Simplifying System Management Through Automated Forecasting, Diagnosis, and Configuration Tuning

by

Songyun Duan

Department of Computer Science
Duke University

Date:_____
Approved:

_____
Dr. Shivnath Babu, Supervisor

_____
Dr. Kamesh Munagala

_____
Dr. Sandeep M Uttamchandani

_____
Dr. Jun Yang

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2010

ABSTRACT

Simplifying System Management Through Automated Forecasting, Diagnosis, and Configuration Tuning

by

Songyun Duan

Department of Computer Science
Duke University

Date: _____
Approved:

_____
Dr. Shivnath Babu, Supervisor

_____
Dr. Kamesh Munagala

_____
Dr. Sandeep M Uttamchandani

_____
Dr. Jun Yang

An abstract of a dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2010

Abstract

Large-scale networked computing systems are widely deployed to run business-critical applications in environments where changes are frequent. Manual management of these complex systems can be tedious and error-prone. Meanwhile, the high costs of application downtime make it critical to ensure system availability and reliability. Recent progress in monitoring tools enables system administrators to collect fine-grained data about system activity with low overhead. This data provides valuable information for system management. However, the monitoring data collected from production systems is massive in size and noisy; which makes it hard for system administrators to fully utilize this data for effective system management.

This dissertation describes a data-management platform, called Fa, where system administrators can pose declarative queries over system monitoring data. Fa automatically finds fairly accurate and efficient execution plans for given queries, and returns query results in easy-to-interpret formats. Fa supports three key query types, namely, forecasting queries (for predicting or detecting performance problems), diagnosis queries (for finding the cause of performance problems), and tuning queries (for recommending changes to system configuration to resolve diagnosed problems):

(a) For processing diagnosis queries, Fa constructs problem signatures from system monitoring data to identify recurrent problems and to reuse past diagnostic information. For a rare or new problem, Fa employs an anomaly-based clustering technique to generate performance baselines and to characterize

the deviation from baselines to pinpoint root causes. Fa also incorporates an active-learning component that identifies diagnosis queries whose results, if provided or confirmed by system administrators, can be used to update problem signatures and to improve the accuracy and efficiency for processing future queries.

(b) For processing tuning queries to resolve problems caused by system misconfiguration, Fa employs an adaptive sampling algorithm that plans experiments to efficiently identify high-impact configuration parameters and high-performance settings. These experiments bring in information—required for generating accurate query results—that is missing in the monitoring data collected so far.

(c) For both one-time and continuous forecasting queries, Fa automatically searches for efficient execution plans in a large space of plans composed of data-transformation operators as well as synopsis-learning and prediction operators. Forecasting queries can be composed with diagnosis and tuning queries to enable proactive system management that avoids potential problems.

We have evaluated the Fa platform with monitoring data collected from database-backed multitier services, and with synthetic data that models the noisy nature of monitoring data from production systems. Our evaluation shows that Fa's query plan selection and execution strategies provide actionable information for system management automatically, accurately, and efficiently. Critical features like reliable confidence estimates, robustness to noise, and providing supporting evidence for query results make Fa a practical and useful platform.

Contents

Abstract **iv**

List of Figures **xii**

List of Tables **xvii**

Acknowledgements **xix**

1 Introduction **1**

1.1 Complexity of System Management 1

1.2 Data-driven System Management 4

1.2.1 Research Questions 6

1.3 Contributions 8

1.4 Organization 11

2 Overview **13**

2.1 Multi-tier Systems and System (Monitoring) Data 13

2.2 System Management Queries 16

2.3 Architecture of Fa 20

2.3.1 Interfacing with the Managed System 21

2.3.2 Interface for Expressing Queries 22

2.3.3 Representation of Base and Derived System Data 24

2.3.4 Query Processor 25
2.4 Illustration with Forecasting Queries 28
2.4.1 Execution Plans 28
2.4.2 Finding a Good Plan Automatically and Efficiently 31
2.5 Summary 32

3 Automated Processing of Diagnosis Queries 34

3.1 Motivation 34
3.2 Abstraction of Diagnosis Queries 36
3.3 Phase I: Generating and Using a Signature Database 39
3.3.1 Generating the Binary Matrix 45
3.3.2 Generating the Separating Functions 47
3.3.3 Weighting the Separating Functions 47
3.3.4 Online Use and Maintenance 49
3.3.5 Error-Aware Signature Databases 52
3.4 Phase II: Anomaly-based Clustering 55
3.4.1 Diagnosis Vectors and Margin Classifiers 57
3.4.2 Strawman: Margin-based Agglomerative Clustering (MAC) 59
3.4.3 Partition-Check-Merge (PCM) Algorithms 62
3.4.4 PCM-Conservative (PCM-C) 64
3.4.5 PCM-Eager (PCM-E) 66
3.4.6 Filtering and Ranking Diagnosis Results 66
3.5 Experimental Evaluation 67
3.5.1 Experimental Setting 67

3.5.2 Evaluation of Phase I 69
3.5.3 Evaluation of Phase II 78
3.5.4 Details of Diagnosis Results for Phase II 84
3.5.5 Experimental Comparison with Previous Work 91
3.6 Related Work 93
3.7 Summary 94

4 Guided Diagnosis Through Active Learning 96
4.1 Motivation 96
4.2 Overview 98
4.3 Active Learners 101
4.3.1 Least-Confidence Sampling (LC) 102
4.3.2 Kernel-Furthest-First Learner (KFF) 103
4.3.3 Hybrid Learner (Hybrid) 104
4.3.4 How Falcon Uses an Active Learner 105
4.4 Clustering in Falcon 105
4.4.1 Distance-based Clustering 105
4.4.2 Time-based Chunking 106
4.5 Experiments 109
4.5.1 Testbed 109
4.5.2 Evaluation Methodology 110
4.5.3 End-to-End Validation: Falcon Vs. Random Sampling . . . 112
4.5.4 Comparing Clustering Methods 113
4.5.5 Comparing Active Learners 115
4.6 Related Work 116

4.7 Summary 118

5 Automated Processing of Tuning Queries 119

5.1 Motivation 119

5.2 Abstraction of Tuning Queries 124

5.3 Overview of Fa's iTuned Approach to Process Tuning Queries . . . 127

5.4 Adaptive Sampling 131

5.4.1 Initialization 131

5.4.2 Picking the Next Experiment 132

5.4.3 Overall Algorithm and Implementation 140

5.5 A Platform for Running Online Experiments 142

5.6 Improving iTuned's Efficiency 146

5.6.1 Eliminating Unimportant Parameters Using Sensitivity Analysis 147

5.6.2 Running Multiple Experiments in Parallel 148

5.6.3 Early Abort of Low-Utility Experiments 149

5.6.4 Workload Compression 149

5.6.5 Using Database-specific Knowledge 150

5.6.6 Other Techniques 150

5.7 Empirical Evaluation 151

5.7.1 Methodology and Summary 152

5.7.2 Performance of the .eX framework for Conducting Experiments 153

5.7.3 Why Parameter Tuning is Nontrivial 154

5.7.4 Tuning Results 158

5.7.5 Sensitivity Analysis 163

5.8 Related Work 166
5.9 Summary 169

6 Automated Processing of Forecasting Queries 171

6.1 Motivation 171
6.2 Abstraction of Forecasting Queries 175
6.3 Execution Plans 177
6.3.1 Initial Plan Space Considered ($\mathbf{\Phi}$) 180
6.4 Plan-Selection Preliminaries 183
6.4.1 Estimating Forecasting Accuracy of a Plan 184
6.5 Processing One-Time Queries 186
6.5.1 Overview of FPS 186
6.5.2 Ranking Attributes 188
6.5.3 Traversal of the Ranked List in Chunks 190
6.5.4 Generating a Plan from a Chunk 191
6.5.5 Decision to Generate a Plan or Not 193
6.5.6 Sharing Computation in FPS 194
6.6 Experimental Evaluation 195
6.6.1 Datasets, Queries, and Balanced Accuracy 195
6.6.2 Ranking Attributes 198
6.6.3 Traversal of Ranked List 200
6.6.4 Selecting a Predictive Attribute Subset 201
6.6.5 Selecting a Synopsis 203
6.6.6 Decision to Generate a Plan or Not 205
6.6.7 Effect of Sharing 207

6.7 Making FPS Concrete 208
6.8 More Transformations 211
6.8.1 Wavelet Transformer 211
6.8.2 Log Transformer 212
6.8.3 Difference Transformer 213
6.9 Continuous Forecasting Queries 214
6.9.1 FPS-Adaptive (FPS-A) 215
6.10 Related Work 219
6.11 Summary 221

7 Conclusions **223**

8 Future Work **228**
8.1 Multi-Goal Experimental Design 229
8.2 Evaluation with Other Systems 230
8.3 System Data Complexity 230
8.4 Incorporating Domain Knowledge 231
8.5 Policy-based System Management 231

Bibliography **233**

Biography **241**

List of Figures

1.1 Overview of Fa 5

2.1 Fa platform for managing a Web service system 15
2.2 Sample system monitoring data 16
2.3 System management queries 19
2.4 Fa architecture 21
2.5 Example execution plan for a forecasting query 29

3.1 Sample monitoring data used in diagnosis query processing 36
3.2 Control and data flow in Fa during diagnosis query processing . . . 38
3.3 (a) Clustering Vs. separating functions; (b), (c) improving robustness of signature databases 40
3.4 Signature databases: (a) SD_1, (b) SD_2 41
3.5 Signature databases: (a) SD_3, (b) SD_4 43
3.6 Sample AC-Curve 52
3.7 Sample data 56
3.8 Plot of data in Fig.3.1 58
3.9 Margin Classifier ($MC(F, C)$) 59
3.10 Dendogram 60
3.11 Margin along a diagnosis vector 61
3.12 Margin-based Clustering (MAC) 62
3.13 PCM-Conservative (PCM-C) 65

3.14 AC-Curves for Rubis-60, Rubis-complex 70

3.15 AC-Curves for Rubis-60 with two groups: (a) existing annotations, (b) new annotations 71

3.16 Normal probability plot for attribute $CPU\_utilization \in (72, 73)$ in OLTP-single 72

3.17 Cluster timeline for a subset of OLTP-single 73

3.18 Cluster transition diagram for a subset of OLTP-single 74

3.19 AC-Curves for Rubis-complex with Gaussian error: (a) error level = 1, (b) error level = 2 76

3.20 AC-Curves for Rubis-complex with Non-Gaussian error: (a) error level = 1, (b) error level = 2 77

3.21 Robustness curves for Rubis-complex: (a) Gaussian error, (b) non-Gaussian error 77

3.22 AC-Curves for Rubis-60 with Gaussian error: (a) error level = 1, (b) error level = 2 78

3.23 AC-Curves for Rubis-60 with Non-Gaussian error: (a) error level = 1, (b) error level = 2 78

3.24 Robustness curves for Rubis-60: (a) Gaussian error, (b) non-Gaussian error 79

3.25 Trend as the number of failures increases 79

4.1 Outline of our Falcon algorithm 100

4.2 Time-based chunking 108

4.3 Setting I: (a) Rubis-1, (b) Rubis-2 112

4.4 Setting II: (a) Rubis-1, (b) Rubis-2 113

4.5 Setting II: (a) Unbalanced Rubis-1, (b) Unbalanced Rubis-2 114

4.6 Setting I: (a) Rubis-1 (b) Synthetic 114

4.7 Setting II: (a) Rubis-1 (b) Synthetic 115

4.8 Setting I: (a) Rubis-2 (b) Synthetic 115

4.9 Setting II: (a) Rubis-2 (b) Synthetic 116

4.10 Setting II: (a) Rubis-1, (b) Unbalanced Rubis-1 116

5.1 2D projection of a response surface for TPC-H Query 18; total database size = 4GB, physical memory = 1GB 122

5.2 Steps in iTuned's Adaptive Sampling algorithm 131

5.3 Example GRS from five samples 137

5.4 Example of EIP computation 140

5.5 The .eX framework in action for standby databases 145

5.6 Impact of shared_buffers Vs. effective_cache_size for workload W4 (TPC-H SF=10) 156

5.7 Impact of shared_buffers Vs. work_mem for workload W5 (TPC-H SF=10) 157

5.8 Impact of shared_buffers Vs. effective_cache_size for workload W5 (TPC-H SF=10) 158

5.9 Comparison of tuning quality. iTuned's tuning times are shown in minutes (m) or hours (h). Ri denotes Rank i 160

5.10 Comparison of iTuned's tuning times in the presence of various efficiency-oriented features 163

5.11 Effect plot for workload W5 (SF=10) 167

5.12 Effect plot for workload W4 (SF=1) 168

5.13 Effect plot for workload W4 (SF=10) 169

6.1 Example datasets. (a) Usage; (b) and (c) are transformed versions of Usage 173

6.2 BN synopsis built from data in Fig. 6.1(c) 180

6.3 Plan selection and execution algorithm for one-time forecasting queries 186

6.4 Pictorial view of FPS processing Example query Q_1 from Section 6.1 187

6.5 (a) Comparing correlation metrics, (b) Correlation Vs. time-based ranking. These two experiments use the Periodic-large-tb dataset. . 198

6.6 Comparing correlation metrics: (a) Aging-real, (b) Multi-large-tb . 199

6.7 (a) Comparing correlation metrics (Complex-syn), (b) CFS-Score Vs. BA for random subsets(Periodic-large-tb) 200

6.8 (a) Importance of chunk-based traversal, (b) Choosing chunk sizes (Δ=30 in Fig. 6.3). These two experiments use the Periodic-large-tb dataset. 200

6.9 (a) Comparing attribute-selection techniques (Aging-real), (b) CFS-Score Vs. BA for random subsets (Multi-large-tb) 202

6.10 Comparing attribute-selection techniques: (a) Aging-variant-syn, (b) Aging-fixed-tb 202

6.11 Comparing attribute-selection techniques: (a) Multi-large-tb, (b) Periodic-large-tb 203

6.12 Comparing synopses: (a) FIFA-real, (b) Aging-fixed-tb 205

6.13 Comparing synopses: (a) Multi-large-tb, (b) Periodic-small-tb . . . 205

6.14 Comparing synopses: (a) Forecasting humidity in Motes-real, (b) Forecasting light in Motes-real 206

6.15 (a) CFS-Score-based pruning (Periodic-large-tb), (b) Improvements with sharing (Periodic-large-tb) 206

6.16 CFS-Score-based pruning: (a) Aging-fixed-tb, (b) Aging-real 207

6.17 CFS-Score-based pruning: (a) Multi-large-tb, (b) Periodic-small-tb 207

6.18 Improvements with sharing: (a) Aging-fixed-tb, (b) Aging-variant-syn 208

6.19 Improvements with sharing: (a) Multi-large-tb, (b) Complex-syn . . 208

6.20 FPS Vs. State-of-the-art synopsis (RF): (a)Aging-real, (b) Complex-syn 209

6.21 FPS Vs. State-of-the-art synopsis (RF): (a) Multi-large-tb, (b) Aging-variant-syn 210

6.22 Extended Plan Selection Algorithm 212

6.23 Using Wavelet transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses 213

6.24 Using Log transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses 213

6.25 Using Difference transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses 214

6.26 FPS-Adaptive (FPS-A) 216

6.27 Adaptivity and convergence properties 218

List of Tables

3.1 Notation table 40

3.2 Monitoring datasets used in the evaluation. Columns a and i are the number of attributes and instances respectively. Datasets 1-3 are used to evaluate Phase I, and datasets 4-9 to evaluate Phase II . . . 67

3.3 Comparing running times (seconds) 80

3.4 Comparing diagnosis accuracy. Numbers like 1st and 2nd indicate the rank of the cluster C whose diagnosis vector contains the relevant attributes in decreasing order of cluster size (smaller rank is better). The % value is the size of C wrt the number of historical points $|H|$. The third integer value indicates the number of diagnosis vectors returned after filtering with a support threshold of 2% 82

3.5 Comparison with previous approaches 92

4.1 Datasets used in Section 4.5 110

5.1 Notation used in this chapter 125

5.2 Features that improve iTuned's efficiency 147

5.3 Overheads of operations in the .eX framework 154

5.4 Comparison of tuning quality in terms of workload running time after tuning (all the tuning times are shown in seconds.) 164

5.5 Sample of iTuned's results 165

5.6 Sensitivity analysis. For W2, W3, W6, W8, rank and performance of best setting (secs) are shown. Lower is better 165

5.7 Sensitivity analysis. For W2, W3, W6, W8, rank and performance of best setting (secs) are shown. Lower is better 166

6.1 Defaults for experiments 195

6.2 Datasets used in experiments; n, m, and I are the number of attributes, tuples, and measurement interval (minutes) respectively; real, tb, and syn represent real, testbed, and synthetic datasets respectively 196

6.3 Synopsis comparisons. The time in seconds to produce the best plan, and the corresponding BA, are shown. The missing data points are cases where Java ran out of heap space before convergence. 204

6.4 FPS Vs. RF; The datasets had to be scaled down to get RF to run within reasonable time 210

6.5 Run-time overhead. Values for FPS-A and Strawman are tuples processed per second. % is FPS-A's degradation relative to Strawman . 219

Acknowledgements

First, I would like to express my deep and sincere gratitude to my adviser Dr. Shivnath Babu for his invaluable guidance and generous support during my graduate studies at Duke. I am lucky to be one of Shivnath's first students and work closely with him on several projects from the very beginning. I learned a lot from Shivnath regarding how to build up solid projects and how to perform high-quality research as a computer scientist. I benefit enormously from Shivnath's vision in database and systems research and his help to improve my writing and presentation skills. This dissertation would not have been possible without Shivnath.

I greatly appreciate the help from my dissertation committee members Kamesh Munagala, Sandeep M Uttamchandani, and Jun Yang for their valuable feedback and suggestions on this dissertation from various angles. I would like to thank the Department of Computer Science at Duke, especially all the members in the database group.

I would like to thank Hui Zhang at NEC Labs America and Badrish Chandramouli and Jonathan Goldstein at Microsoft Research for their mentorship during my internship there. I really enjoyed the collaboration with them. These internships gave valuable experience that broadened my research vision and strengthened my ability to handle large-scale real-world problems.

I own a lot to my parents and parents-in-law. They always stand behind me with their everlasting love and support and try their best to provide me the education

opportunity they never had. All my achievements belong to them.

Finally and most importantly, I would like to convey my deep love for my wife, Jingjing Li, who accompanied me through all the ups and downs of the graduate studies with a firm belief in my potential as a researcher. She always loves me, encourages me, and inspires me to be a better person in many aspects. I cherish all the happiness she continually brings into my life. This dissertation is also for her!

Chapter 1

Introduction

Due to the rapid advances in computing power and network capabilities over the past few decades, large-scale networked computing systems are now deployed widely to run popular Web services and business-critical applications. For example, Google's services were run on about 24 server farms (as per data from 2006) consisting of 450,000 servers around the world [44]. Corporate data centers usually house thousands of interconnected components including storage systems, database servers, applications, and Web services [50]. As system scale increases, so does the complexity that arises from the interactions between system hardware, software, and workloads [25]. Furthermore, these systems are highly dynamic environments where data, workloads, and system characteristics change over time. For example, a Web site may experience a surge in traffic due to a newly released advertisement, which could lead to unexpected system behavior. Even a small change in hardware or software may have a big impact on the overall system performance.

1.1 Complexity of System Management

The scale, complexity, and dynamic characteristics of networked computing systems make it hard for administrators to understand system behavior and perform

effective management. A study [71] found that 72% of the top-40 Web sites suffer user-visible problems, such as slow responses, blank pages or error messages being displayed, items not being added to shopping carts, and unexpected database slowdowns. Although the reliability of individual hardware units has improved significantly in recent years, a networked system comprising tens of thousands of such units, maybe from multiple vendors, is still notoriously unreliable. Meanwhile, despite the progress in programming languages and software design principles, system software is still far from being bug-free. In addition to hardware failure and software bugs, another critical factor that causes system problems or outage is human-operator error; misconfiguration is a leading cause [60]. Networked systems have so many parameters to set that administrators can easily make mistakes during the system deployment or tuning phases. For instance, Web service systems, middleware (e.g., WebSphere) or database systems (e.g., Oracle, DB2) have several hundred parameters that need to be tuned carefully to achieve desired performance. Moreover, these complex components may be integrated with other complex parts to compose a complete system. Since these components are interdependent, tuning configuration parameters of one component may impact the behavior of other components, and even result in a system-wide problem [51].

While managing networked systems is becoming increasingly challenging, it is more important than ever to ensure system reliability, availability, and serviceability. Poor system performance or service downtime can lead to user dissatisfaction and huge loss of revenue. For example, brokerages and banking firms can lose up to $75,000 per minute of downtime [48]. A 22-hour outage at eBay cost the company more than $3 Million in customer credits and $6 Billion in market capitalization [50].

Although there exist several commercial frameworks such as HP's OpenView [59]

and IBM's Tivoli [79] that aggregate system monitoring data and enable administrators to visualize this data, these tools are insufficient to provide the intelligence necessary for effective system management [25]. Administrators often rely on rules-of-thumb or write their own scripts in an *ad-hoc* way for various management tasks. Unfortunately, rules can be easily invalided as the managed system evolves, and the scripts may take a lot of time to write and tune on a per-task basis. Since administrators have so many complex tasks to deal with, they often do not realize the opportunities for system tuning or diagnosis. What is even worse is that administrators may inadvertently cause system misconfiguration that lead to system problems [60].

The scale and complexity of modern networked systems have made it impossible for any individual administrator to understand the entire system behavior. These systems are typically managed by a team of administrative staff, with each individual administrator responsible for a small system component. A challenging situation is that system components could be from different vendors and managed by different teams or even different companies. Consequently, system-wide diagnosis or tuning become time-consuming due to the manual interactions and communication required. Because of the lack of useful tools for system management, the effectiveness of managing complex systems depends highly on the expertise and skills of administrators; but experienced administrators are expensive to get and retain. All these factors conspire to drive the total cost of system ownership up to 18 times the original purchase price [63].

The complexity and indispensability of effective system management have motivated a research trend towards *autonomic computing* [48], i.e., making systems self-managing. However, this field is still at a very initial stage. This dissertation

makes major steps towards realizing the vision of autonomic computing by providing a platform where administrators can specify management tasks easily and get actionable intelligence automatically.

1.2 Data-driven System Management

The progress in monitoring tools (e.g., DTrace [17], SystemTap [77], sar [69], OProfile [58], VTune [85]) enables administrators to collect fine-grained data about system activity and configuration with low overhead. These tools can collect data at multiple system levels, including hardware-level metrics such as power consumption, network-level metrics such as packet rate, operating-system-level metrics such as CPU utilization, and application-level metrics such as throughput. Valuable information for system management may be hidden in the monitoring data. The more detailed the data that is collected, the higher the chance that information needed to address a specific management task (e.g., diagnosis of a system problem) is contained in the data. As system scale and complexity increase, the size of monitoring data also grows drastically. The data collected from a busy corporate data center can be easily over one Terabyte per day, or much larger if more fine-grained monitoring is enabled [90]. Data of this size is well beyond human ability to analyze manually. It is impractical for administrators to wade through the huge amounts of monitoring data to find information relevant to the management tasks at hand; which is like looking for a needle in a haystack.

Recently, there is a trend towards applying data mining techniques to analyze system monitoring data and to extract information for various management tasks, e.g., [24,25,65]. Such a *data-driven* approach to system management has the advantage of being more robust to dynamic characteristics of the managed system than

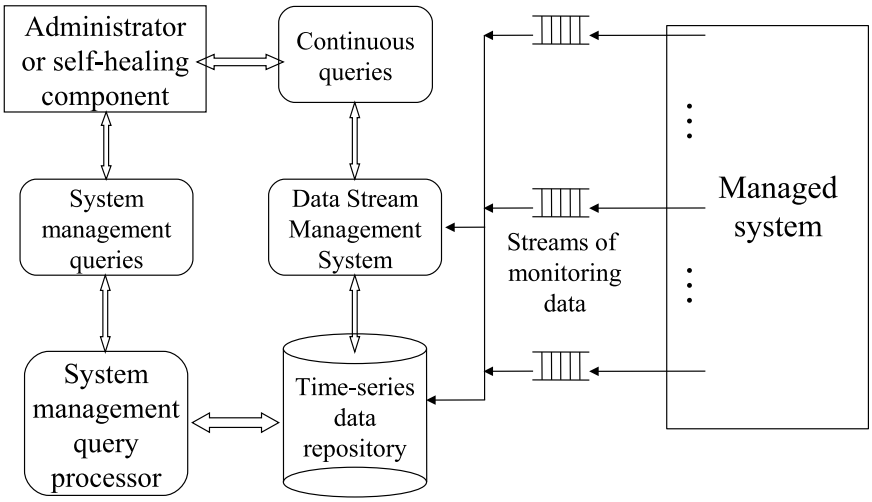

Figure 1.1: Overview of Fa

those approaches dependent on *a priori* knowledge of the system. However, the previous work was mostly done on a per-task basis, each focused on one aspect of system management. What administrators really need is a platform, with a simple and intuitive interface, where they can easily specify system management tasks of interest and get actionable intelligence quickly and with reasonable accuracy.

In this dissertation, we build such a platform, called *Fa*[1], for data-driven system management. Fa treats a complex networked system as a rich source of monitoring data (also called *system data*), and supports queries from administrators or *self-healing components* (that automate the process of maintaining system performance at a satisfactory level). Figure 1.1 shows a high-level overview of Fa. As the managed system runs, monitoring data is periodically collected at specified intervals. The data is fed into a data stream management system to support *continuous queries* over the data [6, 7, 33]. For instance, an *observational query* from a system administrator may ask "how many transactions had a response time over 5 seconds

[1]African god of fate and destiny

during the past hour". On the other hand, administrators or self-healing components can pose *system-management queries* such as "what will average transaction response time be one hour from now". Fa's query processor automatically finds a good execution strategy that analyzes the collected system data and produces a query result, usually with confidence estimates about the result.

1.2.1 Research Questions

To make Fa a practical and useful platform to assist administrators in managing large-scale and complex systems, we need to address the following research questions:

- **What is the right interface for expressing queries over system data?** The interface needs to be simple to use and intuitive to express both (i) observational queries in query languages such as SQL (for *snapshot queries*) and CQL [7] (for continuous queries), and (ii) management queries for tasks such as diagnosing system problems in a declarative fashion.

- **What is the right format to represent query results?** Visualization of the *supporting evidence* used for generating query results is useful for administrators to validate the reasoning behind result generation. Also, features like reliable confidence estimates about query results can help administrators to decide how much to trust the results, and gradually build confidence in the Fa platform.

- **How to find a good execution plan to process a given query automatically and efficiently?** There may be a huge space of execution plans for a given query. Fa's query processor needs to have an efficient plan search

algorithm that automatically identifies good execution plans, even when there is no model to estimate the quality of a plan without running it.

- **How to design practical algorithms to analyze massive, noisy, and streaming system data?** Monitoring data collected from large-scale networked systems is often huge in volume and arrives as continuous data streams, so it is infeasible for any analytic algorithm to make multiple passes over the data. The massive and streaming nature of the data raises challenges from both the efficiency and and the accuracy perspective. In addition to the massive data size, monitoring data is always noisy due to natural system dynamics and external influences. The analytic algorithms need to be *robust* to the superficial noisy property of system data so that reliable results can be generated for given queries.

- **How to detect situations where existing system data is insufficient to generate a high-quality result for a given query?** It is possible that the system data that has been collected does not cover the entire system operating range well, and query results based on such data can be inaccurate and even misleading. For example, the knowledge learned from system data collected for one workload type W_1 may be inapplicable for tuning the system in the face of a workload type W_2 that is quite different from W_1. It is important to detect such situations of *data inadequacy* to avoid producing inaccurate information that might lead to misconfiguration or improper tuning.

- **How to design an active data collection strategy for missing information necessary to process a query with minimal *cost*?** Active data collection may incur extra costs in terms of system resources, time, and hu-

man efforts. For instance, an experiment that collects performance data for a given workload at one configuration setting could take considerable amounts of system resources and several hours to complete. To process a configuration tuning query, it may require multiple such experiments to collect enough performance data in order to produce a good tuning recommendation. Therefore, active data collection needs to maximize the value of each *experiment* (i.e., minimizing the total cost of experiments) while improving the quality of the result for a given query.

1.3 Contributions

This dissertation describes the Fa data-management platform where administrators can pose queries declaratively over monitoring data collected from the managed system, and get reliable and actionable intelligence from Fa's query results, in an automatic and efficient manner. Specifically, this dissertation makes the following contributions.

- Fa provides simple and intuitive interfaces for administrators to express system management queries (specifically, diagnosis, tuning, and forecasting queries are the focus of this dissertation) declaratively [33, 37]. The query results can be visualized by administrators, possibly with supporting evidence and confidence estimates for the results.

- For automated processing of diagnosis queries, Fa uses a two-phase approach to make best use of two distinct types of system data: (i) data about previously diagnosed problems and (ii) data about normal system behavior. In Phase I, Fa constructs a database of problem signatures that distills the essential

properties of problems that have already been diagnosed from system data, and matches undiagnosed problems against the signature database; producing reliable confidence estimates for matches. In this way, Fa detects recurrent problems and leverages past diagnostic efforts through *annotations* (e.g., root cause or corrective action) associated with the problem signatures. For an undiagnosed problem without high-confidence matches, Fa triggers Phase II, where a novel *anomaly-based clustering* technique is used to group normal system behavior data based on how they deviate from the data representing the problem to be diagnosed. Fa characterizes the deviation with a few concise attribute sets that pinpoint possible causes of the problem. Both the signature construction and matching techniques in Phase I and the anomaly-based clustering technique in Phase II are robust to noise that is common in monitoring data from production systems. This part of the dissertation was published in [38].

