of a query Q we have only to input its key input features into the cost model which was trained to turn them into runtime.

We categorize key input features into: *operator features* that relate to the semantics of the query, *data features* that relate to the characteristics of the input dataset and the query, *performance features* that include the costs of executing different operator phases, and *configuration settings* that include the software settings and hardware resource allocation. In the following subsections we describe in more details the process of collecting key input features and that of building cost models.

2.6.2 Collecting Input Features

For building cost models based on learning training data is required. Training data is generated by running a number of synthetic or real workloads on a range of input datasets and execution configurations. The query execution is instrumented such that data features, operator features and performance features are collected by a profiler and stored into a centralized database.

We collect key input features at the granularity of MapReduce tasks and store them into *task profiles*. Concretely, for a MapReduce job with m mappers and r reducers, a number of $m+r$ profiles are collected. Within a task data features, operator features and performance features are collected based on domain knowledge about the operator's semantics. Generally, two categories of features are collected: runtime execution specific (i.e., MapReduce framework dependent), and workload specific (e.g., at the query processing layer, or at the graph processing layer, etc).

We use BTrace [2] to profile the execution of distributed queries. BTrace allows users to write scripts (annotated Java programs) that specify *profiling rules*. BTrace uses java agents to run the compiled script into the same JVM as the actual java program. When the precondition of a BTrace rule is satisfied, the bytecode corresponding to the rule is executed. Thus, collecting query input features is possible for any arbitrary method for which a triggering rule exists, and that is called during the query's execution.

Figure 2.2 shows a code fragment corresponding to a BTrace rule. The BTrace rule intercepts the call to *getOrCreatePartitions* method of *GraphTaskManager* class, located inside the *processGraphPartitions* method. The profiling method *onGraphTaskManager_on_processGraphPartitions* waits until the intercepted method completes, as specified with the where clause, *where = Where.AFTER*, then it accesses two attributes of the returned object i.e., *partition* (lines 216-217). The parameters of the profiling method have the same signature with that of the intercepted method and can be accessed at the interception time. Returned values are also available for profiling if they are intercepted through the *Return* annotation.

[2] https://kenai.com/projects/btrace

```
207  @OnMethod(clazz = "org.apache.giraph.graph.GraphTaskManager",
208          method = "processGraphPartitions",
209          location = @Location(where=Where.AFTER, value = Kind.CALL,
210              clazz="org.apache.giraph.partition.PartitionStore",
211              method="getOrCreatePartition"))
212
213  public static void onGraphTaskManager_on_processGraphPartitions(Integer partitionId,
214          @Return Partition partition) {
215
216      perWorkerVertices.addAndGet(partition.getVertexCount());
217      perWorkerEdges.addAndGet(partition.getEdgeCount());
218  }
219
```

Figure 2.2 – BTrace profiling rule example.

2.6.3 Building a Cost Model

There are multiple ways of building a cost model: *explicitly* through analytical modeling, *implicitly* through machine learning given that training data is available, or as a *hybrid* of the two. Analytical models are built by experts based on domain knowledge about the query processing model, and use explicit formulas among key input features to compute the output feature (runtime). Instead of using explicit formulas, learning based models build *implicit* models through training (i.e., model fitting). In the following we describe multiple model fitting mechanisms that we later use for prediction.

Depending on how much information is available regarding the functional dependency among the key input features and the output feature (i.e., runtime) we can categorize model fitting algorithms into: i) algorithms with a *fixed functional form*, where the canonical form of the modeled cost function is known a priori and the only unknowns are the coefficients of the function which are learned from the training data, and ii) algorithms with *unknown functional form*, where the function corresponding to the modeled output is unknown. For the last case, algorithms that quantify *similarity* among input features based on a distance metric (e.g., nearest neighbors, kernel methods, support vector machine) or that segment the input feature space (i.e., decision trees) are used instead.

Any given fitting algorithm has an objective function (also known as the loss function) that drives the process of optimizing the model using training samples. One of the most common objective function for regressive models, that predict continuous values, is to minimize the mean squared error between the actual and predicted value on the samples from the training set. We further detail the objective function when describing each model fitting mechanism in particular.

Model Fitting for Fixed Functional Forms

Multi-variate Linear Regression: Formally, given a set of input features $X_1, ..., X_k$, and one

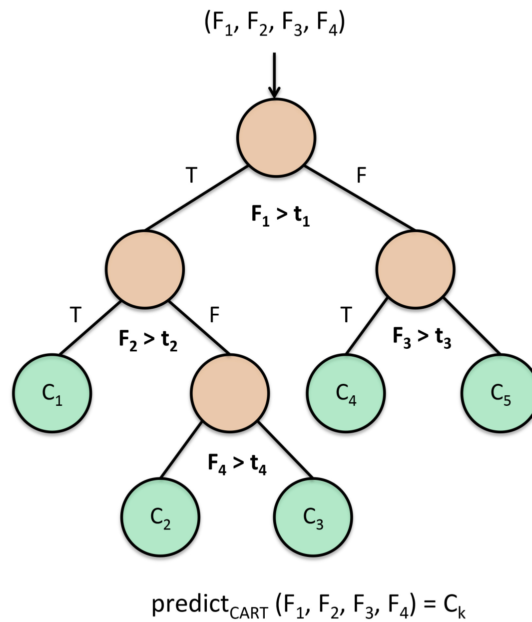


Figure 2.3 – CART decision tree with four input features $F_1 - F_4$, four conditionals, and five possible predicted values $C_1 - C_5$.

output feature Y (i.e., processing phase runtime), the model has the functional form:

$$f(X_1,...,X_k) = c_1X_1 + c_2X_2 + ... + c_kX_k + r$$

where c_i are the coefficients and r is the residual value. The model fitting algorithm for multivariate regression seeks to find the coefficients and the residual value such that the mean squared error among the estimated runtime value and the actual runtime value for the queries from the training set is minimized. In fact the coefficients of the model can be interpreted as the "cost values" corresponding to each input feature.

Model Fitting for Unknown Functional Forms

Decision trees are a good modeling approach when the underlying dependency among the input features and the output feature is not known in advance or when the dependency does not follow a fixed functional form. Decision trees are thus general and applicable to a large class of prediction problems. A large number of fitting algorithms based on decision trees exist [35].

Classification Classification and Regression Trees (CART): CART models are well known among decision tree algorithms due to their generality, practicality, and expressivity. CART models grow a decision tree by classifying the samples from the training set into multiple zones based on a recursive binary tree growing procedure. Initially, all the training samples are located in one single node. In the next step, the split (i.e., the input feature and the

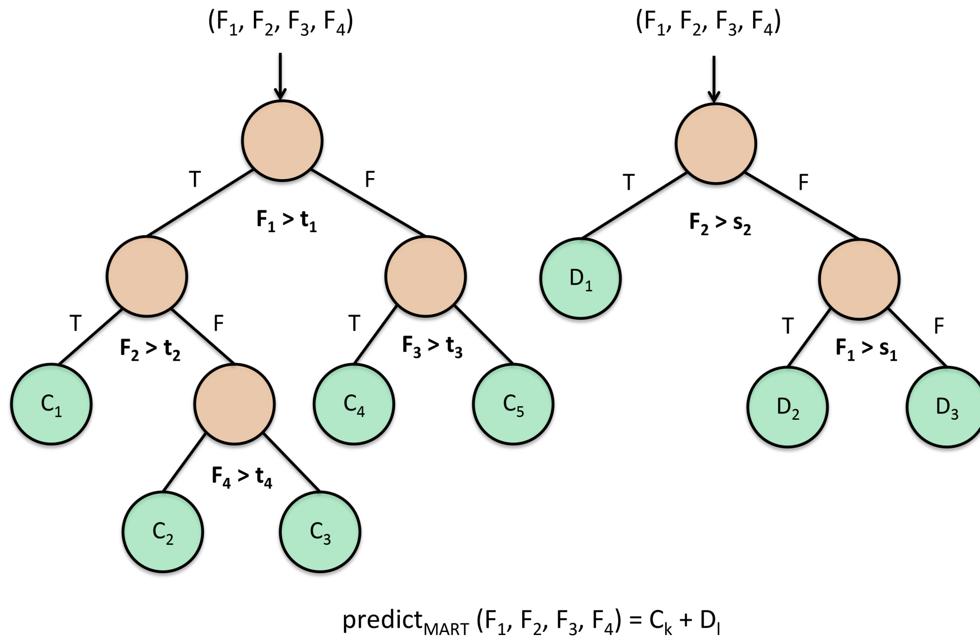


Figure 2.4 – MART decision trees with two boosting iterations (i.e., two trees) and four input features $F_1 - F_4$. The predicted value is a summation over the predicted values of each tree.

threshold value) that best separates the training samples into two subsets is searched for. A good separation is achieved when there is small discrepancy among the output feature value assigned to a tree node (i.e., the *average* output feature value of all samples within that node) and the actual output feature values of the samples within that node. More concretely, the split that reduces the average squared error the most is chosen. The process continues iteratively until there is no more significant error reduction or until the minimum number of samples within a leaf node has been reached (no more splits are allowed). Figure 2.3 shows a CART tree with four input features, four conditionals (the intermediate tree nodes), and five possible predicted values (the leaf nodes).

Multiple Additive Regression Trees (MART): In contrast with CART, MART iteratively builds a *sequence* of regression trees instead of building one single regression tree per model. The advantage is that each subsequent tree in the sequence is built to compensate for the residual errors observed on the training data on the current tree, hence prediction errors can be further reduced. Figure 2.4 shows a MART model with two trees. MART models were shown to have very good properties in the context of runtime and resource prediction [50].

Hybrid Decision Trees: Hybrid decision trees combine the power of decision trees of segmenting the input feature space into multiple zones and the generality of fixed functional forms within a leaf node. That is, instead of estimating the *average* output feature value of *all* samples within a leaf node, a fixed functional form is fitted instead. Thus, different output feature values can be predicted from the same leaf node. This set of features make hybrid

decision trees powerful, and more advantageous to use compared with simple tree models (e.g., CART), and fixed functional form models (e.g., multi-variate linear regression). M5 Tree [65] is a model fitting algorithm that is implementing such a hybrid decision tree model.

2.6.4 Prediction

During the prediction phase the output performance metric is estimated using the models built in the model building phase. For instance, for the decision tree model illustrated in Figure 2.3, and a query Q with feature vector $(F_1, F_2, F_3, F_4) = (2t_1, 0.5t_2, 0.5t_3, 2t_4)$, the estimated value is obtained by finding the leaf node that satisfies all the conditionals encountered during the tree traversal on the input feature vector. Thus, in this example the predicted value is C_2.

2.6.5 Accuracy Metrics of Interest

We quantify the accuracy of the prediction models on multiple accuracy metrics to assess the quality of estimations for different end-to-end use cases, as presented in Section 1.1.

Relative prediction error (RE): Conventional metric used in runtime prediction (e.g., [29, 6, 81]). It is defined as: $RE = \frac{|Predicted - Actual| \times 100\%}{Actual}$ and it measures the relative error of the predicted runtime with respect to the actual runtime.

Average relative prediction error ($\overline{RE}$): It is defined as the average relative prediction error corresponding to a workload of analytical queries: $\overline{RE} = \frac{1}{n} \times \sum_{i=1}^{n} RE_i$.

Cumulative distribution function of relative error (CDF): It shows the distribution of the relative prediction errors corresponding to a workload. It can be used to quantify the proportion of predictions that have a relative error bellow a target error.

Ratio Error: It shows the maximum ratio error among the predicted runtime and the actual runtime. It is defined as: $Ratio_Error = max\{\frac{Predicted}{Actual}, \frac{Actual}{Predicted}\}$.

Order preserving degree (OPD): Given a workload for which predictions are computed, the order preserving degree measures the proportion of predictions for which the relative order among pairs of predictions is maintained the same with to the relative order among the corresponding pairs of actual values. This metric was proposed in query optimization to assess the accuracy of the optimizer to compare alternative query plans (e.g., [85]). The order preserving degree is also very useful in the context of resource allocation and deployment. For instance, when targeting to answer performance boost questions: i.e., that aims to find the execution configuration that can improve the actual performance of the workload by a given factor.

Coefficient of determination (R^2): Measures how well the training data fits the model. We use the following formula for computing this metric: $R^2 = 1 - \frac{\sum_i (Actual_i - Predicted_i)^2}{\sum_i (Actual_i - \overline{Actual})^2}$ The closer the value is to one, the better the model fits the training data.

As there is no one single error metric to capture the quality of predictions for all potential prediction use cases, we choose to quantify the accuracy of our models on multiple prediction metrics. In the sections where we target a particular use case, we focus on the metrics that are the most relevant for that use case.

2.7 Summary

In this chapter we presented distributed processing engines for scale-out data analytics, state of the art prediction approaches, and background concepts for runtime cost modeling. In the first section, we started with the MapReduce execution model that received momentum in the recent years due to its scalability and fault tolerance properties for executing data processing tasks on large cluster deployments. We then discussed iterative processing and distributed graph processing models that can be used for executing iterative computation at scale. We briefly summarized alternative MapReduce-like paradigms for very fast computation using in-memory storage structures. In the following sections we presented an overview of prediction approaches in the context of iterative processing, DBMS, and MapReduce. While in the context of iterative processing there are no empirical approaches aimed to estimate runtime, cost models based on analytical modeling and machine learning are available for conventional SQL and non-iterative ETL. We summarized existing approaches and at the same time pin-pointed prediction requirements that are not yet addresses in the literature. At the end, we briefly presented background concepts for building cost models using machine learning-based models.

3 Runtime Prediction for Iterative Analytics

3.1 Introduction

Today's data management requirements are more complex than ever, going beyond the traditional DBMS operators [20]. Analytical tasks often include iterative machine learning or graph mining algorithms [44, 52] executed on large input datasets. For instance, Facebook uses machine learning to order stories in the news feed (i.e., ranking), and to group users with similar interests together (i.e., clustering). Similarly, LinkedIn uses large scale graph processing to offer customized statistics to users (e.g., total number of professionals reachable within a few hops). These algorithms are often iterative: one or more processing steps are executed repetitively until a convergence condition is met [44].

Predicting the runtime of iterative algorithms poses two main challenges: i) predicting the number of iterations, and ii) predicting the runtime of each iteration. In addition to the algorithm's semantics, both types of prediction depend on the characteristics of the input dataset, and the intermediate results of *all* prior iterations. On one hand, the number of iterations depends on how fast the algorithm converges. Convergence is typically given by a *distance metric* that measures incremental updates between consecutive iterations. Unfortunately, an accurate closed-form formula cannot be built in advance, before materializing all intermediate results. On the other hand, the runtime of a given iteration may vary widely compared with the subsequent iterations according to the algorithm's semantics and as a function of the iteration's current *working set* [26]. Due to *sparse computation*, updating an element of the intermediate result may have an immediate impact only on a limited number of other elements (e.g., propagating the smallest vertex identifier in a graph structure using only point to point messages among neighboring elements). Hence, estimating the time requirements, or alternatively, the size of the working sets of each iteration *before* execution is difficult.

Existing Approaches: Prior work on estimating the runtime or the progress of analytical queries in DBMS (e.g., [4, 16, 25, 29, 51]) or more recent MapReduce systems (e.g., [38, 40, 57, 63]) do not address the problem of predicting the runtime of analytical workflows that include *iterative algorithms*. For certain algorithms theoretical bounds for the number of

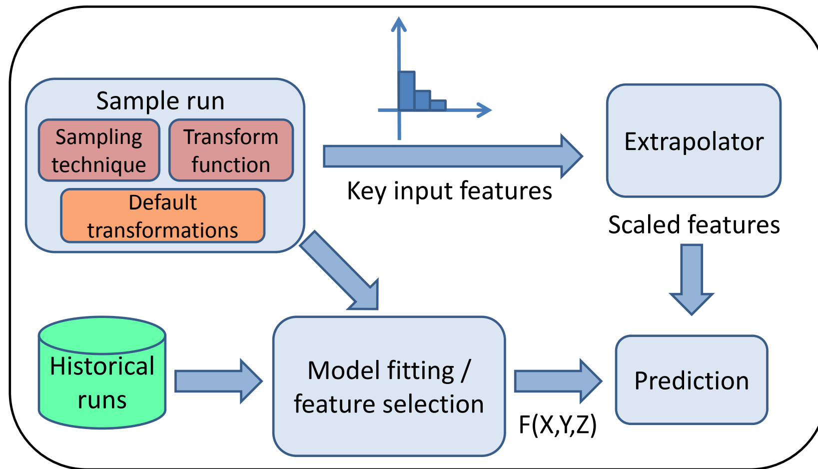


Figure 3.1 – PREDIcT's methodology for estimating the key input features and runtime of iterative algorithms.

iterations were defined (e.g., [34, 44, 46]). However, due to simplifying assumptions on the characteristics of the input dataset theoretical bounds are typically too coarse to be useful in practice.

3.1.1 Sketch of Proposed Approach

In this chapter we introduce PREDIcT, an experimental methodology for iterative algorithms that estimates the number of iterations and per iteration key input features capturing resource requirements (such as function call counters, message byte counters), which are subsequently translated into runtime using a cost model. Figure 3.1 illustrates PREDIcT's approach to estimate the runtime of iterative algorithms. One of the key components of PREDIcT is the sample run, a short execution of the algorithm on a sample dataset. During the sample run key input features are collected and used later as a basis for estimating the processing characteristics of the algorithm on the complete input dataset. However, as some algorithm parameters are tuned to a certain dataset size, a sampling run cannot simply execute the same algorithm with the same parameters on a smaller dataset. We first have to identify the parameters that need to be scaled and then apply the *transform function* to obtain the suitable values for the sample dataset size. One such parameter is the convergence threshold used by PageRank [59] and other algorithms. We illustrate the need to scale the threshold with an example.

Example: PageRank is an iterative algorithm that computes the rank of all vertices of a directed graph by associating to each vertex a rank value that is proportional with the number of references it receives from the other vertices, and their corresponding PageRank values. PageRank converges when the average delta change of PageRank at the graph level from one iteration to the next decreases below a user defined threshold $\tau \geq 0$. For acyclic graphs convergence

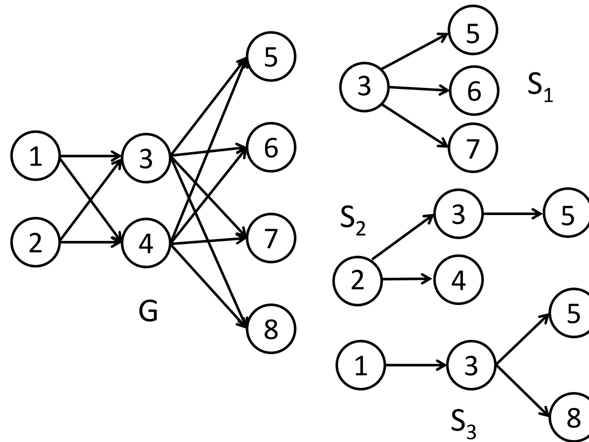


Figure 3.2 – Maintaining invariants for the number of iterations when executing PageRank on sample graphs.

to $\tau = 0$ is given by $D+1$, where D is the diameter of the graph. Consider Figure 3.2 showing an input graph G, and three arbitrary samples S1-S3, with a sampling ratio of 50% of vertices. The complete graph requires three iterations to converge (i.e., $D = 2$). Sample S1 requires only two iterations, while samples S2 and S3 require three iterations as they preserve the diameter. However, none of the samples above maintain invariants for the number of iterations given an arbitrary convergence threshold $\tau > 0$. Due to the different number of vertices, edges or in/out node degree ratios of samples S1-S3 as compared with G, the average delta change of PageRank on the samples is not the same when compared to the corresponding average delta change on G. Computing the average delta change of PageRank for the first iteration results in: $\overline{\Delta}_{S1,1} = 3d/16$, $\overline{\Delta}_{S2,1} = d/8$, $\overline{\Delta}_{S3,1} = d/8$, and $\overline{\Delta}_{G,1} = d/16$, where $d = 0.85$ is the damping factor (for deriving the values please see Section 3.6.1). For this example, for a threshold $\tau = d/16$ the actual run converges after one iteration, whereas all sample runs continue execution. By applying the transformation $T = (\tau_S = \tau_G \times 2)$ during the sample run on samples S2 or S3, the same number of iterations is maintained as on the complete graph. Hence, only by combining a transform function with a sampling technique (which maintains certain properties of G: e.g., diameter), invariants can be preserved.

PREDIcT proposes the methodology for providing transformation functions on a *class of iterative algorithms* that operate on homogeneous graph structures, and have a global convergence condition: i.e., computing an aggregate at the graph level. Examples of such algorithms include: ranking (e.g., PageRank, top-k ranking), clustering on graphs (e.g., semi-clustering), and graph processing (e.g., neighborhood estimation). PREDIcT provides a set of *default rules* for choosing the transformations that work for a representative class of algorithms. At the same time, users can plug in their own set of transformations based on domain knowledge, if the semantics of the algorithm are not already captured by the default rules. Considering that a representative set of iterative, machine learning algorithms are executed *repetitively* on different input datasets [20, 44, 52], and that the space of possible algorithms is not prohibitive,

deriving such a set of customized transformations is also practical and worthwhile.

As Figure 3.1 shows, after key input features (including iterations) are profiled during the sample run and extrapolated to the scale of the complete dataset, a cost model is required for translating key input features into runtime estimates. For this purpose, PREDIcT introduces a framework for building customizable cost models for network intensive iterative algorithms executing using the Bulk Synchronous Parallel (BSP) [73] execution model, in particular the Apache Giraph implementation[1]. Our framework identifies a set of key input features that are effective for network intensive algorithms, it includes them into a *pool of features*, and then uses a model fitting approach (i.e., multivariate linear regression) and a feature selection mechanism for building the cost model. The cost model is trained on the set of input features profiled during the sample run, and additionally, on the set of input features of prior actual runs of the algorithm on *different* input datasets (if such runs exist). Such historical runs are typically available for analytical applications that are executed repetitively over newly arriving data sets. Examples include: ranking, clustering, social media analytics.

3.1.2 Contributions

To the best of our knowledge, PREDIcT is the first approach that targets to predict the runtime of a class of iterative algorithms executing on large-scale distributed infrastructures. Although sampling techniques have been used before in the context of graph analysis (e.g., [30, 47]), or DBMS (e.g., [17]), this is the first approach that proposes the *transform function* for maintaining invariants among the sample run and the actual run in the context of iterative algorithms and demonstrates its practical applicability for prediction. We note that the methodology we propose for estimating key input features is conceptually not tied to Giraph, and hence, could be used as a reference for other execution models operating on graph structures such as GraphLab [52] or Grace [78]. To this end identifying the key input features that significantly affect the runtime performance of these engines is required. For some iterative algorithms (that operate on graphs) our approach for *estimating iterations* can be applied even to non-BSP frameworks like Spark [84] and Mahout[2].

In this chapter we make the following contributions:

- We develop PREDIcT, an experimental methodology for predicting the runtime of a class of network intensive iterative algorithms. PREDIcT was designed to predict not only the number of iterations, but also the key input features of each iteration, which makes it applicable for algorithms with very different runtime patterns among subsequent iterations.
- We propose a framework for building customized cost models for iterative algorithms executing on top of Giraph. Although the complete set of key features and the cost

[1] http://giraph.apache.org
[2] http://mahout.apache.org/

model per se will vary from one BSP implementation to another (in a similar fashion as DBMS cost models vary from one DBMS vendor to another), our proposed methodology is generic. Hence, it can be used as a reference when building similar cost models on alternative BSP implementations.

- We evaluate PREDIcT on a representative set of algorithms using real datasets, showing PREDIcT's practicality over analytical upper bounds. For a 10% sample, the relative errors for estimating key input features range in between 5%-20%, while the errors for estimating the runtime range in between 10%-30%, including algorithms with up to 100x runtime variability among consecutive iterations.

3.2 The BSP Processing Model

Any algorithm executed on top of BSP is inherently iterative: It runs in a succession of *supersteps* (i.e., iterations) until a termination condition is satisfied. Each superstep is composed of three phases: i) concurrent computation, ii) communication, and iii) synchronization. In the first phase, each worker performs computation on the data stored in the local memory. In the second phase, the workers exchange data among themselves over the network. In the last phase, all workers synchronize at a barrier to ensure that all workers have completed. Subsequently, a new superstep is started unless a termination condition is satisfied.

In the context of graph processing, algorithms are parallelized using a *vertex centric* model: Each vertex of the input graph has associated customized data structures for maintaining state information and a user defined compute function for implementing the semantics of the algorithm. Intermediate results are sent to destination vertices using a messaging interface. Any vertex can inspect the state of its neighbors from the previous iteration, and can communicate with any other vertices of the graph based on their identifiers. Messages sent in one superstep are received by the targeted vertices in the subsequent superstep. Note that not all the vertices are active (i.e., executing the compute function) in all supersteps. A vertex that has finished its local computation can vote to halt (i.e., switch to the inactive mode). An inactive vertex can however be re-activated by a designated message received during any of the following supersteps. The algorithm completes when all active vertices vote to halt.

In Apache Giraph the BSP processing model is implemented as a master-slave infrastructure, with one master and multiple workers (or slaves). The master is in charge of partitioning the input data according to a partitioning strategy, allocating partitions to workers and coordinating the execution of each superstep (i.e., synchronization among workers). The workers are in charge of executing the compute function for every vertex of its allocated partition(s) and sending out messages to destination vertices. Each worker owns a pool of threads which are triggered to send out messages whenever the size of message buffers goes beyond a certain specified value. The worker with the largest amount of processing work is on the critical path, and hence determines the runtime of a superstep.

The runtime of an iterative algorithm executed in Giraph can be broken down into multiple phases: the *setup phase*, the *read phase*, the *supersteps phase* and the *write phase*. In the setup phase, the master setups the workers and allocates them partitions of the input graph based on a partitioning strategy; in the read phase, each worker reads its share of the input graph from the Hadoop file system (i.e., HDFS) into the memory; during the supersteps phase, the actual algorithm is executed, while in the write phase, the output graph is written back to HDFS. The supersteps phase includes the runtime of n supersteps (until the termination condition gets satisfied), and hence, it is the most challenging to predict from all the other phases.

3.3 Modeling Assumptions

In our proposed prediction methodology we make the following assumptions:

- All the iterative algorithms we analyze in this chapter are guaranteed to converge.
- Input datasets are graphs, and are amenable to sampling; the sample graph maintains its key properties similar or proportional with those of the original graph.
- Both the *sample run* and the *actual run* use the same execution framework (i.e., Giraph) and system configuration parameters.
- All the worker nodes have uniform resource allocations, hence processing costs among different workers are similar.
- The dominating part of the runtime of the algorithms is networking: i.e., sending/receiving messages from other vertices.

Such assumptions hold for a class of algorithms implemented on top of BSP which are dominated by network processing costs: Some of them have very short per vertex computation (e.g., PageRank), while some others have larger per vertex computation cost which is largely proportional with the size and the number of messages received (sent) from (to) the neighboring nodes (e.g., semi-clustering [55], top-k ranking [45]).

3.4 PREDIcT's Transformations

The *sample run* is the preliminary phase of the prediction approach that executes the algorithm on the sample dataset. As explained in section 3.1.1, two sets of transformations characterize the execution of the algorithm during the sample run: the sampling technique adopted and the transform function. Once the set of transformations is determined, the algorithm is executed on the sample. During the sample run, per iteration key input features are profiled and used later in the prediction phase as a basis for estimating the corresponding features of the actual run.

3.4.1 Sampling Techniques

The sampling technique has to maintain key properties of the sample graph similar or proportional with those of the original graph: Examples of such properties include in/out degree proportionality, effective diameter, clustering coefficient. Hence, we adopt similar sampling techniques with those proposed by Leskovec et al. [47], which show that such graph properties on the sample can be maintained *similar* to those on the actual graph.

Random Jump: We choose Random Jump (RJ) from the set of sampling methods proposed in [47], because it is the sampling method that has no risks of getting stuck in an isolated region of the graph, while maintaining comparable results for all the key properties of the graph with Random Walk and Forest Fire (D-statistic scores, that measure how closely the properties of the sample fit the properties of the graph, are shown in Table 1 of [47]). RJ picks a starting seed vertex uniformly at random from all the input vertices. Then, at each sampling step an outgoing edge of the current vertex is picked uniformly at random and the current vertex is updated with the destination vertex of the picked edge. With a probability p the current walk is ended and a new random walk is started from a new seed vertex chosen at random. The process continues until the number of vertices picked satisfies the sampling ratio. Such a sampling technique has the property of maintaining connectivity within a walk. Random jump achieves connectivity among multiple walks by returning to already visited vertices on different edges. Returning to already visited nodes also improves the probability of preserving the in/out node degree proportionality.

Biased Random Jump: Based on the observation that convergence of multiple iterative algorithms we analyze is inherently dictated by high out-degree vertices (e.g., PageRank, top-k ranking, semi-clustering), we propose *Biased Random Jump (BRJ)*, a variation of Random Jump. BRJ is biased towards high out degree vertices: Compared with RJ which picks the seed vertices uniformly at random from the entire set of graph vertices, the seed vertices of BRJ are comprised of the *top k vertices with the highest out-degree.* Then, for each new random walk performed a starting vertex is picked uniformly at random from the set of seed vertices. The intuition of BRJ is to prioritize sampling towards the "core of the network", that include vertices with high out degrees. Biased random jump trades-off sampling uniformity for improved connectivity: By starting random walks from highly connected nodes (i.e., hub nodes), BRJ has a higher probability of maintaining connectivity among sampled walks than RJ, where jumps to any arbitrary nodes are possible. We empirically find that BRJ has higher accuracy than RJ in maintaining key properties of the graph (such as connectivity), especially at small sampling ratios (the sampling ratio proposed for RJ in [47] is 25%). Hence, BRJ is used as our default sampling mechanism.