- Phase II of diagnosis query processing has a harder task than Phase I. Intuitively, Phase I deals with a *supervised learning* problem, while Phase II deals with an *unsupervised learning* problem. Therefore, Phase II usually has poorer diagnosis accuracy and requires more human efforts than Phase I when the latter produces high-confidence matches from the signature database. However, to construct a high-quality signature database, Phase I requires a sufficient number of diagnosed instances of each problem type; which might not be the case in the available system data. Fa incorporates an *active-learning* component that picks informative undiagnosed problems and presents them to administrators for diagnosis. The signature database is then updated with the data and annotations from the newly diagnosed problems. The objective

of the active-learning component is to maximize the informative value that will result from each undiagnosed problem selected; thus, minimizing the total amount of manual diagnostic effort involved while improving the quality of the signature database to a desired level. This part of the dissertation was published in [36].

- For automated processing of configuration tuning queries, Fa adopts an *experiment-driven* tuning approach that plans and conducts experiments to find optimal settings of system configuration parameters. Gaussian process regression models are learned from system data (collected from historical runs or through planned experiments) to represent the response surface of a target performance metric with regard to system configuration parameters; *confidence intervals* are generated around the predicted performance at each hypothetical configuration setting. The response surface can be visualized by administrators to gain insights into the behavior of the system in its operating range. Fa employs a novel *adaptive sampling* technique that picks iteratively from a set of candidate experiments the one with maximum estimated *utility* for the next run based on the response surface learned so far. The adaptive sampling technique is able to quickly eliminate parameters with low performance impact from the tuning process, and identify potential high-performance settings with confidence estimates. Fa also employs a novel experimentation technique that allows conducting planned experiments in a production environment through a cycle-stealing paradigm while ensuring near-zero overhead on the production workload. This part of the dissertation was published in [40].

- Fa considers two type of forecasting queries, namely, one-time queries over

data snapshots and continuous queries over data streams. For a one-time forecasting query over data snapshots, Fa employs a plan selection algorithm that automatically searches in the huge space of execution plans. Each plan consists of a sequence of various data transformers and synopsis learning and prediction operators. Fa's plan selection algorithm converges quickly to fairly-accurate plans by running as few plans as possible. For continuous queries over data streams, Fa has an adaptive plan selection algorithm that automatically adapts to the time-varying properties of the data with minimal overhead. Along with the prediction result to a forecasting query, Fa also outputs a reliable estimate about the prediction accuracy. A novel feature of Fa is how it balances the accuracy of a prediction and the computation time needed to generate this prediction. This part of the dissertation was published in [34].

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 briefly introduces multi-tier systems and the format of the system data we work with in this dissertation. We also give an overview of the Fa platform.

Chapter 3 presents the techniques for processing diagnosis queries. We first describe how to construct a database of problem signatures from system data and how to use the signature database for diagnosis. We then describe the anomaly-based clustering technique for diagnosing problems without high-confidence matches from the signature database.

Chapter 4 presents an active learning technique to improve the overall accuracy for diagnosis query results. The active learning technique intelligently picks informative undiagnosed problems for administrators to diagnose. The diagnos-

tic information from the newly diagnosed problems is then incorporated into the signature database to improve its coverage and matching quality.

Chapter 5 presents the experiment-driven technique for processing tuning queries. This technique employs adaptive sampling with Gaussian process regression models. We apply the tuning technique in database systems to set their configuration parameters.

Chapter 6 presents the techniques for processing forecasting queries. For snapshot forecasting queries, we present a plan selection algorithm that automatically identifies good execution plans composed of data transformation operators and model learning and prediction operators. For continuous forecasting queries over data streams, we present an adaptive plan selection algorithm to handle time-varying data properties.

Finally, Chapter 7 discusses future work and concludes.

Chapter 2

Overview

2.1 Multi-tier Systems and System (Monitoring) Data

Architecturally, popular Web services deployed for e-commerce, report generation, and other applications consist of three tiers:

- The bottom tier is the database and storage layer which manages persistent data to process queries from the higher tiers.
- The middle tier is the application logic layer which implements the business logic that represents the core functionalities of the Web service.
- The top tier is the presentation layer consisting of Web servers that serve static and dynamic content while interacting with the lower tiers to process incoming user requests.

In this dissertation, we chose Web service systems as our focus system since these systems raise various challenges for management tasks, thereby offering comprehensive evaluation opportunities for the Fa platform:

- In complex multi-tier systems, performance and availability problems happen frequently, and it is hard to find the cause. Recent studies show that 20% to 30% of the problems in Web service systems remain undiagnosed [70], and more than 50% of the problems are recurrent ones [60]. The monitoring data collected from these systems makes it possible to evaluate automated diagnosis query processing techniques in Fa comprehensively.

- The setting of database configuration parameters has a big impact on the overall performance of a Web service system. However, since there are many configuration parameters in database systems (e.g., over 100 parameters in DB2, Oracle, and PostgreSQL), database misconfiguration is a common cause of system problems [51]. The issue of configuration tuning in database systems is thus a good setting for evaluating the tuning query processing techniques in Fa.

- Workloads running on Web service systems often display *periodic* and *trend* patterns that present realistic settings for evaluating the forecasting query processing techniques in Fa.

Figure 2.1 is an overview of the Fa platform with a Web service system as the managed system. Monitoring data can be collected for a Web service system at multiple levels:

- At the operating-system level, a tool like *sar* [69] can track low-level metrics such as processor utilization, memory utilization, and number of I/O transfers to the disk subsystem.

- At the database and Web server level, there are status variables about server operation. For instance, MySQL database systems have over 200 status vari-

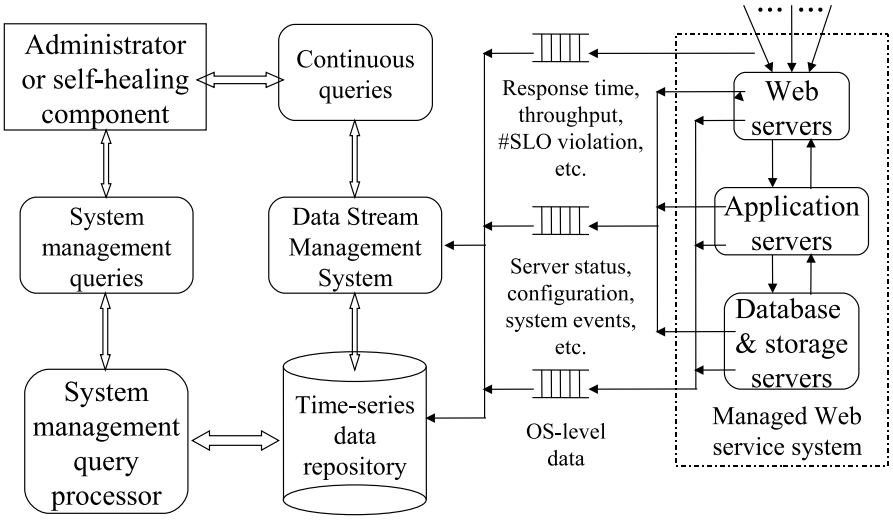

Figure 2.1: Fa platform for managing a Web service system

ables like number of index accesses, number of full table scans, and locks acquired with zero waiting [57].

- At the application level, monitoring tools can track high-level metrics like throughput (i.e., number of requests processed during each measurement interval) and average request response time.

While the system is running, monitoring tools can periodically collect such data at specified measurement intervals and store it into database or storage servers. Figure 2.2 shows a snapshot D in time of the data with a relational schema $\langle \Gamma, X_1, X_2, \ldots, X_n \rangle$. Γ is a *timestamp* attribute with values drawn from a discrete, ordered domain $dom(\Gamma)$. X_i $(1 \leq i \leq n)$ is a system metric. The monitoring data is essentially a high-dimensional time series. Commercial monitoring frameworks such as HP's OpenView [59] and IBM's Tivoli [79] collect similar data.

Web service systems are often required to meet *service level objectives* (SLOs) that are specified by users or third parties implicitly or explicitly. For instance, an

Measurement interval (1 minute)	OS-level data		Server status variables			Application-level data		
	cpu_util	…	#idx_acc	#table_scans	…	resp_time	…	slo_vio
10	16%	…	37	32	…	4	…	0
11	22%	…	41	36	…	3	…	0
12	72%	…	72	67	…	8	…	1
13	84%	…	70	65	…	10	…	1
14	53%	…	39	26	…	5	…	0
15	20%	…	26	18	…	2	…	0
…	…	…	…	…	…	…	…	…

Figure 2.2: Sample system monitoring data

SLO for an online brokerage may stipulate that all transactions complete within 1 second, regardless of how much middleware, databases, or networks are involved. Violations of SLOs indicate that there are system problems or *system failures*. When a system meets all specified SLOs, it is in a *healthy state*; otherwise, it is in a *failure state*. Failures may be caused by a variety of factors including performance problems like resource contention, crashes due to hardware failures or software bugs, and misconfiguration by system administrators.

2.2 System Management Queries

One of the goals of system administrators is to ensure that the managed system continuously meets SLO requirements. Most of the administrators' time is spent looking for answers to various management queries over monitoring data. Next we show some example management queries.

Change-detection queries: Changes in system behavior may indicate performance problems or malicious attacks, so it is important to detect unexpected

changes in real-time. To that end, an administrator may ask change-detection queries like:

- Was there any significant change in the intensity of the workload faced by the Web server within the last hour?
- Has the utilization of system resources such as CPU, memory, and network changed drastically in usage pattern?

Diagnosis queries: Administrators often have a hard time figuring out the causes of service outage or poor system performance, since diagnosis is very challenging in complex networked systems. Below are some sample diagnosis queries for which an administrator may have to find the answer at short notice:

- Why was there a big increase in the processing time of query Q in the database server over the past hour?
- Why was there a slowdown in the end-to-end batch processing of the business intelligence workload, while it was fine last week?

What-if queries: Before applying a change to the production system, a common practice for administrators is to estimate and validate the impact of the proposed change on the overall system performance. What-if queries can be used to estimate the performance impact of a change as follows:

- What will the query processing time be if the database buffer pool size is doubled?
- How will the overall report generation time change if three instances of query type Q' are added to the query batch workload?

Tuning queries: A frequent tuning task that administrators face is to adjust the setting of system configuration parameters, as misconfiguration is a major cause of system problems. For this purpose, administrators may need answers for tuning queries like:

- What is the right database buffer pool size to resolve the performance problem caused by the I/O bottleneck?
- What should be the best multi-programming level setting in the database server to avoid concurrency problems for my business intelligence workload?

Forecasting queries: Accurate and timely prediction of system problems can be very useful — administrators will gain more time for diagnosis and will also be able to take remedial actions proactively to prevent problems from actually occurring. To that end, administrators may ask forecasting queries like:

- Will there be a performance problem in the next hour in the sense that the average query processing time is over 10 seconds in the database system?
- Given the historical data about workloads running on the Web server, what are the chances of a surge in workload intensity in the next hour?

System performance management There are four key steps involved in manual management of system performance:

(1) Detect system problems, by monitoring target performance values and comparing them against thresholds specified in SLOs.

(2) Diagnose the detected problem to find the root cause, by searching past diagnosis history or trying to correlate current and historical system data with the problem.

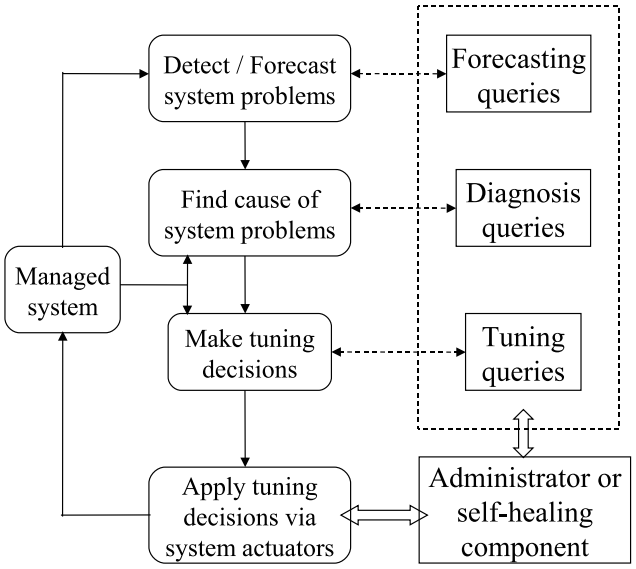

Figure 2.3: System management queries

(3) Decide tuning strategies, by applying rules-of-thumb or custom scripts.

(4) Evaluate the proposed tuning strategy on the managed system to see if it solves the problem; go to step (1) and continue the loop.

As these steps rely much on trial-and-error and the expertise of administrators, manual performance management can be reactive, time-consuming, and error-prone. In comparison, the Fa platform can provide administrators with an altogether different experience (see Figure 2.3):

(1) Rather than being reactive, Fa enables proactive management through forecasting queries that predict future system performance and identify potential system problems automatically.

(2) Administrators can issue a diagnosis query to find the cause of a detected or predicted problem.

(3) Once a problem is diagnosed, administrators can pose a tuning query to solve the problem. Fa's query processor will automatically search and compare the candidate tuning strategies and return the one that is potentially most effective and cost-efficient.

(4) Administrators can either apply the recommended tuning strategy to the production system or wait to get more information to answer the diagnosis or tuning query more accurately.

To process diagnosis queries, Fa may need to detect changes in data patterns to pinpoint the root cause. Also, what-if queries may be triggered to answer tuning queries when comparing different potential tuning recommendations.

2.3 Architecture of Fa

The Fa platform allows administrators or self-healing components to pose queries over system data and get answers through a declarative interface. The technical details of how the queries are processed are hidden from users. As shown in Figure 2.1, administrators may pose conventional continuous queries (over the system data treated as data streams) or system-management queries to Fa's management-query processor. The development of Fa involves several design decisions (see Figure 2.4):

- Interface for interacting with the managed system, including data collection through monitoring tools and changing system settings through *tuning knobs*.
- Query interface for expressing management queries and returning query results; the interface needs to enable system management tasks to be described in a declarative way and intuitive presentation of query results.

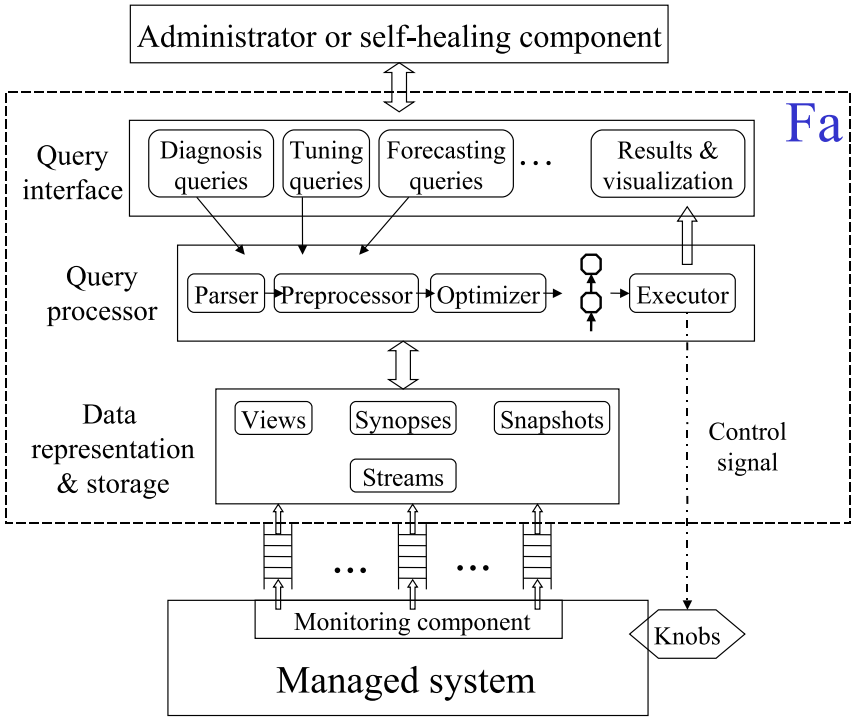

Figure 2.4: Fa architecture

- Data representations that provide the basis for efficient and accurate query processing.

- Query processor, which parses a submitted query Q, does any preprocessing required, finds an efficient and fairly-accurate execution plan for Q, and finally executes the plan to generate query results.

2.3.1 Interfacing with the Managed System

Recent progress in systems research has made system monitoring and operation easier. Some monitoring tools such as DTrace [17] provide flexible control on the granularity and overhead of data collection through *software probes*, which act like hardware sensors to track the status of system components of interest. Fa uses these

tools to collect fine-grained monitoring data with controlled overhead. System operation has also been simplified. For instance, many system configuration parameters can be now changed at run-time without shutting down the system [43, 57]. Mechanisms for *microreboot* [16] enable fine-grained rebooting of system components without restarting the entire system. Microreboots can be done orders-of-magnitude faster than full system reboots. System virtualization techniques make it possible to monitor, isolate, and dynamically adjust system resources easily. All these techniques provide Fa with a lot of opportunities for data collection and system control.

2.3.2 Interface for Expressing Queries

The query interface should be user-friendly with (i) intuitive ways to express queries over system data, and (ii) query results that are easy to interpret. Fa needs to support both *snapshot queries*, which are processed once on a certain version of the data, and continuous queries, the results of which are continuously updated as new data comes in. Furthermore, to realize proactive system management, Fa needs to support smooth composition of forecasting queries with diagnosis and tuning queries. When possible, query results should be visualized to help administrator make sense of the results and the associated confidence estimates. Next we briefly describe how to express forecasting, diagnosis, and tuning queries in a declarative way.

Diagnosis query A diagnosis query can be expressed in Fa as "$Q = Diagnose(F, History)$", where F is the data collected during a system problem (or right before the problem in case of a crash), and $History$ is the system data collected so far. This query asks for diagnosis of the problem represented by F, based on the information

contained in F and the historical data represented by $History$. Diagnosis queries are discussed further in Chapter 3.

Tuning query A tuning query can be expressed in Fa as "$Q = Tune(W, X, DOM, y)$", where W describes the properties of the workload running on the managed system (e.g., number of users accessing the system), X is a set of system configuration parameters (e.g., buffer cache size, CPU resource allocation) that need to be tuned, DOM is the space of possible settings of these parameters that the system can have, and y is the performance metric of interest (e.g., throughput, average response time). The result of this query is a setting of X that is expected to achieve the optimal performance for y. Tuning queries are discussed further in Chapter 5.

Forecasting query A forecasting query can be expressed in Fa as "$Q = Forecast(D, X_i, L)$", where D is a data snapshot or data streams with most recent timestamp τ. The result of this query is the forecast of attribute X_i at time $\tau + L$. Forecasting queries are discussed further in Chapter 6.

Visualization When possible, query results should be visualized to help administrators interpret the results. For example, part of the result of a diagnosis query can be visualized with the failure data F and a group of data points that represent a *system baseline* (i.e., data about a specific type of normal behavior of the system) — the difference between them is characterized to generate the diagnosis result. Such visualization gives administrators deeper understanding of the diagnosis result so that they can take actions for problem resolution with more confidence.

2.3.3 Representation of Base and Derived System Data

Since monitoring data from complex networked systems is massive and noisy, it is critical to transform and store the data in appropriate formats to facilitate query processing. Currently, Fa supports the following representations of system data:

Data stream Monitoring data arrives continuously from the managed system, so it is natural to choose the data stream model. A stream is a bag of elements $\langle t, e \rangle$, where e is a record (with a well-defined schema $[X_1, X_2, \ldots, X_n]$) that is generated at time t. The bag size grows as time goes by and could be infinite. The data stream model is important to extract information from monitoring data in a real-time fashion (e.g., to enable rapid detection or prediction of performance problems). A lot of work has been done on query processing in data stream management systems (e.g., [1, 7]).

Snapshot To provide time-series analysis over historical data, we adopt the snapshot model — a bag of unordered records or a bag of records ordered by their timestamps.

View Like traditional databases, we employ the concept of views to store a version of data that is derived from the base system data for efficient query processing. The views can be in the format of snapshots or data streams, and they could be logical views without materialization. For example, we could create a view composed of system configuration settings and system performance values over time. Then, administrators can pose a diagnosis query to check whether misconfiguration is the cause of a system problem, by detecting the correlation between the configuration settings and the corresponding performance values in the view.

Synopsis A synopsis is an abstract representation (i.e., summarization) of the relationships between system metrics, such as low-level metrics about system activity and high-level metrics about system performance. For instance, a synopsis may be represented with a graphical model (e.g., Bayesian network [88], queueing network model [54]) or a quantitative model such as a multivariate linear regression model [88]. In this dissertation, we evaluate the hypothesis that there are statistical relationships between low-level measurements such as OS-level metrics and high-level performance metrics such as SLO state (i.e., violation or not). Synopses are used to capture such relationships and provide the basis for processing system-management queries. For instance, synopses can be used to:

(1) Construct problem signatures for processing diagnosis queries.

(2) Construct response surfaces for processing tuning queries.

(3) Extract predictive patterns from historical data for processing forecasting queries.

Furthermore, as synopses learned from system data can extract and represent system behavior, query results based on the learned synopses are often more robust to noise in the data compared to results based on base data.

2.3.4 Query Processor

Fa's query processor is responsible for parsing a given system-management query, preprocessing the query, finding a *good* execution plan for it through the query optimizer, and executing the plan to produce query results. In principle, these steps in Fa's query processor are similar to the corresponding ones in conventional database systems, but the actual techniques involved in each step differ significantly.

Query preprocessor Sometimes accuracy and efficiency concerns or requirements make it necessary to rewrite a user query. Consider the case of diagnosis queries. There exist several diagnosis techniques developed for different systems, different performance problems, and different types of system data; and there is no general winner as we have shown [35]. Fa's query preprocessor may transform a user query into a series of queries, each of which can be best processed by a specific diagnosis technique. The transformation depends on the existing system data, the diagnosis techniques in consideration, and the current system state. After rewriting, query processing may take less time, and more importantly, the final query results may be more accurate.

Query optimizer The execution plan for a system management query is significantly different from the execution plans for conventional SQL queries in relational databases. Consider the case of forecasting queries. An execution plan for a forecasting query consists of a series of *data transformers* and a *predictor* which is based on a synopsis learned from the transformed data. (Section 2.4 shows an example execution plan.) For system management queries, it is important to find an execution plan that produces accurate query results. At the same time, it is also important to minimize the overall query processing *cost*, which includes the time required to find such a plan and execute it to generate query results. Conventional query optimizers typically employ a cost model to estimate the quality of a plan in the plan space. However, for system management queries such as forecasting queries, it is almost impossible to get a reliable estimate of the accuracy of a plan without running the plan. Fa's query optimizer then needs to balance the time spent on query optimization and the quality of the best plan found. For system management queries such as diagnosis queries, another factor is the human effort involved to

generate diagnosis results, which needs to be considered in Fa's query optimizer as another optimization metric. All these special characteristics of system management queries require Fa's query optimizer to be novel and fundamentally different from conventional query optimizers.

Query executor Query executor runs the plan found by the query optimizer and returns the results to users. For forecasting queries, the results may contain predicted values, confidence about the prediction, and the synopsis used to generate the prediction. A synopsis is returned to help users understand why the specific prediction was produced [33]. A side benefit is that users may derive useful information from the synopsis itself, especially when the synopsis is a graphical model (e.g., Bayesian network, decision tree). For diagnosis queries and tuning queries, it is possible that existing system data is insufficient to generate query results with high accuracy. The query executor needs to detect such a situation of data inadequacy and also identify what data needs to be collected to improve query results.

- Consider the case of diagnosis queries. If the instances of diagnosed problem types in existing system monitoring data are insufficient to produce a high-confidence result for a diagnosis query, then the output of Fa's query executor may be data about an undiagnosed problem carefully selected for additional user input; once it is diagnosed, the accuracy of diagnosing this type of problem will be improved.

- Consider the case of tuning queries. The execution plan often involves a synopsis that represents the relationship between the target performance metric and the system configuration parameters that need to be tuned. If the existing system data only covers a small part of the operating range, then the

synopsis learned from such data may be inaccurate for the rest of the operating range. In this case, one intermediate output from Fa's query executor could suggest running experiments on the managed system at a set of specific settings. With data collected from the newly performed experiments, the quality of the synopsis will be improved, resulting in a more accurate result for the tuning query.

2.4 Illustration with Forecasting Queries

Next we illustrate how Fa's query optimizer automatically generates an execution plan for a forecasting query "$Q = Forecast(D, X_i, L)$". Recall that this query predicts the value of attribute X_i at time $\tau + L$ based on the data snapshot D, where τ is the current time and L is the *lead time* for forecasting.

2.4.1 Execution Plans

A plan for a forecasting query consists of a summary data representation, *synopsis*, and three types of logical operators—*transformers*, *predictors*, and *synopsis learners*. Figure 2.5 shows the structure of an example execution plan.

- A *transformer* $T(D)$ takes a dataset $D(\Gamma, X_1, X_2, \ldots, X_n)$ as input, and outputs a new dataset $D'(\Gamma, Y_1, Y_2, \ldots, Y_N, Z)$ that may have a different schema from D.

- A *synopsis* $Syn(\{Y_1, \ldots, Y_N\}, Z)$ captures the relationship between attribute Z and attributes $Y_1, \ldots, Y_N$, such that a *predictor* $P(Syn, u)$ can use Syn to estimate the value of Z in a tuple u from the known values of $Y_1, \ldots, Y_N$ in

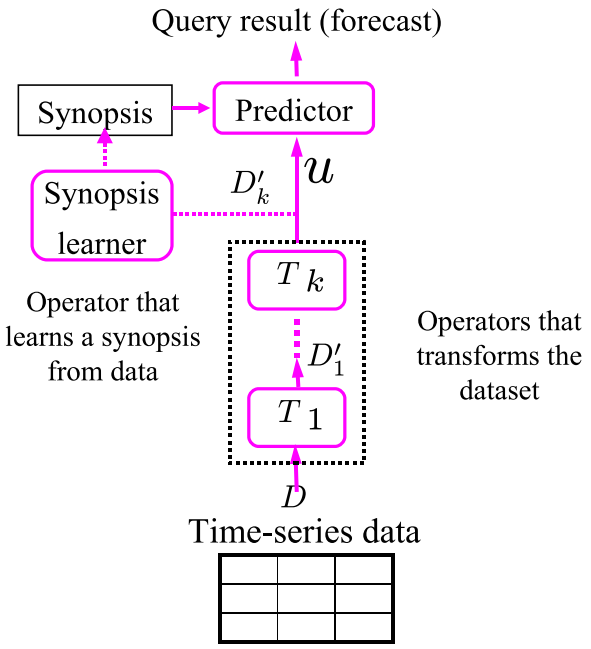

Figure 2.5: Example execution plan for a forecasting query

u. Z is called Syn's *output attribute*, and $Y_1, \ldots, Y_N$ are called Syn's *input attributes*.

- A *synopsis learner* $B(D, Z)$ takes a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$ as input and generates a synopsis $Syn(\{Y_1, \ldots, Y_N\}, Z)$.

Next, we give two example physical implementations of transformers.

Project transformer: A project transformer π_{list} retains attributes in the input that are part of the attribute list *list*, and drops all other attributes in the input dataset; so it is similar to a duplicate-preserving project in SQL.

Shift transformer: Shift(X_j, δ), where $1 \leq j \leq n$ and δ is an interval from $dom(\Gamma)$, takes a dataset $D(\Gamma, X_1, \ldots, X_n)$ as input, and outputs dataset $D'(\Gamma, X_1,$

..., X_n, X') where the newly-added attribute $X'(\tau) = X_j(\tau+\delta)$. When δ is positive (negative), then X' is copy of X_j that is shifted backward (forward) in time.

Next, we discuss two popular synopses and their corresponding learners and predictors from the machine learning literature. There are many other synopses that Fa supports, e.g., support vector machines, classification and regression trees, and random forests [88].

Multivariate Linear Regression (MLR): An *MLR synopsis* with input attributes $Y_1, \ldots, Y_N$ and output attribute Z estimates the value of Z as a linear combination of the Y_j values [88]. Mathematically:

$$Z = c + \sum_{j=1}^{N} \alpha_j Y_j \tag{2.1}$$

The *MLR-learner* uses a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$ to compute the *regression coefficients* α_j and the constant c in Equation 2.1. Note that Equation 2.1 is actually a system of linear equations, one equation for each tuple in D. The MLR-learner computes the least-squares solution of this system of equations, namely, the values of α_js and c that minimize the sum of $(Z(\tau) - \hat{Z}(\tau))^2$ over all the tuples in D [93]. Here, $Z(\tau)$ and $\hat{Z}(\tau)$ are respectively the actual and estimated values of Z in the tuple with timestamp τ in D. Once all α_js and c have been computed, the *MLR-predictor* uses Equation 2.1 to estimate Z in a tuple given the values of attributes $Y_1, \ldots, Y_N$.