3.4.2 Transform Function

The transform function T is formally described by two pairs of adjustments: $T = (Conf_S => Conf_G, Conv_S => Conv_G)$, where $Conf_S => Conf_G$ denotes configuration parameter map-

pings, while $Conv_S => Conv_G$ denotes convergence parameter mappings. For instance, the transformation $T = (d_S = d_G, \tau_S = \tau_G \times \frac{1}{sr})$ for PageRank algorithm denotes: Maintain the damping factor value on the sample run equal with the corresponding value of the actual run, and scale the convergence threshold. Table 3.1 summarizes the notations used for representing the transform function. While the transform function requires domain knowledge about the algorithm semantics, we provide a default rule which works for a set of representative algorithms and can be used as a reference when choosing alternative transformations. For the case that the convergence threshold is tuned to size of the input dataset (i.e., convergence is determined by an *absolute* aggregated value, as for PageRank): $T_{default} = (ID_{Conf}, \tau_S = \tau_G \times \frac{1}{sr})$, while for the case that convergence threshold is not tuned to the size of the input dataset (i.e., convergence is determined by a *relative* aggregated value or a ratio that is maintained constant on a proportionally smaller dataset, as for top-k ranking): $T_{default} = (ID_{Conf}, \tau_S = \tau_G)$. Specifically, we maintain all the configuration parameters of the algorithm during the sample run (identity function over the configuration space) and we scale or maintain the convergence threshold for the sample run.

Notation	Description
T	Transformation
d	Damping factor for PageRank algorithm, by default $d = 0.85$
τ	Convergence threshold
sr	Sampling ratio
ID	Identity function
$Conf$	Configuration parameters
$Conv$	Convergence parameters

Table 3.1 – Notations used for representing the transform function.

3.5 Model Fitting and Prediction

3.5.1 Key Input Features

We identify the key input features for the Giraph execution model based on a mix of domain knowledge and experimentation. Table 3.2 shows the set of key input features we identified for modeling the runtime of network intensive iterative algorithms. The number of iterations is not extrapolated, since the transform function attempts to preserve the number of iterations during the sample run. In order to understand the selection of key input features, consider Figure 3.3 that illustrates the execution phases of an arbitrary iteration of an iterative algorithm that uses BSP. Each worker executes three phases: compute, messaging, and synchronization, as explained in section 3.2.

Compute phase: In this phase the user defined function that implements the semantics of the iterative algorithm is executed for every vertex of the input graph. For a large category

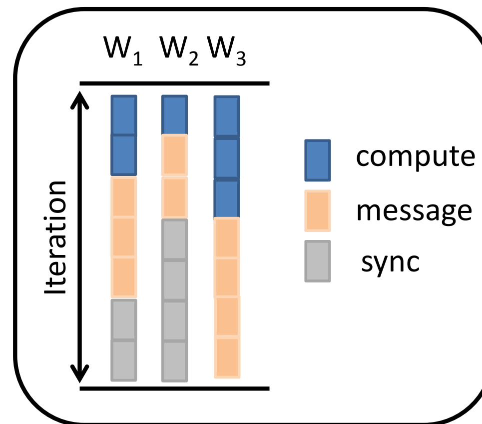


Figure 3.3 – BSP execution phases of an arbitrary iteration.

of network intensive algorithms the cost of local, per vertex computation (executing the semantics of the algorithm) can be approximated by a constant cost factor, while the cost of initiating messages to neighboring nodes is proportional with the number of messages each vertex sends. Hence, the compute time of each worker (which has multiple vertices allocated to it) is proportional with the total number of active vertices (i.e., executing actual work), and the number of messages each worker sends.

Messaging phase: During this phase, messages are sent over the network and added into the memory of the destination nodes. Some BSP implementations can spill messages to disk. Hence, the runtime of this phase is proportional with the number of messages, their sizes, and the number and sizes of messages spilled to disk (if spilling occurs).

Synchronization phase: The synchronization time of a worker w.r.t. the worker on the critical path (the slowest worker) depends on the partitioning scheme adopted, which in turn may result in skewed work assignment among workers. Instead of trying to model the synchronization time among workers explicitly, we model it implicitly by identifying the worker on the critical path, which has close to zero synchronization time.

Name	Description	Extrapolation
ActVert	Number of active vertices	yes
TotVert	Number of total vertices	yes
LocMsg	Number of local messages	yes
RemMsg	Number of remote messages	yes
LocMsgSize	Size of local messages	yes
RemMsgSize	Size of remote messages	yes
AvgMsgSize	Average message size	no
NumIter	Number of iterations	no

Table 3.2 – Key Input Features

While the set of features illustrated in Table 3.2 is effective for network intensive algorithms, they should not be interpreted as complete. Given the generality of selecting input features into the cost model, our proposed methodology can be extended to include additional key input features in the pool of *candidate input features*. For instance, counters corresponding to spilling messages to disk during the messaging phase shall be also considered if spilling occurs. Giraph currently does not support spilling of messages to disk, hence such features were not required in our experiments.

3.5.2 Customizable Cost Model

Based on the processing model breakdown presented in Section 3.5.1, we propose a cost modeling technique for network intensive algorithms that uses *multivariate linear regression* to fit a set of key input features into per iteration runtime. Formally, given a set of input features $X_1, ..., X_k$, and one output feature Y (i.e., per iteration runtime), the model has the functional form: $f(X_1, ..., X_k) = c_1 X_1 + c_2 X_2 + ... + c_k X_k + r$ where c_i are the coefficients and r is the residual value. A modeling approach based on a *fixed functional form* was chosen for several reasons: i) For network intensive algorithms, each phase of the Giraph BSP execution model except the synchronization phase can be approximated by a fixed functional form (multivariate linear regression). The synchronization phase is modeled implicitly, as explained in section 3.5.1. ii) A fixed functional form can be used for prediction on input feature ranges that are outside of the training boundaries (e.g., train on sample run, test on actual run). In fact the coefficients of the model can be interpreted as the "cost values" corresponding to each input feature.

We use the set of features presented in Table 3.2 as *candidates* in the cost model. Customization of the cost model for a given iterative algorithm is done by selecting the actual input features that have a high impact on the response variable Y, and yield a good fitting coefficient for the resulting model. In particular, selecting the actual key features from the above pool of features is based on an *sequential forward selection* mechanism [35] that selects the features that yield the best prediction accuracy on the training data. The forward selection mechanism is a greedy algorithm to select input features from a pool of candidate features. Concretely, the selection procedure adds input features into the model one by one, by always picking the feature that reduces the cross validation error of the current model the most. The process continues until there are no more features to select from or until there is no further model improvement (i.e., the addition of any other feature does not reduce the error any further).

Cost Model Extensions: For the cases where the compute phase is not linearly proportional with the number of active vertices, and the number and size of messages, our proposed cost model is extensible as follows: i) The compute phase and messaging phase are separately profiled; ii) A similar approach as above is used to model the messaging phase; iii) A non linear approach is used to model the compute function (e.g., decision trees). For this purpose, MART scale [51] can be used, as it was designed to be accurate even on key input features outside of

the training boundaries.

Modeling the Critical Path: In the BSP processing model, the runtime of one iteration is given by the worker on the critical path (i.e., the slowest worker). In a homogeneous environment where each worker has the same share of system resources, the worker on the critical path is the worker processing the largest part of the input graph. For a vertex centric partitioning scheme, non-uniform allocations may exist if some vertices are better connected than others, which in turn results into larger processing requirements. This observation holds for network intensive algorithms, where the number of outgoing edges determine the messaging requirements of the vertex, and in turn, the runtime. We adopt the following methodology for finding the worker on the critical path: For a given partitioning scheme of vertices to partitions, and a mapping of partitions to workers, the total number of outbound edges for each worker is computed. The worker with the largest number of outbound edges is considered to be on the critical path. Such a method for finding the slowest worker can be piggybacked in the initialization phase of the algorithm, in the *read phase*, and can be exploited for prediction just before the algorithm starts its effective execution in the *superstep phase*.

Training Methodology: For training the cost model we use both sample runs and measurements of previous runs of the algorithm that were given different datasets as input (if such runs exist). Such a training scenario is applicable for the class of algorithms we address in this chapter, as the underlying cost functions corresponding to each input feature: i.e., cost of sending/receiving messages, or the cost of executing the compute function, are similar when executing the same algorithm on different input datasets. Hence, once a cost model is built, it can be reused for predicting the runtime of the algorithm on different input datasets.

The cost model is trained at the granularity of *iterations*: Key input features are profiled and maintained on a per-worker basis for each iteration of the algorithm. Specifically, the code path of each BSP worker was instrumented with counters for all the input features potentially required in the cost model. Then, all counters are used to train the model.

3.5.3 Prediction

There are two phases in the prediction process: i) extrapolation of key input features profiled during the sample run, and ii) estimating runtime by plugging in extrapolated features into a cost model.

Extrapolator: As shown in Figure 3.1, in the first prediction phase an extrapolator is used to scale-up input features profiled during the sample run. The input metrics that are used in the extrapolation phase are the number of edges and the number of vertices of the sample graph S, and the corresponding number of edges and vertices of the complete graph G. We use two extrapolation factors: i) For features that primarily depend on the number of vertices (e.g., ActVert), we extrapolate with a scaling factor on vertices: i.e., $e_V = \frac{|V_G|}{|V_S|}$. ii) For features that depend both on the number of input nodes and edges (e.g., message counts depend on

how many outbound edges a vertex has) we extrapolate with a scaling factor on edges: i.e., $e_E = \frac{|E_G|}{|E_S|}$. Note that not all key input features require extrapolation: e.g., number of iterations is preserved during the sample run. Extrapolation of input features is done at the granularity of *iterations*: i.e., the input features of an arbitrary iteration of the sample run are extrapolated and then used to predict the runtime of the corresponding iteration of the actual run.

Estimation: In the second phase extrapolated features are plugged into the cost model to compute estimated runtime. The cost model is invoked multiple times, on extrapolated input features corresponding to each iteration of the sample run. Hence, the number of iterations is used *implicitly* rather than explicitly in prediction.

3.6 End-to-end Use Cases

3.6.1 PageRank

PageRank is an iterative algorithm proposed in the context of the Web graph, where vertices are web pages and edges are references from one page to the other. Conceptually, PageRank associates to each vertex a rank value proportional with the number of inbound links from the other vertices, and their corresponding PageRank values. In order to understand how is the rank transfer between vertices affecting the number of iterations, we introduce the formula used for computing PageRank [59]:

$$PR(p_i)_{it} = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)_{it-1}}{L(p_j)} \tag{3.1}$$

where $PR(p_i)$ is the PageRank of the vertex p_i, N is the total number of vertices, d is the damping factor (typically set to 0.85), $p_1, p_2, ..., p_N$ are the vertices for which the rank is computed, $M(p_i)$ is the set of vertices that link to p_i, and $L(p_j)$ is the number of outbound edges of vertex p_j. The rank value of each vertex is initialized to $1/N$.

Convergence: PageRank algorithm converges when the average delta change of PageRank value at the graph level goes below a user defined threshold τ. Formally, the *delta change of PageRank* for an arbitrary vertex p_i, corresponding to an iteration it, is defined as: $\delta_{i,it} = |PR(p_i)_{it} - PR(p_i)_{it-1}|$, and the *average delta change of PageRank* on graph G is: $\overline{\Delta}_{G,it} = \frac{1}{N}\sum_i \delta_{i,it}$. For simplicity, $\overline{\Delta}_{G,it} = \overline{\Delta}_G$ when referring to any arbitrary iteration. It can be shown that for a directed acyclic graph the maximum number of iterations required for PageRank to converge to $\overline{\Delta}_G = 0$ is the diameter of the graph D plus one. For real graphs, however, the DAG assumption does not hold as cycles between vertices are typical. Therefore, an additional number of iterations is required for the algorithm to converge to a convergence threshold $\tau > 0$.

Sampling Requirements: In order to take a representative sample that can maintain the number of iterations of the actual run similar with that of the sample run we make the following observations: i) Maintaining connectivity is crucial in propagating the PageRank transfer

among graph vertices. Therefore, the sampling technique should maintain the connectivity among sampled vertices (i.e., the sample should not degenerate into multiple isolated subgraphs). ii) The PageRank delta change per vertex depends on the number of incoming and outgoing edges. The sample should ideally maintain the in/out node degree ratio similar with the corresponding ratio on the original graph. iii) The diameter of the graph determines the number of iterations required to propagate the PageRank transfer among vertices located at the graph boundaries. Hence, ideally the diameter of the sample graph shall be similar with the diameter of the original graph. In practice, maintaining the *effective diameter* of the graph (as introduced in [44]) is more feasible, i.e., the shortest distance in which 90% of all connected pairs of nodes can reach each other.

Transform Function: Consider the example introduced in Figure 3.2: It can be shown that for any arbitrary iteration, the average delta change of PageRank on graph S3 can be maintained the same with the average delta change of PageRank on graph G (i.e., $\overline{\Delta}_{S3} = \overline{\Delta}_{G}$) by the following transform function: $T = (ID_{Conf}, \tau_S = \tau_G \times \frac{1}{sr})$, where $Conf = \{d\}$, and sr is the sampling ratio.

For a better understanding of transformation T, we compute the PageRank of vertex 5 on graph G, and then on graph S3, for the first iteration of the algorithm. On graph G, the PageRank of vertex 5 is given by: $(1-d)/N + 2d/4N$=$(2-d)/2N$, while on graph S3: $(1-d)/(N/2) + d/(2*(N/2)) = (2-d)/N$. We observe that the PageRank value of node 5 on the sample S3 is twice of the corresponding PageRank value on graph G (equal with the inverse of the sampling ratio), as the sample maintains the structure of the original graph (i.e., in/out node degree ratio and diameter). Similarly, it can be shown that the average delta change of PageRank on the sample graph S3 is twice of the corresponding average delta change of PageRank on graph G (i.e., $\overline{\Delta}_{S3} = \overline{\Delta}_{G} \times 2 = \overline{\Delta}_{G} \times \frac{1}{sr}$). Hence, by applying the transform function T for the sample run, invariants are maintained for the number of iterations. In real graphs such symmetric structures cannot be assumed. Still, we can use such transformations as a basis for an *heuristic approach* that shows good results in practice.

3.6.2 Semi-clustering

Semi-clustering is an iterative algorithm popular in social networks as it aims to find groups of people who interact frequently with each other and less frequently with others. A particularity of semi-clustering as compared with the other clustering algorithms is that a vertex can belong to more than one cluster. We adopt the parallel semi-clustering algorithm as described in [55]. The input is an undirected weighted graph while the output is an undirected graph where each vertex holds a maximum number of C_{max} semi-clusters it belongs to. Each semi-cluster has associated a score value:

$$S_c = \frac{I_c - f_B * B_c}{V_c(V_c - 1)/2} \tag{3.2}$$

where I_c is the sum of the weights of all internal edges of the semi-cluster, B_c is the sum of the weights of all boundary edges, f_B is the boundary edge factor (i.e., $0 < f_B < 1$, a user defined parameter) which penalizes the total score value, and V_c is the number of vertices in the semi-cluster. As it can be noticed, the score is normalized to the number of edges in a clique of size V_c such that large semi-clusters are not favored. The maximum number of vertices in a semi-cluster is bounded to a user settable parameter V_{max}.

Convergence: The algorithm runs in iterations: In the first iteration, each vertex adds itself to a semi-cluster of size one which is then sent to all of its neighbors. In the following iterations: i) Each vertex V iterates over the semi-clusters sent to it in the previous iteration. If a semi-cluster sc does not contain vertex V and $V_c < V_{max}$, then V is added to sc to form sc'. ii) The semi-clusters sc_k that were sent to V in the previous iteration together with the newly formed semi-clusters sc'_k are sorted by score and the best S_{max} semi-clusters (i.e., with the highest score) are sent out to V's neighbors. iii) Vertex V updates its list of C_{max} best semi-clusters with the newly received / formed semi-clusters (i.e., the semi-clusters from the set: sc_k, sc'_k) that contain V. The algorithm converges when there are no further updates to the lists of best semi-clusters, that are maintained at each vertex. As such a stopping condition requires a large number of iterations an alternative stopping condition that considers the proportion of semi-cluster updates is more practical. More precisely: $\frac{updatedClusters}{totalClusters} < \tau$, where $updatedClusters$ represents the number of semi-clusters updated during the current iteration, while $totalClusters$ represents the total number of semi-clusters in the graph.After the algorithm converges, the lists of best semi-clusters are aggregated into a global list of best semi-clusters.

Sampling Requirements: Semi-clustering has similar sampling requirements as PageRank: In particular, the sampling mechanism should maintain the connectivity among vertices (to avoid isolated sub-graphs) and the in/out node degrees proportionality, such that a proportionally smaller number of semi-clusters are sent along the edges of the sample graph in each iteration of the sample run.

Transform Function: For semi-clustering the convergence threshold is not tuned to the size of the dataset as a *ratio* of cluster updates decides convergence. Hence, we use the transform function: $T = (ID_{Conf}, \tau_S = \tau_G)$, with $Conf = \{f_B, V_{max}, C_{max}, S_{max}\}$, and sr is the sampling ratio. Intuitively, the total number of cluster updates on a sample that preserves the structure of the original graph is proportionally smaller than the total number of cluster updates on the complete graph. As for PageRank algorithm, such transformations assume perfect structural symmetry of the sample w.r.t. the original graph. Therefore, we adopt it as an *heuristic*, which shows good results in practice.

3.6.3 Top-k Ranking

Top-k ranking for PageRank [45] finds the top k highest ranks reachable to a vertex. Top-k ranking operates on output generated by PageRank and it proceeds as follows: In the first

iteration, each vertex sends its rank to the direct neighbors. In the following iterations, each vertex receives a list of ranks from all the neighboring nodes, it updates its local list of top-k ranks, and then it sends the updated list of ranks to the direct neighbors. A node that does not perform any update to its list of ranks in one iteration does not send any messages to the neighbors. As the number of messages and the message byte counts sent in each iteration is variable (depending on the number of ranks stored per node, and whether the node performed any updates), the runtime of consecutive iterations is not constant.

Convergence: Top-k ranking it is executed iteratively until a fixed point is reached [45], or alternatively, until the total number of vertices executing updates goes below a user defined threshold: i.e., $\frac{activeVertices}{totalVertices} < \tau$.

Sampling Requirements: There are two main requirements: i) Maintaining connectivity, in/out node degrees and effective diameter among sampled vertices as for PageRank algorithm, and ii) Maintaining the relative ordering of ranks for sampled vertices. Top-k ranking is executed on output generated by PageRank. Assuming an input sample that satisfies the sampling requirements of PageRank, the resulting output generated by PageRank preserves the connectivity and the relative order of rank values. Consider Figure 3.2, the rank of any node on S_3 is twice the rank of the corresponding node on G.

Transform function: We observe that the convergence condition is not tuned to the size of the dataset as it uses a ratio of updates to decide convergence. For a sample that satisfies the sampling requirements, the ratio of rank updates on the sample is maintained in pair with the ratio of rank updates on the complete graph, hence, unlike PageRank algorithm, no scaling is required: $T = (ID_{Conf}, ID_{Conv})$, where $Conf = \{topK\}$, $Conv = \{\tau_S = \tau_G\}$.

3.6.4 Neighborhood Estimation

Estimating the number of vertices reachable from a vertex v within h hops or shortly *the neighborhood of v* is used in social applications today. LinkedIn for instance provides information on the number of professionals reachable within h hops from any given user. We implement neighborhood estimation for *all* the vertices of an input graph using an iterative, probabilistic algorithm similar with estimating effective diameters and radii in large graphs [44]: Each vertex v of the graph stores the number of neighbors reachable from v in h hops as a set of k probabilistic Flajolet-Martin bitstrings $b_k(h, v)$ [27]. In the first iteration, each vertex is initialized with a set of k random bitstrings. After initialization, each vertex sends its own bitstrings to the neighboring vertices. In the following iterations, each vertex updates its bitstrings using a bitwise OR operator among its bitstrings and the corresponding bitstrings received from the neighboring nodes. Only if the bitstrings are updated during the current iteration, the vertex sends again its updated bitstrings to the neighboring nodes. The algorithm has variable resource requirements per iteration as the number of messages sent, and the number of active vertices of each iteration depend on the actual number of vertices updating their bitstrings.

Convergence: Unlike other algorithms, neighborhood estimation is executed until a fixed point is reached. The challenge stands in estimating per iteration key features such as active vertices and message byte counts as they vary from one iteration to the next. The neighborhood of a vertex v after h iterations is computed from the k Flajolet-Martin bitstrings by: $N(h,v) = \frac{1}{0.77351}2^{\frac{1}{k}\sum_{l=1}^{k} b_l(i)}$, where $b_l(i)$ is the position of leftmost 0 bit of the lth bitstring of node v, and k is the number of bitstrings stored at each node (a constant, typically 32 [44]).

Sampling Requirements: Maintaining connectivity among nodes, and effective diameter are primarily required. Preserving distances among sampled vertices contributes in propagating the bitstrings updates of the sample run at the same pace with those of the actual run.

Transform function: For a sample that satisfies the sampling requirements, the neighborhood function on the sample grows with the same rate as on the original graph: Consider vertex 1 in Figure 3.2: The number of vertices reachable within two hops on G, is twice the number of vertices reachable within two hops on S_3. Hence, the processing requirements during the sample run can be maintained proportional with the processing requirements of the actual run using a sample that satisfies the sampling requirements. Considering that the neighborhood is estimated probabilistically starting from a set of k bitstrings that each vertex keeps updating, the only transformation required is to maintain the same set of initial bitstrings on the sampled nodes as on the complete graph. In particular, $T = (ID_{Conf}, ID_{Conv})$, where $Conf = \{K, seed_i\}$, $Conv = \{\}$. K is the number of bitstrings (e.g., 32), and $seed_i$ is the seed used in generating the bitstrings of each node (each vertex sets its seed as the vertex id, such that the same initial bitstrings are generated for both the sample and the actual runs).

3.6.5 Labeling Connected Components

Labeling connected components is an algorithm that finds the number of connected components in a graph by mapping each vertex to a connected component identifier. The algorithm can be implemented in an iterative fashion as follows: Initially, the connected component value (i.e., CCV) of each vertex is initialized with the vertex identifier. In the first iteration, each vertex inspects the CCV of the neighboring vertices. If any of these values is smaller than the current CCV, the vertex changes its CCV with that one and broadcasts a message with the updated value to all of its neighboring vertices. In the following iterations, each vertex checks all the messages received from its neighbors. If any message includes a CCV smaller than the current identifier, the vertex changes its value and broadcasts a message with the updated CCV to all of its neighbors. The algorithm continues in a similar fashion until no new messages are being sent. A main characteristic difference between connected components and the previous algorithms is that the processing requirements of consecutive iterations may vary widely. Typically, a few long iterations are followed by multiple very short iterations.

Convergence: The total number of iterations required for running the connected components algorithm is bounded by the diameter of the graph [44].

Sampling Requirements: All of the three sampling requirements of the PageRank algorithm are equally important for connected components. We emphasize that vertices with a high-out degree (i.e., hub nodes) have a high impact on the convergence speed of the connected components algorithm. As such nodes are highly connected, their corresponding connected component identifier can be propagated towards other regions of the graph in a few steps. Hence, starting the sampling process from such nodes would be beneficial.

Transform function: Prior research showed that for uniform graphs, sampling mechanisms based on random walks typically maintain the diameter of the sample similar with the one of the complete graph [47]. For example, the sample graph S3 presented in Figure 3.2 has the same diameter with graph G. Therefore, an explicit transform function is not required as for the other algorithms. In particular, $T = (ID_{Conf}, ID_{Conv})$, where $Conf = \{\}$, $Conv = \{\}$.

3.7 Limitations

PREDICT was designed for a class of iterative algorithms that operate on homogeneous graph structures and use a *global* convergence condition: e.g., computing an average or a ratio at the graph level. Algorithms for which convergence is highly influenced by the *local state* of any arbitrary vertex of the graph are not amenable to sampling, and hence, PREDIcT methodology cannot be used for these cases. Similarly, PREDIcT cannot be used on degenerate graph structures where maintaining key graph properties in a sample graph is not possible. Similar to traditional DBMS techniques, we cannot use a sample of a dataset to estimate outliers, but we can use it to produce average values. We note that the sampling requirements in our case are more relaxed, as we do not use sampling to approximate results. Instead, sampling is used as a mechanism to approximate the processing characteristics of the actual run. Examples of algorithms where our methodology is not applicable: collaborative filtering (heterogeneous graphs with two entity types: e.g., users and movies) or simulating advertisements in social networks [45] (the decision to further propagate an advertisement depends on the *local* interest of the node receiving the advertisement (i.e., his interest list). Examples of datasets where our methodology is not applicable: e.g., degenerated, non uniform graph structures, e.g., lists.

3.8 Experimental Evaluation

3.8.1 Setup and Methodology

Experimental Setup: Experiments were performed on a cluster of 10 nodes, where each node had two six-core CPUs Intel X5660 @ 2.80GHz, 48 GB RAM and 1 Gbps network bandwidth. All experiments were run on top of Giraph 0.1.0, a library that implements the BSP model on top of Hadoop. We use Hadoop 1.0.3 as the underlying MapReduce framework. Unless specified otherwise each node is set with a maximum capacity of three mappers, each mapper having allocated 15GB of memory. Hence, our Giraph setup has a total of 30 tasks (i.e., 29 workers

and one master).

Datasets: Four real datasets are used for evaluating PREDIcT: Two of them are web graphs: Wikipedia, and UK 2002, and the remaining two are social graphs: LiveJournal and Twitter. The Wikipedia dataset is a subset of the online encyclopedia including the links among all English page articles as of 2010, UK 2002 is the web graph of the .uk domain as crawled by UbiCrawler[3] in 2002, LiveJournal graph models the friendship relationship among an online community of users[4], while Twitter graph[5] models the following relationships among users as crawled in 2009 [15]. Table 3.3 illustrates the characteristics of each dataset.

All datasets are directed graphs. For algorithms operating on undirected graphs we transform directed graphs into the corresponding undirected graphs. In Giraph, which inherently supports only directed graphs, a reverse edge is added to each edge.

Name	Prefix	# Nodes	# Edges	Size [GB]
LiveJournal	LJ	4,847,571	68,993,777	1
Wikipedia	Wiki	11,712,323	97,652,232	1.4
Twitter	TW	40,103,281	1,468,365,182	25
UK-2002	UK	18,520,486	298,113,762	4.7

Table 3.3 – Graph Datasets

Algorithms: We evaluate PREDIcT on a set of representative algorithms for ranking (i.e., PageRank, top-k ranking), clustering (i.e., semi-clustering), and graph processing (i.e., labeling connected components, and neighborhood estimation).

Metrics of Interest: For validating our methodology, we compute standard error metrics used in statistics that show the accuracy of the fitted model on the training data. In particular, we consider: the coefficient of determination (i.e., R^2), and the signed relative error (i.e., negative errors correspond to under-predictions, while positive errors correspond to over-predictions).

Sources of Error: There are two sources of error when providing end-to-end runtime estimates: i) Misestimating key input features; ii) Misestimating cost factors used in the cost model. Depending on the the error sign of the two types of estimates, the aggregated errors can either accumulate or reduce the overall error. Hence, we first provide results on estimating key input features, then, we provide end-to-end runtime results.

Memory Limits: The memory resources of our deployment are almost fully utilized when executing the algorithms on the largest datasets: i.e., Twitter and UK. In Apache Giraph, in addition to the input graph which is read and maintained into the memory, per vertex state and per vertex message buffers are also stored into the memory. Hence, the overall memory

[3]http://law.di.unimi.it/software.php/#ubicrawler
[4]Courtesy of Stanford Large Network Dataset Collection
[5]Courtesy of Max Planck Institute for Software Systems

requirements are much larger than the size of the dataset itself. For instance, executing semi-clustering (which sends a large number of large messages) on the UK dataset requires 90% of the full RAM capacity of our cluster, hence, the memory resources of our setup are almost fully utilized. As Giraph is currently lacking the capability of spilling messages to disk, we run out of memory when trying to run semi-clustering, top-k ranking, and neighborhood estimation on the Twitter dataset[6].

3.8.2 Estimating Key Input Features

In this section we report experimental results for estimating the number of iterations and per iteration key input features that have a high impact on predicting runtime, as summarized in Table 3.2. Runtime prediction results are presented in Section 3.8.4.

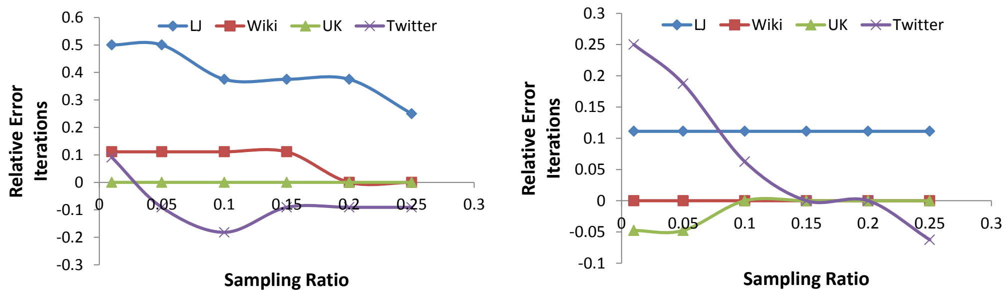


Figure 3.4 – The accuracy of predicting the number of iterations for PageRank for $\epsilon = 0.01$ (left) and for $\epsilon = 0.001$ (right).

PageRank: This set of experiments shows the accuracy of predicting the number of iterations for PageRank algorithm as the size of the sampling ratio increases from 0.01 to 0.25. The convergence threshold value is set as $\tau = 1/N \times \epsilon$, where N is the number of vertices in the graph, while ϵ is the convergence tolerance level, a sensitivity parameter varied between 0.01 and 0.001. Figure 3.4 shows the results for all datasets when BRJ is adopted as the underlying sampling scheme. Sensitivity analysis w.r.t. the sampling method is deferred to section 3.8.5. For a sampling ratio of 0.1, and a tolerance level of $\epsilon = 0.01$ the maximum mis-prediction for the web graphs and Twitter datasets is less than 20%. LiveJournal has 40% relative error for the same sampling ratio. For this dataset, our results on multiple algorithms are consistently showing that the sampling method adopted cannot capture a representative sample as for the other algorithms due to its underlying graph structure which is *not* scale-free[7]. Lower errors correspond to a tolerance level of $\epsilon = 0.001$, when PageRank converges in a larger number of iterations. The relative errors for all datasets are maintained bellow 10% including LiveJournal. This is a desired outcome for a prediction mechanism, as accurate predictions are typically more useful for long running algorithms.

[6] Similar observations w.r.t Giraph are presented in [26].

[7] We have analyzed the out-degree distribution of LJ and we observed that it is not following a power law. Similar observations are presented in the study of Leskovec et al. [49] or Gjoka et al. [30].