Bayesian Networks (BN): A *BN synopsis* is a summary structure that can represent the joint probability distribution $Prob(Y_1, \ldots, Y_N, Z)$ of a set of random variables $Y_1, \ldots, Y_N, Z$. A BN for variables $Y_1, \ldots, Y_N, Z$ is a directed acyclic

graph (DAG) with $N + 1$ vertices corresponding to the $N + 1$ variables. Vertex X in the BN is associated with a *conditional probability table* that captures $Prob(X|Parents(X))$, namely, the conditional probability distribution of X given the values of X's parents in the DAG. The DAG structure and conditional probability tables in the BN satisfy the following equation for all $(Y_1 = y_1, \ldots, Y_N = y_N, Z = z)$ [88]:

$$Prob(y_1, \ldots, y_N, z) = \prod_{i=1}^{N} Prob(Y_i = y_i|Parents(Y_i)) \times Prob(Z = z|Parents(Z))$$

Given a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$, the *BN-learner* finds the DAG structure and conditional probability tables that approximate the above equation most closely for the tuples in D. Since this problem is NP-hard, the BN-learner uses heuristic search over the space of DAG structures for Y_1, ..., Y_N, Z [88].

The *BN-predictor* uses the synopsis generated by the BN-learner from $D(\Gamma, Y_1, \ldots, Y_N, Z)$ to estimate the unknown value of Z in a tuple u from the known values of $u.Y_1$, ..., $u.Y_N$. The BN-predictor first uses the synopsis to infer the distribution $Prob(u.Z = z|u.Y_j = y_j, 1 \leq j \leq N)$. The exact value of $u.Z$ is then estimated from this distribution, e.g., by picking the expected value.

2.4.2 Finding a Good Plan Automatically and Efficiently

Space of Execution Plans: An execution plan for a forecasting query *Forecast*(D, X_i, L) first applies a sequence of transformers to D, then uses a synopsis learner to generate a synopsis from the transformed dataset, and finally uses a

predictor to make a forecast based on the synopsis and the transformed dataset. For each logical operator, there is a wealth of physical implementations from the machine-learning literature [88].

The best plan for a forecasting query Q is the one with the highest prediction accuracy in the space of candidate execution plans. The best plan involves the right combination of a sequence of transformers, a synopsis type, and the synopsis learner and predictor. Therefore, the search space for the best plan is large and complex. Unlike query optimizers in relational database systems, there is no reliable *cost model* to estimate the accuracy of a plan for a forecasting query without running the plan. Since running a plan takes time, it is nontrivial to optimize the time Fa takes to produce a prediction result after a forecasting query is submitted. There is a tradeoff between the processing time required to generate a prediction result and the accuracy of the prediction. Fa's query optimizer needs to take this tradeoff into account during automatic plan generation.

2.5 Summary

This chapter introduced multi-tier Web service systems as our focus systems and described the format of monitoring data we consider in this dissertation. As part of system administration, administrators face the need to answer various management queries over the system monitoring data. We identified three key query types (namely, diagnosis, tuning, and forecasting queries) that are needed routinely in system performance management. Next, we described the Fa platform which contains a simple query interface for expressing the three management queries, a query processor that automatically finds a good execution plan and executes the plan (with the support of appropriate representations of the system data) to generate

the query result. Finally, we illustrated the optimization challenges, using an example forecasting query, that Fa faces while processing system-management queries automatically, accurately, and efficiently.

Chapter 3

Automated Processing of Diagnosis Queries

3.1 Motivation

A recent study [71] found that 72% of the top-40 Web sites suffer user-visible problems, such as slow responses, blank pages or error messages being displayed, items not being added to shopping carts, unexpected database slowdowns, and others. Walmart.com was unavailable for almost 10 hours during the peak U.S. 2006 holiday season. Such deviation of systems from desired behavior may violate *service-level objectives (SLOs)* that specify what an acceptable level of service is. For example, an SLO for an online brokerage may stipulate that all transactions complete within 1 second, regardless of how much middleware, databases, or networks are involved.

SLO violations in a system indicate *failures*. When a system meets all specified SLOs, it is in a *healthy state*; otherwise, it is in a *failure state*. Failures may be caused by a variety of factors including performance problems like resource contention, crashes due to hardware or software faults, and misconfiguration by system administrators. The increasing scale, complexity, and dynamics of modern systems make it laborious and time-consuming to track down the cause of failures manually [25, 48].

At the same time, it is important to diagnose failures and recover systems quickly. Brokerages and banking firms can lose up to $75,000 per minute of downtime [48]. A 22-hour outage at eBay cost the company more than $3 Million in customer credits and $4 Billion in market capitalization. These factors motivate automated processing of diagnosis queries in an efficient and reasonably-accurate way to diagnose failures using system monitoring data; a way that is also easy and intuitive for system administrators to use. However, automated processing of diagnosis queries based on system monitoring data poses nontrivial challenges:

- **Noisy data:** Monitoring data collected from production systems contains various types of errors that can mislead diagnosis: (i) natural system variability injects Gaussian noise; (ii) failures may corrupt observations; and (iii) rapid system state transitions cause observations from different states to get mixed up.

- **High dimensionality:** Some of our monitoring datasets have 100-300 attributes per server, posing challenges from an accuracy as well as running-time perspective.

- **Dynamic systems:** Conventional approaches like defining a baseline system behavior, and pinpointing deviations from the baseline do not work when workloads and system configuration change over time.

- **Reuse:** Since failure diagnosis is expensive in large-scale and complex systems, it is valuable to leverage past diagnosis efforts whenever possible; particularly since 50-90% of failures seen are recurrences of previous failures [14].

- **Trust:** Features like reliable confidence estimates and evidence for diagnosis

	time	lock_time	num_io	failures
h1	1	51.3	76.1	0
h2	2	48.5	63.6	0
h3	4	51.9	97.9	0
h4	5	49.0	43.6	0
h5	8	50.4	13.5	0
h6	9	50.8	51.2	0
h7	11	49.8	119.5	0
h8	12	49.3	141.2	0
h9	13	72.4	65.4	0

(a) Healthy data H

time	lock_time	num_io	failures	annotation
3	95.4	43.5	1	Lock prob
10	89.6	123.2	1	Buffer prob

(b) Annotated failure data L

time	lock_time	num_io	failures
6	75.6	83.5	1
7	72.4	83.8	1

(c) Unannotated failure data U

time	lock_time	num_io	failures
14	70.0	80.7	1
15	71.9	85.6	1

(d) Failure data F

Figure 3.1: Sample monitoring data used in diagnosis query processing

results are important for an automated diagnosis tool to gain administrators' trust; otherwise the tool will not be used in practice.

This chapter describes how Fa addresses the above challenges while performing automated processing of diagnosis queries. We begin with an overview of the process.

3.2 Abstraction of Diagnosis Queries

System Monitoring Data: When a system is running, Fa collects monitoring data periodically and stores it in a database. In this chapter, we consider monitoring data with a relational schema as shown in Figure 3.1. For example, Fa uses the *sar* [69] utility to collect more than 100 performance metrics (e.g., average CPU utilization, number of disk I/Os) periodically from Linux servers. Database servers maintain performance counters (e.g., number of index updates, number of full table scans) that Fa reads periodically. Most enterprise monitoring systems like HP OpenView [59] and IBM Tivoli Monitoring [79] collect similar data.

Over a period of time, the monitoring data collected by Fa will contain three types of instances (as illustrated in Figure 3.1):

- **Healthy data** H, which is monitoring data collected when the system was in a healthy state. Recall that a system is in a healthy state when it experiences no SLO violations; and in a failure state otherwise.

- **Unannotated failure data** U, which is monitoring data collected from failure states of the system where the cause of failure has not been diagnosed so far.

- **Annotated failure data** L, which is monitoring data collected from failure states of the system where the cause of failure has been diagnosed. A successful diagnosis can happen any time after the failure occurs. Upon diagnosis, information about the type and cause of failure is attached as an *annotation* (or metadata) to the corresponding monitoring data. Specifically, the addition of an annotation to an instance t in the unannotated data U, moves t from U to L.

Example 3.2.1. *Figure 3.1(a) displays the historical data for a database server collected by monitoring the server at one-minute intervals. In each interval, attribute* `lock_time` *is the average wait time to acquire locks;* `num_io` *is the number of disk I/Os;* `failures` *denotes whether the average response time of database transactions in that interval exceeded a threshold (causing SLO violations) or not; and* `annotation` *records the cause of each diagnosed failure. In this historical data, healthy data H consists of instances $h1$–$h9$, annotated data L consists of failure instances $l1$ and $l2$, and unannotated data U consists of failure instances $u1$ and $u2$.*

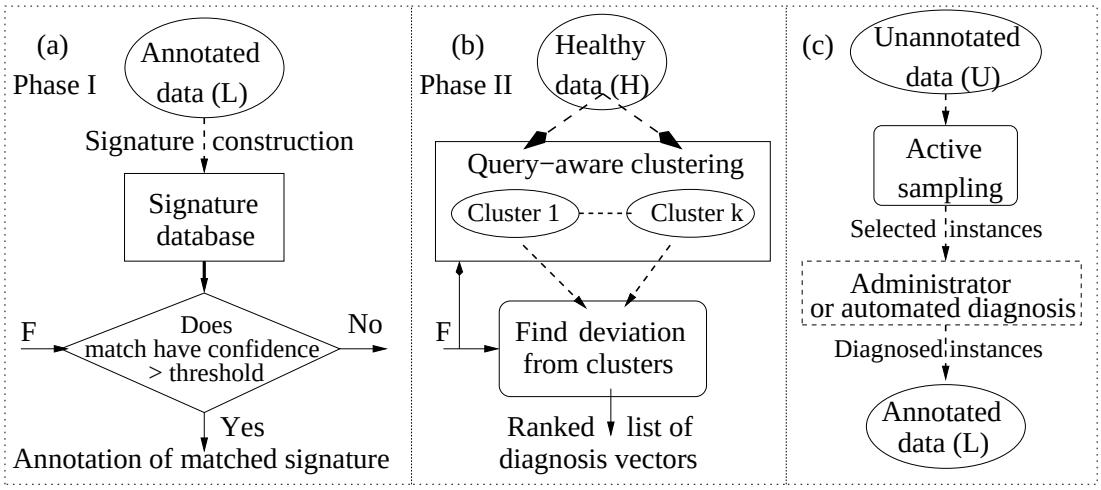

Figure 3.2: Control and data flow in Fa during diagnosis query processing

Diagnosis Queries: When the monitored system experiences a failure, an administrator or system-management software can diagnose the cause of the failure by posing a *diagnosis query* to Fa of the form Q=*Diagnose*$(F, H \cup U \cup L)$.

- F is monitoring data from the system during the failure (or just before the failure in the case of a system crash).
- $H \cup U \cup L$ is the historical data collected so far.

Example 3.2.2. *Figure 3.1(b) shows recent monitoring data F from the same server as in Example 3.2.1. The values of the* `failures` *attribute in F indicate that the server is experiencing some type of failure.*

Fa processes a diagnosis query Q=*Diagnose*$(F, H \cup U \cup L)$ in two phases, as illustrated in Figure 3.2.

Phase I (Figure 3.2(a), Section 3.3): First, Fa finds whether the failure represented by F is the same as a previously-diagnosed failure in L. That is, Phase I translates Q to Q_I=*Diagnose*(F, L). If the diagnosis result produced by this phase

has confidence higher than a specified threshold, then Fa returns this result to the issuer of the query; otherwise Fa goes to a more expensive Phase II of diagnosis.

Phase II (Figure 3.2(b), Section 3.4): Here, Fa compares F with the healthy data H collected so far, to see whether the cause of the failure can be characterized succinctly as attributes whose values in F deviate from their values in the data representing different healthy states of the system. That is, Phase II translates Q to Q_{II}=$Diagnose(F, H)$.

Background Phase (Figure 3.2(c)) : Fa supports techniques to guide manual diagnosis efforts by actively selecting informative unannotated instances from U for diagnosis. These techniques run constantly in a background phase, transferring data from U to L to improve the signature database's accuracy and coverage. This background phase will be described in Chapter 4.

3.3 Phase I: Generating and Using a Signature Database

In this section, we present a systematic way to utilize the diagnosis information exposed by the annotated failure instances L. Table 3.1 summarizes the notation we use in this chapter.

We are given L, the historical data with annotations about m distinct failures, denoted A_1, A_2, ..., A_m. Our goal is to generate a *signature database* from L that contains entries of the form $\langle sig, A_i \rangle$ where sig is a signature for failure A_i. Each failure can be represented by any number of signatures, including zero. As illustrated in Figure 3.2(a), instances F from an undiagnosed failure can be matched

Notation	Description
H	Healthy monitoring data
L	historical failure data with annotations
U	historical failure data without annotations
F	Data from a failure to be diagnosed
$\mathbf{x}_i$	System metrics
$\mathbf{A}$	Annotation
s_i	Separating functions
S	Signature database
$S(i,:)$	ith signature in S
$S(:,j)$	jth column in S
C_i $(1 \leq i \leq l)$	Clusters of H
$\overrightarrow{w_1}, \overrightarrow{w_2}, \ldots, \overrightarrow{w_l}$	Weight vectors of diagnosis results

Table 3.1: Notation table

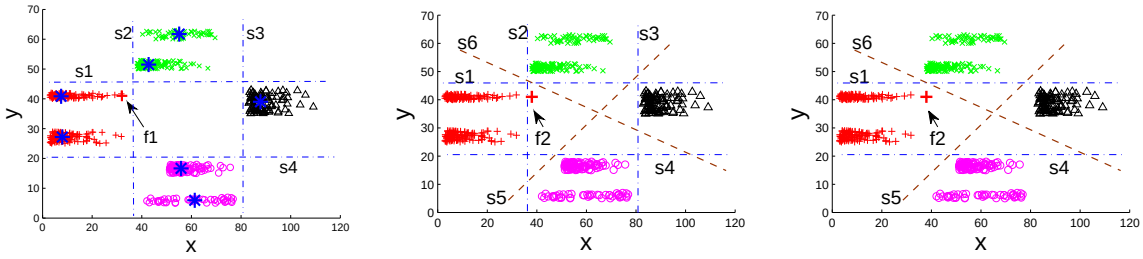

Figure 3.3: (a) Clustering Vs. separating functions; (b), (c) improving robustness of signature databases

against the database to find the signature nearest to F. The annotation of this signature will be returned along with a confidence estimate in the match if the confidence exceeds a threshold.

We will first illustrate our ideas using a series of examples. Suppose L is as shown in Figure 3.3(a). Each instance has two attributes, x (denoting `lock_time`) and y (denoting `num_io`), plotted along the horizontal and vertical axes respectively; and one of four distinct annotations A_1 (cross), A_2 (plus), A_3 (triangle), or A_4 (circle).

Clustering: One way to generate the signature database is by clustering the data in L using a technique like K-means [88]. Figure 3.3(a) shows the clusters per

SD_1

lock_time	num_io	annotation
7.9	27.2	A_1
7.5	40.9	A_1
42.8	51.5	A_2
55.1	61.7	A_2
87.8	39.0	A_3
61.3	5.9	A_4
65.8	16.7	A_4

SD_2

s_1	s_2	s_3	s_4	annotation
1	0	0	0	A_1
0	1	0	0	A_2
0	0	1	0	A_3
0	0	0	1	A_4

s_1 = 1 if y > 45, otherwise 0
s_2 = 1 if x < 38, otherwise 0
s_3 = 1 if x > 80, otherwise 0
s_4 = 1 if y < 20, otherwise 0

Figure 3.4: Signature databases: (a) SD_1, (b) SD_2

failure type. The centroids of these clusters—represented by blue stars ("$\star$") in Figure 3.3(a)—become the signatures for the corresponding failure type, giving a signature database SD_1 as shown in Figure 3.4 (a).

Suppose the query instance $f_1 = \langle 32, 41 \rangle$ in Figure 3.3(a) is a failure instance that we want to diagnose. f_1 can be matched with SD_1 to find the centroid (signature) nearest to f_1; which is a centroid for Failure A_1 (cross) given the data distribution. However, this diagnosis is incorrect since it is obvious from Figure 3.3(a) that f_1 is an instance of Failure A_2 (plus). This example illustrates that although clustering-based signatures are conceptually simple, they have drawbacks. (They work poorly on real data in our experiments.)

Separating functions: Instead of clustering, suppose we identify *separating functions* $s_1(x, y)$, $s_2(x, y)$, $s_3(x, y)$, and $s_4(x, y)$ that separate each type of failure instances from the others. These functions can take many different forms. To convey our ideas while keeping the example simple, we will use a simple form, namely, separating lines in the 2D plane. Figure 3.3(a) shows s_1–s_4 as dotted lines. For example, $s_1(x, y)$ separates the instances of Failure A_1 from the others, and has the form: $s_1(x, y)$=1 if $y > 45$, otherwise 0.

Matrix: Figure 3.4(b) shows a signature database SD_2 generated using s_1–s_4. SD_2 is a matrix with each row representing the signature of some failure. For example, the signature of Failure A_1 is $\langle s_1(x,y) = 1, s_2(x,y) = 0, s_3(x,y) = 0, s_4(x,y) = 0\rangle$, denoted $\langle 1,0,0,0\rangle$. Each column represents a separating function. For example, the first column represents $s_1(x,y)$ which maps instances of Failure A_1 to 1, and instances of all other failures to 0.

To match a query instance $f = \langle x,y\rangle$ with SD_2, we compute $\vec{s}(x,y) = \langle s_1(x,y),\ s_2(x,y),\ s_3(x,y),\ s_4(x,y)\rangle$ and find the signature nearest to $\vec{s}(x,y)$ in SD_2. For example, $\vec{s}(32,41)$ is $\langle 0,1,0,0\rangle$ for the query instance $f_1 = \langle 32,41\rangle$. (Note that $\langle 0,1,0,0\rangle$ matches Failure A_2's signature perfectly.) For now, we will measure distances in terms of the *Hamming distance*, namely, the number of bits that are different. Thus, the distances of $\vec{s}(32,41)$ to the four signatures in Figure 3.4(b) are respectively 2, 0, 2, 2. Since $\vec{s}(32,41)$ is nearest to A_2's signature, f_1 is diagnosed correctly.

Handling errors: Now consider the query instance $f_2 = \langle 39,41\rangle$ shown in Figure 3.3(b). f_2 is of failure type A_2, but has higher error in the x dimension than the instances in L. If we match f_2 against SD_2, $\vec{s}(39,41)$ is $\langle 0,0,0,0\rangle$. Since $\vec{s}(39,41)$ is equidistant from all the signatures in SD_2, f_2 will not be diagnosed correctly by SD_2.

Now suppose we use the signature database SD_3 from Figure 3.5(a) to diagnose f_2. SD_3 contains two new separating functions, $s_5(x,y)$ and $s_6(x,y)$. s_5 separates instances of Failures A_1 and A_2 from those of A_3 and A_4, and s_6 separates instances of A_1 and A_3 from those of A_2 and A_4; as represented by the columns for s_5 and s_6 in Figure 3.5(a). The separating planes for s_5 and s_6 are shown in Figure 3.3(b). Now, $\vec{s}(x,y) = \langle s_1(x,y),\ s_2(x,y),\ s_3(x,y),\ s_4(x,y), s_5(x,y),\ s_6(x,y)\rangle$. For f_2, $\vec{s}(39,41) =$

SD3

S_1	S_2	S_3	S_4	S_5	S_6	annotation
1	0	0	0	1	1	A_1
0	1	0	0	1	0	A_2
0	0	1	0	1	1	A_3
0	0	0	1	0	0	A_4

S_1 = 1 if y > 45, otherwise 0
S_2 = 1 if x < 38, otherwise 0
S_3 = 1 if x > 80, otherwise 0
S_4 = 1 if y < 20, otherwise 0
S_5 = 1 if 0.81*x-y>-16, otherwise 0
S_6 = 1 if 0.36*x+y>61, otherwise 0

SD4

S_1 $\beta 1=1$	S_2 $\beta 2=0$	S_3 $\beta 3=0$	S_4 $\beta 4=1$	S_5 $\beta 5=1$	S_6 $\beta 6=1$	annotation
1	0	0	0	1	1	A_1
0	1	0	0	1	0	A_2
0	0	1	0	0	1	A_3
0	0	0	1	0	0	A_4

S_1 = 1 if y > 45, otherwise 0
S_2 = 1 if x < 38, otherwise 0
S_3 = 1 if x > 80, otherwise 0
S_4 = 1 if y < 20, otherwise 0
S_5 = 1 if 0.81*x-y>-16, otherwise 0
S_6 = 1 if 0.36*x+y>61, otherwise 0

Figure 3.5: Signature databases: (a) SD_3, (b) SD_4

$\langle 0,0,0,0,1,0 \rangle$. $\vec{s}(39,41)$ has least Hamming distance to the signature for Failure A_2 in Figure 3.5(a), so f_2 will now be diagnosed correctly.

Why did SD_3 diagnose f_2 correctly, while SD_2 did not? The reason can be understood from an analogy to *error correction* in telecommunications. Error correction is the ability to reconstruct the original, error-free data at the destination in the presence of errors caused by noise or other impairments during transmission from source to destination. The central idea in error correction is as follows: a bit string b to be transmitted is interleaved with some carefully-chosen extra bits, to transmit a new bit string b' such that a fixed number or less of bit-flip errors during transmission will not convert b' to a new bit string a' that corresponds to the transmitted version of another bit string a, $a \neq b$. (If this case arises, then the destination cannot tell whether a was transmitted or b.)

For f_2, s_2 predicts 0 instead of the correct 1; causing a 1-bit error. SD_3 is robust to 1-bit errors, but SD_2 is not. The Hamming distance between any two signatures

in SD_3 is ≥ 3. Thus, even though $\vec{s}(39, 41)$ was computed as $\langle 0, 0, 0, 0, 1, 0\rangle$—a 1-bit error from the ideal $\langle 0, 1, 0, 0, 1, 0\rangle$—$\vec{s}(39, 41)$ still remained nearest to A_2's signature. However, the Hamming distance between any two signatures in SD_2 is ≥ 2. Thus, a 1-bit error in $\vec{s}(39, 41)$ leaves it in an ambiguous position for SD_2; causing incorrect diagnosis.

The previous example shows that selected redundancy in the set of separating functions can overcome incorrect predictions by some of the functions. Learning more functions increases the cost of generating the signature database. However, that is not a concern since the bulk of this work is done offline, and can be made very efficient with parallel learning of functions. The more pressing issue is that some functions are less reliable than others, and their presence can hurt diagnosis accuracy and confidence significantly.

Functions s_2 and s_3 are two less reliable functions in our example. Note that the data for each type of failure in Figure 3.3(a) shows larger spread along the x axis than the y axis. Intuitively, there are larger chances of error in the x values. Since s_2 and s_3 separate exclusively along the x axis, they are likely to get their predictions wrong when errors in x arise (like what happened for f_2). We can drop s_2 and s_3, and generate a new signature database SD_4 that has the four functions s_1, s_4, s_5, and s_6 only (Figure 3.3(b)). SD_4 is shown in Figure 3.5 (b). Note that SD_4 will give a perfect match for f_2.

Takeaway points: Our series of examples show that the following is a powerful representation of the signature database to achieve both good accuracy and robustness to errors:

- *Binary matrix M:* The ith row in M, denoted $M(i,:)$, is the signature of failure $A_i, 1 \leq i \leq m$. The jth column in M, denoted $M(:,j)$, corresponds to the separating function $s_j(\vec{x})$. The number of columns d depends both on m and the built-in error tolerance desired.

- *Separating functions $s_1(\vec{x})$–$s_d(\vec{x})$:* Each function separates one or more types of failure instances from the others. These functions can take many different forms. Fa uses *Classification and Regression Trees (CART)* [88] that are learned automatically from L.

- *Weights β_1–β_d for the respective functions:* For robustness to errors, the prediction $s_j(\vec{x})$ from a less reliable separating function s_j is given a smaller weight while computing the distance of $\vec{s}(\vec{x})$ to the signatures. For example, reasonable weights for s_1–s_6 in SD_4 are $\{1, 0, 0, 1, 1, 1\}$ because s_2 and s_3 are less reliable.

Next, we describe the offline generation of the signature database, its use for diagnosis, and online maintenance.

3.3.1 Generating the Binary Matrix

There are four rules to generate a valid matrix M:

1. Each row should be distinct since no two failures can have the same signature.

2. Columns that contain all 0s or 1s should be excluded, since they do provide no differentiation among failures.

3. Two columns cannot be the same or complementary since they derive the

same separating function. (A 0-1 exchange in a column generates its complementary.)

4. The *radius* r of M, defined as half the minimum Hamming distance over all $\langle M(i,:), M(j,:)\rangle$ pairs, $i \neq j$, should be above a given threshold. Intuitively, the higher the radius, the higher the error-correction ability of M. For a query instance $\vec{x}$, $\vec{s}(\vec{x})$ can be matched with the correct signature even when up to $r-1$ separating functions produce wrong predictions for $\vec{x}$.

We use a random search algorithm to generate M given a threshold R_t on M's radius. The algorithm is as follows:

1. Generate m random binary vectors of length $d = R_t(2+\delta)$. δ is a positive integer (we set δ =1). The expected Hamming distance between each pair of vectors is $d/2$.

2. Remove columns containing all 0s or 1s (Rule II).

3. For any identical or complementary column pair, retain one column only (Rule III).

4. If the matrix generated by Steps 1-3 has a radius smaller than the threshold R_t, then go to Step 1.

This simple algorithm is surprisingly effective. The radius threshold R_t is a design choice best left to the administrator. R_t balances diagnosis accuracy and robustness against the time to generate the signature database—higher R_t means more columns (functions), and hence longer time to generate the database. Based on our empirical observations, $R_t = 5log_2(m)$ is a balanced choice, and is Fa's default.

3.3.2 Generating the Separating Functions

For each column $M(:, j)$ of the matrix, Fa learns the separating function $s_j(\vec{x})$ as a binary classification tree (CART) [88] separating the instances with annotation $M(i, j) = 1$ from the instances with annotation $M(i, j) = 0$. In separate work [34] with many types of functions from statistical machine-learning, we found CARTs to best balance prediction accuracy and learning time.

3.3.3 Weighting the Separating Functions

Suppose the jth separating function $s_j(\vec{x})$ has weight β_j, and the overall weight vector is $\vec{\beta} = \langle \beta_1, \beta_2, \ldots, \beta_d \rangle$. A query instance $\vec{x}$ whose true annotation is A_i will be matched with A_i's signature $M(i, :)$ if the following condition holds:

$$\min_{k \neq i} \{\texttt{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \texttt{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta})\} > 0 \tag{3.1}$$

Here, `Dist` is the *weighted* Euclidean distance: $\texttt{Dist}(\vec{u}, \vec{v}; \vec{\beta}) = \sqrt{\sum_{j=1}^{d} \beta_j (u_j - v_j)^2}$. For binary vectors $\vec{u}$ and $\vec{v}$, the Euclidean distance is the square root of the Hamming distance.

The larger the difference in Equation 3.1, the higher the chances of matching $\vec{x}$ to the correct signature when errors cause variations in $\vec{s}(\vec{x})$. Thus, to make the signature database robust, we want to choose the weight vector $\vec{\beta} = \langle \beta_1, \ldots, \beta_d \rangle$ that maximizes this difference over all instances $\langle \vec{x}, A_i \rangle$ in the annotated failure data L. Under the condition that $||\vec{\beta}||^2 (= \sum_{j=1}^{d} \beta_j^2)$ is fixed, this optimization is:

$$\max_{\vec{\beta}} \min_{\langle \vec{x}, A_i \rangle \in L, k \neq i} \{\texttt{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \texttt{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta})\}$$

If there exists some weight vector $\vec{\beta}$ that satisfies Equation 3.1 for all instances

$\langle \vec{x}, A_i \rangle$ in L, then the above optimization problem is equivalent to the following one:

$$\min_{\vec{\beta}} \frac{1}{2}||\vec{\beta}||^2$$

such that, $\forall \langle \vec{x}, A_i \rangle \in L$ and $\forall k \neq i$:

$$\texttt{Dist}(\vec{s}(\vec{x}), M(k,:); \vec{\beta}) - \texttt{Dist}(\vec{s}(\vec{x}), M(i,:); \vec{\beta}) \geq 1 \tag{3.2}$$

This equivalence is shown in [82] which reports recent results on learning *Support Vector Machines (SVMs)* for complex problems like sequence alignment and grammar learning. (Reference [82] does not consider signatures or failure diagnosis.) The above optimization problem is a general version of the maximum-margin principle used in SVM learning, and can also be solved using SVM learning to generate $\vec{\beta}$ [82].

However, there are two reasons why we do not want to use the weights learned from Equation 3.2:

- There may not be a feasible $\vec{\beta}$ that satisfies Equation 3.1 for all instances $\langle \vec{x}, A_i \rangle$ in L.
- We want to avoid the classic problem of *overfitting* [88], where $\vec{\beta}$ is too well tuned for L that it performs poorly for undiagnosed failure instances not in L.

Both these issues can be addressed by introducing a *slack variable* $\varepsilon_i \geq 0$ per $\langle \vec{x}, A_i \rangle \in L$ to relax the corresponding constraints in Equation 3.2; for a new optimization problem:

$$\min_{\vec{\beta},\varepsilon} \frac{1}{2}||\beta||^2 + c\sum_{i=1}^{|L|}\frac{\varepsilon_i}{|L|}$$

such that, $\forall\langle\vec{x}, A_i\rangle \in L$ and $\forall k \neq i$:

$$\texttt{Dist}(\vec{s}(\vec{x}), M(k,:); \vec{\beta}) - \texttt{Dist}(\vec{s}(\vec{x}), M(i,:); \vec{\beta}) \geq 1 - \varepsilon_i \tag{3.3}$$

Here, $|L|$ is the number of instances in L. The constant $c > 0$ controls the tradeoff between matching training instances in L correctly and the robustness to errors. Fa uses SVM learning algorithms from [82] to learn the weights $\vec{\beta} = \langle\beta_1, \ldots, \beta_d\rangle$ by solving this optimization problem. A good value of c is determined via 5-fold cross validation over L [88].

3.3.4 Online Use and Maintenance

So far we discussed how the signature database is generated offline from L. We now discuss the online use of the database for diagnosis, and its maintenance as new instances and annotations are added to L. When the database is queried with an undiagnosed failure instance $\vec{x}$, Fa first computes $\vec{s}(\vec{x}) = \langle s_1(\vec{x}), \ldots, s_d(\vec{x})\rangle$, and then finds the signature nearest to $\vec{s}(\vec{x})$, namely, the signature that minimizes $\texttt{Dist}(\vec{s}(\vec{x}), M(i,:); \vec{\beta}), 1 \leq i \leq m$. As shown in Figure 3.2(a), a confidence estimate *conf* is generated for this match and compared with the confidence threshold C_t. If *conf* $\geq C_t$, then the annotation of the matched signature is returned as the diagnosis result; otherwise Phase II is invoked.

Confidence estimate: When $\vec{x}$ is matched with the signature $M(i,:)$, the confidence in this match is defined as:

$$conf = \min_{k \neq i} \{\texttt{Dist}(\vec{s}(\vec{x}), M(k,:); \vec{\beta}) - \texttt{Dist}(\vec{s}(\vec{x}), M(i,:); \vec{\beta})\} \tag{3.4}$$

Intuitively, *conf* is high when the second-nearest neighbor N_2 of $\vec{s}(\vec{x})$ is far from the first-nearest neighbor N_1, indicating an unambiguous match to N_1. As the gap between N_1 and N_2 shrinks, the ambiguity in the match to N_1 increases, and the confidence decreases.

To make the confidence estimate easier for administrators to understand, Fa converts it into a value in $[0, 100]$. This conversion is done using an equi-depth histogram (quantiles) with 100 buckets generated from the distribution of confidence estimates over all instances of L. Let p_k, $0 \leq k < 100$, denote the quantiles of this distribution. For a confidence estimate *conf* from Equation 3.4, Fa finds i such that $p_i \leq conf < p_{i+1}$; and reports i as the confidence estimate.

Setting the confidence threshold C_t: The value of C_t is critical because a low C_t can lead to incorrect diagnosis, while a high C_t can invoke the more expensive Phase II more often than needed. (Compared to Phase I, Phase II involves higher run-time overhead and more efforts from administrators to interpret diagnosis results.) Fa's approach is to let the administrator specify the minimum diagnosis accuracy she wants from the signature database. Then, Fa automatically derives the appropriate C_t that gives this diagnosis accuracy while minimizing the chances of invoking Phase II. The algorithm works as follows:

1. Divide L into a *training set* and a *test set*. Generate a signature database SD from the training set (Sections 3.3.1–3.3.3). For each instance in the test set, use SD to find the matched signature and confidence estimate.

2. Pick an integer value $x \in [0, 100]$. For test instances whose confidence estimate

is $\geq x$, compute the percentage of instances matched to the correct signature in SD. This percentage is the expected diagnosis accuracy when the confidence threshold is x, denoted $acc(x)$. Vary x in $[0, 100]$ to get enough $\langle x, acc(x)\rangle$ points to plot the *accuracy-confidence curve (AC-Curve)* for SD as shown in Figure 3.6. Note that as x increases, the accuracy is being computed on higher-confidence answers from SD; so we expect the AC-Curve to be nondecreasing.

3. Pick the minimum x_t in the AC-Curve such that $\forall x, x \geq x_t$, $acc(x)$ is above the diagnosis accuracy desired by the administrator. x_t is a sample of the desired value of C_t.

4. Repeat Steps 1, 2, and 3 with different training and test sets from L to get multiple independent samples of C_t; and set C_t to their mean.

Figure 3.6 is a sample graph from our experiments which plots the AC-Curves for both Fa's signature database (FA) and the clustering-based signature database (CLUS). If the administrator desires a diagnosis accuracy of 95%, the Fa's $C_t = 20$ while CLUS's $C_t = 80$. That is, Fa is four times less likely to trigger Phase II than CLUS.

Incremental Maintenance: When new instances are added to L, we need to update two components of the signature database: its separating functions and its weight vector. Recall that CARTs are used as the separating functions and the weight vector is determined via an SVM learning algorithm. Both the separating functions and the weight vector can be updated efficiently with new instances using an incremental CART learning algorithm [88] and an incremental SVM learning algorithm [19] respectively.

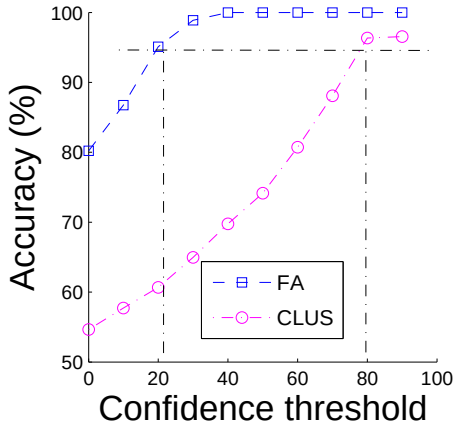

Figure 3.6: Sample AC-Curve

When new annotations are added to L, we also need to update the Matrix. One row (signature) is generated for each added annotation. The existing separating functions can be updated incrementally. If the radius of the new Matrix goes below the threshold R_t specified in Section 3.3.1, then more columns will need to be added to the Matrix to bring its radius above the threshold. New separating functions will have to be learned from scratch for the newly added columns. As we do not expect frequent additions of new annotations, the time amortized for learning new CARTs over a long interval should be small. Again the amortized time for computing the weight vector from scratch via SVM learning is also small. Note that the weight vector can be updated incrementally if the Matrix does not grow in columns.

3.3.5 Error-Aware Signature Databases

This section considers how Fa can make the signature database more accurate and robust if it has models that represent errors expected in the monitoring data. These models could be derived from historical data.

Types of Error: We have seen two types of errors in our monitoring data: *Gaussian* and *non-Gaussian*.

- Gaussian error is caused by natural variability in real systems. If we take multiple observations of an attribute from a particular system state, a goodness-of-fit test to a normal distribution will often be positive.

- Two significant causes of error in our monitoring data cannot be modeled by Gaussian distributions: (i) the onset or presence of failure corrupts readings of some attributes (seen with JBoss application server and MySQL); (ii) observations from different states get mixed up in the same instance due to rapid system state transitions, or due to delays in measuring different attributes under system overload. (A third probable cause is incorrect filling of missing values by monitoring tools.)

Error Models: Both the above types of error can be captured using error models. Plenty of literature exists on error models that vary from simple to complex (e.g., [96]). For example, an attribute x_i with Gaussian error can be represented by a new attribute x_i' of the form $x_i' = x_i + Gaussian(0, \delta_i)$, where δ_i controls the scale of error. The value of δ_i can be learned from historical data. Error modeling is complementary to our robust signature construction techniques.

Error-Aware Matrix: The Matrix's role is to provide redundancy at the level of separating functions. Intuitively, if more error is expected, then we should have a Matrix with a larger radius (recall Section 3.3.1). The radius of a given Matrix can be increased by adding columns.

Error-Aware Separating Functions: Algorithms for learning separating functions can be modified to utilize error information. Fa's CARTs pick attributes to use in decision nodes in the tree based on a metric called *information gain* (*InfoGain*) [88]:

$$\texttt{InfoGain}(A, x_i) = Entropy(A) - Entropy(A|x_i)$$

This classic formula represents the extra information we gain about the annotation A of an instance given the value of attribute x_i in that instance. As expected, attributes with larger InfoGain are preferred while picking attributes to use in decision nodes. However, an attribute x_i with higher chances of error—like attribute x in our running example in Section 3.3—should be preferred less because decisions made based on x_i's value are less reliable. We can achieve this property by rewriting $InfoGain(A, x_i)$ as:

$$\texttt{InfoGain}(A, x_i) = Entropy(A) - Entropy(A|x_i')$$

Here, x_i' is the distribution of x_i once the expected error is added based on the known error model.

Error-Aware Weights: Finally, learning weights for separating functions can be made error-aware by appropriately choosing the set of $\langle \vec{x}, A \rangle$ instances used to learn the weights in Section 3.3.3. Along with the original instances in L, we can use new instances generated by injecting expected error (based on the error models) into the original instances in L.

3.4 Phase II: Anomaly-based Clustering

If the instances F to diagnose in a *Diagnose(F)* query correspond to a failure type that was not seen previously, then the match from the signature database will have low confidence. Phase II of diagnosis runs in this setting. The basic approach in Phase II is to determine how F differs from the data H representing the system in healthy states. We will first illustrate the main ideas using a series of examples.

Suppose H consists of the instances in Figure 3.7(a) shown using the "x" symbol. Each instance has two attributes, x and y, plotted along the horizontal and vertical axes respectively. The figure also shows the failure instances F, indicated using the "+" symbol, in a *Diagnose(F)* query. It is clear from the figure that there are two distinct healthy states of the system: (i) C_1 with $x \in [10, 30]$ and $y \in [65, 80]$, differing from F primarily along the x attribute; and (ii) C_2 with $x \in [60, 80]$ and $y \in [15, 30]$, differing from F primarily along the y attribute. A clustering algorithm like K-means or *locally adaptive clustering (LAC)* [32] can identify these clusters in H, and link both attributes x and y to the failure. The LAC algorithm associates each cluster C with a weight vector that reflects the correlation among instances in C. Attributes on which the instances in C are strongly (weakly) correlated receive a large (small) weight, which has the effect of constricting (elongating) distances along those dimensions.

Next, suppose H ("x") and F ("+") are as shown in Figure 3.7(b). A conventional clustering algorithm will now group the instances in H into three distinct clusters (C_1, C_2, and C_3 in Figure 3.7(b)). Since each of these clusters differs from F along both the x and y attributes, both attributes will be linked to this failure as well. However, a closer look at Figure 3.7(b) indicates that this answer is incorrect. Both the failure data and the healthy data have similar distribution along the y

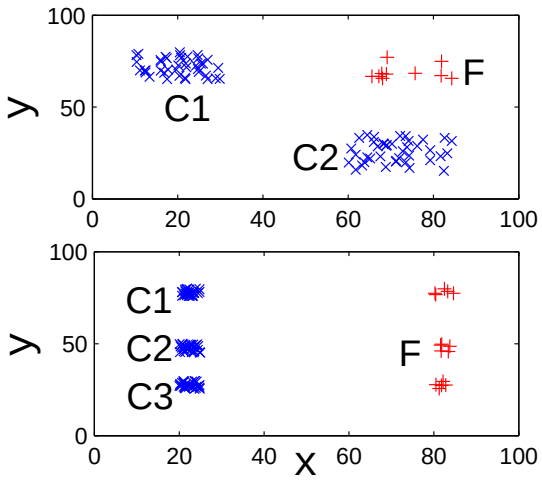

Figure 3.7: Sample data

axis, and differ along the x axis only. So, the correct answer should link the failure to x only.

What went wrong in the second example? Conventional clustering algorithms ignore the failure (query) instances F while deciding how to group the instances in H into clusters. Thus, the clusters generated by these algorithms are independent of the failure instances to be diagnosed, causing two major weaknesses: (i) generating clusters that do not give the correct diagnosis, and (ii) generating many more clusters than needed, which can mislead the system administrator. Section 3.5.3 validates both observations empirically.

We have developed a new algorithmic framework, called *anomaly-based clustering*, that clusters H with consideration of the instances F to be diagnosed. (That is, the same H may be clustered differently for a different F.) Intuitively, anomaly-based clustering will place two instances $h_1, h_2 \in H$ into the same cluster iff they have similar deviations from F. This strategy gives the right answer for the example in Figure 3.7(b), generating a single cluster for H, and linking the failure to

attribute x only. The rest of this section describes anomaly-based clustering.

3.4.1 Diagnosis Vectors and Margin Classifiers

Fa processes a *Diagnose(F, H)* query by first clustering the healthy data H into a set of clusters C_1, C_2, ..., C_l, and then outputting the deviation of F from these clusters in the form $\{\langle\vec{w}_1, C_1\rangle, \langle\vec{w}_2, C_2\rangle, \ldots \langle\vec{w}_l, C_l\rangle\}$ as the diagnosis result. l depends on the query, and is not a predetermined constant. $C_1 \cup C_2 \cup \cdots \cup C_l$ need not include all the instances in H. Thus, outlier instances in H will be ignored.

Each $\vec{w} \in \{\vec{w}_1, \ldots, \vec{w}_l\}$ is called a *diagnosis vector*. $\vec{w}$ has the form: $\vec{w} = \langle w_1, w_2, \ldots, w_n\rangle$, where each attribute $x_j \in \langle x_1, x_2, \ldots, x_n\rangle$ is given a weight w_j such that $-1 \leq w_j \leq 1$ and $\sum_{j=1}^{n} |w_j| = 1$. Intuitively, $\vec{w}_i \in \{\vec{w}_1, \ldots, \vec{w}_l\}$ specifies the weighted list of attributes to which the failure can be localized by comparing the instances in C_i to the failure instances F. C_i serves as the evidence why $\vec{w}_i$ is reported in the diagnosis result.

Computing the Diagnosis Vector: Since we are dealing with high-dimensional data, a most desirable property of each $\langle\vec{w}, C\rangle$ is to make $\vec{w}$ as concise as possible. That is, the weights of all attributes that do not help differentiate between C and F should be zero. This property enables the system administrator to zoom in quickly on likely causes of the failure without being misled by false positives. Fa uses *margin classifiers (MC)* to achieve this property. A margin classifier $MC(F, C)$, $C \subseteq H$, finds the linear combination $\sum_{j=1}^{n} w_j x_j$ of attributes $\langle x_1, \ldots, x_n\rangle$ that produces the maximum separation between C and F. This maximum separation is called the *margin* between C and F.

Example 3.4.1. *Consider query* Q=Diagnose(F, H) *from Example 3.2.1 and Fig-*

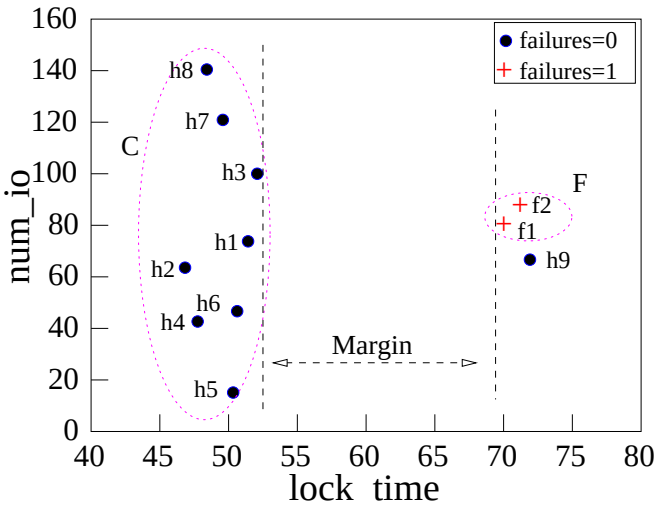

Figure 3.8: Plot of data in Fig.3.1

ure 3.8. Let $C = H - \{h9\}$. *(h9 is an erroneous observation generated while the system transitioned from a healthy state to a failure state.) The margin between* C *and* F *is produced between the two dotted lines in Figure 3.8: the line* `lock_time` $= 51.9$ *and the line* `lock_time` $= 70.0$. *Thus, margin* $= 18.1$. *Since the margin is produced along* `lock_time`, *the diagnosis vector* $\vec{w} = \langle w_1, w_2 \rangle$ *(corresponding to* $\langle$`lock_time`, `num_io`$\rangle$*) that produces the margin is* $w_1 = -1$ *and* $w_2 = 0$. $\langle 1, 0 \rangle$ *also produces the same margin.*

Figure 3.9 shows how a margin classifier $MC(F, C)$, $C \subseteq H$, works by solving a linear program to compute the margin between C and F. $MC(F, C)$ also finds the diagnosis vector that produces the margin. Section 3.5.3 gives an empirical validation of how margin classifiers produce concise and correct diagnosis vectors.

Procedure *Margin Classifier (MC)*
Input: Healthy instances C, Failure instances F
Output: Margin m between C and F, and the diagnosis vector $\vec{w} = \langle w_1, w_2, \ldots, w_n \rangle$ that produces the margin
MC solves the following linear program:
1. Variables in the linear program:
   (i) X_i, Y_i, $1 \leq i \leq n$, such that output $w_i = X_i - Y_i$
   (ii) *High, Low* such that output $m = High - Low$
2. Constraints in the linear program:
   (i) $\sum_{i=1}^{n}(X_i - Y_i)t.\mathbf{x}_i \geq High, \quad \forall t \in C$
   (ii) $\sum_{i=1}^{n}(X_i - Y_i)t.\mathbf{x}_i \leq Low, \quad \forall t \in F$
   (iii) $\sum_{i=1}^{n}(X_i + Y_i) = 1$
   (iv) $X_i \geq 0, Y_i \geq 0, \quad 1 \leq i \leq n$
3. Optimization objective: Maximize $High - Low$
4. Solve the linear program; Return m and $\vec{w}$

Figure 3.9: Margin Classifier ($MC(F, C)$)

3.4.2 Strawman: Margin-based Agglomerative Clustering (MAC)

We begin with a strawman algorithm, called *margin-based agglomerative clustering (MAC)*, for anomaly-based clustering of H. MAC was proposed originally in [56] for analyzing cancer-related microarray data, and we have extended it to process diagnosis queries. MAC starts with an agglomerative hierarchical clustering [88] of the instances in H. The margin from the failure instances F is used as the metric for clustering. (Conventional clustering schemes use distance-based metrics like Euclidean distance.) Each instance in H is first placed in its own active cluster. In each iteration, MAC computes $MC(C_i \cup C_j, F)$ for each pair $\langle C_i, C_j \rangle$ of clusters among the remaining active clusters. MAC then picks the cluster pair $\langle C_{i'}, C_{j'} \rangle$ that gives the maximum margin with respect to F, and merges (agglomerates) them together to form a single combined cluster. The merged clusters $C_{i'}$ and $C_{j'}$

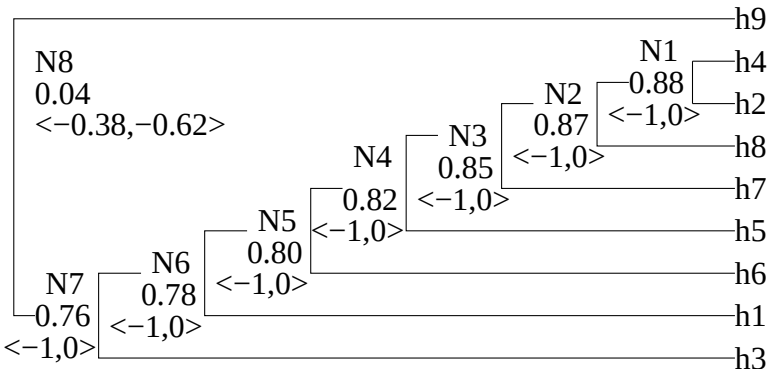

Figure 3.10: Dendogram

are no longer considered active. This process is repeated until all instances are merged into a single cluster. The entire process can be represented as a *dendogram*, which is a tree with the instances in H as leaves, and each new cluster formed by MAC as a nonleaf node.

Example 3.4.2. *Figure 3.10 shows the dendogram generated by MAC for the healthy data in Example 3.2.1 and Figure 3.8. The margin (computed after normalizing the data) and diagnosis vector for the cluster at each nonleaf node are also shown.*

We can generate clusters from the dendogram by selectively deleting nonleaf nodes which will partition the dendogram into a forest of trees. The instances comprising the leaves of each tree form a cluster C that will be output as a $\langle \vec{w}, C \rangle$ pair in the query result after computing $MC(F, C)$.

Consider a node P in the dendogram with child nodes L and R. Let the clusters corresponding to these three nodes be C_p, C_l, and C_r respectively. (Note that C_l and C_r were merged to form C_p in the dendogram, i.e., $C_p = C_l \cup C_r$.) Let the margin and corresponding diagnosis vector for these three clusters be $\langle m_p, \vec{w}_p \rangle$, $\langle m_l, \vec{w}_l \rangle$, and $\langle m_r, \vec{w}_r \rangle$ respectively. We will delete P if any of the following conditions is satisfied: (i) $Margin(C_l, F, \vec{w}_p) < (1 - \alpha)m_l$, or (ii) $Margin(C_r, F, \vec{w}_p) < (1 - \alpha)m_r$.

Procedure *Margin*$(C, F, \vec{w})$
Input: Cluster of healthy instances C, failure instances F, and diagnosis vector $\vec{w} = \langle w_1, w_2, \ldots, w_n \rangle$;
Output: Margin between C and F along vector $\vec{w}$;
/* Find the minimum value of $\vec{w} \cdot h$ among all instances $h \in C$ */
1. $h_{min} = \text{argmin}_{h \in C} \sum_{i=1}^{n} w_i \times h.\mathbf{x}_i$;
/* Find the maximum value of $\vec{w} \cdot f$ among all instances $f \in F$ */
2. $f_{max} = \text{argmax}_{f \in F} \sum_{i=1}^{n} w_i \times f.\mathbf{x}_i$;
3. **Return** $\sum_{i=1}^{n} w_i \times h_{min}.\mathbf{x}_i - \sum_{i=1}^{n} w_i \times f_{max}.\mathbf{x}_i$

Figure 3.11: Margin along a diagnosis vector

Here, *Margin*$(C, F, \vec{w})$ denotes the margin (separation) of a cluster of instances C from the failure instances F along the diagnosis vector $\vec{w}$; the details are given in Figure 3.11. α is a small positive constant, e.g., $\alpha = 0.2$.

Note that $\vec{w}_p$ is the diagnosis vector that gives the margin for the combination of C_l and C_r. Intuitively, if the margin of C_l (C_r) along $\vec{w}_p$ is significantly less than the margin of C_l (C_r), then merging C_l with C_r is diluting the "clusteredness" of C_l (C_r) with respect to the failure instances F. (Note that *MC*(C_l, F) computes C_l's maximum margin across all possible diagnosis vectors.)

This intuition is applied in the second phase of MAC—see Lines 6 and higher in Figure 3.12—to partition the dendogram into clusters that are then output in the query result.

Example 3.4.3. *When the dendogram in Figure 3.10 is partitioned, the nonleaf node* N_8 *will be deleted, to generate two output clusters* $\{h1, h2, h3, h4, h5, h6, h7, h8\}$ *and* $\{h9\}$*; precisely what we expect based on Figure 3.8. Recall that h9 is an erroneous observation generated during system transition.*

Algorithm *Margin-based Agglomerative Clustering (MAC)*
Input: *Diagnose*(H, F) query; Value of α (default is 0.2)
Output: Result in the $\{\langle \vec{w}_1, C_1\rangle, \langle \vec{w}_2, C_2\rangle, \ldots \langle \vec{w}_l, C_l\rangle\}$ format
/* Phase 1: generate a dendogram for the instances in H */
1. Initialize the dendogram by creating a leaf node for each instance in H, and place each H instance in its own cluster;
2. **While** (more than one cluster remains) {
3. **For** each cluster-pair C_i,C_j from the remaining clusters Compute $\langle m_{ij}, \vec{w}_{ij}\rangle = MC(\{C_i \cup C_j\}, F)$;
4. Pick the C_i,C_j pair that has the maximum margin m_{ij};
5. Merge C_i and C_j to form a new cluster C. Create a new node in the dendogram for C, with nodes for C_i and C_j as children. Drop C_i and C_j from consideration for merging;
} /* end while */
/* Phase 2: partition dendogram to generate the output clusters */
6. **For** (each node P in the dendogram) {
7. Let $\vec{w}_p$ be the diagnosis vector of P. Let m_l and m_r be the margins of P's children L and R. Let C_l and C_r be the respective clusters for nodes L and R;
8. Delete P if $Margin(C_l, F, \vec{w}_p) < (1-\alpha)m_l$, or if $Margin(C_r, F, \vec{w}_p) < (1-\alpha)m_r$;
} /* end for */
9. For each tree T in the partitioned dendogram
10. Let $C \subseteq H$ be the cluster formed by the instances in T's leaves; compute $\langle m, \vec{w}\rangle$=$MC(C, F)$, and output $\langle \vec{w}, C\rangle$

Figure 3.12: Margin-based Clustering (MAC)

MAC is inefficient: MAC requires $O(|H|^2)$ invocations of MC since MAC starts by invoking $MC(\{t_1, t_2\}, F)$ for every pair of instances t_1, t_2 in H. Thus, MAC scales poorly with $|H|$, but it gives good diagnosis accuracy; we will validate both observations empirically in Section 3.5.3.

3.4.3 Partition-Check-Merge (PCM) Algorithms

We now propose an algorithmic framework that combines the good features of MAC, which is accurate, but inefficient, with those of conventional Distance-based Parti-

tional[1] Clustering (*DPC*) algorithms that are efficient, but less accurate. We later describe clustering algorithms that instantiate this framework. This new framework is called *Partition-Check-Merge (PCM)* because it has the following structure:

- One or more *partitioning phases* that use an efficient DPC algorithm to partition the data progressively into more and more clusters until the check in Step 2 is satisfied. This progressive cluster refinement is achieved by increasing the input parameter k to the DPC algorithm that specifies the number of clusters to generate.

- One or more *checking phases* that perform checks, namely, evaluating the current partitioning of instances to see whether this partition is good enough to be the set of clusters produced during an intermediate stage of MAC. If a check succeeds, then PCM moves to the merging phase; otherwise, partitioning is continued, possibly with a larger k.

- A *merging phase* where the current set of clusters are merged progressively, like in MAC, to possibly consolidate several small clusters into a minimal set of clusters (representing diagnosis vectors and evidence) that can be output in the query result.

A degenerate case of PCM is one where the check never succeeds, so the partitioning phase eventually places each instance into a separate cluster. In this case, the merge phase will resemble running MAC from scratch. However, for most monitoring datasets, the check will succeed much earlier—e.g., once k becomes equal to or larger than the best k for the data—avoiding the $O(|H|^2)$ complexity of MAC.

[1]Intuitively, partitional algorithms work in a top-down fashion, while agglomerative algorithms work bottom-up.

If the partitioning phase generates more clusters than optimal, then the merging phase will glue back clusters that should not have been split in the first place; at some loss of efficiency. In effect, PCM can be as accurate as MAC, while leveraging the efficiency of DPC. The challenge in PCM is in the implementation of the check phase. Next, we discuss two concrete instantiations of PCM.

3.4.4 PCM-Conservative (PCM-C)

PCM-Conservative (PCM-C) (Figure 3.13) uses a conservative implementation of check to process a *Diagnose*(F, H) query. For each instance $t \in H$, PCM-C first computes m_t, the individual margin between t and the failure instances F.

For partitioning data instances, PCM-C does DPC using the LAC [32] algorithm (Line 7 in Figure 3.13). Suppose the clusters $\{C_1, \ldots, C_k\}$ are produced by a partitioning step. For each $C \in \{C_1, \ldots, C_k\}$, let $\langle m_C, \vec{w}_C \rangle$ be the margin and corresponding diagnosis vector for C and F. PCM-C's check phase lists an instance $t \in C$ as *covered by* C if *Margin*$(\{t\},F,\vec{w}_C) \geq (1-\alpha)m_t$. (Recall from Section 3.4.2 that *Margin*$(\{t\}, F, \vec{w}_C) \leq m_t$, since m_t is t's maximum margin across all vectors.) That is, t is covered by the cluster C that t was assigned to by DPC if t's margin along C's diagnosis vector is close enough to t's individual margin.