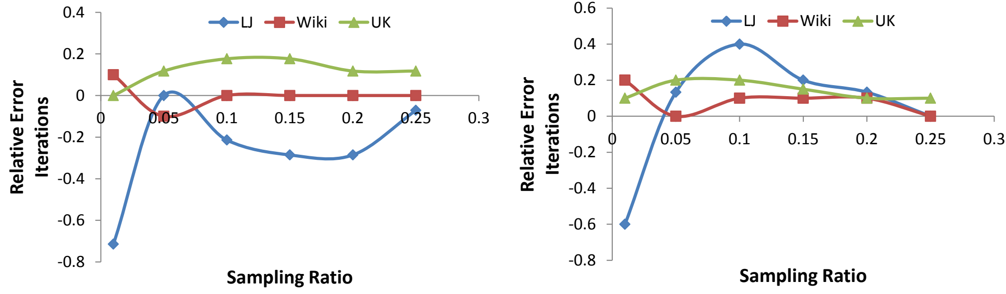


Figure 3.5 – The accuracy of predicting the number of iterations for semi-clustering for $\tau = 0.01$ (left) and for $\tau = 0.001$ (right).

Semi-clustering: In this section we analyze the accuracy of predicting iterations for semi-clustering. The base settings we use in evaluation are: $C_{max} = 1, S_{max} = 1, V_{max} = 10, f_B = 0.1, \tau = 0.001$. Figure 3.5 shows the accuracy results for all datasets but Twitter for two convergence ratios for $\tau = 0.01$, and $\tau = 0.001$. As explained in experimental methodology, as the memory footprint of semi-clustering algorithm on Twitter is much larger than the total memory capacity of our cluster we could not perform experiments on this dataset. For a sampling ratio of 0.1 the relative errors corresponding to all web graphs analyzed are below 20%. Again, LiveJournal dataset shows higher variability in its error trend due to its underlying graph structure which is less amenable to sampling.

We have performed sensitivity analysis w.r.t. S_{max} and V_{max} when running semi-clustering on LJ dataset, which has the highest relative error on the base settings. In particular, we analyzed two cases: i) increasing S_{max} from one to three, and ii) increasing V_{max} from ten to twenty. Compared with the base settings, for a sampling ratio of 0.1 (or larger) the relative errors were maintained in similar bounds for all sampling ratios.

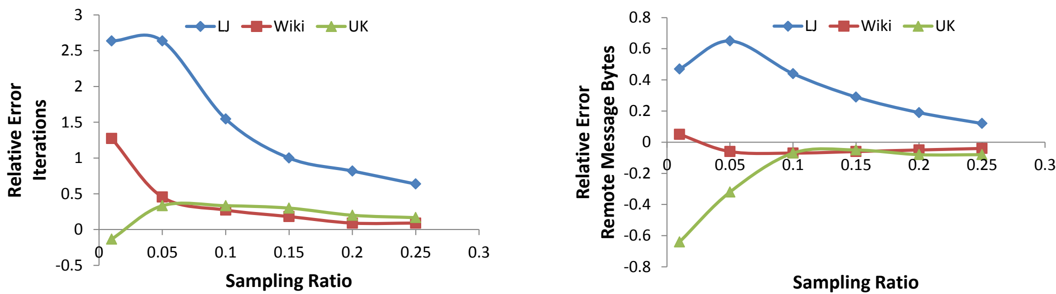


Figure 3.6 – Top-k ranking key input features estimation: a) Estimating iterations (left), b) Estimating remote message bytes (right).

Top-K Ranking: We analyze the accuracy of estimating iterations and the accuracy of estimating key input features (i.e., remote message bytes) in Figure 3.6. We execute sample runs on output generated by PageRank algorithm, and use a convergence threshold of $\tau = 0.001$. We

observe that the relative errors for estimating iterations are bellow 35% for all scale free graphs analyzed, while the errors for estimating remote message bytes are bellow 10%. Similarly to our experiments on PageRank and semi-clustering, higher errors are observed for LiveJournal dataset: for a sampling ratio of 0.1, the number of iterations are over-estimated by a factor of 1.5, while the message byte counts by 40%. An interesting observation for top-k ranking is that the accuracy in estimating the message byte counts is more important than the accuracy of estimating the number of iterations per se. That is because the runtime of consecutive iterations varies and is proportional with the number of message byte counts and the number of active vertices of each iteration (results on estimating runtime are shown later, in Figure 3.11).

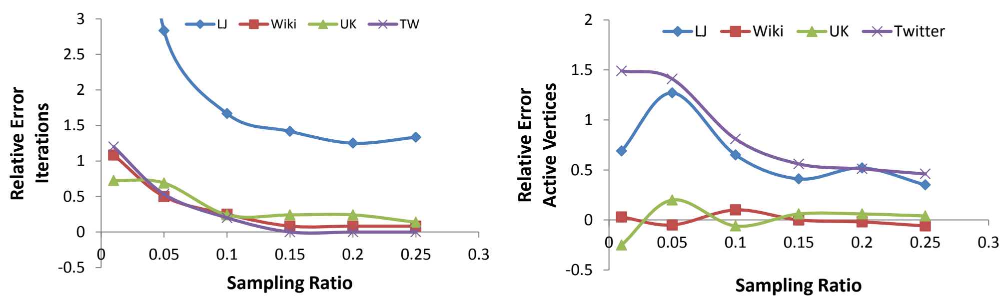


Figure 3.7 – Predicting key input features for connected components: a) number of iterations (left), b) active vertices (right).

Connected components: For this experiment each cluster node was set with a maximum capacity of six mappers, each mapper having allocated 7GB of memory, accounting for a total of 60 tasks (i.e., 59 workers and one master). Due to the large processing variability among subsequent iterations, the number of iterations per se is not sufficient for predicting the runtime of connected components algorithm. Hence, we present the accuracy of estimating active vertices in addition to estimating iterations. Figure 3.7 a) shows the accuracy results when estimating iterations. For a sampling ratio of 0.1, the relative errors for all datasets but LJ are bellow 25%. Figure 3.7 b) shows the estimated total number of active vertices required for the execution of the algorithm (summed up for *all* iterations). For a sampling ratio of 0.1, the relative error for both web graphs is less than 10%. The reason that LJ highly over-estimates the total number of active vertices for a sampling ratio of 0.1 is that it is not scale-free, hence, the sample cannot capture the structure of the original graph. For Twitter, on the other hand, the sample of 0.1 is too small to capture key input features with a better accuracy than 81% due to the density of the graph (i.e., a very large number of incident edges per node): The sampling ratio of 0.1 vertices corresponds to a ratio of *only* 0.002 in terms of edges. Higher sampling ratios improve the accuracy results: For a sampling ratio of 0.25 the error decreases to 46%.

Sensitivity analysis w.r.t. sampling is showing that a smaller number of seed vertices used in BRJ sampling ($k = 100$ instead of $k = 1\%$ of total vertices) improves the accuracy on Twitter

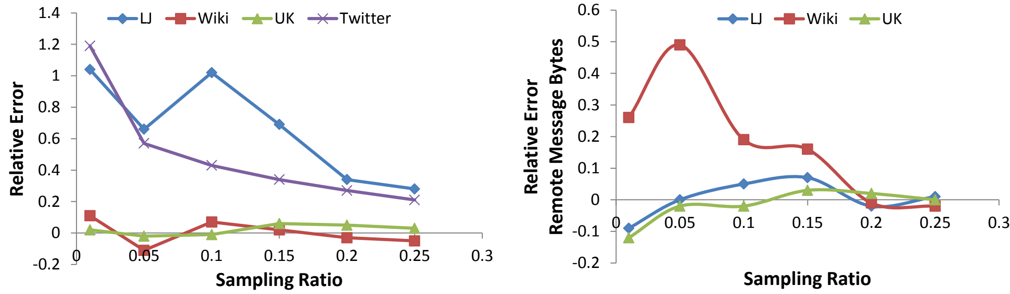


Figure 3.8 – a) Predicting active vertices for connected components with guided sampling (left), b) Predicting remote message bytes for neighborhood estimation (right).

dataset to 21% relative error for a sampling ratio of 0.25 (Figure 3.8 a)). When using a smaller number of seed nodes in sampling, the average out degree of vertices increases (random walks return to already visited nodes more often, and more incident edges are picked), hence, the propagation of the connected components ids to all the vertices of the sample takes a fewer steps, as in the original graph. This result shows that additional information on the characteristics of the dataset and on the algorithm can guide the sampling process to achieve higher accuracy results. Similar trends are observed for predicting other input features such as *message byte counters*.

Neighborhood Estimation: We execute neighborhood estimation for a fixed number of iterations $numIter = 10$: i.e., finding the number of vertices reachable within 10 hops. Figure 3.8 shows results for estimating remote message bytes. For a sampling ratio of 0.1 the relative errors for estimating remote message bytes are less than 19% for all datasets analyzed. Compared with the other algorithms, we observe that the errors for LJ are much smaller for this case: As the number of iterations is fixed and the key input features variability among consecutive iterations is less pronounced than for algorithms like top-k ranking or connected components (a large number of vertices stay active and propagate messages to neighbors for the first 10 iterations), the overall estimations errors are reduced.

3.8.3 Upper Bound Estimates

In the following we analyze the accuracy of predicting iterations for PageRank when using analytical upper bound estimates. In particular, for PageRank iterations are approximated using the analytical upper bound as defined in the detailed survey of Langville et al. [46]: $\#iterations = \frac{log_{10}\epsilon}{log_{10}d}$, where ϵ is the tolerance level as defined above, and $d = 0.85$ is the dumping factor. Note that the formula does not consider the characteristics of the input dataset, and as we show next, such bounds are loose. For instance, for a tolerance level of $\epsilon = 0.001$ we obtain a number of 42 iterations using the above formula, whereas the actual number of iterations is less than 21 for all datasets (a factor of 2x misprediction). Figure 3.9 shows the actual number of iterations, the estimated number of iterations using the above

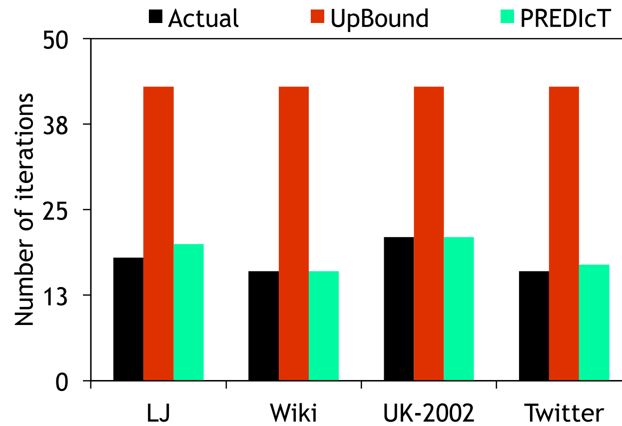


Figure 3.9 – Estimating the number of iterations: Analytical upper bounds versus PREDIcT.

formula, and the predicted number of iterations for PREDIcT when using a sample of 10%. We observe that PREDIcT improves the accuracy of analytical upper bounds for estimating iterations, as it reduces the relative error of analytical upper bounds from $[104, 168]$% to a relative error of $[0, 11]$%.

3.8.4 Estimating Runtime

In this section we show the accuracy of predicting the end-to-end runtime execution for semi-clustering, top-k ranking, connected components, and neighborhood estimation. As they show runtime variability among subsequent iterations, they are more challenging to predict than algorithms with constant per iteration runtime (i.e., PageRank). For training the cost model we show results for two cases: i) no prior executions of the algorithm exist (no history); ii) historical executions of the algorithm on *different* datasets exist. For the case that no history exists, sample-runs on samples of 0.05, 0.1, 0.15 and 0.2 are used for training. For the case that history exists, prior runs on all other datasets but the predicted one are additionally considered. We note that once a cost model is built it is used multiple times, for predicting the runtime of the same algorithm on *different* input datasets.

Semi-clustering: Figure 3.10 a) shows the accuracy of predicting runtime for the case that history does not exist. The coefficient of determination of the cost models corresponding to the three datasets on which predictions are made are as follows: $R^2_{LJ} = 0.82$, $R^2_{Wiki} = 0.89$ and $R^2_{UK} = 0.84$, and are showing that each multi-variate regression model fits the training data (the closer the value to one, the better the model is). The key input features that achieve the highest correlation on the multi-variate model are the local and remote message byte counters. It can be observed that the error trend for each dataset is very similar with the corresponding error trend for predicting iterations (see Figure 3.5 for $\tau = 0.001$). In contrast to predicting iterations, additional errors in estimating per-iteration input features (i.e., message byte counters) and cost model approximations are determining an error difference between

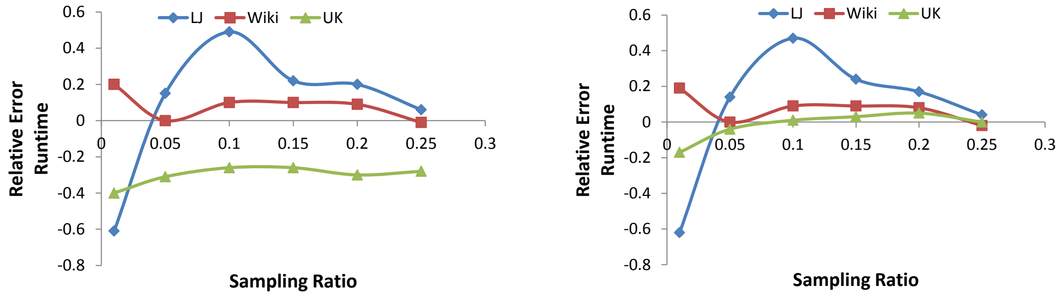


Figure 3.10 – Semi-clustering runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right).

the two graphs. For a sampling ratio of 0.1 the errors are less than 30% for the scale free graphs and less than 50% for LiveJournal.

Figure 3.10 b) shows similar results for the case that history exists. The corresponding coefficient of determination of each of the three models is improved: i.e., $R^2_{LJ} = 0.95$, $R^2_{Wiki} = 0.95$ and $R^2_{UK} = 0.88$. The error trends for Wikipedia and LiveJournal are similar as for the case that sample-runs are used for training. The cost factors for the UK dataset are improved and the errors are reduced to less than 10% when using a sampling ratio of 0.1 or larger.

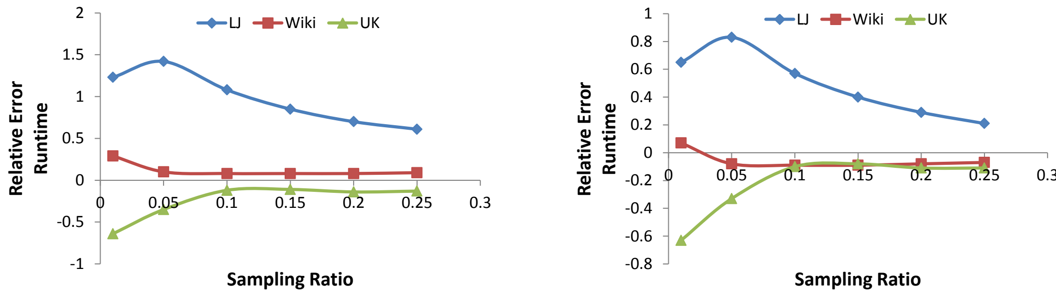


Figure 3.11 – Top-k ranking runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right).

Top-K Ranking: We analyze the accuracy of estimating time in Figure 3.11. We observe that the error trends are less than 10% for the scale free graphs analyzed. The key input features that achieve the highest correlation on the multivariate model are the local and remote message bytes and their corresponding message counts. For the case history is not used, the coefficient of determination of the models are as follows: $R^2_{LJ} = 0.95$, $R^2_{Wiki} = 0.96$ and $R^2_{UK} = 0.99$. Yet, the cost factors corresponding to the cost model for LJ dataset are over-predicted: That is due to the training phase which uses very short sample runs, especially for small datasets such as LJ. As the overhead of running very short iterations surpasses the actual processing cost associated to each key input feature, the coefficients of the cost model are over-estimated. Hence, the end to end relative errors are determined not only by over-predicting key input

features, but also by over-predicting cost factors. In contrast to LJ, for larger datasets fairly accurate cost models can be built using sample-runs. For the case history is used, all the cost models are improved. The coefficient of determination of the models are: $R^2_{LJ} = 0.99$, $R^2_{Wiki} = 0.99$ and $R^2_{UK} = 0.99$. We observe that the error trends are in pair with the error trends for estimating message byte counts (Figure 3.6 b)).

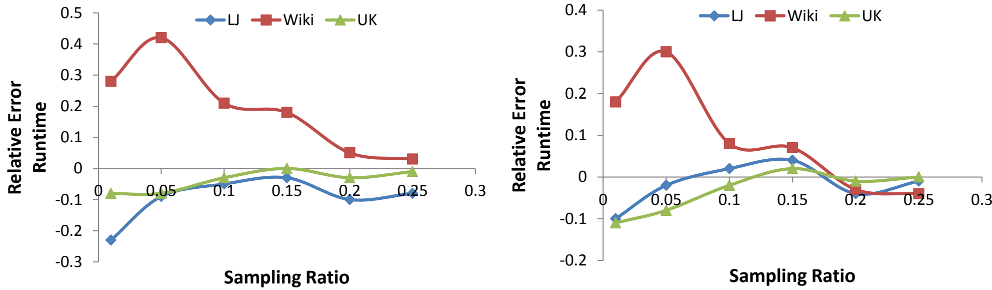


Figure 3.12 – Predicting runtime for neighborhood estimation: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right).

Neighborhood Estimation: Figure 3.12 shows accuracy results for estimating runtime for neighborhood estimation. For a sampling ratio of 0.1 the relative errors for estimating runtime in the case that history is not used are less than 21%, while for the case history is used, all errors are reduced to less than 10% for the same sampling ratio. The key input features that achieve the highest correlation on the multivariate model are the active vertices, the total vertices, and the local and remote message bytes, and the coefficient of determination of the models: $R^2_{LJ} = 0.99$, $R^2_{Wiki} = 0.99$, and $R^2_{UK} = 0.98$.

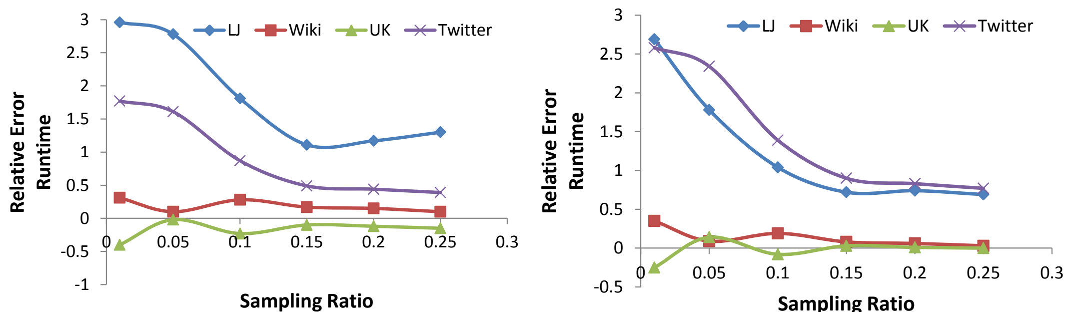


Figure 3.13 – Connected components runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right).

Connected components: Figure 3.13 a) shows runtime results for the case that only sample-runs are used in training. Similar error trends as for the case of estimating active vertices are observed (see Figure 3.7 b)). We note that due to the variability among consecutive iterations, the number of active vertices and the message byte counts have a higher impact on runtime of connected components algorithm than the number of iterations per se.

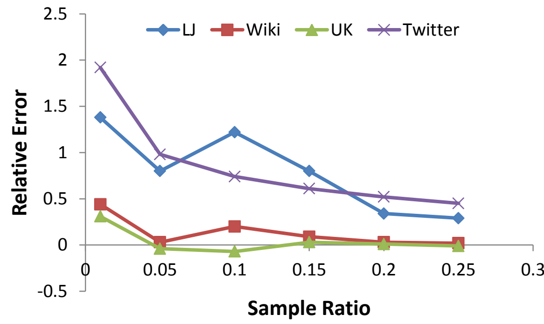


Figure 3.14 – Connected components runtime prediction: Training with sample- and actual-runs for guided sampling.

The key input features that achieve the highest correlation on the multi-variate model are the number of active vertices, the local and the remote message byte counters. The coefficient of determination of the models corresponding to the four datasets on which predictions are made are as follows: $R^2_{LJ} = 0.88$, $R^2_{Wiki} = 0.94$, $R^2_{UK} = 0.98$, and $R^2_{TW} = 0.99$. For a sampling ratio of 0.1 the relative error for Wikipedia dataset is 28% and for UK is -23%. When historical runs are additionally used in training the corresponding errors decrease to 19% and -8% respectively. The high errors on LiveJournal datasets are determined in part by key input features over-predictions and in part by cost factors over-estimations (for a very similar reason as for top-k algorithms explained above). The causes of errors for Twitter are mainly coming from over-predicting key input features. Figure 3.14 is showing the corresponding results when the number of seed nodes used for BRJ sampling is set to 100 (guided sampling). While the web graphs are marginally affected by a smaller number of seed nodes, the accuracy on Twitter is improved by 30% for a 0.25 sampling ratio.

3.8.5 Sensitivity to Sampling Technique

In this section we analyze the accuracy of predicting iterations when varying the underlying sampling technique. In order to analyze the impact of bias on maintaining key properties on the sample, we compare RJ with BRJ. Additionally, we select MHRW [30], another sampling technique based on random walks that in contrast with RJ, removes all the bias from the random walk, which is known to inherently have some bias towards high degree vertices. All sampling techniques use a probability $p = 0.15$ for restarting the walk, while the number of seed vertices for BRJ is $k = 1\%$ of the total vertices of the graph. Figure 3.15 shows sensitivity analysis for predicting iterations for PageRank, semi-clustering and top-k ranking on UK dataset. Figure 3.16 shows sensitivity analysis for predicting key input features for connected components (i.e., active vertices, iterations) and neighborhood estimation (i.e., remote message bytes). We observe that for a sampling ratio of 0.1, the relative error for BRJ sampling are generally better than for all the other sampling techniques. The result shows that the bias towards high out-degree vertices of BRJ contributes to a good accuracy in prediction for the algorithms we analyze in this chapter. The reason is that convergence of these algorithms

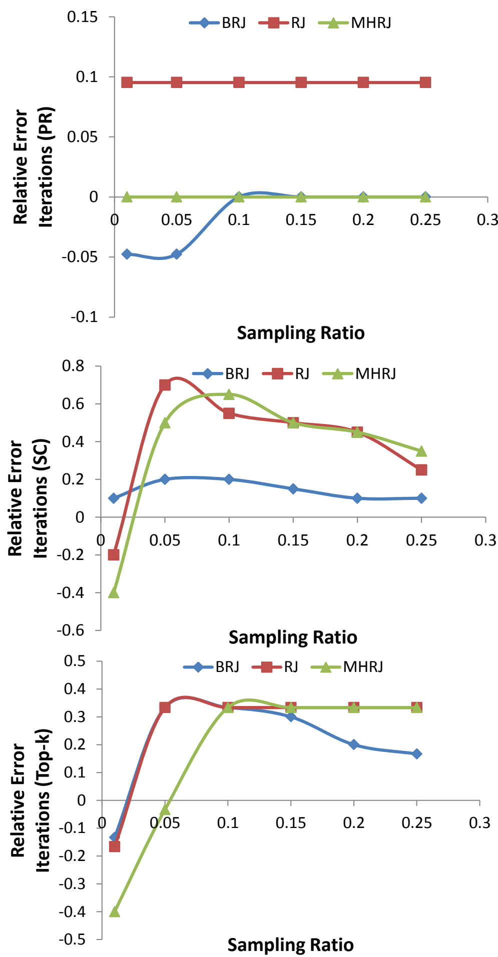


Figure 3.15 – Predicting iterations: sensitivity analysis w.r.t. sampling technique for PageRank (top), semi-clustering (middle), and top-k ranking (bottom) on UK web graph.

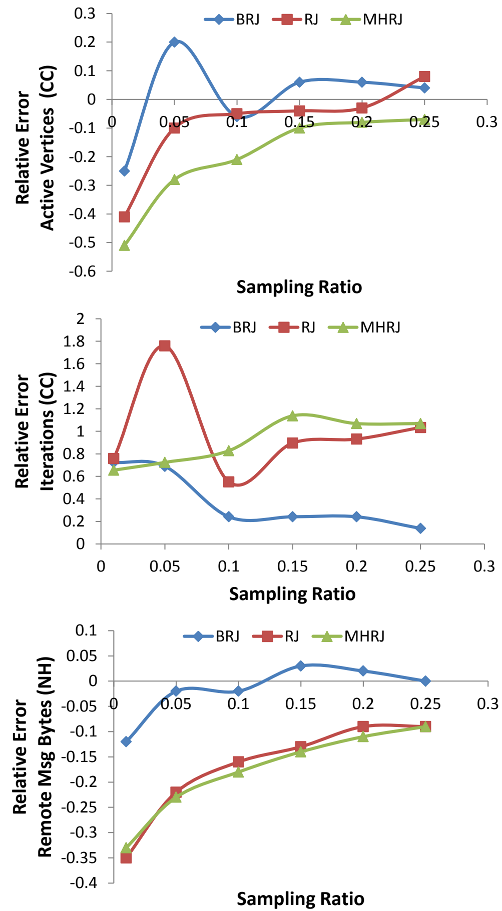


Figure 3.16 – Predicting key input features: sensitivity analysis w.r.t. sampling technique for connected components (top and middle), and neighborhood estimation (bottom) on UK web graph.

is inherently “dictated” by highly connected nodes: For instance, for PageRank such nodes contribute a large share to the average rank value, or for semi-clustering they contribute significantly to the ratio of semi-cluster updates. While other iterative algorithms executing graph processing tasks such as: random walks with restart [44] (proximity estimation), or Markov clustering [74] are expected to benefit from similar sampling methods based on random walks, customized sampling methods may be required for other algorithms. In Figure 3.13 c) we showed one example dataset for connected components algorithm, where guiding the sampling process can further improve the accuracy of results, given that more information about the input dataset is available.

Sampling Consistency: Finally, in order to evaluate the consistency of the sampling method, we perform further sensitivity analysis: For each sampling ratio we take multiple samples (using different starting seeds for the random number generator), we run sample-runs on each of them and evaluate the standard deviation for estimating iterations. For a sampling ratio of 0.1, the largest deviations observed are as follows: For PageRank: 3% for scale free graphs and 0% for LiveJournal, for semi-clustering: 5% on scale free graphs and 14% on LiveJournal, and for connected components: 9% on scale free graphs and 10% on LiveJournal. While there is some inherent variability in the sampling process, the error trends are maintained similar among different sample instances.

Sampling cost: For a sampling ratio of 0.1 the cost of taking a BRJ sample on the in memory graph using a *sequential* random walk implementation ranges between tens of seconds and 14 minutes for our datasets. The cost of taking a similar sample with RJ ranges between tens of seconds and 3 minutes. The cost of BRJ is higher because the probability of reaching new vertices decreases after the hub of highly connected nodes was already sampled. As more rounds of walks are necessary to reach new vertices, more time is required for sampling. Taking a sample can be sped up by using a parallel approach, where multiple workers are used for running independent random walks in parallel. Algorithms on distributed random walks exist and can be used for parallelizing the sampling task [22, 30].

3.8.6 Overhead Analysis

This section compares the runtime of the sample-run for a sample ratio of 0.1 with that of the actual-run. Figure 3.17 shows the runtime of all algorithms for the largest graphs: Twitter and UK. For PageRank, the runtime of the sample-run on a sample of 0.1 of the Twitter dataset accounts for 3.5% of the runtime of the actual-run. The reason is that the our sampling mechanism stops after a given ratio of *vertices* (not edges) is sampled. As Twitter graph is much denser than the others, the average number of incident edges per vertex is almost 9x smaller in the sample graph. For semi-clustering, the runtime of the sample-run on a 0.1 sample of the UK dataset accounts for 4.8% of the runtime of the actual-run for a similar reason as before.

We note that the runtime of the sample-run is much smaller than the runtime of the actual-run particularly for long running algorithms, where the runtime of the iterations dominate the runtime of the algorithm (i.e., the overhead of pre-processing the graph is relatively small). For algorithms where the overhead of pre-processing the graph dominates, the overhead of running sample-runs is higher. Connected components on Twitter is one such example: The actual time spent in running iterations is 19 seconds for the sample-run, which accounts for 4% of the time spent in running iterations for the actual-run (i.e., 465 sec). Yet, due to the overhead of reading, partitioning and outputting the result, that accounts for more than 80% of the sample-run time, the overall runtime of the sample-run relative to the runtime of the actual-run is higher, accounting for 12% of its time.

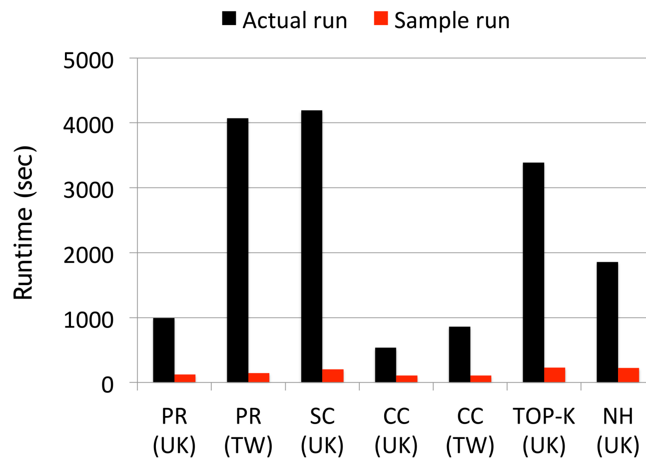


Figure 3.17 – Runtime of sample-runs and actual-runs for PageRank (PR), semi-clustering (SC), connected components (CC), top-k ranking (TOP-K), and neighborhood estimation (NH), in seconds.

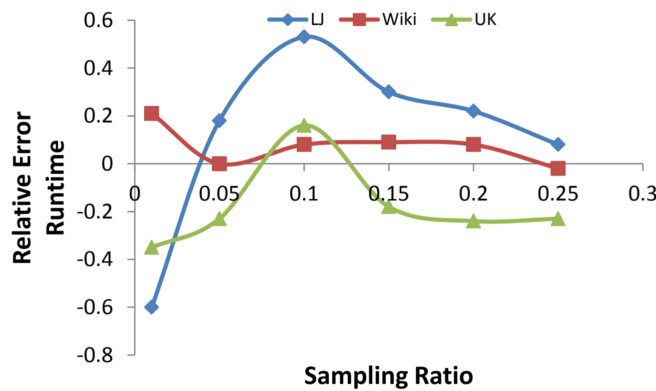


Figure 3.18 – Estimating runtime for semi-clustering for a different slot allocation.