If t is covered by C, then (i) t will not be considered again during partitioning (Line 21), and (ii) t will be associated with C in the input to the merge phase. PCM-C iterates through the partitioning and merge phases until all instances get covered. If check finds that the current set of clusters $\{C_1, \ldots, C_k\}$ do not cover a significant fraction of the remaining instances, then partitioning is redone with a larger k; currently, we double k when this situation arises (Lines 16-17). Thus, while PCM-C starts with a small default value of k, k will get incremented automatically

Algorithm *Partition-Check-Merge-Conservative (PCM-C)*
Input: *Diagnose*(F, H) query; Value of α (default is 0.2)
Output: Result in the $\{\langle \vec{w}_1, C_1\rangle, \langle \vec{w}_2, C_2\rangle, \ldots \langle \vec{w}_l, C_l\rangle\}$ format
/* First compute the individual margin of each instance $t \in H$ */
1. Compute $\langle m_t, \vec{w}_t\rangle = MC(\{t\}, F)$ for each instance $t \in H$;
2. k = default value; /* LAC's number of clusters parameter */
3. $Rem\_pts = H$; /* instances not assigned to clusters yet */
4. $Coverings = \phi$; /* assignment of instances to clusters */
5. **While** $(Rem\_pts \neq \phi)$ {
6. /* Partitioning phase */
7. Partition Rem_pts with LAC into clusters $\{C_1, \ldots, C_k\}$;
8. $Outliers = \phi$;
9. /* Check phase: Lines 10-22 below */
10. **For** (each instance $t \in Rem\_pts$) {
11. Let C_i be the cluster that t was assigned to by LAC;
12. **If** $(Margin(\{t\}, F, \vec{w}_{C_i}) \geq (1-\alpha)m_t)$
13. Mark t as covered by C_i;
14. **Else** Add t to *Outliers*;
15. } /* end for */
16. **If** $(\frac{|Outliers|}{|Rem\_pts|} > 0.9)$
17. $k = k \times 2$; /* increase k, Rem_pts is unchanged */
18. **Else** {
19. **For** (each cluster $C_i \in \{C_1, \ldots, C_k\}$ that covers instances)
20. Add C_i and the instances C_i covers to *Coverings*;
21. $Rem\_pts = Outliers$; /* remove covered instances */
22. } /* end else */
23. } /* end while */
24. /* Merge phase */
25. Initialize a partially-built dendogram with the clusters in
26. *Coverings* as the leaves of the dendogram;
27. Proceed with MAC using this partially-built dendogram;

Figure 3.13: PCM-Conservative (PCM-C)

if required.

Once all instances in H have been covered by a set of clusters generated by DPC, these clusters are input to the merge phase. Merge does MAC-style agglomerative clustering—starting with these clusters as the leaves of the dendogram, instead of the $|H|$ individual instances—to generate the final query output.

3.4.5 PCM-Eager (PCM-E)

We found empirically (Section 3.5.3) that PCM-C tends to generate many clusters as input to the merge phase. While merge can glue back clusters that should not have been split, PCM-C's merge remains inefficient because of the quadratic dependence on the number of input clusters. PCM-E tackles this problem.

Recall that PCM-C's check phase will list a instance t as covered only if t's margin along the diagnosis vector of the cluster C that t was assigned to by DPC is close enough to t's individual margin. *PCM-Eager (PCM-E)* relaxes this condition as follows: PCM-E's check lists t as covered if t's margin along the diagnosis vector of *any* of the clusters generated by DPC so far is close enough to t's individual margin. As before, if t is covered by C, then t will be assigned to C in the input to the merge phase. (Note that DPC may not have assigned t to C.) Intuitively, PCM-E reduces DPC's role to identifying significant diagnosis vectors $\vec{w}$ from the data. The $\mathit{Margin}(\{t\}, F, \vec{w}) \geq (1-\alpha)m_t$ condition is used to associate instances with each $\vec{w}$, creating clusters that are input to the merge phase. The rest of PCM-E is similar to PCM-C.

3.4.6 Filtering and Ranking Diagnosis Results

Fa takes the set of $\langle \vec{w}_i, C_i \rangle$ pairs generated by anomaly-based clustering, and outputs the final diagnosis result as a filtered and ranked list.

- **Filtering:** Fa removes $\langle \vec{w}_i, C_i \rangle$ pairs where the cluster size $|C_i|$ does not satisfy a minimum support threshold; similar to support thresholds in frequent-itemset mining. Clusters composed of outliers get eliminated here.

Name	a	i	Description of data and failures
1. Rubis-60	110	8184	Contains annotated data about 60 distinct single-EJB failures injected in our testbed
2. Rubis-complex	110	1797	Contains annotated data about 14 distinct multiple-EJB failures injected in our testbed
3. Synthetic	16	10992	Synthetic annotated data about 10 distinct failures; patterns in the data are complex
4. Dolphin, 5. ECE	43	4881	OS-level data collected for 55 days from two heavily-used departmental servers at Duke
6. Rubis-bug	110	900	Data access by the *BuyNow* EJB gets null result occasionally (bug in application logic)
7. Rubis-jndi	110	1500	JNDI naming-directory entry of the *SB_SearchItemsByRegion* EJB gets corrupted
8. OLTP-single	42	3660	Occasional CPU contention caused by an application on OLTP server (no disk contention)
9. OLTP-multi	28	696	Both CPU and disk contention caused separately by an application on the OLTP server

Table 3.2: Monitoring datasets used in the evaluation. Columns a and i are the number of attributes and instances respectively. Datasets 1-3 are used to evaluate Phase I, and datasets 4-9 to evaluate Phase II

- **Ranking:** The remaining $\langle \vec{w}_i, C_i \rangle$ pairs are ranked in decreasing order of cluster size $|C_i|$.

3.5 Experimental Evaluation

3.5.1 Experimental Setting

All the diagnosis techniques described in this chapter were implemented in the Fa system. We evaluate these techniques in the context of a three-tier Web service composed of a Web server, an application server, and a database server. The evaluation is based on common types of failures in each tier. Table 3.2 summarizes our datasets and failure scenarios.

Failures injected in a testbed: We have implemented a testbed that runs *Rubis* [20]—a multitier auction service modeled after eBay—on a JBoss application server (with an embedded Web server) and a MySQL DBMS. It has been reported that software problems and operator errors are the common causes of failure in Web services [60]. We inject such failures into a running Rubis instance using a comprehensive failure-injection tool [15]. This setting makes it easy to study the accuracy of Fa's diagnosis algorithms because we always know the true cause of each failure.

Specifically, we can inject 3 independent causes of failure—software bugs, data corruption, and uncaught Java exceptions—into any of the 25 Java modules (*enterprise Java beans (EJBs)*) that comprise the component of Rubis running in the application server. Using this mechanism, we can inject 75 distinct single-EJB failures and any number of independent multiple-EJB failures (concurrent single-EJB failures). Intuitively, multiple-EJB failures are harder to diagnose. The *Rubis-60*, *Rubis-complex*, *Rubis-bug* and *Rubis-jndi* monitoring datasets in Table 3.2 are from this setting. These datasets contain the number of times each distinct EJB procedure call is invoked per minute.

We can also inject failures caused by contention for CPU, memory, and disk resources. The *OLTP-single* and *OLTP-multi* monitoring datasets in Table 3.2 are collected from a MySQL DBMS running an OLTP workload, where we injected resource contention to cause failures. The datasets record OS metrics (e.g., CPU utilization, paging), DBMS performance counters (e.g., number of index accesses and table scans), and transaction-level performance metrics (e.g., average transaction response time) per minute.

Real failures in a production system: *Software aging*—progressive degradation in performance caused by, e.g., memory leaks, unreleased file locks, and

fragmented storage space—is a common cause of system failure, especially in Web servers [84]. The *Dolphin* and *ECE* datasets were collected from two production servers at Duke over the course of two months. This data records OS metrics at 10-minute intervals. Both servers crashed once or more during this period due to aging of different resources, as found by a previous study [84]. We validate Fa's automated diagnosis results with the results from this human-intensive study.

Synthetic data: *Synthetic* is a complex dataset (*PENDIGITS*) from the UCI machine-learning repository. Rather than generating our own synthetic data, we decided to use this dataset for the purpose of experimental repeatability.

3.5.2 Evaluation of Phase I

Queries: We consider *Diagnose*(F, L) queries over the Rubis-60, Rubis-complex, and Synthetic datasets. By default, L contains 60% of the failure instances in each dataset, and is used to generate the signature database. The remaining 40% of the failure instances are used to query the signature database to compute its diagnosis accuracy (% of times the correct annotation is returned).

Techniques: We compare four techniques: (i) CLUS, signature database implemented using K-means clustering with 10 clusters per annotation[2]; (ii) FA, Fa's signature database; (iii) FA-EA, Fa's error-aware signature database; and (iv) CART, a multi-class classifier implemented using classification and regression trees. By treating each annotation in L as a distinct class label, a multi-class classifier learned from L can predict the annotation of a new failure instance. We chose CART over

[2]We also tried the state-of-the-art LAC clustering [32], and got similar results.

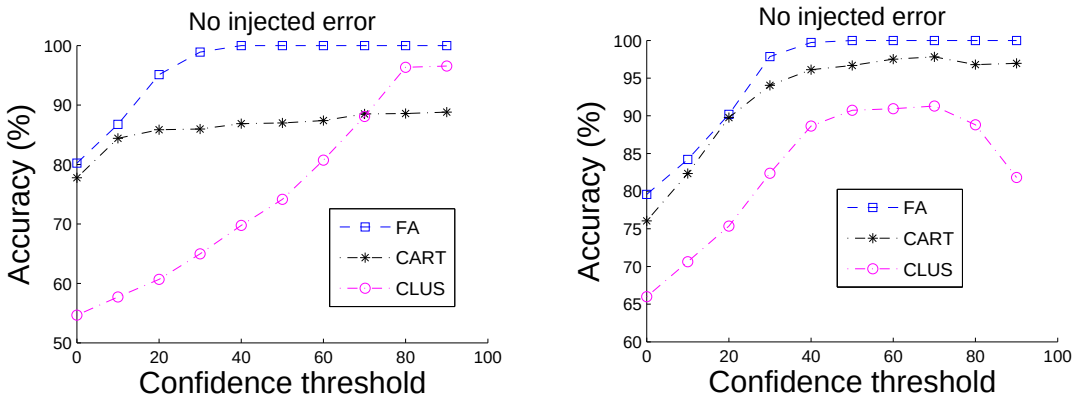

Figure 3.14: AC-Curves for Rubis-60, Rubis-complex

other multi-class classifiers for three reasons: (i) CARTs are being used for diagnosis in production settings like eBay.com [24]; (ii) CARTs provide a principled way to compute confidence estimates; and finally, (iii) Fa uses CARTs as separating functions.

Comparing Accuracy-Confidence Curves: The goal of Phase I is to provide high diagnosis accuracy while invoking Phase II only when required. The Accuracy-Confidence Curves (AC-Curves) in Figure 3.14 show how well each technique meets this goal. No error is injected into the query instances (unlike the experiments in Section 3.5.2). Recall the definition of confidence estimates, confidence thresholds, and AC-Curves from Section 3.3. To diagnose a query instance, CARTs compute a probability distribution over annotations, and output the most-likely annotation. The confidence estimate is the difference in probabilities between the most-likely annotation and the second most-likely annotation, mapped to $[0, 100]$ as discussed in Section 3.3.4.

Suppose the administrator wants a diagnosis accuracy of 90%. Then, Figure 3.14(a) for Rubis-60 shows that the confidence thresholds (C_t) for FA and CLUS

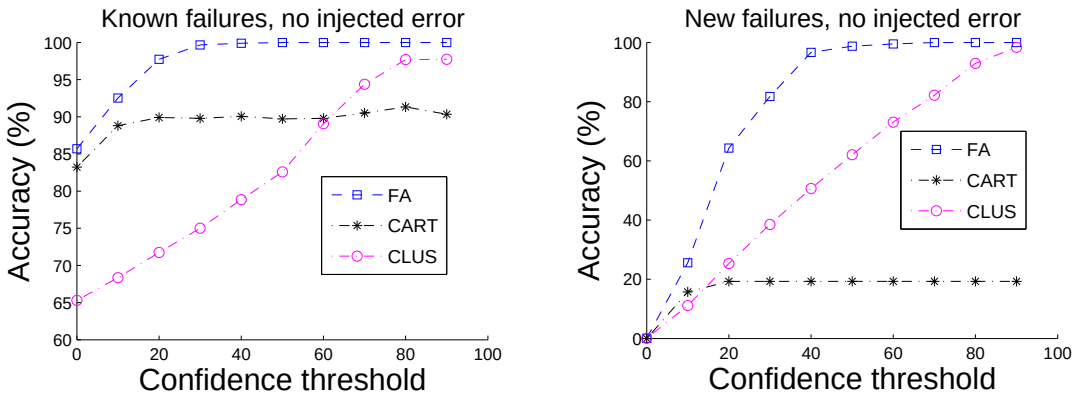

Figure 3.15: AC-Curves for Rubis-60 with two groups: (a) existing annotations, (b) new annotations

can be set to 20 and 80 respectively. Since FA's C_t is 4 times less that of CLUS, FA is four times less likely to invoke Phase II at the same accuracy level. More interestingly, CART is unusable when required accuracy is 90%. FA maintains its superior performance for Rubis-complex, while CLUS now becomes unusable when the required accuracy is over 90%.

Figure 3.14 used our default setting where for each query instance $\langle \vec{x}, A \rangle$, the signature database contains at least one signature for annotation A. That is, we evaluated how good the signature database is in diagnosing previously-seen failures (which does not mean previously-seen instances). This setting is practical because as much as 90% of all software failures reported by users today are previously-seen failures [14]. We will now consider query instances whose correct annotations are not in the signature database. An accurate response from Phase I in this case is an answer with confidence below the threshold C_t; thereby invoking Phase II.

To create this setting, we divide the instances in Rubis-60 into two groups with nonoverlapping annotations: Group G_1 with 40 annotations and Group G_2 with the remaining 20 annotations. A subset of the instances from G_1 are used to construct the signature database. The remaining instances in G_1 (previously-seen failures) and

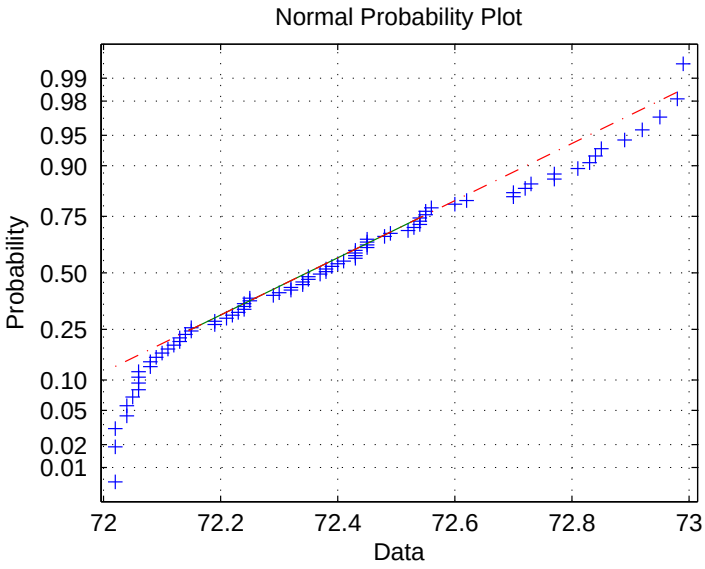

Figure 3.16: Normal probability plot for attribute $CPU\_utilization \in (72, 73)$ in OLTP-single

the instances in G_2 (new failures) form the set of query instances. Figures 3.15(a) and (b) show the diagnosis accuracy of different techniques in these two cases. The behavior of signature databases (FA and CLUS) for both types of failures is as we saw in Figure 3.14(a). However, CART performs poorly on the new failures, which shows a key advantage of using signature databases for Phase I rather than multi-class classifiers. Intuitively, signature databases have a better chance of detecting when a failure does not have the symptoms of any previously-seen failure. Our algorithm for setting C_t (described in Section 3.3.4) considers both types of plots in Figure 3.15.

Verification of Error Models: Section 3.3.5 has discussed the Gaussian error model. To verify the presence of Gaussian error in our monitoring data collected for an attribute (e.g., CPU utilization), we took multiple measurements of this attribute from a particular system state. The normal probability plot of these values in Figure

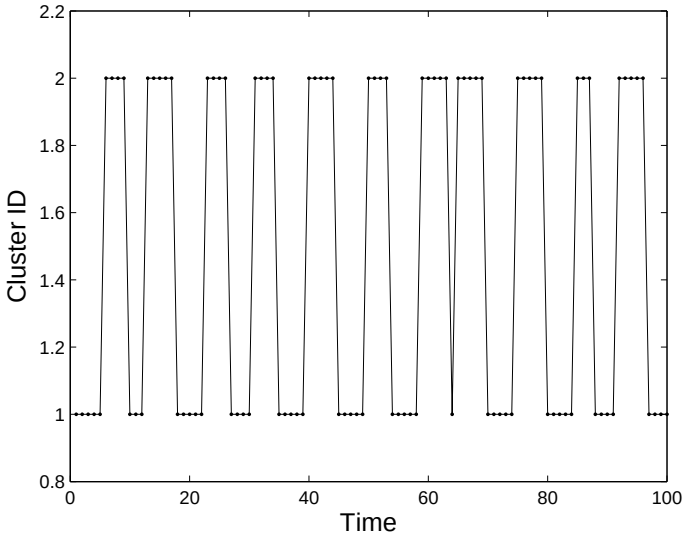

Figure 3.17: Cluster timeline for a subset of OLTP-single

3.16 and a hypothesis test for goodness of fit to a normal distribution (Matlab's Lillietest) prove that Gaussian error exists in the monitoring data.

As mentioned in Section 3.3.5, some errors cannot be captured by the Gaussian distribution, e.g., when observations from different system states get mixed into an instance due to rapid system state transition. We verify such a situation with Figure 3.17 and Figure 3.18. Figure 3.17 is a *cluster timeline* generated from a subset of the OLTP-single dataset when clustered using LAC with $k = 2$. A cluster timeline shows the progress of time on the x-axis and the current cluster identifier on the y-axis. In the OLTP-single dataset, CPU contention happens on the OLTP server in a periodic fashion with a period of 4 minutes; this pattern is clear from the cluster timeline. Figure 3.18 is a cluster transition diagram that shows how the diagnosis vectors change over time. (This figure is best viewed in color.) Like Figure 3.17, Figure 3.18 was also generated from a subset of the OLTP-single dataset when clustered using LAC with $k = 2$. The x-axis and y-axis are two relevant attributes

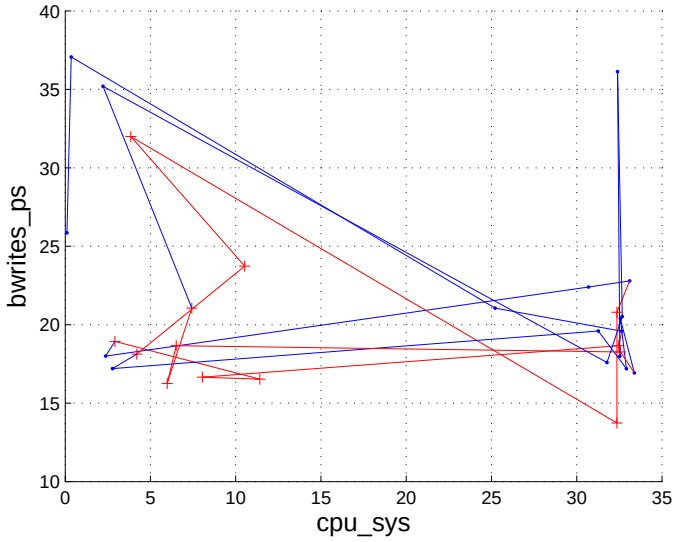

Figure 3.18: Cluster transition diagram for a subset of OLTP-single

in the data, namely, `cpu_sys`, the CPU utilization (in OS space) on the OLTP server, and `bwrites_ps`, the number of disk blocks written per second on the OLTP server. Points belonging to the two clusters in the data are indicated respectively using a red "+" symbol and a blue "." symbol. A line L_{p_1,p_2} from point p_1 to p_2 in Figure 3.18 indicates that the system was in state p_1, and then transitioned to state p_2 in the next measurement interval; the color of L_{p_1,p_2} is the same as the color of point p_2 in Figure 3.18. Figure 3.18 illustrates some interesting aspects of the OLTP-single dataset:

- The red "+" points predominantly have smaller `cpu_sys` than the blue "." points.
- The system tends to stay in the red (blue) state for four measurement intervals, and then transitions to the blue (red) state.
- There are *transitionary points* in the data. These are points that are collected

when the system is transitioning from one state to another, so these points may belong to different (similar) clusters, but have similar (different) attribute values.

Transitionary points contain non-Gaussian errors that are affected by which states are involved in the transition. We adopt a probabilistic model to describe such non-Gaussian errors. This model assigns a probability p_i to attribute x_i that specifies how probable the reading of x_i is an incorrect value, a value in its observation range that is not the true value. p_i represents the scale of non-Gaussian error for x_i.

Comparing Robustness to Error: The query instances used so far to compute the accuracy of our diagnosis techniques were taken directly from the monitoring data. We now inject errors into these query instances based on error models to study how accuracy degrades as error increases.

Note that the model we use for Gaussian error is described in Section 3.3.5. In our experiments with Gaussian-error model, the parameter δ_i for x_i is set to $0.2 * rand * error\_level$ multiplied by x_i's true value, where $rand$ is a random value within [0, 1] and $error\_level \in \{1, 2, 3, 4\}$ controls the scale of errors. For non-Gaussian error model, the parameter p_i for x_i is set to $0.2 * rand * error\_level$. $rand$ is to make different attributes have different scale of errors. Roughly speaking, for an attribute x: (i) Gaussian error of level e means that observations of x have a variance of $10e\%$ from their true value; and (ii) non-Gaussian error of level e means that each observation of x has a $10e\%$ chance of being an arbitrary value from the range of values of x.

Figures 3.19 and 3.20 show the AC-Curves for error levels 1 and 2 for Gaussian

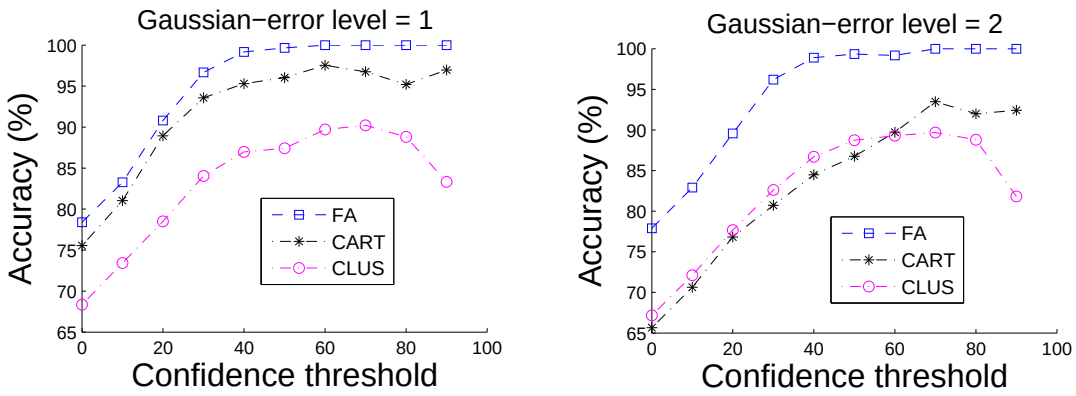

Figure 3.19: AC-Curves for Rubis-complex with Gaussian error: (a) error level = 1, (b) error level = 2

and non-Gaussian error in Rubis-complex respectively. (Note that Figure 3.14(b) is the AC-Curve at error level 0.) It is clear from comparing these graphs that the gap between FA and CLUS/CART increases as the error increases. CLUS is highly sensitive to non-Gaussian errors. Further evidence is provided by Figure 3.21 where the confidence threshold C_t is set at 40. Figure 3.21 shows the accuracy of different techniques as the error level, both for Gaussian and non-Gaussian error, increases from 0 to 4 for Rubis-complex. Also note that FA's performance is very close to that of the FA-EA algorithm which has knowledge of the expected error. The observations validate FA's robustness to error.

Figures 3.22 and 3.23 show the AC-Curves for error levels 1 and 2 for Gaussian and non-Gaussian error in Rubis-60 respectively. Note that Figure 3.14(a) is the AC-Curve at error level 0. Figure 3.24 shows the accuracy of different techniques as the error level (for Gaussian and non-Gaussian error) increases from 0 to 4 for Rubis-60. These figures again validate FA's robustness to error.

Scalability with Number of Annotations: Figure 3.25 shows the trend as the number of annotations—distinct single-EJB failures—is increased from 20 to 80.

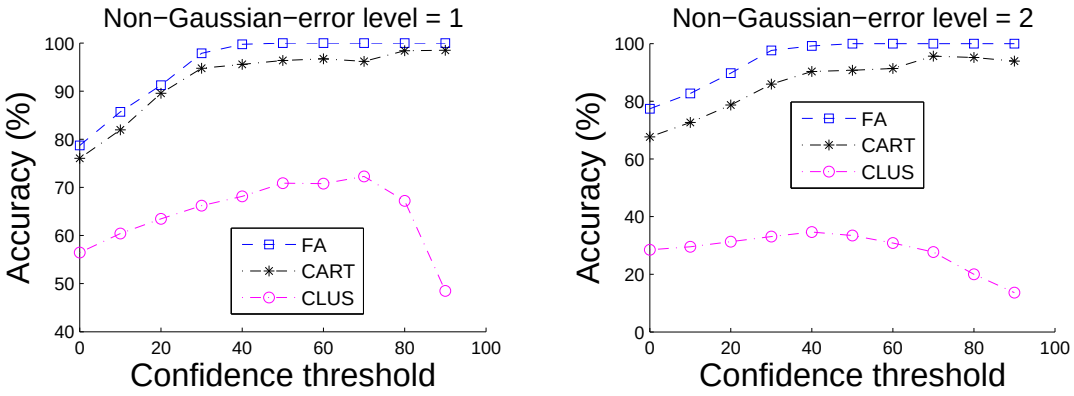

Figure 3.20: AC-Curves for Rubis-complex with Non-Gaussian error: (a) error level = 1, (b) error level = 2

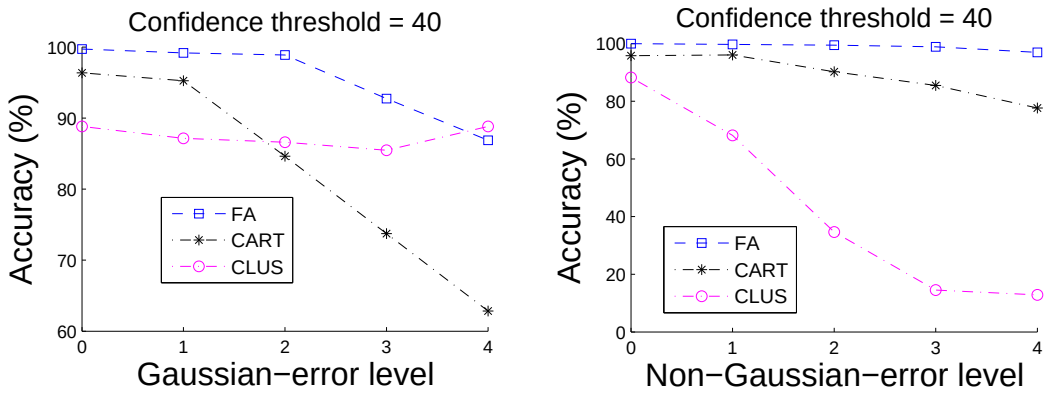

Figure 3.21: Robustness curves for Rubis-complex: (a) Gaussian error, (b) non-Gaussian error

Since we can generate at most 75 distinct single-EJB failures (recall Section 3.5.1), the 80-failure dataset contains 5 multiple-EJB failures as well. The gap between FA and CLUS/CART increases as the number of failures increases.

In the offline phase of signature database generation, FA is less efficient than CART or CLUS. If the separating functions are not learned in parallel, FA can take an order of magnitude more time to generate the database than CART or CLUS. However, these offline efforts make FA much better in the online phase because: (i) FA is comparable to CLUS and CART in the time for Phase I; and (ii) FA invokes

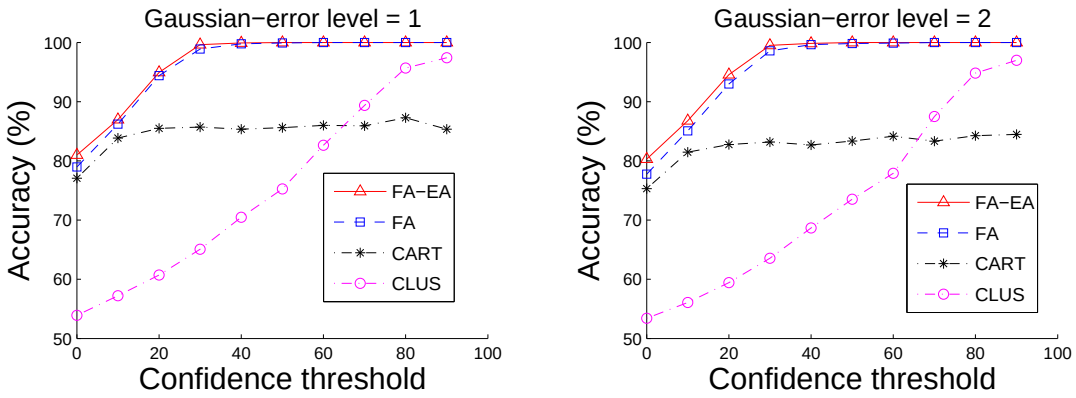

Figure 3.22: AC-Curves for Rubis-60 with Gaussian error: (a) error level = 1, (b) error level = 2

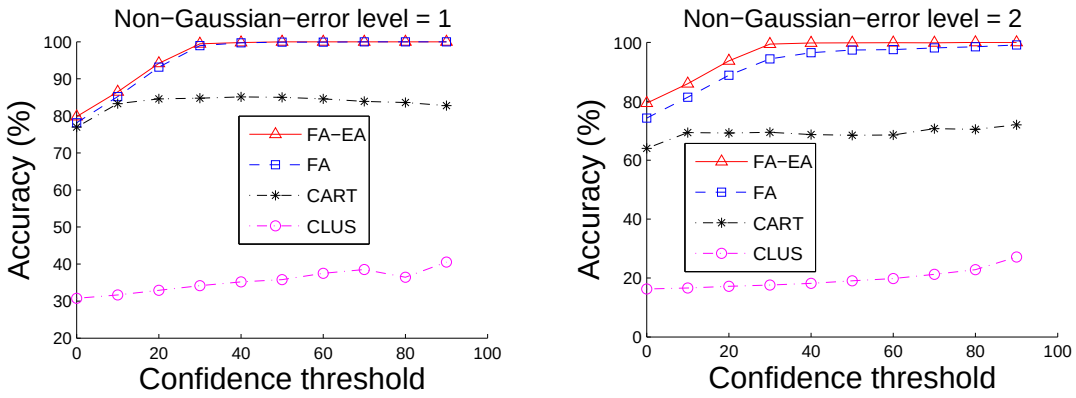

Figure 3.23: AC-Curves for Rubis-60 with Non-Gaussian error: (a) error level = 1, (b) error level = 2

Phase II much less often.