3.8.7 Resource Allocation

We present one experiment that demonstrates PREDIcT's applicability for estimating runtime when a different resource allocation (i.e., number of slots) is used during the sample run: In particular, we use 15 workers for the sample-run, and 29 workers for the actual run. Figure 3.18 shows the results for estimating runtime for semi-clustering algorithm. In contrast with Figure 3.10 b) (where the same slot configuration was used for the sample and the actual runs) increased errors are observed (in particular for UK dataset) due to an additional level of critical path approximation: i.e., given a different number of slots to execute the algorithm, we use an uniform scaling factor to scale the two extrapolating factors (on edges and vertices). For higher accuracy results, our framework is extensible to support *per worker* extrapolating factors, according to the partition size of each worker (i.e., number of edges and vertices each worker is allocated with).

3.9 Summary of Related Work

The main limitations of existing theoretical approaches (e.g., [46, 44, 34, 8]) is that they provide loose upper bounds on the number of iterations an algorithm requires to converge (as shown in Section 3.8.3) and do not model system level resource requirements. Although DBMS-like approaches (e.g., [29, 50, 6]) are conceptually ready for modeling resource requirements of *bulk* iterations (i.e., that have uniform resource requirements per iteration), they lack the mechanism of estimating resources of *sparse* iterations, and do not estimate the number of iterations. PREDIcT uses an experimental approach to overcome the challenges of iterative algorithms: Sample-runs are used to quantify the number of iterations, and per-iteration resource requirements. This set of characteristics enables PREDIcT to accurately estimate the runtime of iterative algorithms.

3.10 Conclusion

In this chapter we presented PREDIcT, an experimental methodology for predicting the runtime of a class of iterative algorithms operating on graph structures. PREDIcT builds on the insight that the algorithm execution on a small sample can be *transformed* to capture the processing characteristics of the complete input dataset. Given an iterative algorithm, PREDIcT proposes a set of transformations: i.e., a sample technique and a transform function, that in combination can maintain key input feature invariants among the sample run and the actual run.

PREDIcT introduces an extensible framework for building customized cost models for iterative algorithms executing on top of Apache Giraph, a BSP implementation. Our experimental analysis of a set of diverse algorithms: i.e., ranking, semi-clustering, and graph processing shows promising results both for estimating key input features and time estimates. For a sample ratio of 10%, the relative error for predicting key input features ranges in between 5%-35%, while the corresponding error for predicting runtime ranges in between 10%-30% for all *scale-free* graphs analyzed.

4 Predicting Runtime of Data Pre-processing

4.1 Introduction

In this chapter we consider analytical workloads produced by data pre-processing tasks: i.e., tasks that Extract, Transform, and Load (ETL) the input for further analysis and more complex processing. In contrast to ad-hoc query workloads, data pre-processing tasks are comprised of *fixed data flows* that are run repetitively over newly arriving data sets or on different portions of an existing data set. For such workloads, mechanisms to predict the runtime performance for incrementally updated input data sets are required.

In contrast with analytical queries consisting of traditional database operators, ETL processing tasks executing on MapReduce often include user defined map and reduce functions written in imperative languages such as Java. The input data is read from *in-situ* files whose structure may be opaque to the system. One of the main differences, is that MapReduce does not always "own" the data or the query's operators. In this context, modeling the query runtime using state-of-the-art analytical modeling is still an open problem.

In this chapter we develop hybrid prediction models customized *per query segment type*. We specialize models per query segment type in order to reduce the corresponding domain knowledge required about the operators' semantics and implementation when collecting the features. Concretely, our models use a small number of key input features (i.e., tuple size, input cardinality) and exploit historical information about prior query executions (i.e., per tuple processing cost). To compute a runtime estimate, our approach combines a set of machine learning models with a global analytical model. Machine learning models are used as building blocks to capture the *processing cost* and the *output cardinalities* of each query segment. An analytical model is then used to compute the query runtime from its segments' estimates.

A query can be modeled using one segment (coarse-grain) or multiple segments (fine-grain). We consider several options for segmentation since different granularities may be useful for different scenarios. For example, *coarse grain segments* are good candidates for dedicated infrastructures where performance interference and runtime variability is low. In contrast,

fine granularity segments are good candidates for shared infrastructures where the dynamics of the system (e.g., slowdown/speed-up) must be captured. For example, such segmentation is used for query progress estimators [53, 57]. Since all of these scenarios are of interest for our workloads, we propose a *generic* prediction mechanism which can be applied at different segment granularities according to the particular use case at hand.

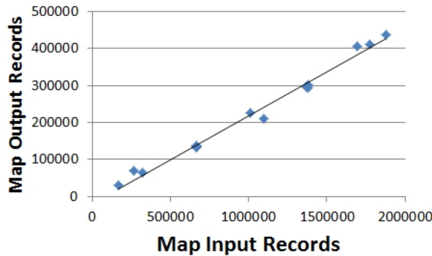


Figure 4.1 – Input / output cardinality correlations for Workload-A

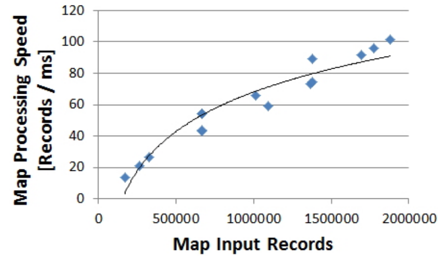


Figure 4.2 – Input cardinality / processing speed correlations for Workload-A

In this chapter, we evaluate our proposed prediction technique in the context of applications that were written using Jaql. We investigated correlations between input / output query features including data characteristics and per segment processing costs for several real workloads such as social media analytics, data pre-process for machine learning algorithms, and general analytics (the set of workloads is described in Section 5.6). As a result of the analysis, we found strong correlations between per segment input / output cardinalities, and between input cardinalities / segment processing speeds. Figure 4.1 and Figure 4.2 show the observed correlations for a a typical task (pre-process for mining step of social media analytics). We note that, if the observed correlations can be mapped to a function, it is possible to model them either using simple linear regression (i.e., for linear functions) or more specialized regression models such as *transform regression* [61], which can handle non-linearities in the data (i.e., for more complex functions).

The proposed technique is applicable to MapReduce jobs in general and other high-level languages so long as sufficient information is available in log files to identify traces from similar MapReduce jobs. We note that identifying job types by only comparing job binaries is not robust because additional configuration parameters may be used to decide the actual code fragments executed by the job. For our case, Jaql's use of *transparent functions* and their parameters facilitated this task.

In this chapter we make the following contributions:

- We develop hybrid prediction models that estimate the runtime of the same set of ETL queries executing on *different input datasets.*
- We analyze the sources of errors when predicting the query runtime and discuss the error propagation pipeline for the models.

- We evaluate and show the feasibility of the prediction models for different levels of segment granularities on real analytical workloads. In our experiments, we obtain less than 25% runtime prediction errors for 90% of predictions.

4.2 Jaql

Jaql is a declarative query language and a runtime system for enterprise data analytics designed to leverage Hadoop's MapReduce distributed processing model. Jaql is used by several products at IBM (including Cognos Consumer Insights, and InfoSphere BigInsights) [11] that influenced the design and development of the language itself and the processing system. Jaql is used by a range of data centric applications including: data search, data cleaning, machine learning, machine learning pre-processing, log analysis.

While Jaql shares similarities with the other declarative languages designed for data analytics at scale: e.g., HiveQL, Pig, DryadLINQ, etc, it has some specific features that make it attractive to use for data analysis both in the early stages of the analysis (i.e., semi-structured data), but also in the later stages of the processing pipeline (i.e., structured data). Concretely, Jaql uses the JSON data format which inherently supports both semi-structured and structured data (i.e., *data model flexibility*). Jaql borrows concepts from functional languages like: allowing lazy evaluation and high order functions, thus it can process expressions for which the input schema is partially known at runtime. Additionally, Jaql allows users to specify scripts at various levels of abstraction (i.e., high level operators can be combined with low level operators) and it exposes control and access to the physical query plan if needed (aka, *physical transparency*).

4.2.1 Query Example

Figure 4.3 shows a query example that from a list of web pages it extracts the web graph corresponding to a particular web domain. The main script (lines 11-13) reads the input pages from HDFS, it calls the *extractWebGraph* jaql function for the *"uk"* domain (defined above (lines 2-5)), finally it writes results back to HDFS. Jaql uses the pipe operator -> to specify the input/output flow of data. The extractWebGraph function applies a *filter* operator to select only the pages corresponding to the selected domain, then it applies a *transform* operator that produces an output record with the schema *page_url*, *valid_links*. *valid_links* is a subset of the web addresses included in *links*. *filterByDomainName* is the jaql function that filters out the links outside of the selected domain. *extractDomainName* is a user defined function (i.e., java UDF) that extracts the domain name from an input URL. Its implementation is defined in "ch.epfl.jaql.examples.ExtractDomainNameFromURL" class (not shown here).

```
[
 {
   page_url: "http://a.ch",
   out_links: ["http://b.ch", "http://c.ch"]
 },
 {
   page_url: "http://a.uk",
   out_links:["http://a.ch", "http://c.uk",
"http://d.uk"]
 },
 {
   page_url: "http://c.uk",
   out_links:["http://d.uk"]
 },
 {
   page_url: "http://d.uk",
   out_links:["http://c.uk"]
 }
]
```

```
[
 {
   page_url: "http://a.uk",
   valid_links: ["http://c.uk", "http://d.uk"]
 },
 {
   page_url: "http://c.uk",
   valid_links: ["http://d.uk"]
 },
 {
   page_url: "http://d.uk",
   valid_links: ["http://c.uk"]
 }
]
```

```
1:  extractDomainName = javaudf("ch.epfl.jaql.examples.ExtractDomainNameFromURL");
2:  extractWebGraph = fn (webPages, domain)(
3:    webPages -> filter extractDomainName($.page_url) == 'uk'
4:    -> transform {$.page_url, valid_links: filterByDomainName($.links, "uk")}
5:  );
6:
7:  filterByDomainName = fn (listOfLinks, domain)(
8:    listOfLinks -> filter extractDomainName($) == domain;
9:  );
10:
11: read(hdfs("crawledPages.dat")
12: -> extractWebGraph("uk")
13: -> write(hdfs("webGraphUK.dat");
```

Figure 4.3 – Jaql query (script) example that extracts from a list of web pages the web graph corresponding to a particular web domain.

4.2.2 Query Compilation in Jaql

For an input query expressed in Jaql the compiler produces an optimized query plan that will be executed as a pipeline of MapReduce jobs. Jaql's compiler associates to each job a portion of the query plan tree that it will execute during the map phase, and a portion of the query plan that will run in the reduce phase of the job. These portions of the query plan corresponding to a MapReduce job are saved inside the job's configuration file as "Jaql strings" and are expressed also in Jaql. At execution time, Jaql's runtime parses the Jaql string of the

map (or reduce) phase and executes the underlying operators.

4.3 Modeling Assumptions

First, we assume that the cluster configuration settings are constant. This assumption typically holds in practice if we consider that the best set of configuration settings is usually chosen at the deployment time per workload rather than per each input query. Second, we assume that data distribution of the inputs does not change. Increasing the table sizes maintains the relative distribution of values constant, i.e. all datasets *sample data* from the same distribution. An important effect of this assumption is data proportionality. I.e., for an input schema, the average record size remains constant. We experimentally validated the last assumption on the workloads that we investigated, which were typically composed of multiple UDFs that were executed on semi-structured data. However, if the data distribution assumption does not hold for workloads which store data in more traditional, structured format, orthogonal approaches may be employed to build histograms on the columns of interest. For instance, online aggregation techniques as proposed in [60] may be used to build *approximate histograms* at a low cost.

4.4 Model Fitting and Prediction

4.4.1 Sketch of Proposed Approach

We separate queries into query types and we build prediction models per query-type as follows. Each query type is defined by the set of MapReduce jobs it requires in the query execution. Further, each MapReduce job is identified by the set of Jaql functions that describe the query semantics of the given job (e. g., filter, aggregate, join, etc). In order to filter the log files of a workload on a particular query-type, we use the following definition of job similarity: *two jobs are considered similar iff all of their Jaql functions are equal.* While using a less restrictive definition of similarity is possible, our definition of job similarity allows us to use a feature vector consisting of *only data processing characteristics* instead of a query feature vector that combines query semantics with data processing characteristics. Building models per query type with this definition of job similarity is in fact not very different than the *sample run* prediction step we use in the context of iterative processing to capture the processing characteristics of the analytical task (as shown in Section 3.4). One of the main differences is that we do not explicitly run the query on a sample, but we exploit prior reference executions of the query on different input datasets, that are already available inside the query logs.

A typical Jaql query is composed of several MapReduce jobs. A MapReduce job consists of several phases (i.e., the map and reduce phases). In turn, each phase has several processing steps (i.e., read, map, sort, write, shuffle, reduce). In our approach we break the query into several segments and build prediction models at segment granularity. Then, we compute the query runtime using a global model that aggregates each segment's performance. A

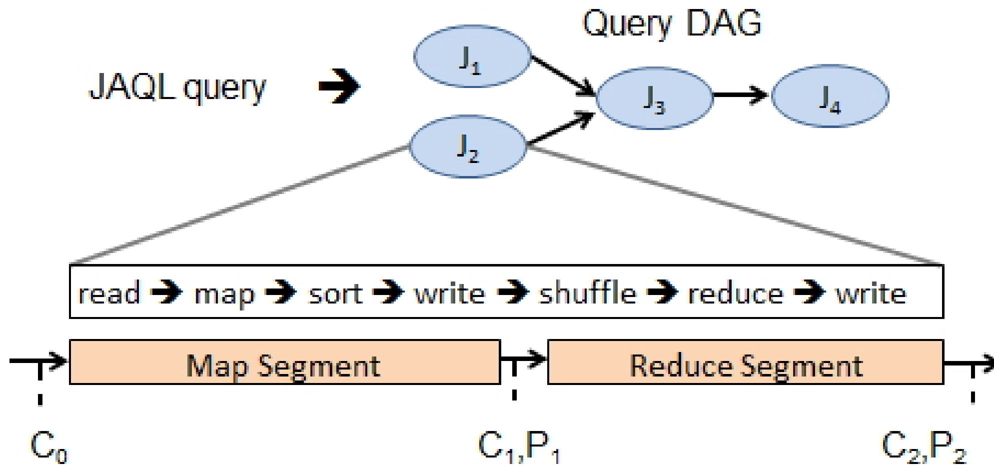


Figure 4.4 – Modeling per segment cardinality functions (i.e., C_i) and processing speed functions (i.e., P_i) for phase-level segments.

segment can be a query, a job, a phase or a processing step according to the level of granularity considered. Figure 4.4 illustrates phase-level segments.

There was no overhead to collect the data needed to build the models since existing logs were used 'as-is'. The time to build the models for the experiments used in this chapter ranged from seconds to minutes, depending the amount of log files analyzed. This overhead and the required disk space needed to store the logs can be tuned as needed by limiting the maximum number of instances stored per job type.

4.4.2 Modeling Segment Performance

We use two machine learning models to predict segment performance. A model is used to predict the processing speed of the segment and another model is used to predict the output cardinality of the segment. For constructing these models, we use uni-variate linear regression as follows: For predicting the processing speed we use a feature vector *(input cardinality, processing speed)*, while for predicting the output cardinality of a segment we use a feature vector *(input cardinality, output cardinality)*. These models are later used to compute the runtime estimate of the segment. Using the input cardinality and the processing speed we compute the system utilization time of the segment, while the output cardinality is used as the input into the subsequent segment. For the first segment of the query pipeline we compute the input cardinality based on the input and tuple size of the input datasets.

4.4.3 Modeling Query Runtime

To predict the query runtime we combine the performance of each segment on the critical path of the query using a global analytical model. Depending on the level of segment granularity, there are several factors that may need to be considered such as: the level of parallelism (i.e., the number of map / reduce tasks), scheduling overheads, segment overlaps and data skew. In the following we present the methodology for computing the query runtime performance for prediction models that use phase-level segments. This methodology can be easily adapted for other segment granularities (e.g., job, query), and therefore is not presented here.

In order to compute the effective running time of a segment, we divide the system utilization time of the segment by the actual number of tasks used to execute the segment (i.e., multiple tasks are used to increase the degree of parallelism). The actual number of tasks is determined by the cluster configuration, the job configuration and the amount of input data processed. For instance, the number of map tasks is usually computed based on the size of the input data, while the number of reduce tasks is typically taken from the configuration file.

Given that there are no queuing delays in the system and that the MapReduce cluster is configured such that the reduce phase starts after the map phase finishes, we can use the following formulas to compute the runtime estimate of a query:

$$SegmentRuntime = (\overline{TaskRuntime} + \overline{SO_{task}}) \times numWaves \tag{4.1}$$

where $\overline{TaskRuntime}$ is the average runtime of a map task or a reduce task, $\overline{SO_{task}}$ is the average scheduling overhead per task, and $numWaves$ is the number of waves (i.e., the maximum number of tasks that a worker node is expected to run sequentially) required to execute the job.

The job runtime is computed as follows:

$$JobRuntime = \sum_{k} SegmentRuntime_k + SO_{job} \tag{4.2}$$

where $SegmentRuntime_k$ is given by the previous formula and the SO_{job} is the scheduling overhead per job. Currently, all the MapReduce jobs of a given Jaql query are executed sequentially. Therefore, the query runtime estimate is given by adding up the runtime of all MapReduce jobs. For parallel job executions, identifying the jobs on the critical path of the query is further required to compute the query runtime.

4.4.4 Sources of Errors

There are two categories of factors that contribute to inaccurate runtime predictions: i) Prediction errors caused by non-representative feature vectors or insufficient training at

the segment level; in the same category, we also include prediction errors caused by interconnecting segment models together (i.e., using the predicted output cardinality of one segment as the input of the subsequent segment). ii) Simplification assumptions about the scheduler (i.e., potential schedules, scheduling overheads), simplification assumptions about data skew and hardware homogeneity assumptions across cluster nodes;

In order to compare the errors introduced by the segment level models (case i)) with the errors introduced by simplification assumptions used in the global analytical model (case ii)), we introduce a new metric called the *aggregated runtime*. The aggregated runtime is the query runtime computed using the global analytical model presented in Section 4.4.3 that takes as input *perfect* segment level runtime values. Thus, the aggregated runtime exposes the errors that are introduced by the global analytical model and the simplification assumptions (i.e., it is effectively quantifying the second category of errors).

We currently account for data skew at the reduce tasks by modeling the skew exposed by earlier job runs on already seen data sets (i.e., we model the performance of the longest reduce task rather than that of the average task). Yet, we omit possible block size differences at the map tasks which may cause additional estimation errors (i.e., we use the average performance of a map task in the global analytical model).

4.5 Experimental Study

We evaluate our prediction techniques on a standard benchmark on decision support systems and on several real workloads.

TPC-DS [71]: TPC-DS is a decision support workload modeling a retail supplier. We use TPC-DS because it covers a large variety of decision support queries (e.g., reporting, iterative, data mining) which were designed to cover more realistic scenarios [62] as compared with its precursor (i.e., TPC-H [72]).

Workload-A: Social media data analysis. The categories of queries investigated include: mining pre-process, general pre-process and analytics.

Workload-B: Data pre-processing for machine learning algorithms. The categories of queries include: summarization, cleansing, and statistics computation.

4.5.1 Experimental Methodology

Each of the above workloads was run on a dedicated cluster. Each time a MapReduce job is executed, it outputs a historical file that summarizes how it ran. For Workload-A, we used existing historical files instead of re-executing the queries. For evaluating our models, we used *k-fold cross-validation* [35]. Historical files corresponding to each query were split into k sets where k-1 sets were used for training the models, and 1 set was used for testing the model. For

building these sets, we considered only historical files corresponding to query executions on *different* input data sets. This process was repeated k times. All prediction errors are computed as the relative error between the predicted and the actual values. We report all prediction errors as cumulative distribution functions.

4.5.2 Experimental Setup

We use several different cluster infrastructures. For the TPC-DS benchmark we run our experiments on a 10 node cluster, each of the node having two 6-core CPUs Intel X5660 @ 2.80GHz, 48 GB RAM and 1 Gbps network bandwidth. Workload-A uses a 4 node cluster, while Workload-B uses a 20 node cluster. In all experiments we use Hadoop 0.20.2 configured with FAIR scheduler. The reason for using several infrastructures is that for particular workloads we use existing historical log files from production clusters instead of re-playing all the workloads on the same cluster infrastructure.

4.5.3 Job-Level Predictions

We evaluate job-level predictions at multiple segment granularities: i.e., job and phase level. We use 3-level cross-validation to validate our prediction models.

Our first workload consists of a mix of three TPC-DS queries (i.e., Q3, Q7, Q10) and three synthetic queries, all of them using the TPC-DS data. We choose these queries because they include a different number of joins and aggregates, and hence have different complexity (i.e., with query pipelines varying from one single MapReduce job up to a maximum of seven MapReduce jobs). The job runtime varies in the range of [25sec, 4mins]. Figure 4.5 shows the cumulative distribution function of errors for a total of 186 predictions. For 95% of the workload the prediction errors were less than 20% for all the prediction models analyzed, while job-level models were more accurate, with 10% error for 95% of the workload. The reason that job-level models were more accurate is that they do not require to model the scheduling overheads or the critical path of the query explicitly. The effects of these factors are implicitly included into the features of the job-level models. The small differences between the aggregated runtime and the predicted runtime for phase-level segments show that the main causes that induced a large part of errors for phase-level segments were the simplifying assumptions presented in Section 4.4.4 rather than the fine grain models per se.

Figure 4.6 illustrates the absolute predicted values as compared with the actual values for phase-level segments. With a few outliers the predicted values closely match the actual values. This is also illustrated by traditional metrics used in prediction: the *coefficient of determination* R^2*=0.98* (the closer to 1, the better), the *normalized root-mean-squared error NRMSE=0.09* (the closer to 0, the better), and the *maximum under-prediction error MUPE=22%* (for a job of 136 sec). A full description of these metrics can be found in [35] and a summary in Section 4.2 of [85].

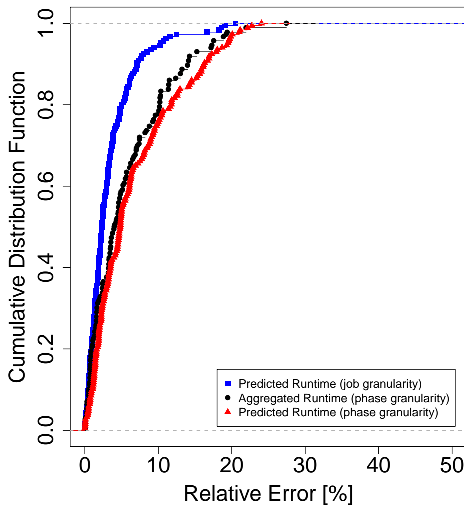


Figure 4.5 – Job runtime estimation for TPC-DS

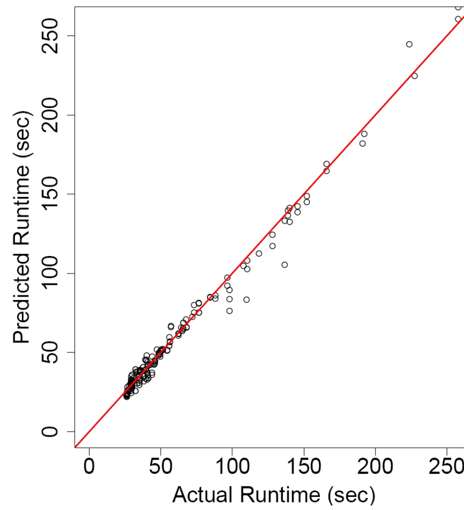


Figure 4.6 – Actual Runtime vs. Predicted Runtime for TPC-DS

Similar results for Workload A and Workload B are illustrated in Figure 4.7. These graphs show the prediction errors for phase-level segments only. For Workload A, the job running time varied in the range [16sec, 7.5 hrs], while the job runtime estimation error is less than 15% for 80% of the workload. For Workload B, the job running time varied in the range [1min, 30mins], while the job runtime estimation error is 30% for 80% of the workload. In both cases, our predictions are very close to the aggregated runtime, effectively showing that the prediction models per se have a good accuracy. Similarly, most of the prediction errors were caused by scheduling and critical path approximations.

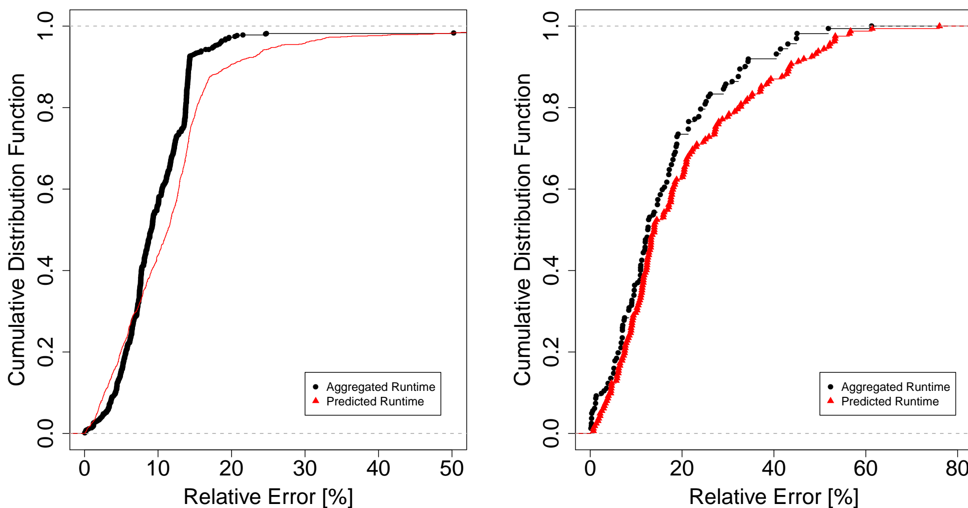


Figure 4.7 – Job runtime estimation for Workload-A (left), and Workload-B (right).

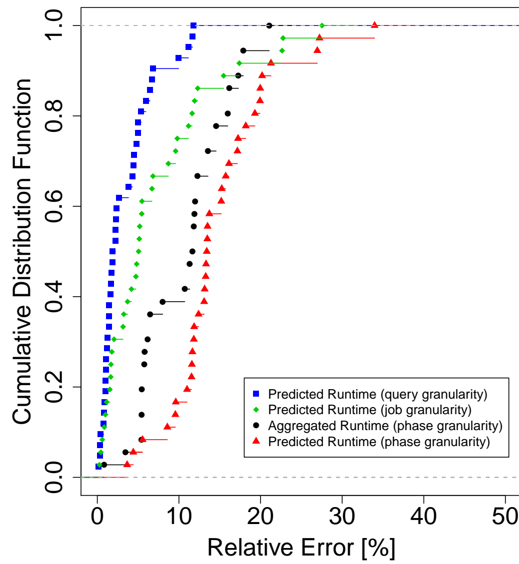


Figure 4.8 – Query runtime estimation for TPC-DS

4.5.4 Query-Level Predictions

We evaluated query level predictions at various levels of segment granularities: i.e., query, job and phase levels. We used 3-level cross-validation to validate our prediction models. Figure 4.8 shows the distribution of prediction errors for the TPC-DS workload. We use the same set of queries as presented in Section 4.5.3. The errors introduced by all prediction schemes was kept under 25% for 90% of the workload. Similarly with job-level predictions, coarse granularity models (i.e., that use query level segments) achieved better accuracy than fine granularity models (i.e., that use job or phase level segments).

Typically, queries with a larger number of MapReduce jobs accumulate more errors than queries with a fewer number of jobs. Yet, an interesting observation is that fine granularity models do not only cumulate errors on the critical path of the query, but may also neutralize cumulated errors if both over- and under- estimations are present. This is one of the reasons that phase granularity models accumulate only 10% more errors than query granularity models for query pipelines composed of up to seven MapReduce jobs.

The total number of predictions is less than for the case of predicting job-level performance because only query-level predictions are reported. For the job-level case, prediction errors for *all* the jobs of a query were reported. Traditional metrics used in prediction are still in reasonable limits as follows. For phase granularity models: R^2=0.97, *NRMSE=0.25*, and *MUPE=45%*, while for query granularity models: R^2=0.99, the *NRMSE=0.07*, and *MUPE=12%*.

Fine vs. Coarse Granularity Query Segments: In the context of dedicated cluster infrastructures, our technique is more accurate when applied on coarse grain segments rather than on fine grain segments. This result is not surprising, considering the additional sources of errors for fine granularity models (i.e., scheduling approximations, data skew) and the cu-

mulative errors caused by connecting a larger number of models together. This point is also corroborated by small differences between the predicted runtime and the aggregated runtime, which show the maximum achievable accuracy for fine granularity models. An interesting direction of future work is to combine fine granularity models with coarse granularity models to further improve runtime estimations. The idea is to use the fine granularity models that predict the size and the speed of processing intermediate results and then to use the predicted values as additional inputs in the feature vector of the coarser grain models. Such an approach resembles the models proposed in [86] with the difference that some of the input features of the model are at their turn predicted in a preliminary phase.

4.6 Summary of Related Work

Previous works on runtime prediction, in the context of traditional DBMS [29, 4, 25] or in the context of MapReduce [28], focus on estimating the runtime performance of *similar* queries on the *same* input datasets. Such techniques use a similarity metric to correlate the query of interest, whose runtime is being predicted, with other *similar* queries from the training set, for which the runtime is known. For ETL analytics, applications use *fixed* data flows that are run at regularly scheduled intervals over *newly* arriving data sets. For such cases, traditional approaches require *re-training* on each of the datasets to provide accurate estimates, or runtime must be extrapolated. In contrast, our approach can accurately model the *processing cost functions* corresponding to various query pipelines and input data sets.

Herodotou et al. propose Starfish [40, 38], a self-tuning system for Hadoop that aims to find the best set of configuration settings. The key building block of Starfish is the *job profile*, which models the processing characteristics of each job. Compared with Starfish, our models use *several* prior query executions on *multiple* data sets to fit processing cost functions instead of assuming that the cost functions are constant.