3.5.3 Evaluation of Phase II

Queries: We now evaluate the processing of *Diagnose*(F, H) queries in Phase II. For datasets 4-9 listed in Table 3.2, H contains the historical healthy monitoring data and F contains 5-10 instances from the listed failures. For Dolphin and ECE, F contains 5 instances collected just before each server's first crash. We consider two cases for OLTP-multi, one where F contains failure instances from CPU contention,

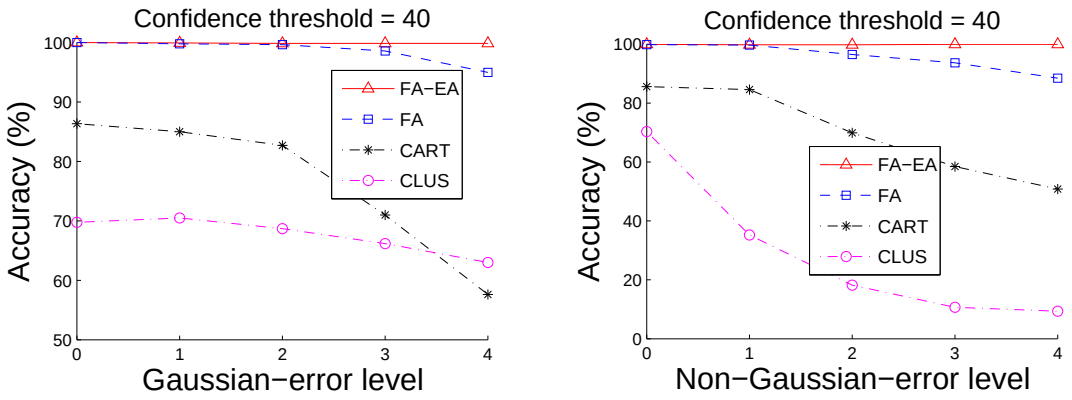

Figure 3.24: Robustness curves for Rubis-60: (a) Gaussian error, (b) non-Gaussian error

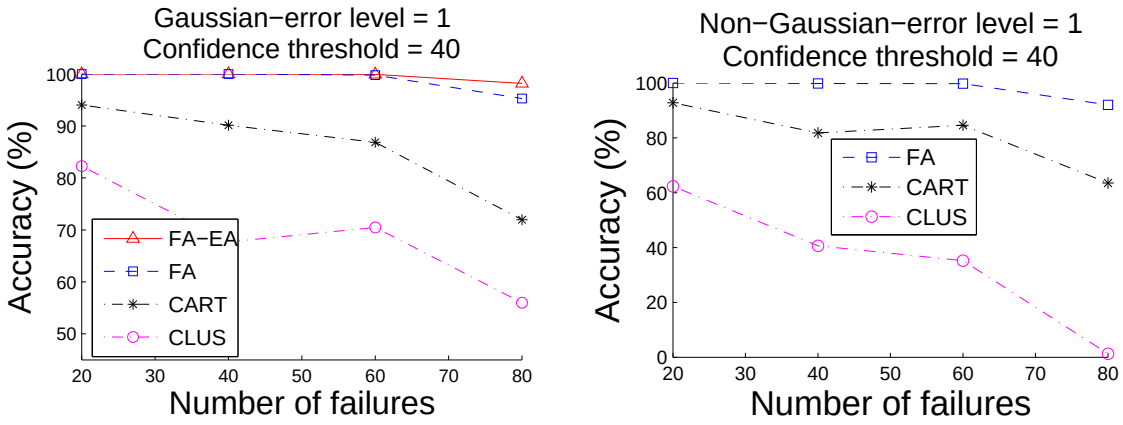

Figure 3.25: Trend as the number of failures increases

and the other where F contains failure instances from disk contention. We can evaluate the accuracy of diagnosis results since the cause of failure in each case is known.

Algorithms and Defaults: We evaluate four algorithms for Phase II: (i) MAC (Section 3.4.2), (ii) PCM-C (Section 3.4.4), (iii) PCM-E (Section 3.4.5), and (iv) *LAC-Silhouette (LAC-S)*. LAC-S applies LAC on H after computing the number-of-clusters parameter k that maximizes a validity index called *Silhouette* [12]. Silhouette aims to maximize the inter-cluster distances (the average distance of pairs

Dataset	LAC-S	MAC	PCM-C	PCM-E
Dolphin	260.3 k=19	5137.0 ($\|H\|$= 1000)	87.4 T_p=23.5 T_m=63.9	32.7 T_p=18.8 T_m=13.9
ECE	229.6 k=2	5187.9 ($\|H\|$= 1000)	89.1 T_p=18.9 T_m=70.2	26.1 T_p=17.2 T_m=8.9
Rubis-bug	28.3 k=3	1318.5 ($\|H\|$= 600)	45.8 T_p=40.1 T_m=5.7	34.7 T_p=28.8 T_m=5.9
Rubis-jndi	75.1 k=2	1399.6 ($\|H\|$= 600)	108.2 T_p=92.6 T_m=15.6	95.8 T_p=87.0 T_m=8.8
OLTP-single	214.5 k=2	5116.0 ($\|H\|$= 1000)	173.3 T_p=137.4 T_m=35.9	88.1 T_p=70.6 T_m=17.5
OLTP-multi,F= CPU contention	7.2 k=15	1526.1	7.1 T_p=4.3 T_m=2.8	3.8 T_p=3.2 T_m=0.6
OLTP-multi,F= Disk contention	6.2 k=14	1516.1	7.4 T_p=3.3 T_m=4.1	4.1 T_p=3.6 T_m=0.5

Table 3.3: Comparing running times (seconds)

of points from different clusters) and minimize the intra-cluster distances (the average distance of pairs of points from the same cluster). For each cluster $C \subseteq H$ generated by LAC, LAC-S outputs $\langle \vec{w}, C \rangle$ computed using $MC(F, C)$.

Comparing Running Times: Table 3.3 shows the running time of our algorithms on the different datasets. Each reported time was averaged over 10 runs. For LAC-S, the time shown is the time to compute silhouette indices for 10 different values of k. This time is an optimistic estimate of the running time of LAC-S because we expect that more than 10 choices of k will have to be explored before finding the k that maximizes the silhouette index. We currently try all values of $k \in [2, 30]$. The best k is reported in LAC-S's column in Table 3.3.

Because of its poor scalability, we had trouble running MAC on the full version of all but the smallest dataset (OLTP-multi) in Table 3.2. Therefore, the times for MAC are for scaled-down versions of the datasets. The scaled-down size is shown in MAC's column in Table 3.3. The times for PCM-C and PCM-E are split into the time for the partitioning and checking phases, denoted T_p in Table 3.3, and the time for the merge phase, denoted T_m. The following trends are clear in Table 3.3.

- MAC is very inefficient because of $O(|H|^2)$ *MC* calls.
- PCM-E is by far the most efficient algorithm. Note that PCM-E's T_m is usually significantly better than that of PCM-C. This trend is because PCM-E's aggressive strategy to map points to clusters leads to a much lower number of clusters being input to the (quadratic) merge phase. Furthermore, PCM-E's T_p is usually better than that of PCM-C because PCM-E's aggressive strategy gets all points covered in fewer iterations of the partitioning and checking phases.
- PCM-E typically matches or outperforms LAC-S, which is because the silhouette computations in LAC-S perform $O(|H|^2)$ distance computations.

Comparing Diagnosis Accuracy of Phase II: Table 3.4 summarizes the diagnosis accuracy of our algorithms on the datasets. Numbers like 1st and 2nd in Table 3.4 indicate the smallest rank of a cluster C whose diagnosis vector gives non-zero weights to attributes relevant to the failure (smaller rank is better). The non-zero weights in this diagnosis vector are shown for PCM-E, with the weights for attributes relevant to the failure shown in bold font. Each cell also shows the % size of C with respect to the number of historical points $|H|$, and the number

Dataset	LAC-S	MAC	PCM-C	PCM-E	Diagnosis vector for PCM-E
Dolphin	8th, 5%, 14	Not found	1st, 39%, 8	1st, 37%, 7	2 nonzero weights, (**0.77**, **0.23**)
ECE	Not found	2nd, 9%, 2	1st, 68%, 3	1st, 73%, 4	2 nonzero weights, (**0.75**, **0.25**)
Rubis-bug	2nd, 25%, 3	1st, 14%, 7	1st, 19%, 3	1st, 21%, 6	2 nonzero weights, (**0.61**, 0.39)
Rubis-jndi	Not found	1st, 43%, 2	1st, 21%, 2	1st, 48%, 5	1 nonzero weight, **1**
OLTP-single	Not found	2nd, 10%, 2	1st, 33%, 4	2nd, 26%, 5	4 nonzero weights, (**0.3**, 0.34, 0.11, 0.25)
OLTP-multi,CPU	3rd, 10%, 15	2nd, 25%, 4	2nd, 19%, 4	2nd, 26%, 5	3 nonzero weights, (**0.49**, 0.43, 0.08)
OLTP-multi,Disk	1st, 20%, 14	1st, 69%, 4	1st, 70%, 4	2nd, 74%, 4	3 nonzero weights, (**0.55**, 0.24, 0.19)

Table 3.4: Comparing diagnosis accuracy. Numbers like 1st and 2nd indicate the rank of the cluster C whose diagnosis vector contains the relevant attributes in decreasing order of cluster size (smaller rank is better). The % value is the size of C wrt the number of historical points $|H|$. The third integer value indicates the number of diagnosis vectors returned after filtering with a support threshold of 2%

of $\langle \vec{w}, C \rangle$ pairs in the result after filtering with a support threshold of 2%. The following trends are clear in Table 3.4.

- PCM-C and PCM-E give the best accuracy in all cases.
- MAC gives good accuracy in most cases. Recall from Section 3.5.3 that MAC uses only a subset of the full historical data since we had to scale down the datasets to get MAC to run in reasonable time.
- LAC-S gives poor accuracy in many cases.

As shown in Table 3.4, the diagnosis accuracy of LAC-S is poor for the Dolphin dataset when we use $k = 19$ for which the silhouette cluster validity index is maximized. We tried LAC on this dataset with different values of k ranging from 2 to

30. In most cases, the relevant attribute—the attribute measuring available swap space, since the failure in Dolphin is because of swap space exhaustion—was not part of the diagnosis result. In the few cases where the relevant attribute was part of the diagnosis result, it appears in some lowly-ranked cluster (as for $k = 19$ in Table 3.4) and/or as one among many attributes with nonzero weight in a diagnosis vector. Furthermore, as we increase k, LAC reports more and more clusters in the result, each cluster with its own diagnosis vector; so we can't be confident about any of the output clusters or diagnosis vectors.

On the other hand, note from Table 3.4 that both PCM-C and PCM-E report the relevant attribute (as one of two attributes) in the diagnosis vector of the top-ranked cluster, which contains close to 40% of the total historical data. This result illustrates the power of PCM's anomaly-based clustering. The Dolphin data contains many patterns because of the general effects of software aging, causing LAC's query-unaware clustering to generate clusters that are not relevant to the query-specified failure points.

Conciseness of PCM-E's Diagnosis Vectors: Diagnosis vectors from PCM-E usually have very few attributes of nonzero weight, even for high-dimensional datasets. This property makes it easy to interpret results. The last column of Table 3.4 gives the diagnosis vector of the relevant cluster produced by PCM-E. For example, PCM-E's diagnosis vector for Rubis-bug contains only 2 attributes (out of 110) with nonzero weights: one with weight 0.39, and the other with weight 0.61. (Sum of absolute weights is 1.) The latter attribute pinpoints the buggy Java bean. PCM-E's diagnosis vector for Dolphin contains only 2 attributes (out of 41) with nonzero weights: one with weight 0.77 and the other with weight 0.23; both pinpoint the swap space exhaustion problem causing the crash.

3.5.4 Details of Diagnosis Results for Phase II

In this section we give the details of the results summarized in Table 3.4. The results are given per dataset.

Dolphin

The most relevant attribute for the Dolphin dataset measures available swap space (usedSwapSpace), since the failure in Dolphin is because of swap space exhaustion. Also relevant are attributes that are related to the temporary space available, e.g., tmpDirUsed and tmpDirAvail.

LAC-S: There are 14 clusters with a support of 2%. The diagnosis vector of the top-ranked cluster has weight 0.1 for tmpDirUsed, and three other attributes. This cluster has a margin of 0.38. The number of tuples in this cluster is 667, of which 100% comes from points that are not categorized as failures. The 8th-ranked cluster has weight 0.19 for userSwapSpace, and seven other attributes. This cluster has a margin of 0.1. The number of tuples in this cluster is 260, of which 98% comes from points that are not categorized as failures.

MAC: There are 3 clusters with a support of 2%. The relevant attributes are not part of the diagnosis vectors of these three clusters.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.74 for tmpDirUsed and weight -0.26 for usedSwapSpace. This cluster has a margin of 0.39. The number of tuples in this cluster is 1923, of which 96.5% comes from points that are not categorized as failures. There are 8 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.77 for tmpDirAvail and and weight -0.23 for usedSwapSpace. This cluster has a margin of 0.39. The number of tuples in this cluster is 1790, of which 96.2% comes from points that are not categorized as failures. There are 7 clusters with 2% support.

ECE

The failure is because of exhaustion of free memory. The relevant attribute in the data is realMemFree.

LAC-S: There are 2 clusters with a support of 2%. The relevant attribute is not part of the diagnosis vectors of these clusters.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight -0.99 for realMemFree. (realMemFree is also part of the top-ranked cluster, but with a very small weight.) This cluster has a margin of 0.48. The number of tuples in this cluster is 94, of which 89.4% comes from points that are not categorized as failures. There are 2 clusters with 2% support.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.98 for realMemFree. This cluster has a margin of 0.47. The number of tuples in this cluster is 3200, of which 85.1% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.74 for realMemFree. This cluster has a margin of 0.27. The number of tuples in this cluster is 3450, of which 84.7% comes from points that are not categorized as failures. There are 4 clusters with 2% support.

Rubis-bug

The relevant attributes measure the number of invocations per minute for different procedures of the *BuyNow* Java module in the application server.

LAC-S: The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the getItemId procedure in the *BuyNow* Java module. This cluster has a margin of 0. The number of tuples in this cluster is 363, of which 50.1% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

MAC: The top-ranked cluster has a diagnosis vector with weight 1 for the invocation of the getItemId procedure in the *BuyNow* Java module. This cluster has a margin of 0.1. The number of tuples in this cluster is 83, of which 98.8% comes from points that are not categorized as failures. The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the getNextBuyNowID procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This cluster has a margin of 0.11. The number of tuples in this cluster is 75, of which 100% comes from points that are not categorized as failures. There are 7 clusters with 2% support.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.26 for the invocation of the getItemId procedure in the *BuyNow* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 173, of which 98% comes from points that are not categorized as failures. The 3rd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the getNextBuyNowID procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This

cluster has a margin of 0.06. The number of tuples in this cluster is 48, of which 98% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.21 for the invocation of the getItemId procedure in the *BuyNow* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 186, of which 97.8% comes from points that are not categorized as failures. The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the getNextBuyNowID procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This cluster has a margin of 0.06. The number of tuples in this cluster is 121, of which 99% comes from points that are not categorized as failures. There are 6 clusters with 2% support.

Rubis-jndi

The relevant attributes measure the number of invocations per minute for different procedures of the *SearchItemsByRegion* Java module in the application server.

LAC-S: There are 2 clusters with a support of 2%. The relevant attributes are not part of the diagnosis vectors of these clusters.

MAC: The top-ranked cluster has a diagnosis vector with weight 0.73 for the invocation of the create procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.11. The number of tuples in this cluster is 257, of which 96.1% comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-C: The top-ranked cluster has a diagnosis vector with weight 0.92 for the invocation of the create procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.09. The number of tuples in this cluster is 316, of which 99.4% comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weight 1 for the invocation of the create procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 717, of which 95.7% comes from points that are not categorized as failures. There are 5 clusters with a support of 2%.

OLTP-single

The relevant attribute measures CPU utilization on the OLTP server.

LAC-S: There are 2 clusters with a support of 2%. The relevant attribute is not part of the diagnosis vectors of these clusters.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight 1 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.76. The number of tuples in this cluster is 103, of which 0% comes from points that are not categorized as failures.

PCM-C: The top-ranked cluster has a diagnosis vector with weight 0.48 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.61. The number of tuples in this cluster is 1222, of which 88.9% comes from points that are not categorized as failures. There are 4 clusters with a

support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weight 0.3 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.01. The number of tuples in this cluster is 1162, of which 78.66% comes from points that are not categorized as failures. There are 5 clusters with a support of 2%.

OLTP-multi (CPU-based Failure)

The relevant attribute measures CPU utilization on the OLTP server.

LAC-S: The 3rd-ranked cluster has a diagnosis vector with weight 0.47 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.34. The number of tuples in this cluster is 71, of which 28.2% comes from points that are not categorized as failures. There are 15 clusters with a support of 2%.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight 0.58 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.33. The number of tuples in this cluster is 176, of which 42.6% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-C: The 2nd-ranked cluster has a diagnosis vector with weight 0.47 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.34. The number of tuples in this cluster is 130, of which 42.3% comes from points that are not categorized as failures. There are 4 clusters with a

support of 2%.

PCM-E: The 2nd-ranked cluster has a diagnosis vector with weight 0.57 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.33. The number of tuples in this cluster is 179, of which 39.1% comes from points that are not categorized as failures. There are 5 clusters with a support of 2%.

OLTP-multi (Disk-based Failure)

The relevant attributes measure disk-usage (e.g., number of bytes or disk blocks read or written) on the OLTP server.

LAC-S: The top-ranked cluster has a diagnosis vector with weight -1 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.81. The number of tuples in this cluster is 137, of which 64.96% comes from points that are not categorized as failures. There are 14 clusters with a support of 2%.

MAC: The top-ranked cluster has a diagnosis vector with weight -1 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.8. The number of tuples in this cluster is 482, of which 44.8% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.74 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.29. The number of tuples in this cluster is 485, of which 45.6%

comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weights -0.47, -0.23, and 0.24 for the attributes measuring disk blocks read per second, disk blocks written per second, and number of OS-level writes per second respectively on the OLTP server. This cluster has a margin of 0.04. The number of tuples in this cluster is 513, of which 41.3% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

3.5.5 Experimental Comparison with Previous Work

We used the Dolphin dataset in Table 3.2 to compare our algorithms for diagnosis query processing with the following algorithms from previous work:

- The *metric-attribution approach* from [25] first learns a Tree-augmented Bayesian network classifier [88] using $H \cup F$, and then infers which attributes in $A_1, \ldots, A_n$ correlate highly with the attribute that tracks failures in the data (e.g., `failures` in Example 3.2.1). The highly-correlated attributes are output as the single diagnosis vector in the query result. Note that this technique does not partition the data into one or more clusters; the entire data is considered to be a single cluster. We did not consider the extended approach in [26]—which first partitions $H \cup F$ into windows of fixed size W, and then applies metric attribution per window—because there is no principled approach for choosing the window size W.

- The CART-based diagnosis approach [24] learns a decision tree T from $H \cup F$, and then outputs the attributes used in the top-level decision nodes in T

Diagnosis approach	Causative attributes found
Correct result	Primary attribute: usedSwapSpace attribute (see Section 3.5.3); Secondary attributes: tmpDirAvail, tmpDirSize, or tmpDirUsed
PCM-E	usedSwapSpace ($w_1 = -0.23$), tmpDirAvail ($w_2 = -0.77$) in first cluster with 37% points
Metric attribution	None of the primary or secondary attributes in top-5 attributes in diagnosis vector
CART-based diagnosis	None of the primary or secondary attributes in top-5 attributes in diagnosis vector
KDE-based diagnosis	Attribute usedSwapSpace has weight = 1.0 and is one among five attributes in the diagnosis vector with weight = 1.0, and one among 20 attributes with weight > 0.99
Gaussian-based diagnosis	None of the primary or secondary attributes in top-5 attributes in diagnosis vector

Table 3.5: Comparison with previous approaches

as part of the single diagnosis vector. Partitioning H into clusters is not considered in [24].

- The baselining-based approach described in [11] first captures the distribution D_{A_i} of each attribute A_i, $1 \leq i \leq n$, using the historical data H. The distribution can be approximated by a Gaussian distribution or using *Kernel Density Estimation (KDE)* [62]. Then, [11] computes the probability of D_{A_i} having produced the measurements of A_i in F. If this probability is low, then A_i is included in the output diagnosis vector with a weight equal to the inverse of this probability.

The experimental results on the Dolphin dataset are summarized in Table 3.5. For the metric attribution, CART-based diagnosis, and baselining-based approaches, we include the five most relevant metrics from the output diagnosis vector. Note that the results show a clear advantage of our PCM-E algorithm over the previous

approaches in terms of diagnosis accuracy.

3.6 Related Work

Fa's diagnosis query processing differs from all previous work on automated diagnosis in three significant ways: (i) integration of Phase I (diagnosing recurrent failures) and Phase II (diagnosing failures not seen previously); (ii) considering robustness of diagnosis to errors in monitoring data; and (iii) providing reliable confidence estimates.

Diagnosis Phase I: Automated diagnosis of recurrent problems has been considered in previous work, e.g., [14, 26, 92, 95]. Reference [95] builds a multi-class classifier on system event traces to classify previously-solved problems. We have empirically shown the advantages of signature databases over multi-class classifiers, especially in terms of robustness. Reference [92] gives signature-generation techniques which assume that symptoms of each failure have been identified in the monitoring data; which is impractical in the settings we consider where only raw monitoring data is available. Reference [14] considers a very different type of monitoring data—function call stacks from system failures—and gives stack matching and indexing algorithms. Reference [26] extracts indexable signatures from failure data by finding metrics that differentiate a failure state from the healthy states. Instead, Fa captures the difference between one or more failure states from other failure states using annotation information which is ignored in [26].

Fa's signature database generation resembles solving a multi-class classification problem using an ensemble of binary classifiers (e.g., see [5]). However, our focus on robustness to errors and reliable confidence estimates, and our new weighting algorithm in Section 3.3.3, differentiate Fa's Phase I from previous work by the

machine-learning community.

Diagnosis Phase II: Unlike anomaly-based clustering, previous work on Phase II of diagnosis predominantly takes one of the *correlation-based* (e.g., [24, 25]) or *baselining-based* approaches (e.g., [11]). Reference [24] applies decision-tree learning techniques to rank different system components based on their correlation with system failures. Reference [25] applies Bayesian-network learning techniques to correlate performance metrics with high-level system behavior. Reference [11] proposes a heuristic to represent the baseline behavior of a Web service; and applies anomaly detection techniques to categorize deviation from the baseline behavior. We have compared PCM-E empirically on real monitoring datasets with the algorithms in [11, 24, 25]. PCM-E performed the best due to the noisy and dynamic nature of the monitoring data.

Failure diagnosis in database systems: Oracle's recent ADDM tool [31] shows the growing interest among database vendors on automated diagnosis of database failures. Fa differs from ADDM in two ways: (i) Fa targets a broader class of systems (e.g., multitier services); and (ii) ADDM relies on a static knowledge base gathered by Oracle experts over the years, while Fa automates the process of generating the signature database from monitoring data. ADDM and Fa can complement each other.

3.7 Summary

We showed how Fa's new contributions address the five challenges from Section 3.1 for automated processing of diagnosis queries:

- *Noisy data:* Our empirical evaluation (Sections 3.5.2 and 3.5.3) showed how Fa maintains high diagnosis accuracy in the presence of errors in the monitoring data.

- *High dimensionality:* Fa can pinpoint the 1-2 attributes related to a failure even in the presence of 50-100 attributes (Section 3.5.3).

- *Dynamic systems:* Both Fa's signature database generation and anomaly-based clustering can deal with multiple healthy and failure system states and rapid state transitions.

- *Reuse:* Fa's techniques for Phase I increase reuse of previous diagnosis results by enabling high accuracy while minimizing invocations of Phase II (Section 3.5.2).

- *Trust:* Our empirical evaluation showed how Fa's confidence estimates and diagnosis vectors are very reliable.

Chapter 4

Guided Diagnosis Through Active Learning

4.1 Motivation

In Chapter 3, we described how Fa processes diagnosis queries in two phases:

(i) Phase I — constructing and using problem signatures to identify recurrent problems and reusing past diagnosis results if the match with the signature database has high confidence.

(ii) Phase II — constructing system baselines through anomaly-based clustering and characterizing the deviation of the failure data from the baselines to pinpoint the root cause.

While both Phase I and Phase II have their pros and cons, Phase II has a harder task to solve than Phase I. Consequently, diagnosis results from Phase II tend to be less accurate than those from Phase I. In machine-learning terminology, Phase I is a *supervised learning* task and Phase II is an *unsupervised learning* task [88]. In supervised learning, the training data consists of pairs of input objects (L in Fa) and the corresponding outputs (annotations in Fa). The goal is to create a function (e.g., a classifier like a decision tree) that can predict the output for any

legal input. However, in unsupervised learning, the training data contains input objects only—the outputs are unknown—and the goal is to learn a model that fits the input (e.g., a clustering of the input).

Recall that our monitoring datasets contain 100-300 attributes. Supervised learning often has an accuracy advantage over unsupervised learning on such high-dimensional datasets. Reference [27] empirically evaluates techniques for Phase I and Phase II and comes up with similar observations. For example, Phase II is shown to be prone to misdiagnosis under multiple correlated failures.

A New Background Phase III: The above observations about Phases I and II motivated us to consider a new phase in Fa where data is moved proactively from U to L; with the goal of increasing the accuracy and coverage of Phase I with the least manual effort. Figure 3.2(c) illustrates this step. We can move a failure instance t from U to L after annotating t with the cause of the failure that it represents. To get the correct annotation for t, we can leverage the manual diagnosis efforts of system administrators.

Phase III is implemented using a new algorithm, called *Falcon*[1], that can select some instances u from U, and pose an *annotation query* to the system administrator of the form: What are the annotations for the instances in u? To answer this annotation query, the system administrator will have to actually diagnose the cause of the failure represented by u. She can take the help of Fa's Phase II for this purpose. If the system administrator is able to respond back with the annotations for u, then Falcon will remove u from U, and add u and its annotations to L. Otherwise, these instances are left in U. Falcon then iterates by selecting a new set of instances from U, and posing a new annotation query to the system administrator.

[1]Fa's Active Learning with Clustering Online

Phase III can be run continuously in the background as the system executes, or it can be invoked on demand by the system administrator—e.g., when she has the time to do more diagnosis, or when she feels that the classifier trained from the current L needs to be improved. As new annotated instances appear in L, the classifier used by Phase I can be retrained on the new L to potentially improve its accuracy and coverage. The main challenge we need to address in Phase III is to design the sequence of annotation queries posed to the system administrator. Since diagnosis is expensive and time-consuming, Fa's goal is to make the best use of manual diagnosis efforts while maximizing the information gained from the newly diagnosed instances.

In Section 4.2, we present guidelines that algorithms for Phase III should meet. We then give an overview of how Falcon adheres to these guidelines. Sections 4.3 and 4.4 will describe the components of Falcon.

4.2 Overview

Recall that Fa's Phase III poses a sequence of annotation queries to the system administrator. For efficiency and ease of use, we require this sequence to adhere to three guidelines G_1, G_2, and G_3 that we discuss next.

Guideline G_1: Each individual annotation query posed to a system administrator should contain *multiple* instances belonging to a *single* type of failure. The intuition behind this guideline is that it will be hard for a system administrator to diagnose the cause of a failure from a single instance of monitoring data. Multiple distinct instances per failure make it easier to spot patterns both manually as well as when Fa's Phase II is used [10]. It is even more important to ensure (as much as possible)

that the instances in an annotation query correspond to the same type of failure. A query that mixes instances from an assorted set of failures will easily confuse system administrators. The consequences can be higher cost and labor for diagnosis, higher chances of misdiagnosis, and subsequent loss of faith in the usefulness of Phase III.

Guideline G_2: The instances selected in each annotation query should be sufficiently different from the existing annotated instances in L. Adhering to this guideline ensures that manual diagnosis efforts are not duplicated needlessly.