Morton et al. propose ParaTimer [57], a progress estimator for MapReduce DAGs. ParaTimer splits each MapReduce job into segments and builds the estimated time left until the query completes execution using the processing speeds and the input cardinalities of each query segment. Our approach complements ParaTimer as it builds models that predict the cardinality and the processing speed of each query segment.

4.7 Conclusion

In this chapter we presented an approach for predicting the runtime of Jaql queries executing data pre-processing tasks when the input datasets change. We developed a hybrid prediction method which combines localized machine learning models with a global analytical model. Machine learning models are customized per *query segment type*. This modeling decision allowed us to use a small number of data features for building the models. Two types of machine learning models were used: *processing cost* models for estimating the normalized processing

cost of a query segment and *output cardinalities* models for estimating the selectivity of a query segment. After estimating per segment runtime using machine learning models, an analytical model is used to compute the query runtime by summing up the segment-level estimates on the critical path of the query.

We evaluated the feasibility of our approach at various levels of segment granularities on several real workloads used at IBM, and on a selection of TPC-DS queries. In our experiments, the 90th percentile of predictions have an average relative error less than 25%. The sensitivity analysis we performed shows that in the context of dedicated cluster infrastructures, coarse granularity models are more accurate than fine grain models. Fine granularity models have additional sources of error than coarse grain models (e.g., task scheduling approximations), and they accumulate additional errors when inter-connecting a larger numbers of models together.

5 Runtime Prediction for Reporting SQL Analytics

5.1 Introduction

With the prevalence of using hardware infrastructure as a service (IaaS) for data management tasks, answering feasibility analysis questions for hypothetical execution configurations and identifying the execution configurations that can boost the actual performance of the workload by a given factor are fundamental requirements for many analytical applications. In this context, questions like: "What hardware configuration and which execution strategy for the workload can improve the actual performance by 2x?” are common. In order to answer such performance questions, mechanisms that estimate the workload performance for a set of potential execution configurations are required.

In this chapter we consider workloads of *reporting SQL analytics* that are run repetitively to compute periodic reports about the operational state of a business (e.g., computing log analysis, statistical summaries, etc). Reporting SQL analytics are prevalent in decision support queries [72, 71], data warehousing, web and social media analysis, and are often times executed as part of mixed analytical workflows that include other analytical tasks such as data pre-processing and iterative machine learning (as shown in Section 1.2.1). Executing reporting queries on large input datasets demands for a distributed processing engine that parallelizes the execution across a cluster of machines. In this work, we focus on reporting queries that are expressed in HiveQL and that are executing at scale on top of the MapReduce infrastructure.

We have seen in Section 2.5 that there are two main approaches to estimate the runtime of SQL queries: that of using analytical models and that of using machine learning models. While building accurate analytical models is a very challenging task, prior research showed that training-based models can be more accurate than pure analytical models when the training datasets cover well the space of testing queries. The reason is that training-based models can capture a wide range of runtime execution effects that are hard to model otherwise (e.g., [29, 6, 50]). While training based prediction models can alleviate the inaccuracies introduced by simplifying modeling assumption of conventional analytical models, they have two main limitations: *high training* cost, which is required each time the testing workload or the execu-

tion setting changes, and *reduced accuracy* outside of the training boundaries.

As recent work shows [29, 28, 6, 50, 81] and Table 2.1 summarizes, running benchmark queries for building the training set is an expensive task, as hundreds or even thousands of queries are executed to achieve a good coverage of the testing set. While for *fixed* deployments training incurs an one-time cost, and hence it is justified, in the context of *elastic* workload deployment, where a large number of potential hardware configurations are made available to end applications on demand and re-training is often required, high training cost is unacceptable.

To alleviate the inaccuracies of conventional analytical models and the high cost of training based approaches we develop TITAN: i.e., Training Methodology and Translation Models for runtime prediction. TITAN takes a new approach to training in order to reduce the training cost of state of the art prediction approaches. In particular, in this chapter we propose: i) A methodology for generating synthetic benchmark queries complemented with a pruning algorithm that altogether produces a concise benchmark that takes a short time to execute while not losing much on the prediction accuracy (i.e., covered input feature space). ii) For the cases that the input feature space of the testing workload is outside the boundaries of the training workload, and hence re-training is required, we propose novel translation models that exploit the existing training data to build *relative performance models* among different operator implementations. Such relative models prove to be useful for repeatedly run workloads where a reference query execution is available, and a better execution setting (in terms of operator implementation or deployment) is sought.

TITAN targets repetitive reporting workloads executed on MapReduce and it extends the state of the art approaches in this context, i.e., Starfish [38] and Elastisizer [39] at multiple levels: it uses a hybrid prediction approach and a relative performance model to estimate per operator *processing cost factors* for a *range of workload characteristics and execution settings* (as we further detail in Section 5.2.4), it reduces the modeling errors introduced in Starfish's analytical model by the *average task profile*, and it models HiveQL [70] operators in addition to ETL.

Contributions and Outline: In this chapter we make the following contributions:

1) We propose a training methodology and a pruning algorithm that reduce the number of benchmark queries to a minimum. Through the pruning algorithm TITAN reduces the time of running benchmark queries from *days to hours* while maintaining a good level of accuracy for the models.

2) To improve the accuracy of analytical models and reduce the cost of training based models, we develop a hybrid prediction approach that limits the use of machine learning for modeling *processing cost factors* at *operator phase granularity* for a range of execution settings and workload characteristics.

3) When the training queries do not cover the input feature space of the testing workload, we propose *translation models*, relative performance models that can exploit reference executions

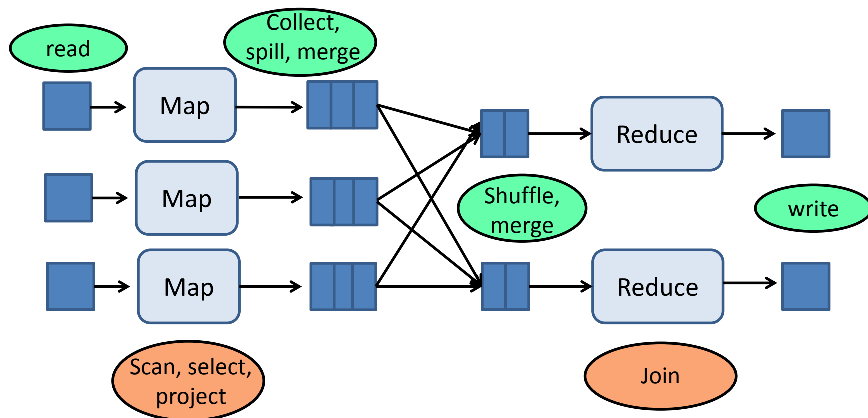


Figure 5.1 – Query Processing Model in HiveQL

of the workload corresponding to *different* execution settings.

5.2 Foundations and Overview

5.2.1 Query Execution in HiveQL

We choose HiveQL as the distributed data processing engine for executing SQL analytics at scale. HiveQL translates SQL queries into workflows of jobs that are executed on top of MapReduce. In particular, pipelines of SQL operators are plugged into the map and reduce functions of the job. Figure 5.1 shows the workflow resulting from executing a select-project-join query: *SELECT S.A, T.B from S JOIN T where S.A=T.B and S.A < 100*. Within a MapReduce job we notice two types of tasks: MapReduce specific tasks (e.g., read, collect, spill, merge, shuffle, write) which are implemented in the underlying MapReduce framework, and SQL operator specific tasks which are implemented inside the map() and reduce() functions and provided by the HiveQL engine. The scan, selection and projection operators are executed in the map function of the job, while the join operator per se is executed inside the reduce function of the job. The join operator implementation that is executed inside the reduce phase of the job is called the *common join*, and is the default join implementation in HiveQL.

5.2.2 Problem Definition

This chapter focuses on estimating the runtime of *reporting SQL queries*, i.e., SQL queries that are executed periodically on similar datasets or on different portions of the input dataset to answer pre-defined analytical questions about the operational state of a business: E.g., run reporting query Q on log datasets collected during the last week. In contrast to ad-hoc query execution, for repeatedly run workloads input statistics change slowly with time. Therefore, such workloads open up opportunities for *workload re-optimization* ([37, 9]) and *elastic workload deployment*, where the deployment setting is chosen such that application

pre-specified time constraints can be met ([39]).

While traditionally workload/query re-optimization is a risky task due to inaccurate statistics of intermediate results (and unaccounted correlations among table attributes: e.g., [19, 32, 12]), in this chapter we address a related, but *different* problem: That of estimating query runtime performance for a potential set of *execution settings* consisting of: i) query plan re-writes in terms of different operator implementation and possibly different packings of operators within one or several MapReduce jobs, and ii) a pool of potential hardware deployments, starting from a prior query execution for which input data statistics were collected in a database. This problem has a high practical applicability when transiting workloads from development into production and when seeking an elastic deployment that can *safely* guarantee user requested SLAs as exemplified in the prediction use cases in Section 1.1. In the rest of this chapter, we use the terms *execution setting* and *execution configuration* interchangeably.

5.2.3 Starfish's Limitations

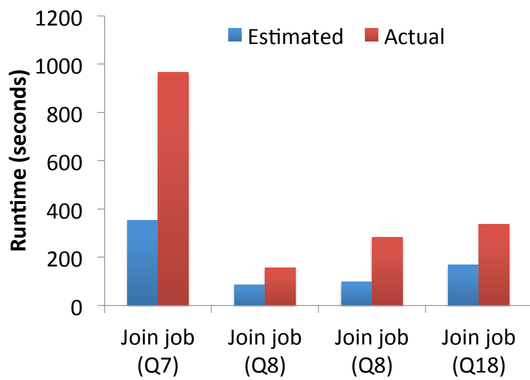


Figure 5.2 – Selection of common join jobs where the main source of error is Starfish's analytical model.

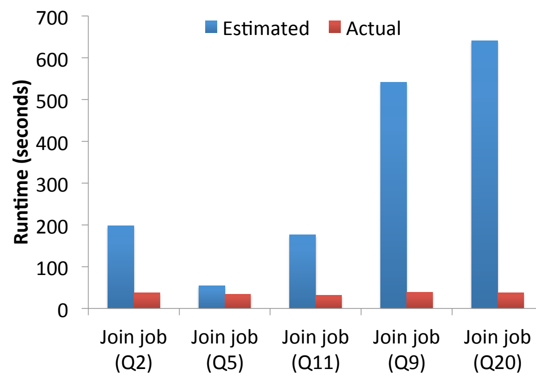


Figure 5.3 – Selection of common join jobs where the main source of error is the task uniformity assumption.

To better understand why pure analytical models are problematic for estimating the query runtime for the capacity planning problem defined above we present one experiment in Starfish [38], state of the art analytical model for the MapReduce ecosystem. The workload consisted of TPC-H queries translated into HiveQL that were executed on top of five databases (scaling factors 1, 10, 30, 60, and 100, as specified in Section 5.6.1). We used the query profiles corresponding to the workload execution on scaling factor ten as the *reference* profiles for the Starfish's What-If engine. That is, they are used as input into the What-If engine (i.e., *reference workload*). The What-If engine is then invoked to estimate the runtime of the workload on scaling factors 1, 30, 60, and 100 (*testing workload*). A summary of the Starfish's What-If engine is presented in Section 5.3.2.

The 95-percentile average ratio error of all MapReduce jobs of the workload that are executing

a common join operator is 2.77 (as opposed to an ideal ratio of 1.0), and the corresponding 95-percentile average relative error is 164% (as opposed to an ideal relative error of 0%). We identify two sources of error in the Starfish's what-if engine: i) *in-accurate analytical model*: that assumes that the processing time varies linearly with the number of input rows processed; ii) *task uniformity assumption*: that is, per tuple processing costs corresponding to a given operator phase are uniformly averaged among all tasks executing the operator phase, independently on the workload characteristics of each task (e.g., level of batching, input size), as we further detail in Section 5.3.2.

Figure 5.2 shows the estimated runtime versus the actual runtime for a selection of join jobs where the main source of error is the analytical model. For this set of jobs Starfish under-predicts the runtime by a factor of 1.8 up to a factor of 2.9. Figure 5.3 shows the runtime for a selection of join jobs where the main source of error is the task uniformity assumption. For this selection of jobs the main source of error is the shuffle phase, where the network transfer cost is heavily over-estimated. Concretely, varying levels of batching for the reference job's tasks executing the shuffle phase combined with cost averaging (independently on the level of batching) causes an over-estimated network transfer cost in the reference profile. As a consequence, the runtime of the jobs is over-estimated at prediction time by a factor of 1.6 up to a factor of 17.3. TITAN addresses these issues by extending Starfish profiler with SQL constructs, by making its analytical model aware of non-linear SQL operators (such as joins), and finally, by using a trained model that learns a range of processing cost factors as a function of the workload characteristics.

5.2.4 TITAN Overview

In the following we present an overview of our proposed hybrid approach that overcomes the inaccuracies of conventional analytical models, while at the same time it reduces the training time of training based prediction approaches. TITAN achieves its goal through a set of modeling design principles as we show next.

Training Methodology: We propose a training methodology at *operator phase granularity* that aims to cover the space of SQL operators and MapReduce phases. Using minimum amount of information about the testing workload (i.e., input schema, query operators used in the workload, and few configuration settings), we generate synthetic tables, query templates, and then query instances such that each operator phase and MapReduce phase is profiled multiple times, for a range of input data properties. Finding the minimal training dataset that maximizes the models accuracy is nevertheless a very hard problem, unsolved to date [35]. Therefore, instead of trying to generate a concise collection of benchmark queries in the first place we propose a heuristic that prunes a large collection of training query instances. Our algorithm shrinks the size of the training set iteratively, as long as the generalization error of the models from one iteration to the next changes less than a threshold value. Our approach uses geometric progressive sampling [64] to shrink the size of the original training set, and

k-fold cross validation [35] to compute the generalization error of the models for each pass.

Localized Training Based Models: To reduce training costs we propose a hybrid modeling approach that uses *localized* training based models that are compensated with global analytical models. Localized models require fewer input features and additionally do not need to capture the logic of a full-fledged cost model implicitly. Similar observations that motivate the use of fine grain, *operator level models* were proposed in the context of DBMS [6, 50]. We apply similar ideas in the context of SQL analytics executed on MapReduce, and go one level further by splitting up operators into multiple phases if the operator is sensitive to input data distributions (e.g., joins). Reducing the impact of input data distributions on the processing cost factors of an operator has beneficial outcomes in reducing training set sizes. We experimentally validate that fine grain models are more accurate than coarse grain models at small training set sizes.

Translation Model: For the cases that the training datasets do not cover the input parameter space of the testing workload, problem that often occurs in practice, we propose the *translation model*, a relative performance model that in contrast with conventional modeling learns *processing cost ratios* among pairs of operator pipelines having the same semantics but running with different execution settings (e.g., operator implementation, deployment). Once built, the translation model estimates the processing cost factor of a query pipeline P for execution setting E2 using a *reference* run of the same query pipeline P for execution setting E1 and the *relative performance ratio* learned during the training phase. For example, the translation model can be used to estimate the processing cost of a map join (i.e., the hash join of HiveQL), given the processing cost of the corresponding common join (i.e., the repartitioning join of HiveQL), and their relative performance ratio $r_{MJ/CJ}$. The translation model has two advantages over an absolute model: i) it exploits a prior, reference execution of the query being predicted, corresponding to a *different* execution setting; ii) as the trained model is relative, it is more likely to hold beyond the boundaries of the training dataset.

5.3 TITAN Prediction Approach

In this Section we present the end to end approach we propose for estimating the runtime of SQL analytics executed at scale. We first introduce the modeling assumptions and the main components of the prediction engine. Then, we present the hybrid prediction model in detail.

5.3.1 Modeling Assumptions

We make the following modeling assumptions:

- Input data statistics corresponding to the input tables and queries are collected into a database during the reference execution of the workload. Hence, they can be re-used during prediction.

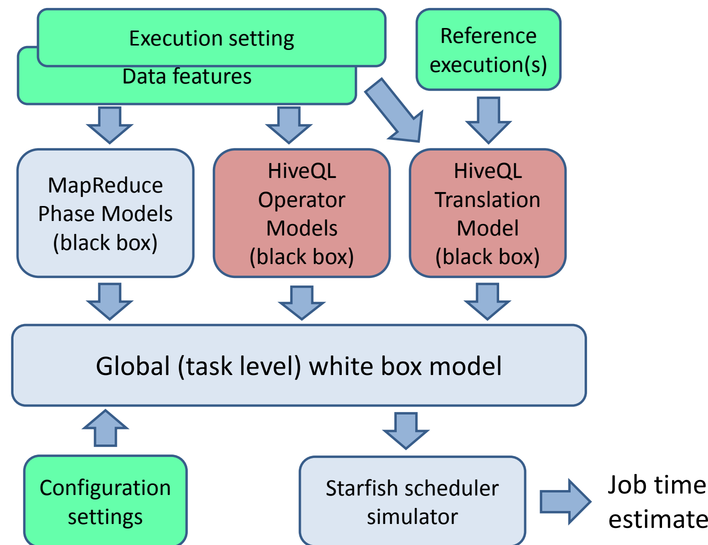


Figure 5.4 – Hybrid prediction engine for estimating the runtime of HiveQL queries.

- Without loss of generality, we implement prediction models for join/select/project operators. Extending our approach for other SQL query operators such as aggregations can be performed by following a similar methodology.

5.3.2 Hybrid Prediction Model

At the high level, our proposed approach is a hybrid of analytical models, machine learning models, and simulation. Machine learning models are used to estimate the processing cost corresponding to each SQL operator phase (e.g., join operator phase) and MapReduce specific phase (e.g., read, collect, merge, shuffle). Analytical modeling is used to estimate the runtime of each MapReduce task by summing up the runtime of each operator phase. Once the runtime of all map and reduce tasks was estimated, simulation is used to produce an ordering of tasks according to a scheduling policy (e.g., FIFO). The job runtime is the longest sequence of tasks that are executed sequentially. The end to end runtime of a query, that may be composed of multiple MapReduce jobs, is computed by summing up the runtime of each job on the critical path of the query. Figure 5.4 shows the main components of the prediction engine. We observe that there are two possibilities for modeling the processing cost of HiveQL operators: i) to use conventional machine learning models that learn *absolute processing costs*, ii) to use translation models which learn *processing cost ratios* among pairs of operator phases with similar semantics, but corresponding to different execution settings: E.g., estimate the processing cost of the map join, given the processing cost of the common join. While the two alternative modeling techniques are applied differently, they share the same prediction framework: the set of features, the fitting algorithm, and the global analytical model. The rest of the section describes the end to end prediction framework; translation models are presented in detail in Section 5.5.

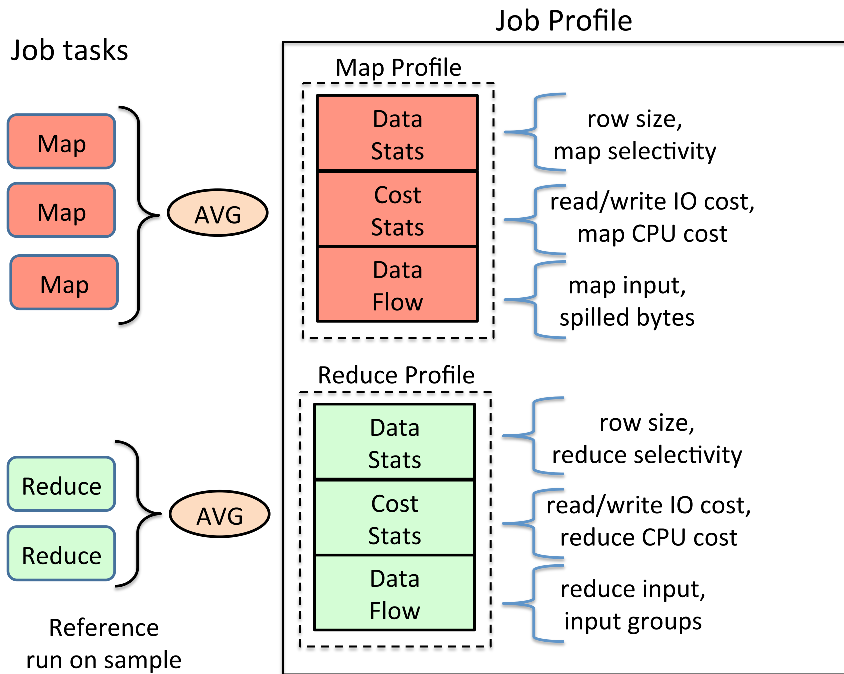


Figure 5.5 – Starfish reference profiles summarize the processing characteristics of a MapReduce job.

Starfish's What-If Engine: TITAN is built on top of Starfish's What-If engine that was designed to estimate the runtime of ETL workloads with the end goal of workload tuning, i.e., given a MapReduce job, an input dataset and a cluster deployment, Starfish finds the best set of configuration settings that optimizes the job performance. The key building block of the What-If engine is the *job profile*, which summarizes the processing characteristics of an input job. The processing characteristics are grouped into: processing cost factors (or cost statistics), data statistics, and data flow information. Figure 5.5 illustrates the processing characteristics that are collected in the job profile when executing the input job on a *sample* of the input dataset. Processing cost factors include per tuple or per byte costs for executing MapReduce phases (e.g., per tuple map cost, per tuple reduce cost, per byte cost of reading from disk / HDFS, etc), data statistics include row sizes and per phase selectivities (e.g., map row size, map selectivity, reduce selectivity, etc), while data flow information include data characteristics as captured at different points in the query pipeline (e.g., map input/output records, map input/output size, reduce input groups, etc). Processing characteristics are collected for each task executing the MapReduce job, then they are summarized into a set of *reference profiles* corresponding to each task type (i.e., one map profile and one reduce profile; for multi-input operators, such as joins, one profile per logical input is built). Starfish uses the *task uniformity assumption* for computing the reference profiles, that is, processing cost factors and data statistics are *averaged* uniformly among all tasks of the same type.

Once the reference profiles are built the What-If engine can be invoked to estimate the runtime

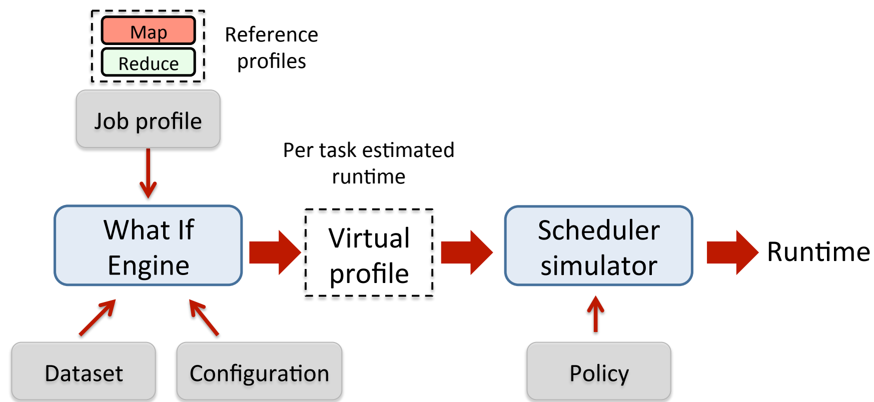


Figure 5.6 – Runtime prediction using Starfish's What-If engine.

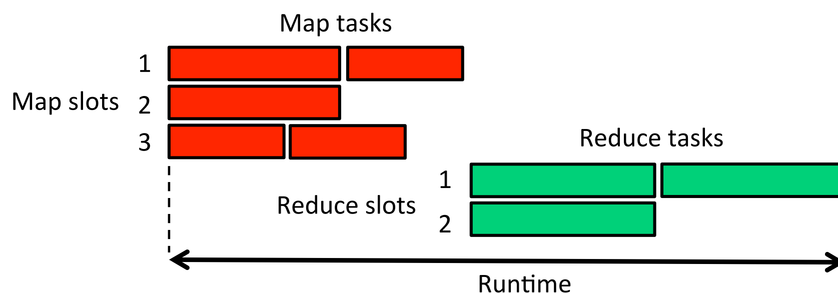


Figure 5.7 – Critical path modeling for a MapReduce job with five map tasks and three reduce tasks.

of the job on a larger input dataset. The prediction steps used by Starfish's What-If engine are illustrated in Figure 5.6. In the first step, the What-If engine estimates the number of tasks and it computes the data flow information of each task using some basic information about the input dataset that will be processed (e.g., input size, chunk size, # of input files), the data statistics taken from the reference profiles, and the configuration settings. Then, it computes the duration of each task using a set of analytical formulas that take into consideration the data flow information, the processing costs of each MapReduce phase as taken from the reference profiles, and the set of configuration parameters. A scheduler simulator is used to provide an ordering of tasks according to a scheduling policy (e.g., FIFO, Fair, etc) and a resource allocation configuration (e.g., number of map/reduce slots). The runtime of the job is the longest sequence of map and reduce tasks (also known as the critical path of the job). Figure 5.7 shows an ordering of tasks for a MapReduce job with five map tasks, three reduce tasks, and a resource allocation with three map slots and two reduce slots. Two map profiles, and one reduce profile were used as reference in the What-If engine for this example.

Localized Training based Models: As we had shown with experiments in Section 5.2.3, the task uniformity assumption used in Starfish introduces significant errors for reporting SQL

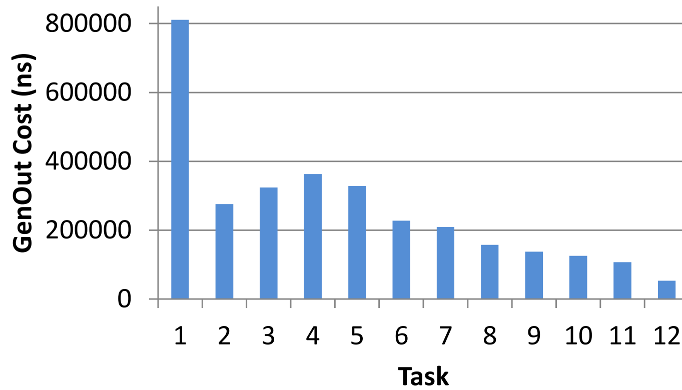


Figure 5.8 – $GenOut$ processing cost variation with the number of rows processed.

workloads. One of the sources of error is that per tuple processing costs corresponding to different processing phases of the MapReduce job, as used in Starfish What-If engine, vary the workload characteristics of *each task* (e.g., vary with the level of batching, and the tuple size). For instance, Figure 5.8 shows the $GenOut$ processing cost (i.e., the cost of joining and outputting tuples) corresponding to all the concurrent tasks executing a common join query in HiveQL of the form *SELECT B.VAL from A JOIN B ON (A.KEY = B.KEY)* on two synthetic tables of sizes of 1GB and 5GB (for the experimental setup please see Section 5.6.1). As different tasks of the job process a different number of rows (i.e., different level of batching), per tuple processing costs can vary heavily among different tasks (up to a factor of 15 for this experiment). Hence, the average per tuple cost, computed as the *average* of all $GenOut$ processing cost observations, is prone to either under-predict or to over-predict the actual runtime.

Therefore, instead of taking the average processing cost factor among all task instances executing the operator phase, which would either under-predict or over-predict the runtime of the phase, we propose to model it using a learning approach. Figure 5.9 shows a decision tree example that models the $GenOut$ processing cost of the common join operator as a function of the input workload characteristics. Each internal node in the tree splits the training samples of the node into two subsets according to an input feature value that categorizes the samples the best. The leaf nodes correspond to the output variable (i.e., estimated processing cost). For example, for $OutRows > 1000$, and $OutBytes < 48MB$, the GenOut cost is $350,000$ ns. For $OutRows > 1000$, $OutBytes > 48MB$, and $OutRowSize > 100B$, the GenOut cost is $200,000$ ns, etc. Hence, in contrast with Starfish and with calibration based approaches which take the *average* cost value among all task instances executing the same operator phase (e.g., [38, 81]), TITAN models *multiple* processing cost values for each operator phase by taking into consideration the characteristics of the input workload.

Global Analytical Models: We motivate a hybrid prediction approach that uses a global analytical model for the following reasons: i) Given that input data properties are available,

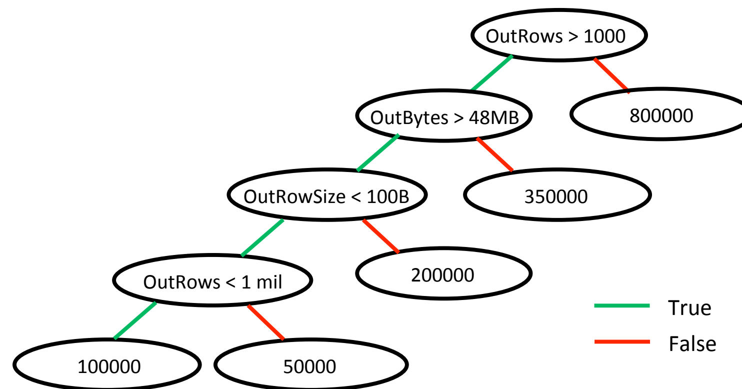


Figure 5.9 – Decision tree example for modeling the *GenOut* cost factor.

certain runtime decisions can be modeled precisely using analytical models: For instance, in a MapReduce job, the spilling phase occurs only if the amount of data collected into the memory buffer of the map phase is higher than a threshold value (i.e., $io.sort.mb$) [36]. While the analytical rule is straightforward, the alternative learning approach requires multiple executions with different input feature combinations to build a model. Additionally, for the case that the configuration threshold changes, new training executions are required. ii) Adopting a machine learning approach at coarse granularity (e.g., operator pipeline, job, query) requires more training samples than when it is used at fine granularity (e.g., operator phase) as the model needs to also capture the logic of the global cost model implicitly.

5.3.3 Localized Training based Models

In this Section we describe in detail the localized training based models that are used to model the processing cost factors of SQL operators and MapReduce specific phases. In particular, we present the model fitting mechanism, and the key input features that we use.

Model Fitting: We use decision trees to fit the processing cost of each MapReduce specific phase or join operator phase. Decision trees are a good modeling approach when the underlying dependency among the input features and the output feature is not known in advance or when the dependency does not follow a fixed functional form. Decision trees are thus general and applicable to a large class of prediction problems. A large number of fitting algorithms based on decision trees exist [35]. We choose Classification and Regression Trees (CART) as our default algorithm due to their generality, practicality, and the fact that they do not have minimum requirements regarding the training set size.