Guideline G_3: The instances selected in each annotation query should be representative of the failures seen in the system that are not covered by the existing annotated instances in L. Adhering to this guideline ensures that manual diagnosis efforts are spent on failures actually seen in the production system.

Our goal is to design an algorithm that adheres to all three guidelines. Guidelines G_2 and G_3 can be met using techniques for *active learning* from supervised machine learning [61]. In conventional supervised learning, a classifier C is trained on a pool of instances that are all annotated. C can then be used to predict annotations for instances with unknown annotations. Active learning is used when the training pool consists largely of unannotated instances for which getting annotations are costly. These techniques have been applied to many applications, e.g., text and image classification, speech recognition, and software testing [61].

Starting with a small set of annotated instances, an *active learner* searches the unannotated pool for instances that provide useful information in creating an accurate classifier. Once an unannotated instance t is chosen, a request is made to a human expert (in general, an oracle) to provide the correct annotation for t. The expert annotates t at some cost, and the classifier is retrained on the new set

Algorithm *Falcon* /* Fa's implementation of diagnosis Phase III */
1. Let L be the current set of annotated failure instances, and U be the current set of unannotated failure instances;
   /* Clustering step to adhere to guideline G_1 */
2. Group instances in U into a minimal set of clusters where each cluster has instances of same failure type with high probability;
   /* Use of active learning to adhere to guidelines G_2 and G_3 */
3. Use an active learner to pick one cluster from the set of clusters generated in the previous step;
4. Pick some $k \geq 1$ instances from the chosen cluster in U to pose an annotation query to the system administrator;
5. Move the annotated instances returned by the system administrator, if any, from U to L. Update the classifier trained from L;
6. Go to Step 1;

Figure 4.1: Outline of our Falcon algorithm

of annotated instances. Based on the newly gained information, the active learner searches the unannotated pool again; and the process repeats.

Notice that an active learner is exactly what we need to implement Phase III. However, conventional active learners (described in Section 4.3) adhere to guidelines G_2 and G_3, but not to G_1. The complication posed by G_1 is that the cost incurred by the system administrator to answer an annotation query is very high if the instances belong to more than one type of failure. Falcon, illustrated in Figure 4.1, addresses this issue.

Falcon proceeds in iterations where each iteration picks an annotation query—i.e., a set of unannotated instances from U—that is posed to the system administrator for annotation. Falcon adheres to guideline G_1 by ensuring that each query contains multiple instances that all correspond to the same type of failure with high probability. To meet this requirement, Falcon first groups instances in U into clusters such that instances from the same failure type go into the same cluster. Section 4.4 describes Falcon's clustering techniques.

Once the clusters are generated, Falcon uses an active learner to pick one cluster from which all instances in the current annotation query will be chosen. This step requires some modifications to conventional active learners which are designed to pick one instance from a pool of unannotated instances, rather than one cluster from a pool of clusters. In Section 4.3, we describe some conventional active learners and our extensions that enable Falcon to adhere to guidelines G_2 and G_3.

After picking a cluster, Falcon has to decide which subset of instances from this cluster to include in the annotation query posed to the system administrator. This decision is discussed in Section 4.4. The response given by the system administrator will be annotations for some subset of the queried instances. This subset could range from all queried instances to an empty set. The cost incurred by the system administrator for finding the annotations adds to the overall cost of Falcon so far. The newly annotated instances will be added to L, and the classifier that Phase I trains from L will be updated. Falcon then proceeds to design the next annotation query.

The next two sections present the details of each step of Falcon from Figure 4.1.

4.3 Active Learners

In this section, we first describe three popular active learners from the machine-learning literature. We will then describe how Falcon adapts these learners to also adhere to guideline G_1 from Section 4.2. Conventional active learners are best described by where they are positioned in the classical "exploration" Vs. "exploitation" spectrum in machine learning [61].

A popular active learner that we consider in Falcon is called the *least-confidence learner (LC)* (or uncertainty sampling) [61]. The unannotated instance $t \in U$ that

LC will pick for manual annotation next is the one on which the classifier C trained from the current L is the least confident about the true annotation. A generic way to quantify the confidence in C's prediction of t's annotation is to measure how close t is to a *decision boundary* in C. (Intuitively, each side of a decision boundary in a classifier will give a different prediction of t's annotation.) The closer t is to a decision boundary in C, the less C is confident about its prediction of t's true annotation; hence, the larger the uncertainty in t.

LC is good at exploitation—namely, acquiring annotations for instances near decision boundaries so that the boundaries can be refined—but, it does not conduct exploration where the goal is to acquire annotations for instances so as to create new decision boundaries if required. Pure exploitation will not find regions of the input space that contain many unannotated instances for which the current classifier learned from L predicts the true annotation incorrectly. Exploration searches for such regions. A popular active learner in the exploration category that we consider in Falcon is called *Kernel Furthest First*; discussed in Section 4.3.

The third type of active learner that we consider balances exploration and exploitation by defining a probability—varied suitably over time—of choosing whether to explore or exploit whenever an annotation query has to be chosen [61]. The rest of this section gives the details of how we implemented these three active learners in Fa.

4.3.1 Least-Confidence Sampling (LC)

LC first learns a classifier C from the current set of annotated failure instances L. For each instance $t \in U$, LC then uses C for two things: (i) predicting t's (unknown) annotation; and (ii) estimating the confidence in this prediction. The

details of confidence estimation are specific to the type of classifier we train from L, and works as follows for the two types of classifiers discussed in Section 4.1:

- While predicting the annotation of an instance $t \in U$, a decision tree classifier can compute the probability of t having each possible annotation from the space of all annotations (i.e., failure types). The annotation predicted for t will be the one with the highest probability. The confidence in this prediction is the difference in probabilities between the most probable annotation and the second most probable annotation.

- A signature-based classifier will predict t's annotation to be the same as that of the signature whose distance to t is minimum, i.e., t's *nearest neighbor* in the signature database. (The distance metrics we consider will be defined momentarily.) The confidence in this prediction is $d_2 - d_1$, where d_1 is t's distance to its nearest neighbor in the signature database, and d_2 is t's distance to its second nearest neighbor in the database.

Notice that both the above differences estimate the distance to a decision boundary. When LC has to pick k, $k \geq 1$, instances to pose an annotation query to the system administrator, it will pick the k instances from U whose predictions have the lowest confidence. Ties are broken randomly.

4.3.2 Kernel-Furthest-First Learner (KFF)

For each instance $t \in U$, KFF computes t's distance to its nearest neighbor in L, i.e., the instance in L that t is closest to among all instances in L. A popular metric for estimating distances is called the L_2 norm (or Euclidean distance). The L_2 norm for a pair of instances t and t' is $\sqrt{\sum_{i=1}^{n}(t.A_i - t'.A_i)^2}$, where $A_1, \ldots, A_n$ are the

data attributes in each instance. Another distance metric, which we use by default is the *cosine distance*: $1 - \cos(\theta)$. Here, θ is the angle between instances t and t' treated as vectors. $\cos(\theta) = \frac{\langle t,t' \rangle}{||t||\ ||t'||}$, i.e., the inner product of t and t' normalized by the product of their lengths. When KFF has to pick k, $k \geq 1$, instances to pose an annotation query to the system administrator, it will pick the k instances from U with the largest distance to their nearest neighbor in L. Ties are broken randomly.

4.3.3 Hybrid Learner (Hybrid)

Whenever an annotation query has to be chosen, Hybrid decides whether to do an exploration with probability p, or to do an exploitation with probability 1-p [61]. KFF is used if exploration is chosen, and LC is used if exploitation is chosen. A simple option is to use a fixed p. Reference [61] describes a better approach which we implemented as Hybrid in Falcon. Hybrid varies p dynamically such that p is high initially, and p is reduced gradually as the classifier trained from L becomes more accurate. After each exploration step, Hybrid estimates how "successful" this step was. Intuitively, if the exploration was successful, then p should be kept high; otherwise it should be reduced.

Let C_b and C_a be the classifiers trained from the set of annotated instances L before and after an exploration step. Let H_b (H_a) be the vector containing the annotations predicted by C_b (C_a) for the instances in $L \cup U$. If H_a is significantly different from H_b—which can be computed using distance metrics from Section 4.3.2—then Hybrid estimates that the exploration was successful; otherwise it reduces p. The full details of Hybrid are given in [61].

4.3.4 How Falcon Uses an Active Learner

The conventional active learners discussed so far in this section work at the level of individual instances in U. However, recall from Section 4.2 that guideline G_1 requires Falcon to work at the level of clusters in U, where each cluster contains instances of the same failure type with high probability. When LC is used as the active learner, Falcon will compute the confidence of each cluster as the average confidence across all instances in that cluster. The cluster with the least confidence is chosen for the annotation query in Step 3 of Falcon in Figure 4.1.

A similar approach is used for KFF. Here, for each cluster, Falcon will compute the average, over all instances in the cluster, of the distance to the nearest neighbor in L. The cluster with the largest average distance will be chosen for the annotation query. Note that Hybrid uses one of LC or KFF in each iteration.

4.4 Clustering in Falcon

We first considered conventional *distance-based* clustering techniques to group instances in U into clusters that contain instances of the same failure type with high probability. However, these techniques performed poorly, so we developed a new technique that we call *time-based chunking*. Both techniques are described next.

4.4.1 Distance-based Clustering

We will give a brief description of *K-means* which is one of the most commonly-used distance-based clustering techniques [88]. K-means uses an iterative refinement algorithm that starts by partitioning the input instances in U into k initial sets, either at random or using some heuristic. It then calculates the mean instance,

or *centroid*, of each set; and constructs a new partition of the instances in U by associating each instance with its closest centroid. Any of the distance metrics from Section 4.3.2 can be used for measuring distances between instances. The centroids are recalculated for the new clusters, and the algorithm is repeated by alternate application of these two steps until the instances no longer switch clusters or the centroids no longer change.

Falcon's active learner will pick one of the clusters generated by K-means, as discussed in Section 4.3.4. The annotation query posed to the system administrator will consist of the centroid of this cluster, and, if $k > 1$, a random sample of $k - 1$ instances from this cluster. As we report in Section 4.5, K-means performed poorly when used in Falcon. Recall that the monitoring datasets collected by Fa contain 100-300 attributes, i.e., these datasets are high dimensional. Distance-based clustering suffers from the *curse of dimensionality* in high dimensional spaces [32]. For any pair of instances in such spaces, it is highly likely that there are some attributes on which the instances are highly distant from each other. Thus, the clusters generated by K-means from U tend to be impure in that they contain instances from different failure types.

4.4.2 Time-based Chunking

We developed a different technique to address the problems with distance-based clustering. In the system management domain, it is reasonable to expect a strong time-based correlation among annotation values. That is, it is more likely that two failure instances that are close together in time belong to the same failure type, compared to two failure instances that are distant in time. This property can be leveraged while clustering instances in U. However, the challenge that we need to

solve is how to identify *change-points* in U—where one type of failure finishes, and another type of failure starts.

Often, there are external indicators that make it easy to detect change-points. For example, instances from two different types of failure may be separated in time by a long intermediate phase where the system was in a healthy state. While such external indicators are useful, we cannot rely solely on such indicators to identify all change-points. For example, a type of failure may be workload dependent, causing failure instances to be interleaved naturally with healthy instances. We have developed a technique to identify change-points that can leverage external indicators where available, but does not depend on them.

The basic idea behind our technique is to use patterns in the confidence estimates of instances in U that are close together in time. As discussed in Section 4.3.1, these confidence estimates are generated by first training a classifier C on the current set of annotated instances L; and then using C to predict the annotation and associated confidence for each instance $t \in U$.

Figure 4.2 illustrates the type of patterns we hope to see, namely, the values of confidence estimates for instances of the same failure that are contiguous in time are relatively close to one another (compared to the confidence estimates for instances of other failure types). The x axis in Figure 4.2 corresponds to the instances in U laid out in increasing order of timestamp. These instances are from a real experiment, and were generated in our testbed by injecting different failure types at different points of time. The top graph in the figure shows the actual change-points in the data that we generated by changing the type of failure injected.

A small random sample of the failure data was used as the current set of annotated instances L, and a decision tree classifier C was trained. The middle graph

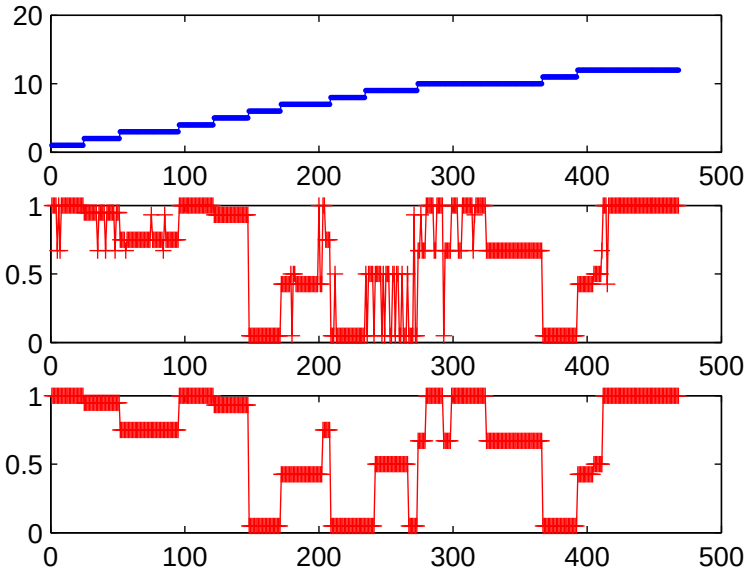

Figure 4.2: Time-based chunking

in Figure 4.2 shows the confidence estimate from C for each respective instance in the top graph. Note that for most of the failure types, the majority of confidence estimates for instances of this type tend to fall within a small range of one another. Thus, most of the actual change-points in the data can be captured by change-points in the confidence-estimate plot. This idea forms the crux of time-based chunking.

Time-based chunking can be implemented in many ways, e.g., [2, 52]. We have implemented a relatively straightforward technique that has worked satisfactorily so far. The bottom graph in Figure 4.2 shows the chunks generated by our implementation; which are reasonably accurate. We scan the instances in U in increasing order of timestamp, grouping the instances into chunks. Let N be the current chunk, and let e be the confidence estimate of the earliest instance in this chunk. The scan will close chunk N, and start a new chunk, when it finds an interval that contains many instances whose confidence estimates fall outside $[e-\delta, e+\delta]$. Here, δ

is a user-specified constant. Once Falcon's active learner picks one of the generated chunks (as discussed in Section 4.3.4), the annotation query posed to the system administrator will consist of the instance at the center of this chunk, and $k-1$ instances around it that belong to the same chunk.

4.5 Experiments

4.5.1 Testbed

Our experiments are run with monitoring data from a controlled testbed, which is built from a software framework developed by the UC Berkeley / Stanford Recovery-Oriented Computing (ROC) project [63]. The testbed runs a multitier Web service named *Rubis* [20]—an auction service modeled on eBay—running on a JBoss application server, with an embedded Web server, and a MySQL DBMS. The monitoring data primarily includes the number of procedure invocations per minute of various Java modules (*Java beans*) in the application server while the Web service is in operation. We employ a fault injection tool [15] to systematically inject faults into Rubis and the JBoss application server. The types of faults injected in our experiments include Java exceptions, message drops, deadlocks, jndi-corruptions, data corruptions, memory leaks, and infinite loops. In addition to independent faults, we also injected correlated faults, because correlated faults are common in large scale production systems [15]. The testbed is run on a machine with 1GHz CPU and 1GB memory.

Name	#Attributes	#Instances	#Distinct failures
1. Rubis-1	105	1334	9
2. Rubis-2	105	1796	14
3. Synthetic	10	528	11

Table 4.1: Datasets used in Section 4.5

4.5.2 Evaluation Methodology

Datasets: Table 4.1 summarizes the datasets used in our experiments. Rubis-1 contains independent failures and two-way correlated failures, and Rubis-2 contains independent, two-way correlated, three-way correlated, and four-way correlated failures. As we have knowledge of the failures injected, each failure instance in Rubis-1 and Rubis-2 is annotated with the actual causes, which provides ground truth for evaluating diagnosis accuracy of our techniques. To test our approach on datasets with complex patterns, we also consider a synthetic multi-class dataset (which is actually the VOWEL dataset from UCI machine-learning repository [83]).

In each dataset, 30% of the instances are used as the independent test data to compute the accuracy of the classifier trained from the current set of annotated instances in L. The rest of the instances appear in the pool of unannotated instances (U). We randomly pick 20% instances from U and get their annotations to generate the initial annotated dataset L.

We consider two experimental settings: (I) all the failures that appear in U also appear in the initial annotated data L, and (II) some types of failure that appear in U are missing from the initial annotated data L. For setting II, we assume the ratio of known failures in L is 75% of all the failures in $L \cup U$ in our experiments.

Annotation query: Each annotation query poses one *representative instance* R and $k-1$ *supporting instances* for the system administrator to diagnose; k is set to

10 in our experiments. We approximate the diagnosis effort for an annotation query as the number of distinct failures in it. The intuition is that system administrators may be distracted from diagnosing the failure represented by R if the supporting instances belong to failure types that are different from the failure type of R. In response, we assume that the system administrator annotates the representative instance R, and all supporting instances that has the same annotation as that of R. Thus, it is desirable to submit a query with k instances from the same failure state.

Algorithms and Defaults: The Falcon algorithm can be implemented with various combinations of clustering algorithms (K-means or time-based chunking) and active learners (LC, KFF, or Hybrid). Decision tree is used as the classifier, as its paths from the root to the leaf nodes help understand the classification results for diagnosis. We set 10 as a default value for the number of clusters in K-means clustering. The combination of time-based chunking and hybrid learner is our default strategy for the Falcon algorithm.

Evaluation Metrics : The accuracy metric preferred for a classifier in the system management domain is called *balanced accuracy* [65]. Suppose there are N types of annotations, and for the given annotation A_i, the classifier makes M_i' correct predictions for the M_i instances present in the test data that have annotation A_i, then the balanced accuracy is $\frac{1}{N} * \sum_i \frac{M_i'}{M_i}$. In all the plots in this section, the value of each point on the x-axis records the accumulated system administrators' diagnosis efforts for the submitted annotation queries so far, and the value on the y-axis records the balanced accuracy of the latest classifier (i.e., the classifier trained on the current L) on the test data. The maximum number of annotation queries in

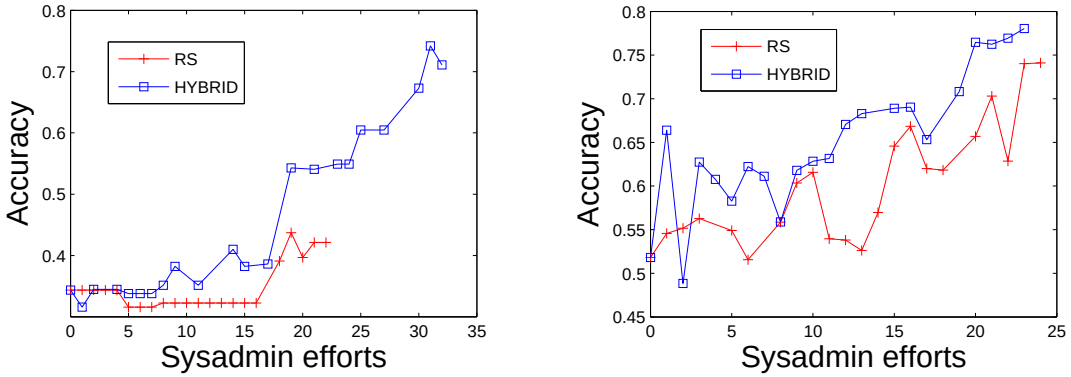

Figure 4.3: Setting I: (a) Rubis-1, (b) Rubis-2

our experiments is set to 30. A good instance selection algorithm is expected to help the classifier achieve good balanced accuracy at minimal diagnosis efforts from system administrators.

4.5.3 End-to-End Validation: Falcon Vs. Random Sampling

We compare our Falcon algorithm using the default strategy (Hybrid) with random sampling (RS) which works as follows: a representative instance is randomly picked from U, and its $k-1$ neighbors (in time) are used as supporting instances. Figure 4.3 and Figure 4.4 plots the performance of Falcon and random sampling in experimental setting I and setting II respectively.

In setting I, it is clear that Falcon requires less diagnosis efforts from system administrators (i.e., sysadmin efforts in the figures) than random sampling and achieves even better balanced accuracy. In setting II, as we do not have information for the failures not seen in L, it requires some exploration effort to get instances with failures not seen in L to improve the coverage of L. Random sampling is good at exploration, which explains why random sampling performs comparable to

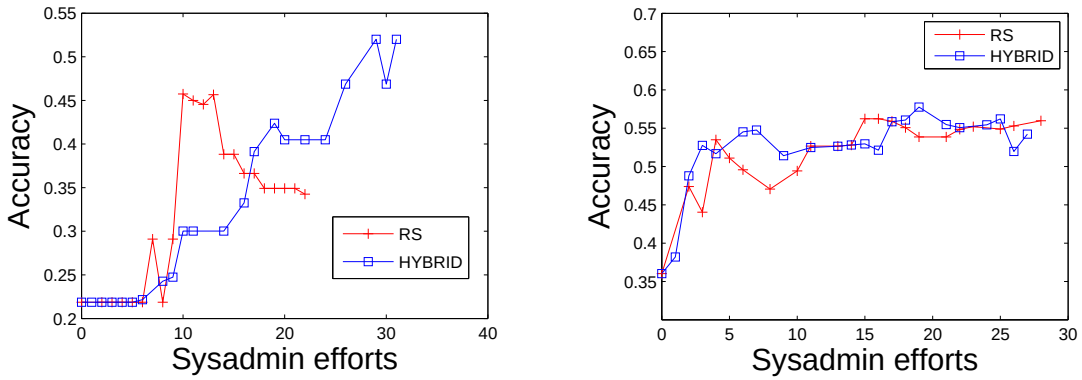

Figure 4.4: Setting II: (a) Rubis-1, (b) Rubis-2

Falcon for Rubis-1 in setting II.

In production systems, it is possible that $U \cup L$ has *unbalanced* failure instances, which means that some failures can have many more instances than other failures. To simulate this situation, we intentionally replicate the instances with some specific failures to make our datasets unbalanced. Then we compare Falcon again with random sampling in two settings. Figures 4.4 and 4.5 show the results. It is clear that our Falcon algorithm is consistently better than random sampling, because random sampling could keep picking instances with the most popular failure type, say A_i, although the information contained in the newly annotated instances of failure A_i drops significantly.

These experiments show that Falcon can perform significantly better overall that simple strategies for guiding the diagnosis efforts of system administrators.

4.5.4 Comparing Clustering Methods

In Figure 4.6 and Figure 4.7, we compare the performance of time-based chunking (CHUNK) with K-means clustering when they are combined with an active learner (LC) to select unannotated instances for diagnosis in the two experimental settings

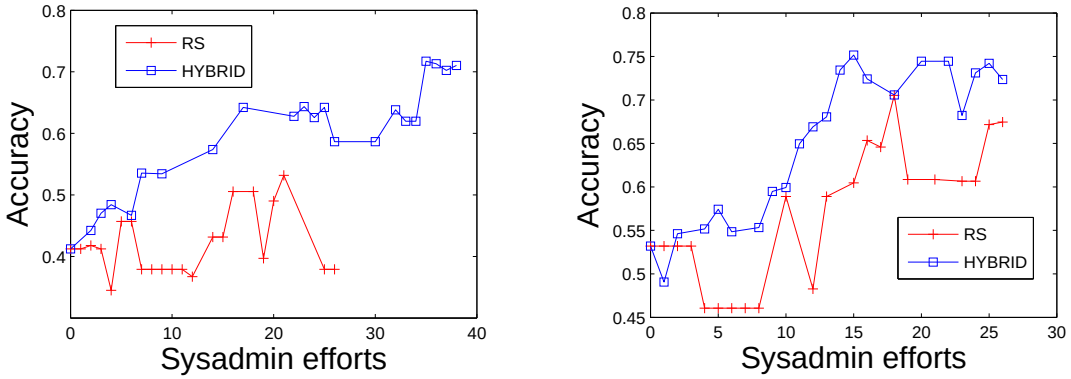

Figure 4.5: Setting II: (a) Unbalanced Rubis-1, (b) Unbalanced Rubis-2

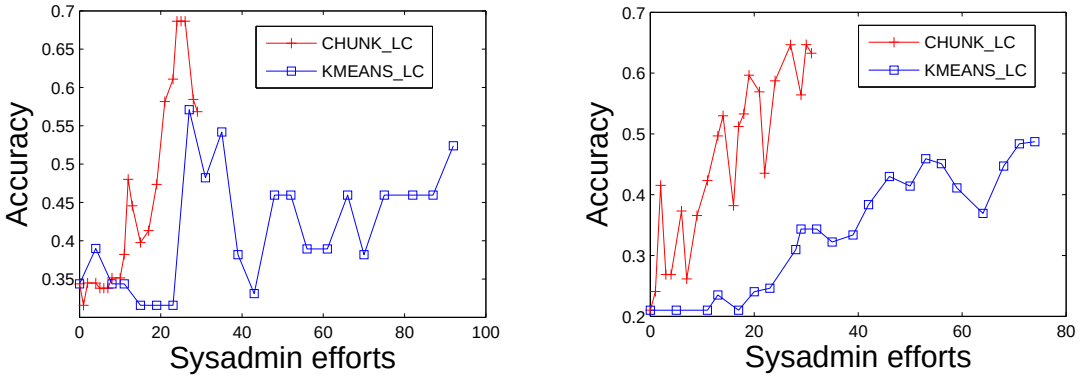

Figure 4.6: Setting I: (a) Rubis-1 (b) Synthetic

respectively. The trend is that time-based chunking is much better than K-means clustering. Note that our datasets contain 105 attributes. Distance-based clustering like K-means can perform poorly by putting instances from different failures into one cluster. Therefore, the instances selected from a cluster picked by the active learner may contain several distinct failures, which incur high diagnosis cost for each annotation query with little information added into L.

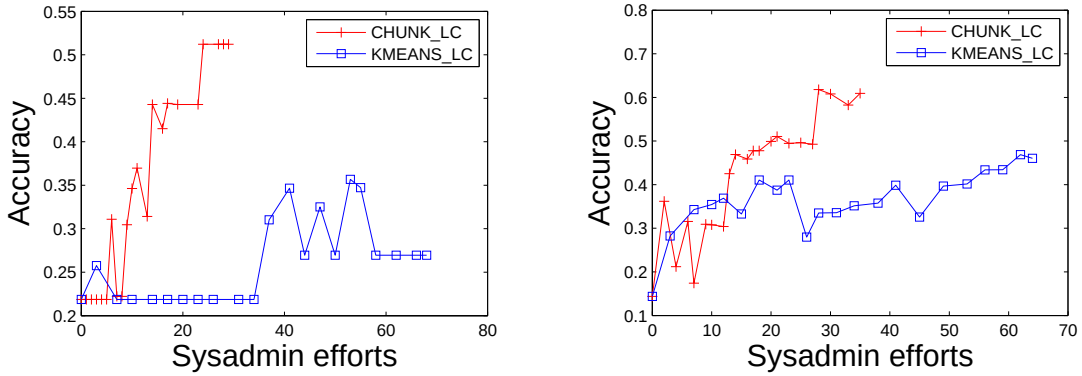

Figure 4.7: Setting II: (a) Rubis-1 (b) Synthetic

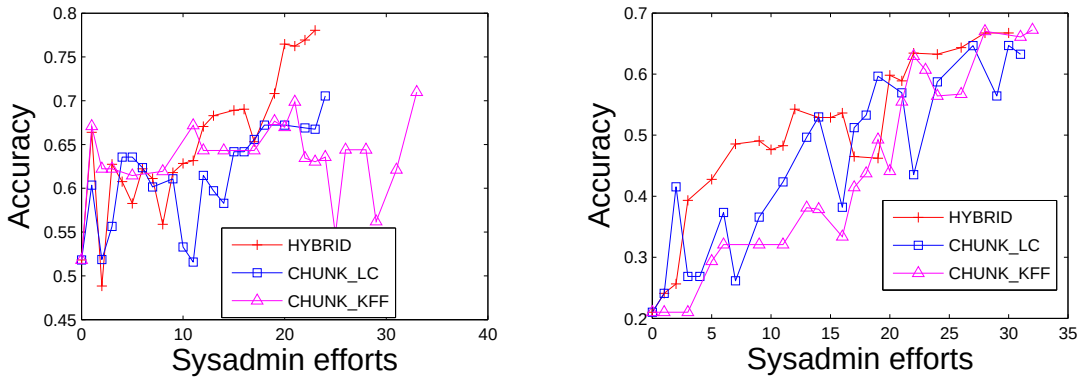

Figure 4.8: Setting I: (a) Rubis-2 (b) Synthetic

4.5.5 Comparing Active Learners

We now compare the three active learners described in Section 4.3, namely, LC, KFF, and Hybrid. Figures 4.8 and 4.9 compare these three active learners for datasets Rubis-2 and Synthetic in the two respective settings. Figure 4.10 considers Setting II for Rubis-1 in the regular and unbalanced cases. The results are mostly along expected lines: Hybrid is able to match up to the best of LC and KFF in each setting. Also note that in Figure 4.10(b), KFF performs well. This is a setting where exploration is very important.