Input Features: Two categories of input features are used to fit the prediction models: MapReduce specific features, used for estimating the MapReduce phases (i.e., read, write, sort, merge, spill, shuffle), and operator specific features used for estimating the HiveQL operators executed inside the map() and reduce() functions. We choose features based on a mix of domain

knowledge about the semantics of the operators and experimentation. Table 5.1 summarizes the input/output features used for modeling MapReduce specific phases, while Table 5.2 shows the features used for modeling join operators. We make several modeling design choices: i) We do not use configuration settings as input features into the learning models. Instead, we model the impact of configuration settings analytically. For instance, the number of map and reduce slots are used as input by the scheduler simulator, memory buffer sizes are used as input into the analytical model (i.e., analytical rule) which decides whether a specific MapReduce phase will occur or not (e.g., spilling, merging, etc). ii) For multi input operators (i.e., joins) we break down the operator execution into multiple phases (e.g., hash table loading phase, streaming input phase, generating output phase), such that the processing cost of each phase can be normalized by the number of input/output tuples, and thus, can be modeled separately.

Name	Feature Type
Map/Reduce Input/Output Records	Input
Map/Reduce Input/Output Bytes	Input
Reduce Shuffle Bytes	Input
Spilled Records	Input
Spilled Record Size	Input
Records Per Buffer Spill	Input
Local Bytes Read/Written	Input
HDFS Bytes Read/Written	Input
Read/Write local cost (per byte)	Output
Read/Write HDFS cost (per byte)	Output
NetworkCost (per byte)	Output
PartitioningCost (per record)	Output
SortCost (per record)	Output
MergeCost (per record)	Output

Table 5.1 – MapReduce specific features

Limitations: Through the set of input features described in the previous section TITAN addresses the runtime prediction problem for distributed queries running with dedicated CPU and memory resources, and with uniformly shared disk bandwidth among concurrent tasks. While modeling inter-query interference is outside of the scope of this chapter, we note that the hybrid prediction approach TITAN proposes is extensible to support the impact of inter-query interference through a set of additional features such as the mix of query operators executing concurrently, altogether with the corresponding slow-down factor on to the processing cost of the operator. Such features for modeling interference among concurrent query workloads were proposed in the context of PostgreSQL [4, 25].

Name	Description	Notes
InBytes/OutBytes	Input/output bytes	all
InRows/OutRows	Input/output rows	all
InRowSize/OutRowSize	Avg. row size	all
FilteredBytes	Join input bytes	all
FilteredRows	Join input rows	all
NumProject	# of projections	all
HashEntries	Hash entries	Map join
HashEntrySize	Hash entry size	Map join
HashSize	Hash size	Map join
LoadedTuples	Loaded tuples	Map join
LoadedBytes	Loaded bytes	Map join
MapInitCost	initialization cost	all
ReduceInitCost	initialization cost	all
StreamInCost	streaming phase	all
GenOutCost	joining phase	all
BuildHashCost	hash building	Map join
LoadHashCost	hash loading	Map join

Table 5.2 – Join specific features

5.3.4 Global Analytical Models

Cost factors estimated through localized machine learning models are plugged into a global analytical model to compute the end to end runtime. Configuration settings such as the number of map/reduce slots, buffer sizes, threshold memory sizes (e.g., that control when spilling occurs), are taken as input into the analytical model as shown in Figure 5.4 and summarized in Table 5.3.

We extend Starfish's performance models [36] proposed in the context of ETL workloads to make them HiveQL operator aware. In particular, we model the map(), and reduce() functions by taking into consideration the HiveQL operator(s) they execute. In the following we present as an example the analytical models we use to model the runtime of two types of joins in HiveQL: the map join and the common join.

Map join: The map join resembles the hash join from traditional DBMS and it is entirely executed during the map phase of the job (map only job). When using the map join, it is assumed that one of the input tables is small enough that can be buffered entirely into the memory of each map task. We split the map function into three phases: initialization (i.e., Init), hash loading (i.e., HashLd), streaming inputs (i.e., StreamIn), and generating outputs (i.e., GenOut). During the initialization phase, input tuples are deserialized, filtered horizontally and vertically (select and filter operators). In the HashLoad phase the hashtable corresponding to the small input is read and stored into the memory. During the StreamIn phase, input tuples from the input tables are saved into an in memory buffer if the probe operation finds matching

Name	Description
MapSlots	# of map slots
ReduceSlots	# of reduce slots
TaskMem	Map/reduce task memory
IoSortMb	Buffer size for sorting files
IoSortFactor	# of files to merge at once
IoSortSpillPercent	Threshold value that triggers the spill
IoSortRecordPercent	Memory quota for storing record metadata
InmemMergeThreshold	# of map output files that trigger the merge
ShuffleInputBufferPercent	Percentage of heap allocated for storing map output
ShuffleMergePercent	Memory quota that triggers the in memory merge
ReduceInputBufferPercent	Maximum quota for map output (reduce phase)

Table 5.3 – Examples of configuration settings considered in the analytical model.

tuples into the hash. During the GenOut phase, result sets are generated and forwarded to the file sink operator which outputs the results. We note that HiveQL uses serialization/deserialization interface (i.e., SerDe) to serialize/deserialize tuples at the end/beginning of the processing pipeline. In contrast with traditional DBMS, the end of a processing pipeline in MapReduce is determined by the end of the map/reduce phases as they materialize intermediate results to disk. The equation we use to compute the runtime of the map function when executing the map join is the following:

$$Runtime_{Map,MJ} = MapInRec \times c_{Init} + HashSz \times c_{HashLd} + (InRows_{LargeTab} + LoadedTuples_{SmallTab}) \times c_{StreamIn} + OutRows \times c_{GenOut} \quad (5.1)$$

where c_{Init}, c_{HashLd}, $c_{StreamIn}$, and c_{GenOut} are the normalized processing costs of each phase, $MapInRec$ and $InRows_{LargeTab}$ are the number of unfiltered / filtered tuples from the large table, $HashSz$ is the hash size, $LoadedTuples_{SmallTab}$ is the total number of matching tuples from the small table that were successfully probed, and $OutRows$ is the number of tuples of the result set.

Common join: The common join resembles the traditional repartitioning join in MapReduce. In the map phase the two input tables are read, filtered, and partitioned on the joining key. Tuples corresponding to the same joining key are sent to one single reducer which performs the join in the reduce phase. We similarly split the reduce function into three phases: *Init*,

StreamIn, and *GenOut*. There are several differences when compared with the map join: i) Filtering input tuples is performed during the map phase of the job, not during the Init phase of the join; ii) Input tuples from all the input tables are sorted on key, so each input tuple is loaded into the in memory buffers only once (i.e., no probe operation exists); iii) Results are forwarded to the file sink operator after all the input tuples corresponding to the current join key were buffered, or alternatively after all tuples corresponding to the n-1 tables, and additionally a threshold number of tuples from the last table were buffered. Hence, result sets corresponding to a join key are generated at once in the same batch, or in multiple batches, in contrast with the map join where result sets are generated after each successful probe operation. The equation we use to compute the runtime of the reduce function when executing the common join is:

$$\begin{aligned} Runtime_{Reduce,CJ} = RedInRec \times c_{Init} + (&InRows_{LargeTab} \\ &+ InRows_{SmallTab}) \times c_{StreamIn} + OutRows \times c_{GenOut} \end{aligned} \quad (5.2)$$

5.4 Training Methodology

In this Section we propose a training methodology targeted to reduce the training time in terms of running benchmark queries compared with extensive training with state of the art analytical benchmarks adopted in prior work (e.g., TPC-H used in [50, 6], and TPC-DS used in [29]) without sacrificing on accuracy.

5.4.1 Query Template Pruning

Starting from a large set of training queries we propose an iterative query pruning procedure as shown by Algorithm 1. In the first iteration, the training set T is instantiated with a random sample of n_0 queries from the full set of benchmark queries Q. Then, in each subsequent iteration the training set size is augmented progressively as long as the accuracy of the prediction models improves beyond a threshold value th. To quantify the model accuracy improvement of each iteration we use *k-fold cross validation*, a widely used method for estimating the prediction error [35]. K-fold cross validation divides the training set into k folds of approximately of equal size, then uses k-1 folds to train the models and uses one fold to test the models. The process is repeated k times, for each possible train/test folds combination. For all the queries of the testing fold F_i the aggregated prediction error is computed as the squared error of the predicted value w.r.t. the actual value (line 13). Then the cross validation estimate of the prediction error is computed on line 15. Finally, the rate of accuracy improvement is computed as the ratio of the cross validation estimate of the previous iteration to the cross validation estimate of the current iteration. With respect to the sampling procedure we use progressive sampling with geometric rate (i.e., $2^i * n_0$, line 21) inspired by the work of Provost et al. [64] which shows that geometric sampling is remarkably efficient in finding the convergence plateau in a few number of steps (compared with linear sampling). While the algorithm

requires to execute two times more queries than the minimum (i.e., in order to compare the model improvement with the previous step), all sub-sequent re-training phases benefit from a smaller subset of queries that have to be re-executed.

Algorithm 1 Iterative query pruning procedure. Notations: initial number of training queries n_0, query set Q, number of folds k, convergence threshold th, number of training queries n, training query set T, iteration it, rate of accuracy improvement R, cross validation error CV.

1: Input: n_0, Q, k, th
2: $n = n_0$, $T = \emptyset$, $R = 0$, $it = 1$
3: **while** $n \leq |Q|$ and ($it \leq 2$ or $R > th$) **do**
4:   **if** $it == 1$ **then**
5:     $T = \{$random set of n queries from $Q\}$
6:   **else**
7:     $T = T \cup \{$random set of $n/2$ queries from $\{Q - T\}\}$
8:   **end if**
9:   separate T into k folds F_i, $1 <= i <= k$
10:   **for** $i = 1; i <= k; i++$ **do**
11:     train models with all F_l s.t. $l <> i$, $1 <= l <= k$
12:     test models on queries Q_j from fold F_i
13:     $CV_i = \sum_{j=1}^{|F_i|}(Predicted_j - Actual_j)^2$
14:   **end for**
15:   $CV^{it} = \frac{1}{n}\sum_{i=1}^{k} CV_i$
16:   **if** $it >= 2$ **then**
17:     $R = CV^{it-1}/CV^{it}$
18:   **end if**
19:   $it = it + 1$
20:   $n = n \times 2$
21: **end while**

5.4.2 Synthetic Query Generation

Assuming that some minimal information about the testing workload is available (i.e., schema information, query operators in the workload) we suggest a methodology for generating synthetic datasets and queries that can be used as input into the pruning algorithm. As we show with experiments, synthetically generated benchmark queries can be more concise (have less training data overlap) than state of the art query template instances and reduce further the training time. In the following we summarize the ideas we use for generating synthetic benchmark queries and present the key differences compared with state of the art training that uses TPC-H/-DS query templates. A synthetic workload instance that follows the methodlogy described bellow is presented in Section 5.6.1.

- **Synthetic queries and data**: We generate *both* synthetic datasets and synthetic queries to make the training methodology generally applicable for a larger set of testing workloads instead of using an existing benchmark that is tight to a fixed schema.

- **Reducing overlap**: In order to remove unnecessary profile data overlap that inherently occurs for long running workflows, queries are generated such that they include only short pipelines of query operators. For instance, a TPCH query includes multiple instances of the same operator implementation in one single query. While we also profile any given operator multiple times, we explicitly map different executions of the operator to *different* input data characteristics (e.g., input sizes, row sizes, data distributions).

- **Fine granularity, systematic training**: Queries are generated such that all processing phases that occur in the workload execution are covered for a *range* of row level data properties established during the workload characterization phase (e.g., the join operator processes input tuples with sizes in the range of 100 to 200 bytes). Building training datasets for each operator in isolation reduces the number of queries required for training because the number of operators is limited, unlike the number of potential queries which is unbounded.

- **Task heterogeneity / Data skew**: Given the MapReduce execution model where multiple tasks are executed in parallel, task heterogeneity in terms of data processing requirements is effective by capturing different "execution patterns" in one single query. Hence, we propose to execute synthetic queries on skewed datasets such that different tasks process a different number of rows.

- **Reducing materialization**: Training queries are generated such that the number of MapReduce jobs in a query is minimized. Proceeding this way we aim to reduce the cost of materializing intermediate results.

5.5 Translation Models

Translation models are prediction models that estimate the runtime of an operator given a set of input features that include the performance metrics of a *reference operator* corresponding to a *different* execution setting (i.e., different implementation). As we show in this section, there are multiple scenarios from practice where such models are applicable and can be used to boost prediction accuracy by exploiting prior executions of the workload.

5.5.1 Semantics

For a given operator phase, we build translation models of the form: $C_{e2} = T_{e1 \to e2}(D_{e2}, C_{e1}, D_{e1})$, where C_{e2}, D_{e2} are the processing cost factor and the data features corresponding to the execution setting $e2$, while C_{e1}, D_{e1} are the processing cost and the data features of the operator corresponding to the execution setting $e1$. In contrast to conventional models, the translation model estimates the processing cost of the operator phase for the execution setting $e2$ using not only the workload features of its execution setting but also performance and data features of the reference execution: E.g., estimate the $GenOut_{MJ}$ processing cost of the map join, given

the $GenOut_{CJ}$ processing cost and the data features of a reference common join.

To make translation models applicable for testing workloads outside of the training boundaries, we build *relative translation models*, which learn relative cost ratios instead of absolute cost values: E.g., For data feature vector D_k, the map join is 2x faster than the corresponding common join; for data feature vector D_l, the map join and the common join have the same performance, etc. Then, using the cost ratio, $r_{MJ/CJ}$ and the cost of the common join t_{CJ}, the cost of the map join can be simply computed as the product of the two. To a certain degree, the relative translation model is similar to the cost modeling technique proposed in query optimization where each processing cost is expressed as a relative ratio w.r.t. the processing cost of the sequential IO (e.g., the cost of random read = 4x cost of sequential read in PostgreSQL, etc). The main difference is that such relative relations are *learned* for multiple zones of the input data feature space and applied only for *similar processing pipelines* operating under two different execution conditions.

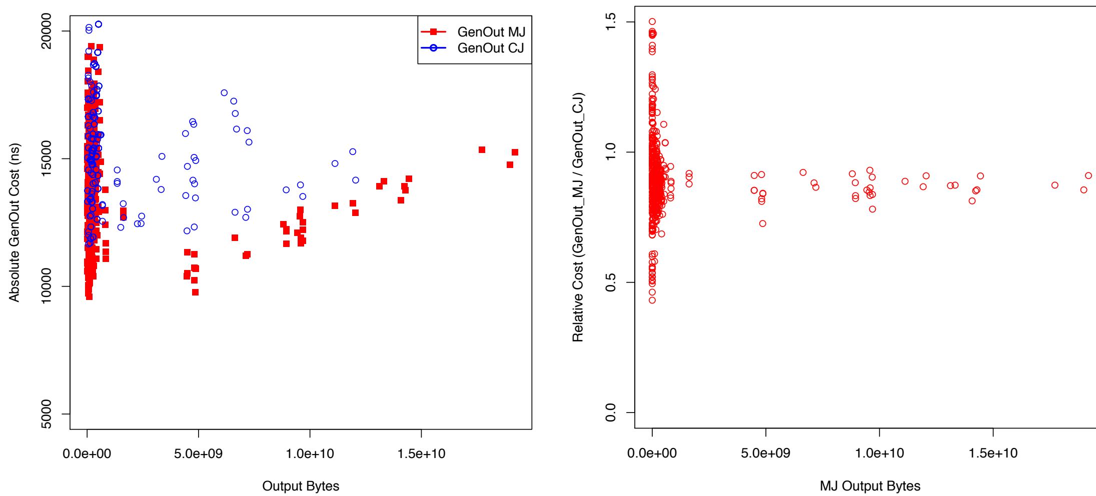


Figure 5.10 – a) Absolute GenOut cost as a function of output bytes (left), and b) Relative GenOut cost: $GenOut_{MJ/CJ}$ (right).

Figure 5.10 illustrates the intuition of the relative translation models with an example. Specifically, Figure 5.10 a) shows the absolute $GenOut$ cost of processing output tuples for the MJ and CJ operators, while b) shows the cost of generating output tuples for the MJ relative to the cost of the CJ: $GenOut_{MJ/CJ}$=$GenOut_{MJ}$ / $GenOut_{CJ}$. We observe that the absolute processing cost values vary over the span of output byte values (as they are dependent on other input features such as the output row size, and level of batching). On the other hand, the relative processing cost $GenOut_{MJ/CJ}$ is more stable. The reason is that corresponding tasks of the MJ and CJ executing the $GenOut$ phase of the join have similar data characteristics. Hence, the relative cost ratio is less dependent on the workload characteristics and more dependent on the actual performance of the two operator implementations. As we show with experiments in Section 5.6, performance ratios are safer to use when the testing workload is not well covered by the training data.

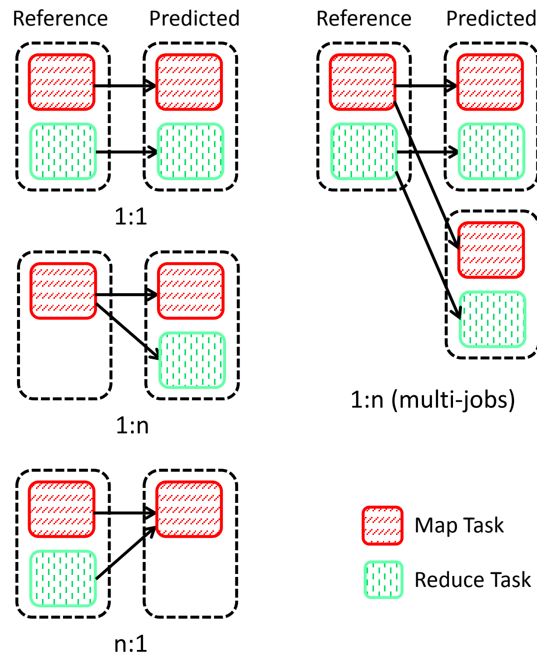


Figure 5.11 – Type of code/data mappings among MapReduce tasks.

5.5.2 Operator Phase and Data Mappings

Two steps are required to apply translation models for operator runtime prediction: i) phase mapping: identifying corresponding task phases that have similar semantics; ii) input mapping: mapping task phases operating on inputs with similar row level data properties for multi-input operators (i.e., joins).

Phase mappings: In the MapReduce ecosystem, an operator (or a set of packed operators) is executed as one or multiple MapReduce jobs, where each job consists of multiple tasks which are executed in parallel. Therefore, phase and data mappings have to be addressed among corresponding task phases. Figure 5.11 illustrates possible phase mappings among MapReduce tasks executing HiveQL operators. We observe that several phase mappings are possible: i.e., 1:1 mapping, n:1 mapping, and 1:n mapping. In general, each task can have multiple reference tasks, and any given task can serve as a reference for multiple other tasks. The type of input mapping used depends on the operator implementation, and the execution setting scenario.

Input mappings: In this step we map tasks operating on inputs with *similar row characteristics*. We note that in practice it is highly unlikely to map tasks operating *exactly* on the same input partition due to different data distribution, different operator placement (e.g., inside map vs. inside map and reduce), and different configuration settings (e.g., the number of map slots controls the number of input splits seen by each map task). Hence, we only target to maintain the same logical input for multi-input operators, which in turn maintains the same row level

data properties and schema.

5.5.3 Use Cases

In the following we present several use cases from practice where we can apply translation models. For each case in turn we explain the model semantics, and the phase and data mappings we perform.

Common Join to Map Join

The common join to map join model estimates the runtime performance of a map join operator given the data, and performance characteristics of the corresponding common join operator. We recall that the map join operator performs the join inside the map task of the MapReduce job (map only job) by joining input rows from the large input with corresponding rows from the small input(s) that are probed from the local memory of each task. On the other hand, the common join repartitions the input tables on the joining key in the map phase of the job, before starting to join tuples in the reduce phase. While the two implementations are different at a global level, a code mapping among join sub-phases can be performed at a semantic level: In essence, each join implementation can be split into two phases: i) *streaming phase*: in this phase the joining tables are fetched from the downstream operators that are filtering the tables horizontally and vertically (i.e., project/select), and stored into the memory. ii) *joining phase*: in this phase the tables are joined and the output tuples forwarded to the file sink operator that materializes them on disk. We note that in HiveQL, the code fragment corresponding to this phase is shared among all the join implementations.

Phase/Data Mappings: In terms of phase mappings, we map the streaming phase of the common join (occurring in two sub steps inside the map and reduce tasks) to the streaming phase of map join (occurring inside the map task), and the join phase of the common join (occurring in the reduce task) to the corresponding join phase of map join (occurring in the map task). In terms of data mappings, we target to match tasks based on the logical input tables they process. For homogeneous task mappings (i.e., map/map, and reduce/reduce) matching tasks operating on the same logical input(s) is possible. For heterogeneous task mappings (reduce/map, map/reduce) where a 1:1 mapping is not possible, we relax the matching problem to task phases operating on a portion of the same logical input(s). Figure 5.11, n:1 case, shows the task mappings for the $T_{CJ \to MJ}$ model.

Map Join to Common Join

The map join to common join model is the reverse of the model above: It estimates the runtime performance of a common operator given the data, and performance characteristics of the corresponding map join operator.

Phase/Data Mappings: Figure 5.11, n:1 case, shows the code/data mappings for $T_{MJ \to CJ}$.

Common Join and Map Join to Skewed Join

The skew join is essentially a hybrid of a common join operator and a map join operator which balances the load of generating result sets for a skewed key from one task to multiple tasks. The skew join is composed of two jobs: the first job is a common join, with the difference that skewed keys are identified at runtime and result sets are generated only for the join keys that are not skewed, while the second job is a map join which performs the join of the skewed keys identified in the previous job.

For this use case, the translation model estimates the performance of a skewed join operator given the data, and the performance characteristics of *two* reference joins: the corresponding common join, and map join operators. Two translation models are used: $T_{CJ \to CJ}$ and $T_{MJ \to MJ}$, hence, this use case corresponds to a *composable translation model*.

Phase/Data Mappings: For this use case we have identity mapping at the phase level: corresponding streaming phases and join phases are mapped among the same join operator types (MJ to MJ and CJ to CJ). For data input mappings we match tasks operating on inputs with similar row characteristics. Figure 5.11, 1:1, shows the code/data mappings for $T_{CJ,MJ \to SJ}$.

5.6 Evaluation

5.6.1 Setup and Methodology

Hardware and Software Setup: We evaluate prediction models on two setups: on a private cluster of eight machines, and on the public cloud, i.e., Amazon EC2 instances. Our private cluster consists of eight nodes, where each node has two six-core CPUs Intel X5660 @ 2.80GHz, 48 GB RAM and 1 Gbps network bandwidth. All experiments were run on top of HiveQL 0.11.0, and Hadoop 1.0.3 as the underlying MapReduce framework. Unless specified otherwise each node was set with a maximum capacity of eight mappers and two reducers, where each task was allocated with 4GB of memory. Hence, our setup has a total capacity of 64 map slots and 16 reduce slots.

We validate part of our experiments on three deployments of Amazon EC2 instances: *Deployment A*: Cluster of ten slave nodes, $m2.2xlarge$ instances, and one master, $m1.xlarge$. Each slave node was set with a maximum of four mappers and two reducers, where each task was allocated with 6GB of memory. *Deployment B*: Cluster of ten slave nodes, $m2.4xlarge$ instances, and one master, $m2.2xlarge$. Each slave node was set with a maximum of eight mappers and four reducers, where each task was allocated with 6GB of memory. *Deployment C*: Cluster of twenty slave nodes, $m2.4xlarge$ instances, and one master, $m2.2xlarge$. Each slave node was set with a maximum of eight mappers and four reducers, where each task was allocated with 6GB of memory.

Workloads: We evaluate our approach on TPC-H [72], state of the art decision support benchmark, a selection of TPC-DS [71] queries, and on synthetic workloads. The TPC-H workload consists of all queries except queries which either do not contain any joins (Q1, Q6) or they include outer joins, currently not supported by our prediction framework (Q13, Q21, Q22). We execute the TPC-H queries on five scaling factors: 1, 10, 30, 60 and 100, and we use the skewed version of the TPC-H data generator [1] with a Zipf skew factor of 2. The TPC-DS workload consists of a selection of join queries with different resource requirements, i.e., Q3, Q7, Q15, Q19, Q24, Q25, Q26, Q29, and Q82, selection that was inspired by the TPC-DS workload analysis paper [62]. Similarly with the TPC-H workload, we execute the selection of TPC-DS queries on five scaling factors: 1, 10, 30, 60, and 100.

Synthetic Workload (i.e., MBench): For training prediction models we generate a synthetic workload following the methodology described in Section 5.4. Based on TPC-H schema information we generate twelve tables with row sizes ranging between 50 and 200 bytes, and join keys with both uniform and skewed distributions, with a skew factor ranging between 0 and 2. Then, we use a query template of the following form for generating query instances: *SELECT A.KEY, A.VAL, B.VAL from A JOIN B ON (A.KEY = B.KEY and A.KEY > t1 and A.KEY < t2)*, where we varied: the joined tables (20 possibilities), the selected columns (2 possibilities), and the selectivity of the query (3 possibilities: 0.1%, 0.5% and 1.0% of total range of values). The resulting workload included = $20 \times 2 \times 3 = 120$ queries, which were executed and profiled. The workload instance above shall not be interpreted as complete for any testing scenario; given more information about the testing workload and execution setting, more queries shall be generated to ensure that the testing space is well covered. The pruning algorithm is in charge of actually determining the minimum number of query instances that are actually used in training. For evaluating the effectiveness of translation models outside of the training boundaries (in Section 5.6.3), we build a second instance of the synthetic workload above where instead of generating tables with rows sizes ranging in between 50 and 200 bytes, we generate tables with rows sizes ranging in between 500 and 2000 bytes. The synthetic tables that we generate have sizes in between 500MB and 10GB.

Prediction Schemes: In addition to **Starfish**, we compare the accuracy of the models with **MART** [50], the best performing prediction approach at operator granularity proposed in the context of DBMS, and **KCCA** [28] the machine learning approach that was proposed both in the context of DBMS and MapReduce.

• Starfish: Prediction approach that uses reference profiles, analytical modeling, and simulation to compute runtime estimates, as summarized in Section 5.3.2. For both TPC-H and TPC-DS workloads, we used the query profiles corresponding to the workload execution on scaling factor ten as the *reference* query profiles, that are used as input into Starfish's What-If engine. The What-If engine is then invoked to estimate the runtime of the workload on all the other scaling factors.

• MART: Prediction approach that builds prediction models at *operator granularity* and uses

Multiple Additive Regression Trees as the underlying fitting mechanism (for details about the model fitting algorithm please see Section 2.6.3). We use the same set of input features as for TITAN.

• KCCA: Prediction approach that builds prediction models at *job granularity* and uses Kernel Canonical Correlation Analysis as the underlying model fitting mechanism. We input configuration parameters and data characteristics as the input features into the models as suggested in the work of Ganapathi et al. [28].

• TITAN: Our proposed prediction approach that builds prediction models at *operator phase granularity*. In contrast with MART and KCCA, TITAN's models are used to estimate per phase *processing costs* instead of absolute runtime. Runtime is computed analytically using the analytical model presented in Section 5.3.4. By default, TITAN uses Classification and Regression Trees (CART) as the underlying model fitting mechanism.

For performing sensitivity analysis with respect to the *level of granularity of the models*, and to assess the usefulness of *translation models* outside of the training boundaries, we implement multiple prediction policies with different modeling characteristics as follows. Unless specified otherwise, we use Classification and Regression Trees (CART) as the underlying model fitting mechanism.

• Predicted Fine (Pred-Fine): It is equivalent with TITAN.

• Predicted TM (Pred-TM): Prediction approach that uses *translation models* for estimating the processing costs of HiveQL operator phases inside the map(), and reduce() functions, and MapReduce phase models for estimating the processing costs of all the other phases.

• Predicted Coarse (Pred-Coarse): In contrast with Pred-Fine, which splits the map(), and reduce() functions into multiple operator phases, and then builds one model per operator phase, Pred-Coarse uses one single model for modeling the runtime of map() function, and one single model for modeling the runtime of reduce() function. Additionally, these models are built to estimate runtime instead of processing costs. The set of input features is shared with Pred-Fine.

• Predicted MapReduce (Pred-MR): Prediction approach that uses only MapReduce specific features for estimating the processing costs of *all* MapReduce phases: i.e., the models do not use HiveQL specific features, thus, they are agnostic to the HiveQL operator semantics of the underlying MapReduce job. One model per MapReduce phase is built (i.e., the granularity of phases is as follows: read, map, spill, merge, collect, write, shuffle, sort, reduce, write).

With the purpose of identifying the sources of errors in the prediction models above, we also show results for several "played back" policies. They are only used to *quantify the errors* introduced by the scheduler simulator and the analytical model, and *shall not be interpreted as valid prediction policies.*

• Scheduler simulator (PB-Sched): Played back policy that uses the *actual* runtime of tasks as input into the scheduler simulator of Starfish. The scheme shows the errors caused by the scheduling assumptions.

• Analytical Model (PB-AM): Played back policy that uses the *actual* processing cost factors corresponding to each MapReduce phase. The scheme shows the cumulated errors introduced by the scheduler simulator and the analytical model.

• Analytical Model Average (PB-AM-AVG): Played back policy that uses the *average* processing cost factors corresponding to each MapReduce phase as computed and summarized into Starfish's reference profiles. Thus, this policy shows the cumulated errors introduced by the scheduler simulator, the analytical model, and the task uniformity assumption.

Prediction Metrics: We evaluate the accuracy of the prediction schemes above using a number of metrics: relative prediction error (RE), ratio error, cumulative distribution of relative error (CDF), and order preserving degree (OPD) as summarized in Section 2.6.5. The order preserving degree in our context measures the proportion of predictions for which the relative order among the estimated runtime values of the workload corresponding to *alternative execution settings* is maintained the same with the relative order among the actual runtime values.

For pinpointing the errors that are on the *critical path* of a task, we introduce the *task normalized relative error (TNE)* of a MapReduce phase as: $|Pred_{phase} - Actual_{phase}|/Actual_{task}$. This metric is useful to identify MapReduce phases that are on the critical path and require model improvement.

Model Fitting: For the cases that we use Classification and Regression Trees (CART), we set the minimum split to 10, the complexity parameter to 0.005, and the number of cross validation folds to three. For the cases that we use Multiple Additive Regression Trees (MART), we set a number of 1000 boosting iterations, and a minimum number of ten observations in the trees terminal nodes, as suggested by prior work [50]. Additionally, we use a shrinkage parameter of 0.01, ten cross validation folds, and a training fraction of 0.75. For the cases that we use Kernel Canonical Correlation Analysis (KCCA), we set the number of neighbors $k = 3$, and we use a Gaussian Kernel with a scale factor of $\sigma = 0.1$, as suggested by prior work in the context of query performance prediction [29].

Execution Settings: For each workload we consider two execution settings corresponding to alternative join implementations used in the workload execution:

• Common join (CJ): For this setting the common join implementation is used by all join operators of the workload.

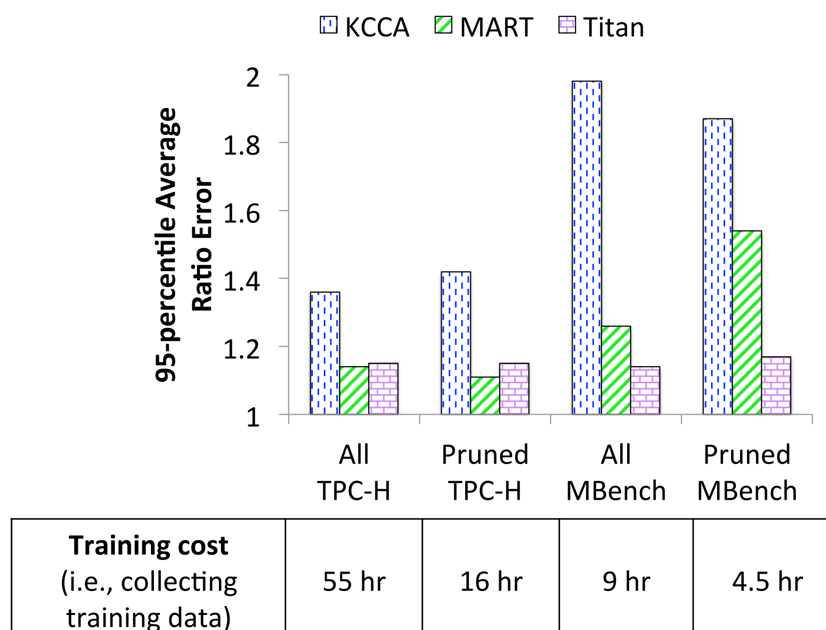
• Map join (MJ): For this setting part of the join operators of the workload use the map join implementation, while the rest use the common join. The reason that not only map joins are used is that not all of the joining operators have a small input table that can fit entirely into the memory.

We note that when predicting the runtime of a query corresponding to the CJ execution setting, the translation model uses a reference execution of the query corresponding to the MJ execution setting, and vice-versa.

5.6.2 Training Models

In this section we compare two training approaches for building prediction models: i) training with a pre-defined workload as proposed by state of the art training-based approaches [29, 6, 50]; ii) training with a synthetic workload as we propose in Section 5.4. For both cases, we analyze the trade-off among the models' accuracy versus the training cost when using the pruning algorithm that we propose in Section 5.4.1.

We clarify that the training cost is generally characterized by two components: i) the time to run training workloads and to collect training data, and ii) the time to build the prediction models (model fitting). In the context of runtime performance prediction for data analytics, the training cost is dominated by the first component. E.g., It ranges in between (tens of hours, few days) for running training workloads versus (tens of seconds, few minutes) for building the models. That is why our goal in this work is to reduce the time required to run training workloads. Unless specified otherwise, when using the term of *training cost*, we refer to the time required to run and collect training data. We note that the time required to build the models ranges between tens of seconds to a few minutes, depending on the amount of training data available, on the model fitting algorithm, and on the number of models that are built.



Training cost (i.e., collecting training data)	55 hr	16 hr	9 hr	4.5 hr

Figure 5.12 – Impact of pruning algorithm on prediction accuracy for training-based modeling approaches.

Training with Pre-defined Workload

For the first training approach, the training workload consists of TPC-H query instances that were randomly generated from TPC-H query templates using the TPC-H query generator (i.e., *qgen* tool). In our experiment, we randomly generate 25 queries per query template accounting for a total number of 425 query instances. We execute the workload of TPC-H query instances on a database of scaling factor 10, we collect the corresponding execution profiles, then we build prediction models for two training scenarios: i) training with all the 425 query instances; ii) training with a subset of query instances, as determined by the pruning algorithm. We set the parameters for the pruning algorithm (described in Section 5.4.1) as follows: $n_0 = 15$ queries, model improvement threshold to $th = 1.10$, and the number of cross validation folds to $k = 5$. We run the pruning algorithm with TITAN as the underlying prediction policy. We evaluate the accuracy of the models on the TPC-H workload consisting of *different* query instances than the training queries that were executed on scaling factors 1, 10, 30, 60, and 100, as described in Section 5.6.1.

The first two sets of bars of Figure 5.12 show the job level accuracy on the testing workload for the two training scenarios. In addition to TITAN, we show accuracy results for KCCA and MART, the two competing approaches that also use training-based models. We observe that while there is a small decrease in accuracy for KCCA, neither MART, nor TITAN worsen their accuracy significantly when using the pruned workload for training. Thus, the pruning algorithm can select the minimum number of training queries for a given threshold on accuracy (e.g., model improvement ratio < 1.10). In terms of the cost of running training queries, the pruning algorithm outputs a number of 120 queries, which accounts for 16 hours of the total benchmark time of 55 hours.

Training with Synthetic Workload

For the second training approach, the training workload consists of synthetic queries that were generated as described in Section 5.6.1 (i.e., the first instance of *MBench* workload). As for training with a pre-defined workload we test our models on the TPC-H workload for two training scenarios: i) training with all the micro-benchmark query instances, ii) training with the pruned queries as determined by the pruning algorithm. The last two sets of bars of Figure 5.12 show the job level accuracy results. While prediction models that model absolute runtime such as KCCA (job level models), and MART (operator level models) have less prediction accuracy as compared with the case of training with the TPC-H workload, TITAN, which models processing costs instead of absolute runtime, has similar accuracy for all training scenarios (i.e., having a ratio error of less than 1.15). In terms of the cost of running training queries, the pruning algorithm outputs a number of 60 queries, which accounts for 4.5 hours of the total benchmark time of 9 hours. Compared with the above approach that uses the TPC-H workload for training, the cost of running training queries for the synthetic workload case is further reduced by a factor of four.

We make several observations: i) The pruning algorithm and the hybrid modeling technique reduce the number of queries required for training for both the cases that a pre-defined workload or a synthetic workload are used in training. ii) The time required to run the complete set of synthetic queries was 9 hours on our local cluster, while the time to run the complete set of TPC-H query template instances lasted 55 hours. iii) Taking into consideration that only a subset of 60 synthetic query instances was enough to train TITAN's models (accounting for 4.5 hours of benchmarking), we use them as the default training benchmark for the rest of the experiments [1].

5.6.3 Testing Models

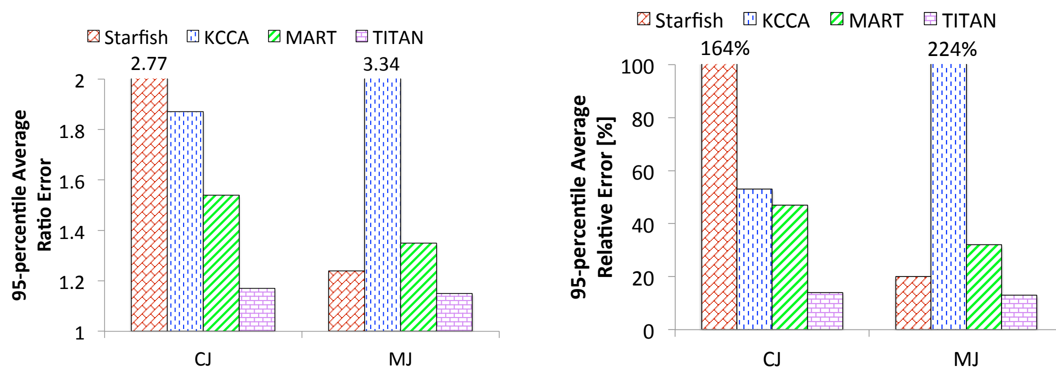


Figure 5.13 – Job level accuracy results for TPC-H workload: a) Job level 95-percentile average ratio error (left), and b) Job level 95-percentile average relative error (right).

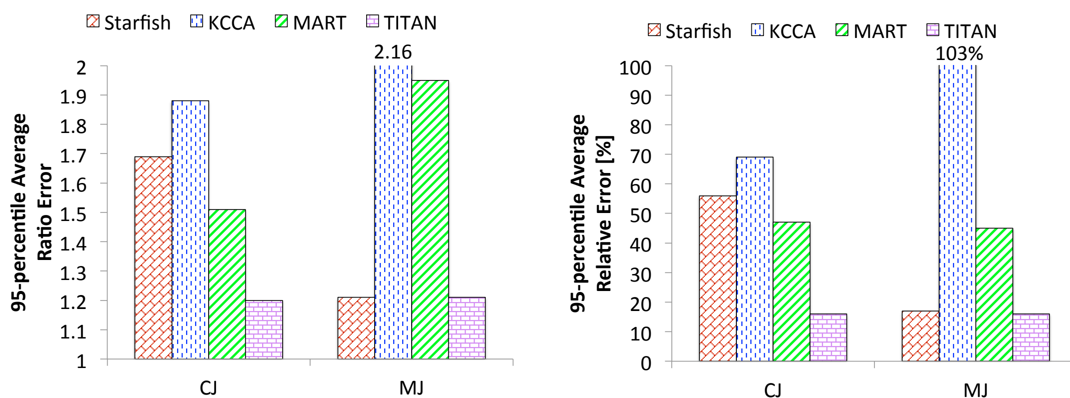


Figure 5.14 – Job level accuracy results for TPC-DS workload: a) Job level 95-percentile average ratio error (left), and b) Job level 95-percentile average relative error (right).

[1] We note that although we use a small number of queries in training, the number of training samples is larger (in the order of hundreds), and is equal with the total number of tasks that were used to execute the training queries on top of Hive.

TITAN versus Alternative Approaches

In this section we compare TITAN with alternative prediction approaches proposed in the literature: Starfish, the baseline for our proposed approach, MART, the best performing policy proposed in the context of DBMS, and KCCA, the machine learning approach that was proposed both in the context of DBMS and MapReduce.

In this experiment, we train prediction models with pruned micro-benchmark queries and we test them on the TPC-H workload. Figure 5.13 a) shows the 95-percentile average ratio error. The largest errors are observed for Starfish and the KCCA policy. Starfish has large estimation errors for the CJ execution setting. The main sources of error are: i) the analytical model, which was not designed to support non-linear operators (e.g., joins) and ii) the task uniformity assumption, which introduces significant errors when the MapReduce job is composed of tasks with very different workload characteristics, as summarized in Section 5.2.3. In contrast, the workload characteristics of the map join tasks are fairly uniform when compared with the corresponding common join tasks. Thus, for the MJ execution setting the task uniformity assumption does not have a negative impact when computing the average cost values in Starfish's reference profiles. In terms of training-based models, the pruned micro-benchmark queries are not sufficient for the KCCA policy to accurately predict the runtime. That is because KCCA requires *k* jobs in the training set that are very similar with the job that is predicted (i.e., k nearest neighbors) in order to achieve a good prediction accuracy. We note that having a number of k similar jobs in the training set for each possible testing query involves extensive training which is in opposition with our goal to limit the training set size to a minimum. MART and TITAN provide much more accurate predictions than KCCA at small training set sizes by using models at *operator phase granularity* instead of job granularity. For both execution scenarios, TITAN achieves the best prediction accuracy by using fine granularity models and by modeling processing costs instead of absolute runtime. Figure 5.13 b) shows the 95-percentile *average relative error* for the TPC-H workload. The error trends for all policies are similar with those for the average ratio error.

We also evaluate our prediction models trained with pruned micro-benchmark queries on a subset of the TPC-DS workload [71]. Figure 5.14 a) shows the 95-percentile average ratio error, while Figure 5.14 b) shows the corresponding average relative error. While the ratio errors for Starfish are smaller than for the TPC-H workload, the task uniformity assumption still introduces large errors for the CJ execution setting (i.e., an average ratio error of 1.7). KCCA has the largest error bars among all the training-based approaches. MART models severely mispredict the runtime of the map() phase for the MJ execution setting. TITAN's finer granularity models improve the accuracy of MART by a factor of two or more for both execution settings.

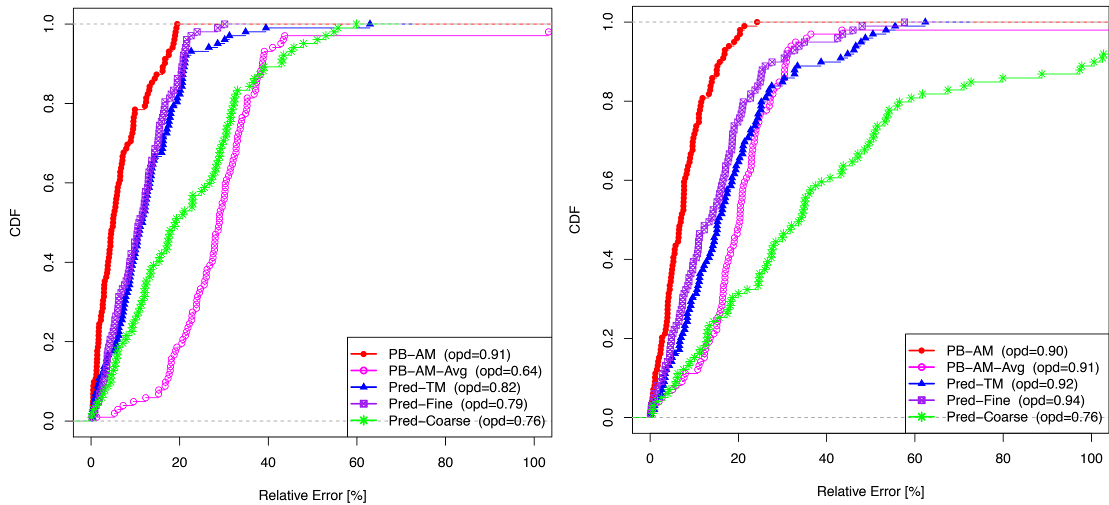


Figure 5.15 – Query level accuracy results for TPC-H workload: a) CJ setting (left), and b) MJ setting (right).

Fine vs. Coarse Models: Sensitivity Analysis on TPC-H

We perform sensitivity analysis with respect to the level of granularity of the models. As in Section 5.6.3, we train models with pruned micro-benchmark queries and we test them on the TPC-H workload. Figure 5.15 shows *query level* accuracy results for the TPC-H workload: i.e., the cumulative distribution functions of the relative prediction errors for the CJ execution setting (left) and for the MJ execution setting (right). In the legend corresponding to each scheme we also show the order preserving degree ratio (OPD) among alternative execution settings of the workload. Pred-Fine and Pred-TM schemes perform the best in terms of relative prediction errors and order preserving degree for both execution settings, with less than 20% relative error for 80% of workload for the CJ setting and less than 25% error for the MJ setting. We observe that the relative prediction errors for Pred-Coarse are higher than for Pred-Fine for both execution settings: i.e., for 80% of the workload, the relative errors are less than 35% for the CJ setting, and less than 60% for the MJ setting. As coarse granularity models rely solely on trained-based models for modeling the runtime of map() / reduce() functions, they require significant training data to fully model the underlying cost model of the functions. Fine granularity models split the map() / reduce() functions into *multiple phases*, and use trained-based models only for modeling the *processing costs* of each phase. The runtime of the functions is then computed analytically. This is the reason why fine granularity models are more accurate that coarse models at small training set sizes.

Fine vs. Coarse Models: Sensitivity Analysis on Micro-benchmark

In this experiment we train prediction models using the subset of micro-benchmark queries identified by the pruning algorithm for the Pred-Fine scheme, and test them on the remaining set of the micro-benchmark. Figure 5.16 a) shows the cumulative distribution function of the

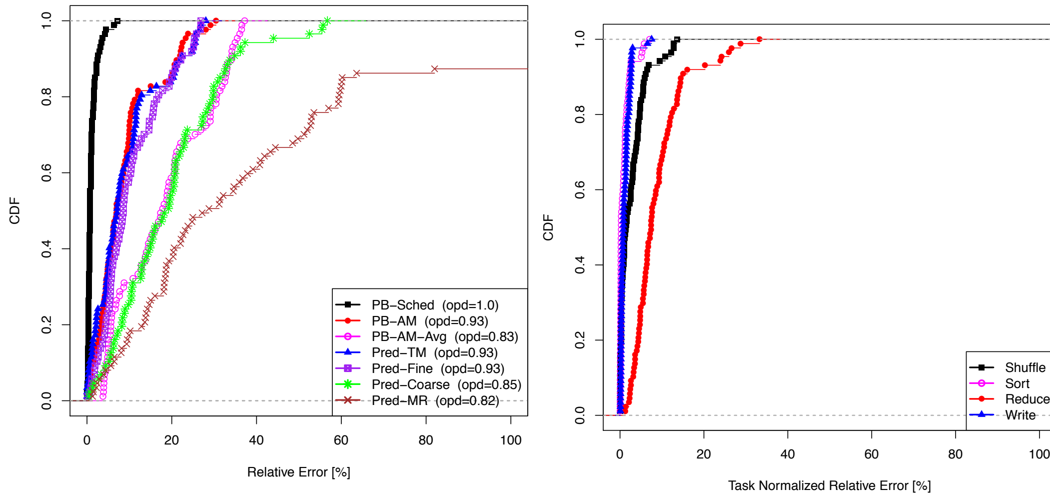


Figure 5.16 – Micro-benchmark results for the CJ setting: a) Query level relative prediction errors (left), and b) Reduce phase task normalized errors (right).

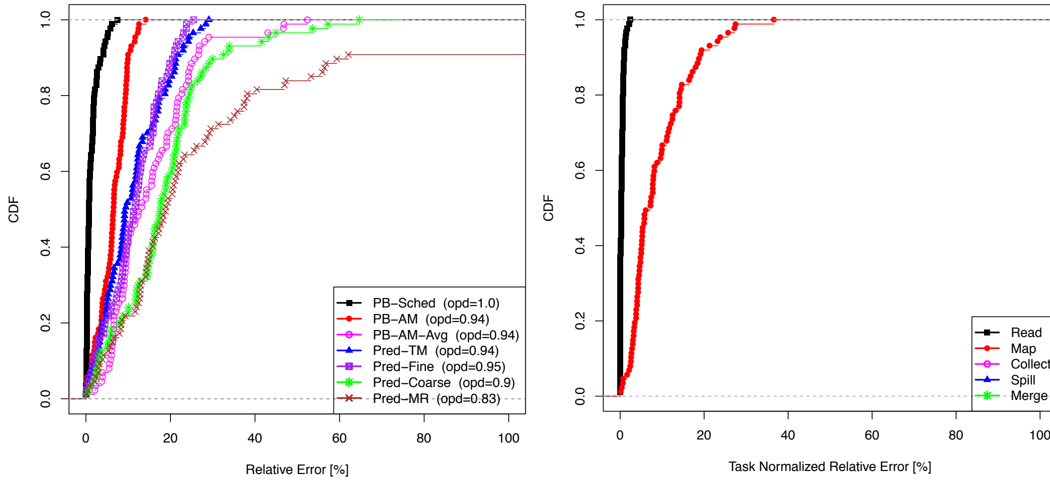


Figure 5.17 – Micro-benchmark results for the MJ setting: a) Query level relative prediction errors (left), and b) Map phase task normalized errors (right).

relative prediction errors corresponding to the CJ execution setting. For 80% of the workload, the relative prediction errors are less than 20% for both Pred-Fine and Pred-TM prediction schemes, less than 30% for Pred-Coarse, while for Pred-MR are less than 60%. We observe that both Pred-Fine and Pred-TM prediction schemes closely follow the PB-AM played back scheme. We note that the OPD values for Pred-Fine and Pred-TM achieve a similar score with PB-AM. Prediction errors at the granularity of reduce phases for the Pred-Fine scheme are shown in Figure 5.16 b). We observe that most of the errors correspond to the reduce() function per se, where the common join operator is being executed.

Figure 5.17 a) shows the error bounds corresponding to the MJ execution setting, where the

Table 5.4 – Prediction results for testing workloads outside of the training boundaries.

	MJ exec. setting		CJ exec. setting	
	$\overline{RE}$	OPD	$\overline{RE}$	OPD
PB-AM	13%	0.97	10%	0.96
Pred-Fine	31%	0.91	93%	0.89
Pred-TM	21%	0.92	45%	0.92
Pred-Coarse	38%	0.91	19%	0.79

common joins of the micro-benchmark are replaced with map joins. While the error margins stay in similar bounds for both Pred-Fine and Pred-TM (less than 20% relative error for 80% of the workload), they are 10% higher compared with the PB-AM scheme, the main source of prediction error being the map function, as shown in Figure 5.17 b).

Runtime Prediction Outside the Training Boundaries

We evaluate the accuracy of prediction models when the testing set is outside the boundaries of the training set. Specifically, we change the training set with a new set of micro-benchmark queries operating on tables with much larger row sizes, in the range of 500-2000 bytes. Then, we test the models on the TPC-H workload. Table 5.4 shows the results. We observe that for both execution settings, the translation models used by Pred-TM reduce the average relative errors of Pred-Fine from 31% to 21% for the MJ setting, and from 93% to 45% for the CJ setting while maintaining a high level of OPD. While for the CJ setting the smallest relative errors are for the Pred-Coarse models, they are less competitive in preserving a high value of OPD. Pred-TM achieves a good balance in terms of reduced relative errors compared with Pred-Fine (which over-predicts runtime) and better OPD than Pred-Coarse (which under-predicts runtime).

5.6.4 Answering Performance Boost Questions

In the following, we evaluate TITAN when answering the *performance boost* question as defined in Section 1.1. Given a reference execution of a workload on deployment A that uses the CJ execution strategy, we seek to find a new execution setting that reduces the workload execution time by a factor of 2x. We evaluate three possible deployment configurations Amazon EC2 cloud infrastructure: *Deployment A, B, and C*, and two join operator implementations: MJ and CJ. For each hardware configuration that has different underlying hardware than the reference execution, we run the micro-benchmark queries selected by the pruning algorithm to build prediction models. Figure 5.18 shows the absolute predicted values for each (operator implementation, deployment) pair. We observe that neither a different operator implementation running on the original deployment, nor a different deployment when using the original join implementation could achieve in isolation the target performance improvement of the workload. Hence, the joint space of possible execution settings has to be considered. TITAN

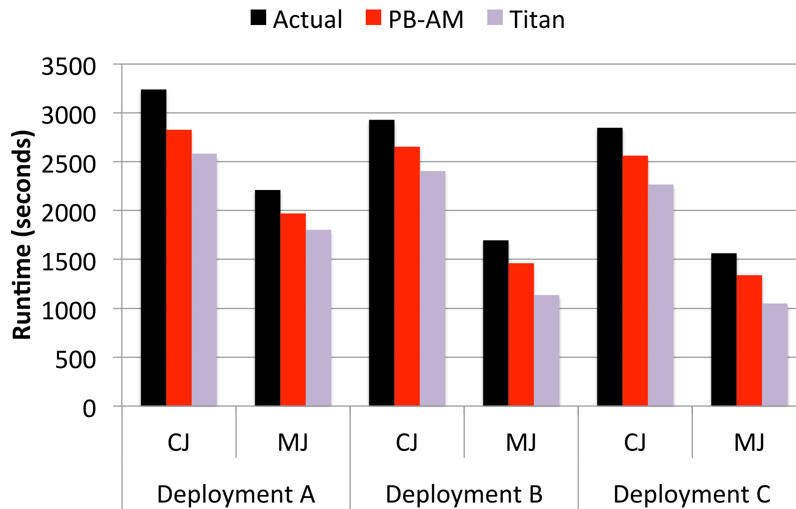


Figure 5.18 – Estimating query runtime of a selection of TPCH queries on multiple cluster allocations towards reducing the workload runtime of the original setting by 2x.

estimates that (MJ, Deployment B), and (MJ, Deployment C) are the execution settings that can reduce the runtime of the workload by a factor of 2x or more. By measuring the actual runtime reductions for the two configurations we obtain: a reduction of 1.9x for (MJ, Deployment B), and a reduction of 2.07x for (MJ, Deployment C). While the predictions were not exact, TITAN correctly identified the execution settings that closely approach the target runtime reduction. We also observe that the order preserving degree among alternative execution settings of the workload is 1.0 in this experiment.

5.7 Summary of Related Work

The closest work to TITAN is Starfish [38], a self-tuning system for Hadoop that uses performance models with the goal of workload tuning, and Elastisizer [39] that applies Starfish's approach for cluster sizing, i.e., finding the cluster size and the type of resources to use that best meet workload requirements. TITAN extends Starfish and Elastisizer at multiple levels: it models HiveQL like operators executed at scale on top of MapReduce, it proposes a methodology for collecting and pruning training datasets, and it reduces the modeling errors introduced in Starfish's analytical model by the *average task profile* through a hybrid prediction approach that models processing cost factors.

TITAN uses similar fine grain models with those proposed in [6, 50] with several key differences that target to *reduce the training cost* in terms of running benchmark queries: i) It splits operator phases operating on multiple inputs (i.e., joins) into multiple sub-phases such that the processing cost of each sub-phase can be *normalized* by its corresponding input feature. Hence, it *always* models *processing cost factors* instead of absolute time. ii) Outside of the

training boundaries it proposes *translation models* to exploit prior executions of the workload corresponding to different execution settings.

Wu et al. [80] propose analytical cost models for HiveQL operators with the underlying goal of query optimization. Their work is tailored towards reducing the size of intermediate results by adaptively grouping join operators that can be processed in one single MapReduce job. Similarly with PostgreSQL cost model, or the query cost calibration approaches proposed for PostgreSQL [81, 68], processing cost factors are assumed *constant* for a given hardware infrastructure. In our work we go a step further by modeling the processing cost of each operator phase for a range of workload characteristics, and execution settings.

In the context of relative performance modeling, Mesnier et al. [56] propose relative models for storage devices for estimating performance of a workload on device D1 based on performance of the same workload on device D2. We adopt a similar idea when building translation models with the difference that we apply it in a different context (e.g., query operators), and with a different goal: to compensate for the cases that the training data set does not cover the input feature space of the testing data set. Our contribution is to propose the level of granularity at which relative performance models can be applied in the context of HiveQL operators, the set of features, and the modeling methodology.

5.8 Conclusions

In this chapter we presented TITAN, a hybrid prediction approach for reporting SQL queries and a training methodology for generating benchmark queries that altogether reduce the cost of training based modeling approaches from days to hours while maintaining a good level of prediction accuracy of the models. The 95-percentile average relative error is less than 25% on the testing benchmarks. For the cases that the training dataset does not cover the testing data, TITAN introduces novel translation models that exploit prior workload executions corresponding to different configuration settings.

Our experiments show the feasibility of the proposed approach both in the context of static deployments, and also in the context of elastic deployments (i.e., using platform as a service on Amazon AWS). Our sensitivity analysis shows that fine granularity models are more accurate than coarse granularity models at small training set sizes (by a factor of two on average). Translation models outperform conventional prediction models outside of the training boundaries. TITAN can be used in all of the use cases presented in Section 1.1. For instance, Figure 5.18 shows how TITAN can be successfully used to find the execution setting that can satisfy the user requested time constraints (i.e., $2x$ performance boost).

6 Conclusions

Analytics today comprise of mixed workflows that include a variety of analytical tasks starting from the traditional SQL-like queries executing at scale, continuing with ETL, data preprocessing, up to more complex iterative analytics and data mining algorithms [82, 20]. Traditional DBMS systems originally designed for SQL analytics, had already started to include interfaces for ETL and more complex analytics into their engines[1]. Likewise, data processing systems originally designed for scalable ETL (i.e., MapReduce) have been extended to offer better support for SQL operators and complex analytics [84, 82]. Optimizing execution of such mixed workflows on a distributed cluster of machines either on a dedicated infrastructure or in the cloud (i.e., using Infrastructure as a Service on Amazon Web Services) motivated the work of this thesis. Workload runtime estimates corresponding to a set of potential resource allocation configurations are very useful to identify the resource allocation that satisfies the desired performance for the end user. At the same time runtime estimates are also very useful for resource managers and schedulers that aim to maximize the utilization of the cluster resources (and thus, reducing the over-provisioning costs).

In the following section we summarize the prediction approaches we developed for estimating the runtime of a class of iterative analytics, data pre-processing workflows, and reporting SQL, while emphasizing the main contributions for each workload category. In the last section we present future research avenues that are worth pursuing in the context of runtime prediction.

6.1 Summary of Contributions

Prediction Methodology for Iterative Analytics: In Chapter 3 we presented a methodology that we developed for estimating the number of iterations and per iteration resource requirements for an important class of iterative analytics used today in social media. The core of our methodology is represented by the *transform function* and the *sampling technique* that altogether preserve similarity among the execution patterns of the analytical task executed on

[1] For instance, Greenplum, Vertica, Teradata provide interfaces to execute business intelligence and ETL queries in parallel using MapReduce-like execution engines.

a sample of the input dataset with the execution patterns of the analytical task executed on the complete dataset. To the best of our knowledge, PREDIcT is the first empirical approach designed for estimating iterations and runtime for a class of iterative analytics executing on graphs. We exemplify and evaluate the set of transformations we propose for a number of iterative algorithms including: PageRank, semi-clustering, top-k ranking, neighborhood estimation, and labeling connected components.

Hybrid Models for Data Pre-processing Tasks: In Chapter 4 we presented prediction models for a class of repetitive workloads executing data pre-processing and ETL tasks on different datasets. By specializing models per query segment type we limited the domain knowledge required to build the models, and consequently it allowed us to use a small number of generic data features. For these workloads, we developed a hybrid prediction approach which combines the benefits of localized machine learning models with that of a global analytical model.

Training Methodology and Hybrid Models for Reporting SQL: One of the stringent requirements in the context of *elastic workload deployments* is how to built accurate and generic prediction models at a small training cost. Our work on estimating the runtime of reporting SQL analytics addresses this challenge in Chapter 5. We show that selecting and limiting the number of training queries to a minimum is possible without affecting the accuracy of the models significantly. To this goal, we propose a training methodology and hybrid prediction models at fine, *per operator-phase granularity* that have higher accuracy compared with competing approaches at small training set sizes.

6.2 Impact

The prediction techniques we describe in this thesis are advancing the state-of-the-art in three ways: i) by providing prediction mechanisms for a class of iterative analytics that were not empirically addressed before and are widely used today in analytical workflows; ii) by providing hybrid prediction models for different categories of data analytics and by analyzing the trade-offs at varying levels of model granularities; iii) by providing mechanisms to reduce the training cost while maintaining a competitive level of accuracy for the models.

PREDIcT improves the accuracy of analytical upper bounds for estimating iterations for PageRank from a relative error of $[104, 168]$% to $[0, 11]$%. Overall, the runtime estimates have an error of $[10-30]$% for all scale-free graph analyzed. Our training methodology for reporting analytics reduces the time for running training queries from *days to hours* while the 95-percentile average relative error is less than 25% on our testing benchmarks. We also show the utility of predictions in an end-to-end use case in Section 5.6.4. There we show that predictions that TITAN provides can be used to answer successfully resource allocation questions, i.e., identifying the resource allocation(s) that can satisfy a target performance goal.

6.2.1 Generality of Techniques to Similar Problems

Resource Allocation for Iterative Processing: In this thesis we answer resource allocation questions in the context of reporting analytics. The prediction techniques we develop for iterative processing: i.e., sampling technique, transform function, and hybrid cost modeling approach can be used as building blocks to answer similar resource allocation questions for iterative analytics. We note that estimating the number of iterations for the BSP execution model is independent on the resource allocation configuration. Hence, the sample-based approach we develop to estimate the number of iterations can be re-used as it is. As the worker on the critical path changes with the number of workers available and the partitioning strategy used, estimating per task key features starting from the observations of the sample run will require extensions to explicitly model the critical path of the actual run. Of particular importance is to build mechanisms that model the critical path of each worker of the actual run as a function of the data statistics collected during the sample run, a set of basic statistics about the input dataset, and the partitioning strategy.

Incorporating Predictions into Online Estimation Techniques and Vice-versa: The prediction techniques we propose in this thesis can be used in conjunction with online estimation techniques such as those proposed in the context of query progress estimators [16, 53, 57], and dynamic query re-optimization [69, 24] to improve the accuracy of estimations at runtime as more information becomes available. While conventionally, runtime predictors target resource allocation where runtime estimates are required *before* the query starts execution, and online estimation techniques target dynamic re-optimization and query progress monitoring, we envision for the near future *hybrid estimation approaches* that combine the advantages of predictors (i.e., runtime estimates before query execution) with those of online estimation techniques (i.e., accuracy refinement by exploiting runtime data).

Concretely, incorporating predictions into online runtime estimators and progress estimators is beneficial as it enables them to exploit the *initial* estimate information for the inactive query pipelines for which runtime data is not yet available. In addition, certain processing characteristics cannot be estimated at runtime. For instance, the number of iterations for iterative analytics cannot be estimated at runtime, before the iterative task completes its execution. The prediction techniques we develop in this context, such as: the sampling technique and the transform function, can be used to provide *initial* runtime estimates.

Incorporating runtime data into runtime predictors can at its turn facilitate problematic prediction models with high errors (i.e., model over-fitting or model under-fitting), and MapReduce jobs for which input data statistics collected during a prior reference run are insufficient or unavailable. For such cases, deliberately bring in more training data at runtime can improve the prediction models. In addition, input data statistics collected for the currently executing MapReduce job (e.g., task selectivities) can be used to refine the existing data statistics corresponding to that job. Of particular importance is the *adaptive mappers* approach proposed in the context of adaptive MapReduce [77]. While adaptive mappers were mainly used to balance

the workload of a MapReduce job among concurrent tasks, the adaptive sampling component could be also used to take approximate histograms at runtime. Thus, the tasks' selectivity information could be refined at runtime and used as input into prediction models.

Estimating Other Performance Features: While the focus of this thesis is to estimate the *runtime* of a class of analytical workloads for a pre-specified execution setting, similar models can be built to predict other performance metrics such as: the memory utilization and the CPU utilization, averaged over the duration of the workload execution. Performance metrics like these can be subsequently used to identify the *optimal allocation of resources* (in terms of memory buffer sizes and task slots) that does not only aim to execute the workload within the given deadline, but it also targets a high level of utilization for the resources that were allocated. While the full set of input features for such estimation problems will include additional, specific features to the performance metric that is modeled, the modeling approach per se at operator phase granularity, and the training methodology can be re-used.

6.3 Predictable vs. Non-Predictable Analytics

This thesis proposes estimation techniques for a class of analytical workloads that are amenable to sampling. For such cases, data characteristics observed during a short execution on the sample (or during the reference execution on a similar input dataset) can be collected and used later for prediction. The problem that we addressed in this context was two fold: i) how to take such a sample (in particular for iterative analytics), and ii) how to build cost models that estimate runtime as a function of the workload characteristics (for all classes of analytics we analyze in this thesis). For such analytical workloads we show that we can answer feasibility analysis questions and performance boost questions as the ones we summarize in Section 1.1. Thus, in contrast with conventional query optimization techniques that aim to compare the relative performance among alternative execution strategies of a workload on a fixed deployment, prediction techniques, like the ones we develop in this thesis, are enablers for identifying the execution settings that can satisfy an *absolute performance goal* (i.e., deadline). In addition, predictions can be used to assess the relative performance improvement of the workload among alternative execution settings that can include *different* deployments.

As the accuracy metric that quantifies the quality of predictions is highly dependent on the end application (use case), we show results for multiple accuracy metrics. We focus on *relative prediction error* and *ratio error* when the goal is to estimate the absolute runtime and on *order preserving degree* when the goal is to rank alternative execution settings. We note that estimating the relative order among alternative execution settings (i.e., ranking potential execution settings as in optimization) is generally easier than estimating the absolute performance. The main reason for that is that ranking can tolerate errors in the models as long as the modeling errors introduced are consistent for all the execution settings that are compared. In contrast, all the modeling errors are visible when estimating the absolute performance.

For the cases that the data statistics collected during a prior reference run of the query are insufficient, and for the cases that the cost models are in-accurate due to little data overlap among the training set and the testing set (e.g., model under-fitting or model over-fitting), we can deliberately bring in more training data to improve the prediction models. The runtime estimation techniques that are summarized in Section 6.2.1 can be used to refine the data statistics as more information becomes available. For fixing the cost model fitting errors, we can generate synthetic workloads with a large spectrum of workload characteristics such that we can better cover the multi-dimensional input feature zone of the testing set. Then, we can use the iterative training procedure we propose in Section 5.4.1 to update the models. From our experience, we find that in many cases we can successfully use cross validation techniques to identify cost modeling errors before the actual execution of the query. While cross validation techniques can be used also to identify the consistency of a sampling technique, as we show in the context of iterative processing in Section 3.8.5, such an approach is expensive to be performed each time a prediction is sought. Thus, the modeling errors introduced by the sampling technique are usually identified at runtime.

This thesis does not address the problem of estimating the runtime of data analytics that are not amenable to sampling techniques, and that of ad-hoc analytics, where either the data statistics or the processing characteristics of the input query are very different than the training data that is currently available. For such cases, the only feasible option to produce an estimate is to apply online estimation techniques (as the ones presented in Section 6.2.1). We summarize the limitations of our prediction techniques in the context of iterative analytics in Section 3.7, in the context of ETL in Section 4.3, and in the context of reporting SQL in Section 5.2.2.

6.4 Looking Ahead

In this thesis we provide the fundamental prediction mechanisms to estimate the runtime of three categories of analytics that are widely used today in social media and web analysis and are executed at scale using the MapReduce execution framework. Analytics are continuously increasing in complexity as users pose analytical queries that require to find insights and complex correlations from the ever increasing input data sets. At the same time, distributed processing frameworks for big data analysis are constantly evolving adding more capabilities to their core engines to support these queries. This evolution of queries and engines rises new challenges and opportunities in query runtime prediction.

6.4.1 SLA Driven Job Scheduling

Rayon [21] is an example of an SLA-oriented scheduler that introduces the concept of *predictable resource reservations* for a pre-specified duration of time. Rayon assumes that the end user can accurately assess his resource requirements and the maximum time allocation for which his workload will use the resources. It is worthwhile to analyze the efficiency of

such a scheduler when using *runtime predictions* instead of ground truth values. Studying the tolerance of the scheduling policy with respect to the prediction accuracy is very useful to assess both the effectiveness of the scheduler, and that of the prediction method in another end-to-end use case. While this future direction of research is worth pursuing for all workload categories, particular interest is represented by applications with *gang-scheduling* requirements, i.e., applications that have to receive all the resources they asked for before starting execution. Iterative analytics executing on Giraph BSP fall into this category.

6.4.2 Cost Models for In-memory Analytical Engines

Estimating the runtime of data analytics on in-memory analytical engines that emerge as an alternative for executing data analytics at scale (such as Spark [84]) open up new opportunities in runtime modeling. While the prediction building blocks and methodologies we propose for: i) estimating iterations, ii) building a hybrid prediction model at phase granularity, and iii) training, are applicable in this context as well, the cost models per se require adjustments to take into consideration new key input features corresponding to a different engine. Similarly with the MapReduce execution model, two types of key features shall be considered: core engine level features (e.g., Spark operators), and upper layer library features (e.g., Hive features, when executing HiveQL queries on Spark).

6.4.3 Sharing Cluster Resources Among Analytical Engines

With the goal of reducing provisioning costs, resource managers often collocate multiple analytical engines specialized for different classes of analytics within the same cluster deployment (e.g., [41]). Sharing cluster resources among concurrent workloads increases the resource utilization of the cluster. Yet, in order to keep up the performance demands of the end applications that have stringent time requirements, resource managers shall carefully consider the impact among concurrent workloads. While workload interference has been studied recently especially in the context of single node DBMS (e.g., [4, 25]), new challenges occur in a distributed setting where *different engines* execute concurrently. Of particular importance is to identify the modeling granularity at which training cost is feasible yet effective (due to the exponential size of the possible input feature combinations), and the key features that can be collected and used to calibrate predictions.

List of Figures

1.1 Iterative Processing: S, input dataset that does not change as a result of executing the iterative task (i.e., input graph structure), T, input that gets updated at the end of every iteration. . . . 4

1.2 Pipeline of Analytical Tasks in Web Data Analysis 5

1.3 Using analytical upper bounds to approximate the number of iterations for PageRank algorithm (left), and per iteration resource requirements (i.e., message bytes) for connected components (right). . . . 6

2.1 MapReduce Execution Model: Map tasks transform the input data and output intermediate results as key-value pairs. The reduce tasks copy and merge all the values corresponding to the same key, then apply the reduce function to produce the final result. . . . 12

2.2 BTrace profiling rule example. . . . 25

2.3 CART decision tree with four input features $F_1 - F_4$, four conditionals, and five possible predicted values $C_1 - C_5$. . . . 26

2.4 MART decision trees with two boosting iterations (i.e., two trees) and four input features $F_1 - F_4$. The predicted value is a summation over the predicted values of each tree. . . . 27

3.1 PREDIcT's methodology for estimating the key input features and runtime of iterative algorithms. . . . 32

3.2 Maintaining invariants for the number of iterations when executing PageRank on sample graphs. . . . 33

3.3 BSP execution phases of an arbitrary iteration. . . . 39

3.4 The accuracy of predicting the number of iterations for PageRank for $\epsilon = 0.01$ (left) and for $\epsilon = 0.001$ (right). 49

3.5 The accuracy of predicting the number of iterations for semi-clustering for $\tau = 0.01$ (left) and for $\tau = 0.001$ (right). 50

3.6 Top-k ranking key input features estimation: a) Estimating iterations (left), b) Estimating remote message bytes (right). 50

3.7 Predicting key input features for connected components: a) number of iterations (left), b) active vertices (right). 51

3.8 a) Predicting active vertices for connected components with guided sampling (left), b) Predicting remote message bytes for neighborhood estimation (right). 52

3.9 Estimating the number of iterations: Analytical upper bounds versus PREDIcT. 53

3.10 Semi-clustering runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right). 54

3.11 Top-k ranking runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right). 54

3.12 Predicting runtime for neighborhood estimation: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right). 55

3.13 Connected components runtime prediction: a) Training with sample-runs (left), b) Training with sample- and actual-runs (right). 55

3.14 Connected components runtime prediction: Training with sample- and actual-runs for guided sampling. 56

3.15 Predicting iterations: sensitivity analysis w.r.t. sampling technique for PageRank (top), semi-clustering (middle), and top-k ranking (bottom) on UK web graph. 57

3.16 Predicting key input features: sensitivity analysis w.r.t. sampling technique for connected components (top and middle), and neighborhood estimation (bottom) on UK web graph. 57

3.17 Runtime of sample-runs and actual-runs for PageRank (PR), semi-clustering (SC), connected components (CC), top-k ranking (TOP-K), and neighborhood estimation (NH), in seconds. 59

3.18 Estimating runtime for semi-clustering for a different slot allocation. 59

4.1 Input / output cardinality correlations for Workload-A 62

4.2 Input cardinality / processing speed correlations for Workload-A 62

4.3 Jaql query (script) example that extracts from a list of web pages the web graph corresponding to a particular web domain. 64

4.4 Modeling per segment cardinality functions (i.e., C_i) and processing speed functions (i.e., P_i) for phase-level segments. 66

4.5 Job runtime estimation for TPC-DS 70

4.6 Actual Runtime vs. Predicted Runtime for TPC-DS 70

4.7 Job runtime estimation for Workload-A (left), and Workload-B (right). 70

4.8 Query runtime estimation for TPC-DS 71

5.1 Query Processing Model in HiveQL 77

5.2 Selection of common join jobs where the main source of error is Starfish's analytical model. 78

5.3 Selection of common join jobs where the main source of error is the task uniformity assumption. 78

5.4 Hybrid prediction engine for estimating the runtime of HiveQL queries. 81

5.5 Starfish reference profiles summarize the processing characteristics of a MapReduce job. 82

5.6 Runtime prediction using Starfish's What-If engine. 83

5.7 Critical path modeling for a MapReduce job with five map tasks and three reduce tasks. 83

5.8 $GenOut$ processing cost variation with the number of rows processed. 84

5.9 Decision tree example for modeling the $GenOut$ cost factor. 85

5.10 a) Absolute GenOut cost as a function of output bytes (left), and b) Relative GenOut cost: $GenOut_{MJ/CJ}$ (right). 92

5.11 Type of code/data mappings among MapReduce tasks. 93

5.12 Impact of pruning algorithm on prediction accuracy for training-based modeling approaches. 99

5.13 Job level accuracy results for TPC-H workload: a) Job level 95-percentile average ratio error (left), and b) Job level 95-percentile average relative error (right). . . 101

5.14 Job level accuracy results for TPC-DS workload: a) Job level 95-percentile average ratio error (left), and b) Job level 95-percentile average relative error (right). . . 101

5.15 Query level accuracy results for TPC-H workload: a) CJ setting (left), and b) MJ setting (right). 103

5.16 Micro-benchmark results for the CJ setting: a) Query level relative prediction errors (left), and b) Reduce phase task normalized errors (right). 104

5.17 Micro-benchmark results for the MJ setting: a) Query level relative prediction errors (left), and b) Map phase task normalized errors (right). 104

5.18 Estimating query runtime of a selection of TPCH queries on multiple cluster allocations towards reducing the workload runtime of the original setting by 2x. 106

List of Tables

1.1 Conceptual differences between reporting and ad-hoc query processing. 3

2.1 Recent prediction approaches for analytical workloads. 23

3.1 Notations used for representing the transform function. 38

3.2 Key Input Features 39

3.3 Graph Datasets 48

5.1 MapReduce specific features 86

5.2 Join specific features 87

5.3 Examples of configuration settings considered in the analytical model. 88

5.4 Prediction results for testing workloads outside of the training boundaries. . . . 105

Bibliography

[1] Program for TPC-H Data Generation with Skew. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[2] A. Aboulnaga and S. Babu. Workload Management for Big Data Analytics. In *SIGMOD*, pages 929–932, 2013.

[3] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-Reduce Extensions and Recursive Queries. In *EDBT*, pages 1–8, 2011.

[4] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Interaction-aware Prediction of Business Intelligence Workload Completion Times. In *ICDE*, pages 413–416, 2010.

[5] K. J. Ahn, S. Guha, and A. McGregor. Analyzing Graph Structure via Linear Measurements. In *SODA*, pages 459–467, 2012.

[6] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based Query Performance Modeling and Prediction. In *ICDE*, pages 390–401, 2012.

[7] Apache Mahout Framework. Project Website: http://mahout.apache.org/.

[8] D. Arthur and S. Vassilvitskii. How Slow is the k-means Method? In *SCG*, pages 144–153, 2006.

[9] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-optimization. In *SIGMOD*, pages 107–118, 2005.

[10] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *SIGMOD*, pages 16–52, 1986.

[11] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *VLDB*, 2011.

[12] R. Borovica, I. Alagiannis, and A. Ailamaki. Automated Physical Designers: What You See is (Not) What You Get. In *DBTest*, 2012.

[13] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160, 2012.

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3:285–296, 2010.

[15] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, 2010.

[16] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators for SQL Queries? In *SIGMOD*, pages 575–586, 2005.

[17] S. Chaudhuri, R. Motwani, and V. Narasayya. Random Sampling for Histogram Construction: How Much is Enough? In *SIGMOD*, pages 436–447, 1998.

[18] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *PVLDB*, 5(12):1802–1813, 2012.

[19] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evauation. *ACM Trans. Database Syst.*, 9(2):163–186, 1984.

[20] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[21] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *SoCC*, 2014.

[22] A. Das Sarma, D. Nanongkai, and G. Pandurangan. Fast Distributed Random Walks. In *PODC*, 2009.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[24] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, 2007.

[25] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *SIGMOD*, pages 337–348, 2011.

[26] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 5(11):1268–1279, 2012.

[27] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[28] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven Workload Modeling for the Cloud. In *ICDEW*, 2010.

[29] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, pages 592–603, 2009.

[30] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *INFOCOM*, pages 2498–2506, 2010.

[31] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI*, pages 17–30, 2012.

[32] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K.-U. Sattler. 10381 Summary and Abstracts Collection – Robust Query Processing. In *Robust Query Processing*, Dagstuhl Seminar Proceedings, 2011.

[33] S. Guha and A. McGregor. Graph Synopses, Sketches, and Streams: A Survey. *PVLDB*, 5(12):2030–2031, 2012.

[34] S. Har-Peled and B. Sadri. How Fast is the k-means Method? In *SODA*, pages 185–202, 2005.

[35] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference and Prediction, Second Edition. Springer, 2008.

[36] H. Herodotou. Hadoop Performance Models. Technical Report, CS-2011-05, Duke University, 2010.

[37] H. Herodotou and S. Babu. Xplus: A SQL-Tuning-Aware Query Optimizer. *PVLDB*, 3(1-2):1149–1160, 2010.

[38] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.

[39] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *SOCC*, 2011.

[40] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.

[41] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[42] 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*, pages 349–362, 2012.

[43] P. Hu and W. C. Lau. A Survey and Taxonomy of Graph Sampling. *CoRR*, abs/1308.5865, 2013.

[44] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *ICDM*, pages 229–238, 2009.

[45] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys*, pages 169–182, 2013.

[46] A. N. Langville and C. D. Meyer. Deeper Inside PageRank. *Internet Mathematics*, 1(3):335–380, 2003.

[47] J. Leskovec and C. Faloutsos. Sampling from Large Graphs. In *KDD*, pages 631–636, 2006.

[48] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

[49] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical Properties of Community Structure in Large Social and Information Networks. In *WWW*, pages 695–704, 2008.

[50] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *PVLDB*, 5(11):1555–1566, 2012.

[51] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *PVLDB*, 5(11):1555–1566, 2012.

[52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[53] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL Progress Indicators. In *EDBT*, pages 921–941, 2006.

[54] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB*, pages 149–159, 1986.

[55] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[56] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the Relative Fitness of Storage. In *SIGMETRICS*, 2007.

[57] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, pages 507–518, 2010.

[58] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, 2008.

[59] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.

[60] N. Pansare, V. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. In *VLDB*, 2011.

[61] E. Pednault. Transform Regression and the Kolmogorov Superposition Theorem. In *IBM Research Report RC 23227, IBM Research Division*, 2004.

[62] M. Poess, R. O. Nambiar, and D. Walrath. Why You Should Run TPC-DS: A Workload Analysis. In *VLDB*, 2007.

[63] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki. Same Queries, Different Data: Can We Predict Runtime Performance? In *ICDE Workshops*, pages 275–280, 2012.

[64] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *KDD*, pages 23–32, 1999.

[65] J. R. Quinlan. Learning With Continuous Classes. In *Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[66] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *SOSP*, pages 472–488, 2013.

[67] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[68] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic Virtual Machine Configuration for Database Workloads. *TODS*, 35(1):7:1–7:47, 2008.

[69] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[70] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *PVLDB*, 2:1626–1629, 2009.

[71] Transaction Processing Performance Council. TPC Benchmark DS Draft Specification Revision 32. http://www.tpc.org/tpcds.

[72] Transaction Processing Performance Council. TPC Benchmark H Standard Specification Revision 2.12.0. http://www.tpc.org/tpch.

[73] L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, 1990.

[74] S. van Dongen. A Cluster Algorithm for Graphs. Technical Report INS-R0010, CWI, 2000.

[75] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, pages 5:1–5:16, 2013.

[76] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *ICAC*, pages 235–244, 2011.

[77] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce Using Situation-aware Mappers. In *EDBT*, pages 420–431, 2012.

[78] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[79] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Middleware*, pages 1–20, 2010.

[80] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query Optimization for Massively Parallel Data Processing. In *SOCC*, 2011.

[81] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *ICDE*, 2013.

[82] R. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. Technical report, AMP Lab, 2012.

[83] 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 *OSDI*, pages 1–14, 2008.

[84] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[85] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *VLDB*, pages 289–300, 2005.

[86] Q. Zhu and P.-A. Larson. Building Regression Cost Models for Multidatabase Systems. In *PDIS*, 1996.

Adrian Daniel Popescu — Curriculum Vitae

Personal Information

Birth date: 11 April, 1983
Nationality: Romanian
Phone (office): +41 21 693 1427
Phone (mobile): +41 762 209 880
E-mail: adrian.popescu@epfl.ch
Web: http://people.epfl.ch/adrian.popescu

Education

PhD candidate in Computer and Communication Sciences, (Sept. 2009 - present)
Ecole Polytechnique Federale de Lausanne (EPFL)
MASc Electrical and Computer Engineering, (Sept. 2007 - Aug. 2009)
University of Toronto (UofT)
BSc Computer Science and Engineering, (Oct. 2001 - Sept. 2006)
University "Politehnica" of Bucharest (UPB)

Research and Work Experience

Research and Teaching Assistant at EPFL (2009-present)
Active Projects: PREDIcT: Predicting the Runtime of Iterative Analytics,
Predicting the Runtime Performance of Reporting Analytics
Past Project: HBaseSQL: Adaptive Query Processing in the Cloud

TAing for: Advanced Databases, Object Oriented Programming,
Programming Fundamentals
Activities: student advisor, preparing/grading assignments.

Summer Research Intern at IBM Almaden Research Center (July-September 2011)
Project: Predicting the Runtime Performance of Jaql Queries
Source code contributions into the IBM BigInsights product.

Research and Teaching Assistant at University of Toronto (2007-2009)
Thesis Project: SAAB: SLA-Aware Adaptive Data Broadcasting

TAing for: Communication and Design, Programming Fundamentals,
Computer Fundamentals
Activities: student advisor, grading assignments.

Research Assistant at FOKUS Fraunhofer Institute, Berlin (Sept. 2006 - July 2007)
Research on Next Generation Networks (NGN) and IP Multimedia
Subsystem (IMS). Projects: FOKUS Presence Service & FOKUS Home
Subscriber Server (FHoSS)

Internship at FOKUS Fraunhofer Institute, Berlin (March 2006 - Aug. 2006)
Diploma thesis: Presence Service in the IMS Playground - design,
implementation and integration into the IMS environment.

Teaching Assistant at University "Politehnica" of Bucharest (Oct. 2005 - March 2006)
TAing for: Microprocessor Based System Design
Activities: student advisor, preparing/grading assignments.

PhD Research Direction

Predicting the runtime performance of large scale analytical workloads is motivated by a number of data management tasks including workload optimization, resource management and scheduling, query progress monitoring. More recently, with the prevalence of using hardware infrastructure as a service (IaaS) for data management tasks, answering feasibility analysis questions for hypothetical configurations became critical.

My broad research interests lie at the intersection of database management systems with distributed systems with a main focus on query performance modeling. My PhD thesis topic focuses on modeling the runtime performance of analytical workloads that include workflows of iterative machine learning algorithms and traditional SQL-like operators executing on large scale infrastructures such as MapReduce. In particular, the set of prediction models I developed recently estimate the runtime performance of: i) a class of iterative algorithms executing on graph datasets, and ii) reporting queries running HiveQL operators for a set of execution settings (consisting of different operator implementations and hardware deployments).

More information: http://dias.epfl.ch/PREDIcT

Publications

"Query Runtime Prediction for Large Scale Reporting Analytics". **A. D. Popescu**, S. Babu, and A. Ailamaki. *Submitted, under peer review.*

"PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics". **A. D. Popescu**, A. Balmin, V. Ercegovac, and A. Ailamaki. In *the Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1678-1689, September 2013.

"Same Queries, Different Data: Can we Predict Query Performance?". **A. D. Popescu**, V. Ercegovac, A. Balmin, M. Branco and A. Ailamaki. In *the 7th International Workshop on Self Managing Database Systems (SMDB), colocated with ICDE*, Washington DC, USA, April 2012.

"Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, **A. D. Popescu**, A. Ailamaki and B. Falsafi. In *the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

"Adaptive Query Execution for Data Management in the Cloud". **A. D. Popescu**, D. Dash, V. Kantere, A. Ailamaki. In *the Second International Workshop on Cloud Data Management (CloudDB), colocated with CIKM*, Toronto, Canada, October 2010.

"SLA-Aware Adaptive On-Demand Data Broadcasting in Wireless Environments". **A. D. Popescu**, M. A. Sharaf and C. Amza. In *the Tenth International Conference on Mobile Data Management (MDM)*, Taipei, Taiwan, May 2009.

"Dynamic Resource Allocation for Databases Running on Virtual Storage". G. Soundararajan, D. Lupei, S. Ghanbari, **A. D. Popescu**, J. Chen and C. Amza. In *the 7th USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, February 2009.

Academic Awards

- SoCC Travel Grant (2014)
- Rogers Scholarship, University of Toronto (2007 - 2009)
- Merit Scholarship awarded to (approx.) 3% students, UPB (2006)
- 3rd prize at UPB Annual Projects Competition, Hardware Section (2005)
- Study Scholarship awarded to (approx.) 10% students, UPB (2003 - 2006)
- 1st prize at the Regional Physics Olympiad Miercurea Ciuc, Romania (2001)

Service

External reviewer at the following venues: Middleware 2008, WWW 2009, HDMS 2010, VLDB 2011, SMDB 2012, Transactions on Computers 2014, CIDR 2015.

Professional Skills

- Programming Languages: Java, C, C++
- Scripting Languages: Bash, Python
- Query Languages: SQL, Jaql, HiveQL, Pig
- Distributed Data Management: Hadoop, HBase, Giraph
- DBMS: IBM DB2, Oracle, MySQL, PostgreSQL
- Statistical computing: R, Matlab, Weka
- Distributed Systems: RPC, RMI, MPI, Open MP, JAVA2EE
- Operating Systems: Linux & Windows
- Compilers: Flex and Yacc
- AWS Experience: EC2, EMR, S3
- Developing Tools: Eclipse, gdb, gprof, svn, git

Pre-PhD Research Projects

SLA-Aware Adaptive Data Broadcasting in Wireless Environments (2008-2009)
UofT, Master's project, Advisor: Prof. Cristiana Amza

Most of the existing broadcasting scheduling policies focused on either minimizing response time, or drop rate when requests are associated with hard deadlines. The inherent inaccuracy of hard deadlines in a dynamic mobile environment motivated the use of Service Level Agreements (SLAs) where a user specifies the utility of data as a function of its arrival time. Hence, I proposed SAAB which is an SLA-aware adaptive data broadcast scheduling policy for maximizing the system utility under SLA-based performance measures. To achieve this goal, SAAB considers both the characteristics of disseminated data objects as well as the SLAs associated with them.

A Study on Performance Isolation Approaches for Consolidated Storage (2008)
UofT, Operating Systems course project, Advisor: Prof. Cristiana Amza

Due to maintenance and operational costs, production lines tend to consolidate several storage servers on a single, dedicated server, which serve multiple workloads. In this context, it is essential to deploy mechanisms that isolate the performance of each individual workload. In this project I studied and evaluated several non-intrusive isolation approaches that are interposed between the application workloads and the storage server. The metrics considered in the evaluation were the QoS guarantees: i.e., latency predictability and head seek overhead.

FOKUS Home Subscriber Server (FHoSS) (2007)
FOKUS Fraunhofer Institute

As part of this project I re-designed and improved the Home Subscriber Server (HSS) implementation of the OpenIMSCore environment. Specifically I worked on the interface between the HSS and the Call Session Control Function nodes (Cx interface) and on the interface between the HSS and the Application Servers (Sh interface). I also implemented a web interface for user provisioning.

Presence Service in the IP Multimedia Subsystem (2006)
FOKUS Fraunhofer Institute, Bachelor degree diploma project
Advisors: Prof. Valentin Cristea (UPB), Prof. Thomas Magedanz (FOKUS)

The Presence Service goal was to track and disseminate the presence status of IMS clients to their corresponding subscribers. The basic features of the service included: a generic publish-subscription mechanism, support for enriched presence information and buddy lists management. In addition, modalities to reduce the presence information traffic over the low bandwidth networks were considered. Specifically, partial messages and versioning techniques were used to achieve performance improvements.

Languages

English - fluent, Romanian - fluent (mother tongue)
Hungarian, French, German - conversational

References

Available upon request.