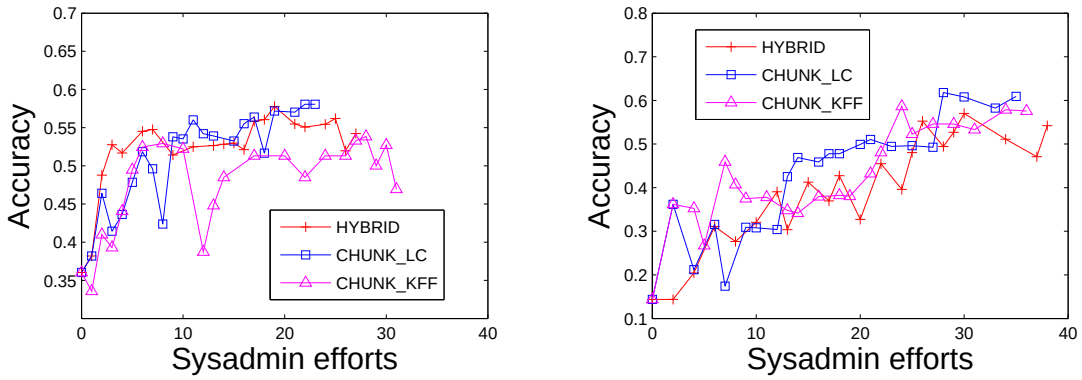

Figure 4.9: Setting II: (a) Rubis-2 (b) Synthetic

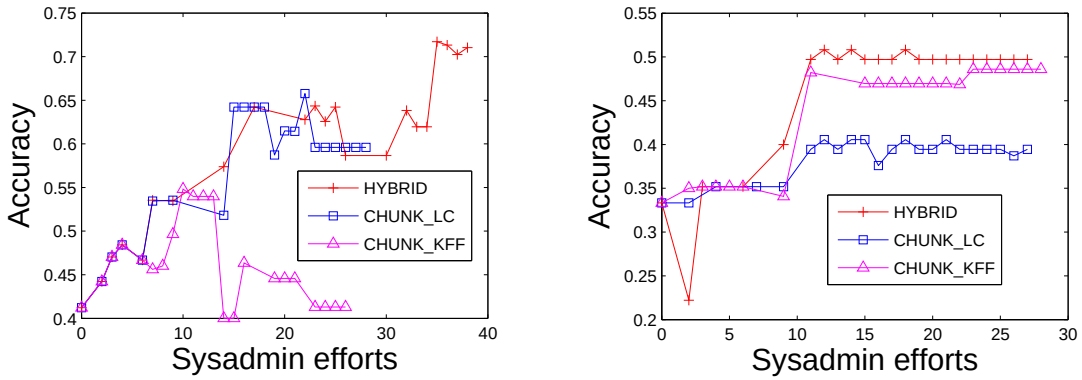

Figure 4.10: Setting II: (a) Rubis-1, (b) Unbalanced Rubis-1

4.6 Related Work

There has been plenty of previous work on wholly- or partially-automated techniques for diagnosing performance and availability problems in systems. However, previous techniques tend to focus on what we identified in Section 4.1 as Phases I and II of diagnosis. To the best of our knowledge, ours is the first work to focus on Phase III, where the goal is to make the best use of the manual diagnosis efforts of system administrators while maximizing the information gained from the newly diagnosed instances.

References [14] and [95] are recent examples of work on Phase I that build

different forms of classifiers to map current failures to previously-diagnosed failures. There has also been work on constructing signatures to characterize different system states [26, 92]. Reference [26] extracts indexable signatures from system states, which are characterized by correlations between low-level system metrics and the overall performance metric, so that a searchable database of historical system states can be created to identify recurrent problems. Reference [92] utilizes signatures to represent correlations between failures and their symptoms in networking systems.

Previous work on automated or semi-automated diagnosis based on unannotated monitoring data (i.e., Phase II) predominantly takes one of the *correlation-based* or *baselining-based* approaches. Recent examples of the correlation-based approach include [24, 25]. Reference [24] applies decision-tree learning techniques to rank different system components based on their correlation with system failures. Reference [25] applies Bayesian-network learning techniques to correlate performance metrics with high-level system behavior. Reference [11] is a recent example of the baselining-based approach where a heuristic is proposed to capture and represent the baseline behavior of a Web service; and two techniques—one based on the χ^2 statistical test, and another based on naive Bayesian networks—are proposed to detect and categorize deviation from the baseline behavior. Reference [27] empirically evaluates techniques for Phase I and Phase II, and points out many challenges that Phase II faces.

Active learning and change-point detection are two techniques that we leverage in our algorithm for Phase III. Reference [61] contains a survey of existing active learners, including detailed descriptions of the active learners that Falcon uses. References [2] and [52] are examples of recent work on change-point detection that Falcon could leverage in future.

4.7 Summary

Processing of diagnosis queries in Fa based on system monitoring data consists of multiple phases. Phase I of diagnosis uses annotated failure data L, which is monitoring data collected from failure states of the system where the cause of failure has been diagnosed and attached as annotations with the data. Previous work (including ours) has shown that this phase can be extremely valuable—because failures often reoccur—and accurate. However, this phase can be effective only if unannotated failure instances are diagnosed accurately, and moved to L in a timely fashion. Such movement requires manual diagnosis effort from system administrators. Since manual diagnosis is expensive and time-consuming, we proposed an algorithm to make the best use of manual effort while maximizing the benefit gained from newly diagnosed instances. An extensive experimental evaluation using data from a variety of failures—both single failures and multiple correlated failures—injected in a testbed, as well as with synthetic data, showed the effectiveness of our algorithm.

Chapter 5

Automated Processing of Tuning Queries

5.1 Motivation

Once a problem in the system is diagnosed, the system administrator's usual next task is to find an effective and cost-efficient strategy to resolve the problem. As there are many configuration parameters in database systems (e.g., more than 100 parameters in DB2, Oracle, and PostgreSQL), misconfiguration of these parameters is a common cause of performance problems in Web service systems [51]. Therefore, there is a strong need in Fa for automated techniques to process tuning queries that ask for settings of database configuration parameters to eliminate misconfiguration problems. This chapter first introduces the approach of *experiment-driven* query processing for system tuning and then describes algorithms for processing tuning queries in the context of configuration parameters in database systems.

Consider the following scenario from a small to medium business (SMB) enterprise. Peter, a Web-server administrator by training, maintains the Web-site of a ticket brokering company that employs eight people. Over the past few days, the Web-site has been sluggish. Peter collects system monitoring data, and tracks the problem down to poor performance of queries issued by the Web server to a backend

database.

Realizing that the database needs tuning, Peter runs the database tuning advisor. (SMBs often lack the financial resources to hire full-time database administrators, or DBAs.) Peter uses system logs to identify the workload W of queries and updates to the database. With W as input, the advisor recommends a database design (e.g., which indexes to build, which materialized views to maintain, how to partition the data). However, this recommendation does not solve the current problem: Peter has already designed the database this way based on a previous invocation of the advisor.

Peter recalls that the database has *configuration parameters*. For lack of better understanding, he had set them to default values during installation. Maybe the parameters need tuning, so Peter pulls out the 1000+ page database tuning manual. He finds many dozens of configuration parameters like buffer pool sizes, number of concurrent I/O daemons, parameters to tune the query optimizer's cost model, and others. Being unfamiliar with most of these parameters, Peter has no choice but to follow the tuning guidelines given. One of the guidelines look promising: if the I/O rate is high, then increase the database buffer pool size. However, on following this advice, the database performance drops even further. (We will show an example of such behavior.) Peter is puzzled, frustrated, and undoubtedly displeased with the database vendor.

Most of us would have faced similar situations before. Tuning database configuration parameters is hard but critical: bad settings can be orders of magnitude worse in performance than good ones. Changes to some parameters cause local and incremental effects on resource usage, while others cause drastic effects like changing query plans or shifting bottlenecks from one resource to another. These effects vary

depending on hardware platforms, workload, and data properties. Groups of parameters can have nonindependent effects, e.g., the performance impact of changing one parameter may vary based on different settings of another parameter.

With Fa, Peter can pose a declarative query that asks for a good setting of the database configuration parameters. The overall technique used by Fa to process tuning queries is called *iTuned*. iTuned can provide a very different experience to Peter. When Peter submits a tuning query to Fa with the database workload W as input, iTuned starts in the background; and Peter can resume his other work. He checks back after half an hour, but iTuned has nothing to report yet. When Peter checks back thirty minutes later, iTuned shows him an intuitive visualization of the performance impact each database configuration parameter has on W. iTuned also reports a setting of parameters that is 18% better than the current one. Another hour later, iTuned has a 35% better configuration, but Peter wants more improvement. Three hours into its invocation, iTuned reports a 52% better configuration. Now, Peter asks for the configuration to be applied to the database. Within minutes, the actual database performance improves by 52%; and Peter is very happy.

To understand the technical innovations in iTuned, let us now consider a simple, but real, example. Figure 5.1 is a *response surface* that shows how the performance of a complex TPC-H query [80] in a PostgreSQL database depends on the *shared_buffers* and *effective_cache_size* parameters. *shared_buffers* is the size of PostgreSQL's main buffer pool for caching disk blocks. The value of *effective_cache_size* is used to determine the chances of an I/O hitting in the OS file cache; so its recommended setting is the size of the OS file cache. Some observations from Figure 5.1:

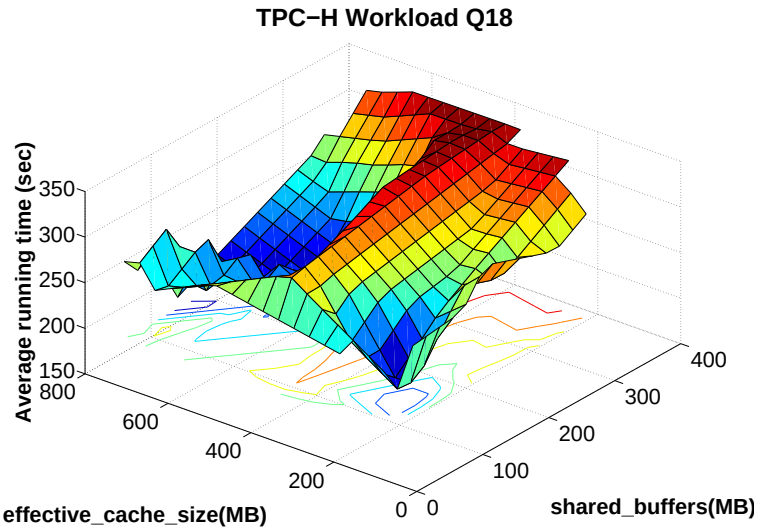

Figure 5.1: 2D projection of a response surface for TPC-H Query 18; total database size = 4GB, physical memory = 1GB

- The surface is complex and nonmonotonic.
- Performance drops sharply as *shared_buffers* is increased beyond 20% (200MB) of available memory; causing a "increase buffer pool size" rule of thumb to degrade performance.
- The effect of changing *effective_cache_size* is different for different settings of *shared_buffers*. Surprisingly, the best performance comes when both parameters are set low.

Typical database systems contain few tens of parameters whose settings can significantly impact workload performance.[1] What automated tools do users have today for holistic tuning of these parameters? Perhaps shockingly, the answer would be "very few or none".

[1]The total number of parameters may be more than a hundred, but most have reasonable defaults.

The majority of tuning tools focus on the logical or physical design of the database. For example, index tuning tools are relatively mature (e.g., [23]). These tools use the query optimizer's cost model to answer *what-if questions* of the form: how will performance change if index I were to be created? Unfortunately, such tools do not apply to parameter tuning because the settings of many high-impact parameters are not accounted for by these models.

Many tools (e.g., [75,81]) are limited to specific classes of parameters like buffer pool sizes. IBM DB2's Configuration Advisor recommends default parameter settings based on answers provided by users to some high-level questions (e.g., is the environment OLTP or OLAP?) [53]. These tools are based on predefined models of how parameter settings affect performance. Developing such models is nontrivial [86] or downright impossible because response surfaces can differ markedly across database systems (e.g., DB2 Vs. PostgreSQL), platforms (e.g., Linux Vs. Solaris; databases that are run on virtual machines), workloads, and data properties.[2] Furthermore, DB2's Configuration Advisor is helpless if the recommended defaults are still unsatisfactory.

Users are forced to rely on trial-and-error or rules-of-thumb from manuals and experts. The following tuning rule from an authoritative PostgreSQL source [66] highlights their predicament (*work_mem* is memory used by sort and hash operators):

> Adjust *work_mem* upwards for: large databases, complex queries, lots of available RAM. Adjust it downwards for: low available RAM or many concurrent users. Finding the right balance spot can be hard.

How do expert DBAs overcome these hurdles? They often run *experiments* to

[2]Section 5.7 provides empirical evidence.

perform what-if analysis during parameter tuning. A typical experiment would consist of:

- Create a replica of the production database on a test system.
- Initialize database parameters on the test system to a chosen setting. Run the workload that needs tuning, and observe the resulting performance.

iTuned takes a leaf from the book of expert DBAs. Each experiment gives a point on the response surface. Since reliable techniques for parameter tuning have to be aware of the underlying response surface, a series of carefully-planned experiments is a natural approach to parameter tuning. iTuned is not the first to advocate an experiment-driven approach for parameter tuning. [76] applied such an approach to tune four parameters in BerkeleyDB. The tuned settings were impressive, however, 37 days were spent in running experiments in parallel on five machines.

Users don't always expect instantaneous results from parameter tuning; they would rather get recommendations that work as described. (Reference [53] estimates that configuring large database systems takes on the order of 1-2 weeks.) Nevertheless, to be practical, an automated approach to parameter tuning has to produce good results within few hours. In addition, several questions need to be answered like: which experiments to run? where to run experiments? what-if the SMB does not have a test database platform?

5.2 Abstraction of Tuning Queries

Consider a database system with workload W and d parameters $X = [x_1, \ldots, x_d]$ that a user wants to tune. (The notation used throughout this chapter is summarized in Table 5.1.) The values of parameter x_i, $1 \leq i \leq d$, come from a known

Notation	Description
$X = x_1,...,x_d$	Parameters for tuning
X^t	Transpose of X
y	Performance metric of interest
$\hat{y}(X)$	Estimate of y at the setting X
$v(X)$	Variance of the estimation at X
$Y(X)$	Density distribution function of the estimate of y
$dom(x_i)$	Domain of feasible settings for x_i
$< X^{(i)}, y^{(i)} >$	Samples collected so far through experiments
$\vec{f}(X)$	A vector of basis functions
$\vec{\beta}$	A vector of regression coefficients
GPR	Gaussian process representation of response surface
$corr(X, X')$	Correlation function in GPR
$Z(X)$	Zero-mean Gaussian process in GPR
EIP(X)	Expected improvement at setting X
W	Workload under consideration

Table 5.1: Notation used in this chapter

domain $dom(x_i)$. Let DOM, where $DOM \subseteq \Pi_{i=1}^{d} dom(x_i)$, represent the space of possible settings of $x_1, \ldots, x_d$ that the database can have. Let y denote the performance metric of interest (e.g., throughput, average response time). With Fa, the user can submit a tuning query like the following:

Q: Tune(W, X, DOM, y)

The result of this query generated by Fa is a setting ($X^\star \in DOM$) that is expected to achieve the optimal performance of y for workload W.

Response Surfaces: There exists a response surface, denoted S_W, that determines the value of y for workload W for each setting of $x_1, \ldots, x_d$ in DOM. That is, $y = S_W(x_1, \ldots, x_d)$. S_W is unknown to iTuned to begin with. The core task of iTuned is to find settings of $x_1, \ldots, x_d$ in DOM that give close-to-optimal values of y. In iTuned:

- Because iTuned runs experiments, it is very flexible in how the database workload W can be specified. iTuned supports the whole spectrum from the conventional format where W is a set of queries with individual query frequencies [23], to mixes of concurrent queries at some multi-programming level, as well as real-time workload generation by an application.

- y is any performance metric of interest; for example, y in Figure 5.1 is the time to completion of the workload. In OLTP settings, y could be, for example, average transaction response time or throughput.

- Parameter x_i can be one of three types: (i) database or system configuration parameters (e.g., buffer pool size), (ii) knobs for physical resource allocation (e.g., % of CPU), or (iii) knobs for workload admission control.

Experiments and Samples: The execution plan for a tuning query in Fa consists of two interrelated phases: (i) a planning phase that plans experiments, and (ii) an execution phase that conducts experiments using some novel features proved by the *.eX framework* (see Fa's architecture in Figure 2.4). An experiment involves the following actions:

1. Set each x_i in the database to a chosen setting $v_i \in dom(x_i)$.

2. Run the database workload W.

3. Measure the performance metric $y = p$ for the run.

The above experiment is represented by the setting $\langle X \rangle = \langle x_1 = v_1,\ldots,x_d = v_d \rangle$. The outcome of this experiment is a *sample* from the response surface $y = S_W(x_1, \ldots, x_d)$. The sample in the above experiment is $\langle X, y \rangle = \langle x_1 = v_1, \ldots, x_d = v_d, y = p \rangle$.

As iTuned collects such samples through experiments, it learns more about the underlying response surface. However, experiments cost time and resources. Thus, iTuned aims to minimize the number of experiments required to find good parameter settings.

5.3 Overview of Fa's iTuned Approach to Process Tuning Queries

Gridding: *Gridding* is a straightforward technique to decide which experiments to conduct. Gridding works as follows. The domain $dom(x_i)$ of each parameter x_i is discretized into k values $l_{i1}, \ldots, l_{ik}$. (A different value of k could be used per x_i.) Thus, the space of possible experiments, $DOM \subseteq \Pi_{i=1}^{d} dom(x_i)$, is discretized into a grid of size k^d. Gridding conducts experiments at each of these k^d settings. Gridding is reasonable for a small number of parameters. This technique was used in [76] while tuning four parameters in the Berkeley DB database. However, the exponential complexity makes gridding infeasible (curse of dimensionality) as the number of parameters increase. For example, it takes 22 days to run experiments via gridding for $d = 5$ parameters, $k = 5$ distinct settings per parameter, and average run-time of 10 minutes per experiment.

SARD: The authors of [29] proposed *SARD* (Statistical Approach for Ranking Database Parameters) to address a subset of the parameter tuning problem, namely, ranking $x_1, \ldots, x_d$ in order of their effect on y. SARD decides which experiments to conduct using a technique known in Statistics as the *Plackett Burmann (PB) Design* [47]. This technique considers only two settings per parameter—giving a 2^d

grid of possible experiments—and picks a predefined $2d$ number of experiments from this grid. Typically, the two settings considered for x_i are the lowest and highest values in $dom(x_i)$. Since SARD only considers a linear number of corner points of the response surface, it can be inaccurate for surfaces where parameters have nonmonotonic effects (Figure 5.1). The corner points alone can paint a misleading picture of the shape of the full surface.[3]

Adaptive Sampling: The problem of choosing which experiments to conduct is related to the sampling problem in databases. We can consider the information about the full response surface S_W to be stored as records in a (large) table T_W with attributes $x_1, \ldots, x_d, y$. An example record $\langle x_1 = v_1, \ldots, x_d = v_d, y = p \rangle$ in T_W says that the performance at the setting $\langle x_1 = v_1, \ldots, x_d = v_d \rangle$ is p for the workload W under consideration. Experiment selection is the problem of sampling from this table. However, the difference with respect to conventional sampling is that the table T_W is never fully available. Instead, we have to pay a cost—namely, the cost of running an experiment—in order to sample a record from T_W.

The gridding and SARD approaches collect a predetermined set of samples from T_W. A major deficiency of these techniques is that they are not *feedback-driven*. That is, these techniques do not use the information in the samples collected so far in order to determine which samples to collect next. (Note that conventional random sampling in databases is also not feedback-driven.) Consequently, these techniques either bring into too many samples or too few samples to address the parameter tuning problem.

[3]The authors of SARD mentioned this problem [29]. They recommended that, before invoking SARD, the DBA should split each parameter x_i with nonmonotonic effect into distinct artificial parameters corresponding to each monotonic range of x_i. This task is nontrivial since the true surface is unknown to begin with. Ideally, the DBA, who may be a naive user, should not face this burden.

iTuned is based on a novel feedback-driven algorithm, called *Adaptive Sampling*, for experiment selection in parameter tuning. Adaptive Sampling analyzes the samples collected so far to understand how the surface looks like, and where the good settings are likely to be. Based on this analysis, more experiments are done to collect new samples that add maximum utility to the current samples.

Suppose n experiments have been run at settings $X^{(i)}$, $1 \leq i \leq n$, so far. Let the corresponding performance values observed be $y^{(i)} = y(X^{(i)})$. Thus, the samples collected so far are $\langle X^{(i)}, y^{(i)} \rangle$. Let $X^\star$ denote the best-performing setting found so far. Without loss of generality, we assume that the tuning goal is to minimize y.

$$X^\star = \arg \min_{1 \leq i \leq n} y(X^{(i)})$$

Which sample should Adaptive Sampling collect next? Suppose the next experiment is done at setting X, and the performance observed is $y(X)$. Then, the improvement *IP(X)* achieved by the new experiment X over the current best-performing setting $X^\star$ is:

$$IP(X) = \begin{cases} y(X^\star) - y(X) & \text{if } y(X) < y(X^\star) \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

Ideally, we would like to pick the next experiment X so that the improvement *IP(X)* is maximized. However, a proverbial chicken-and-egg problem arises here since the improvement depends on the value of $y(X)$ which will be known only after the experiment is done. We can only estimate $y(X)$ based on information captured in the experiments that have been done; denote the estimation of $y(X)$ as $\hat{y}(X)$. Ideally there is probability distribution about this estimation to represent

uncertainty in it. We can compute *EIP(X)*, the *expected improvement* when the next experiment is done at setting X. Then, the experiment that gives the maximum expected improvement is selected.

$$X_{next} = \arg \max_{X \in DOM} EIP(X) \tag{5.2}$$

$$EIP(X) = \int_{p=0}^{p=y(X^\star)} (y(X^\star) - p)\texttt{pdf}_{\hat{y}(X)}(p)dp \tag{5.3}$$

Here, $\texttt{pdf}_{\hat{y}(X)}(p)$ is the probability density function of the predicted performance $\hat{y}(X)$ at setting X. Recall that *DOM* is the set of all feasible parameter settings.

iTuned's Workflow: The challenge in Adaptive Sampling is to compute *EIP(X)* based on the $\langle X^{(i)}, y^{(i)} \rangle$ samples collected so far. The crux of this challenge turns out to be the generation of the probability density function of the predicted performance at X.

Figure 5.2 lists the steps involved in iTuned's Adaptive Sampling algorithm to process a tuning query. Once invoked, iTuned starts with an initialization phase where some experiments are conducted for bootstrapping. Adaptive Sampling starts with the initial set of samples, and continues to bring in new samples through experiments selected based on *EIP(X)*. Experiments are conducted in a seamless fashion in the production environment using mechanisms provided by the .eX framework.

Adaptive Sampling: Algorithm run by iTuned's Planner

1. Initialization: Conduct experiments based on Latin Hypercube Sampling, and initialize GRS and $X^{\star}$=arg $\min_i y(X^{(i)})$ with collected samples;
2. Until the stopping condition is reached, do
3. Find $X_{next} = \arg \max_{X \in DOM} EIP(X)$;
4. .eX framework conducts the next experiment at X_{next} to get a new sample;
5. Update the GRS and $X^{\star}$ with the new sample; Go to Line 2;

Figure 5.2: Steps in iTuned's Adaptive Sampling algorithm

5.4 Adaptive Sampling

5.4.1 Initialization

As the name suggests, this phase bootstraps Adaptive Sampling by bringing in samples from an initial set of experiments. A straightforward technique is random sampling which will pick the initial experiments randomly from the space of possible experiments. However, random sampling is often ineffective when only a few samples are collected from a fairly high-dimensional space. More effective sampling techniques come from the family of *space-filling designs* [68]. iTuned uses one such sampling technique, called *Latin Hypercube Sampling (LHS)* [47], for initialization.

LHS selects m experiments from a space of dimension d (i.e., parameters $x_1, \ldots, x_d$) as follows: (1) the domain $dom(x_i)$ of each parameter is partitioned into m equal subdomains; and (2) m experiments are chosen from the space such that each subdomain of any parameter has one and only one sample in it. LHS has two important advantages:

- LHS samples are very efficient to generate because of their similarity to *permutation matrices* from matrix theory. Generating m LHS samples involves generating d independent permutations of $1, \ldots, m$, and joining the permuta-

tions on a position-by-position basis.

- In general, experiments done through LHS give much better space coverage than through random sampling. LHS guarantees that the settings in the chosen experiments are spread evenly over the ranges of each parameter.

However, LHS by itself does not rule out bad spreads (e.g., all samples spread along the diagonal). iTuned addresses by problem by generating multiple sets of LHS samples, and finally choosing the one which maximizes the minimum distance between any pair of samples. That is, suppose l different sets of LHS samples $L_1, \ldots, L_l$ were generated. iTuned will select the set $L^\star$ such that:

$$L^\star = \arg\max_{1 \le i \le l} \min_{X^{(j)}, X^{(k)} \in L_i, j \neq k} \texttt{dist}(X^{(j)}, X^{(k)})$$

Here, `dist` is a common distance metric like the Euclidean distance. This technique avoids bad spreads.

5.4.2 Picking the Next Experiment

Let the samples collected so far be $\langle X^{(i)}, y^{(i)} \rangle$, $1 \le i \le n$. As discussed in Section 5.3, we need to compute the expected improvement that comes from doing the next experiment at a setting X. One approach is to derive a *regression model* [47] that can estimate $y(X)$ based on the $\langle X^{(i)}, y^{(i)} \rangle$ samples available so far. Such a regression model would have the form:

$$y(X) = \vec{f}^t(X)\vec{\beta} + \varepsilon(X) \tag{5.4}$$

Here, $\vec{f}(X) = [f_1(X), f_2(X), \ldots, f_h(X)]^t$ is a vector of basis functions, and $\vec{\beta}$ is the corresponding $h \times 1$ vector of regression coefficients. The t notation is used to

represent the matrix transpose operation. $\varepsilon(X)$, given by $\varepsilon(X) = y(X) - \vec{f}^t(X)\vec{\beta}$, is called the *residual* because it represents the difference between the true value and the value estimated via regression. The residuals are assumed to follow identical and independent normal distributions.

For example, some response surface may be represented well by the regression model: $\hat{y} = 0.1 + 3x_1 - 2x_1x_2 + x_2^2$. In this case, $\vec{f}(X) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]^t$, and $\vec{\beta} = [0.1, 3, 0, -2, 0, 1]^t$.

Problems with conventional regression models, and iTuned's solution: Conventional regression models assume that the residuals ε_i and ε_j at any pair of settings $X^{(i)}$ and $X^{(j)}$ are independent. However, the response surface of performance with respect to parameter settings is predominantly continuous. Thus, the residuals at two nearby settings tend to be correlated, violating the assumption of independent errors in the model. A related, but bigger, problem with these models is that they do not capture the probability density function $\texttt{pdf}_{\hat{y}(X)}(p)$ of the performance metric. The regression model could be inaccurate due to bias incurred by too few experiments, so it is necessary to capture uncertainty about the estimation of performance to reduce the bias of the regression model.

iTuned addresses both these problems by modeling the residual $\varepsilon(X)$ using a *Gaussian process* $Z(X)$. We first define Gaussian processes, and then describe how iTuned uses them to create the *Gaussian process Representation of a response Surface (GRS).*

Definition 5.4.1. Gaussian Process: *Let χ be a subspace of* DOM*. We say that $Z(X)$, for $X \in \chi$, is a Gaussian process provided that for any $l \geq 1$ and any choice of $X^{(1)}, \ldots, X^{(l)}$ in χ, the vector $[Z(X^{(1)}), \ldots, Z(X^{(l)})]$ has a multivariate normal distribution. $Z(X)$ is determined by its mean and covariance functions.* □

Intuitively, a Gaussian process is a stochastic process for which any finite linear combination of samples are normally distributed.

Definition 5.4.2. Gaussian process Representation of a response Surface (GRS): *A GRS represents a response surface* $\hat{y}(X)$ *as:* $\hat{y}(X) = \vec{f^t}(X)\vec{\beta} + Z(X)$. *Here, the residual in the regression is modeled by a Gaussian process* $Z(X)$ *with zero mean and covariance function* $\texttt{Cov}(Z(X^{(i)}), Z(X^{(j)})) = \alpha^2 corr(X^{(i)}, X^{(j)})$. *corr is a pairwise correlation function defined as* $corr(X^{(i)}, X^{(j)}) = \Pi_{k=1}^{d} \texttt{exp}(-\theta_k |x_k^{(i)} - x_k^{(j)}|^{\gamma_k})$. $\alpha, \theta_k \geq 0, \gamma_k > 0$, $1 \leq k \leq d$ *are constants.* $\square$

GRS's covariance function $\texttt{Cov}(Z(X^{(i)}), Z(X^{(j)}))$ represents the predominant phenomenon in response surfaces that if settings $X^{(i)}$ and $X^{(j)}$ are close to each other, then their respective residual values are correlated. As the distance between $X^{(i)}$ and $X^{(j)}$ increases, the correlation decreases. The parameter-specific constants θ_k and γ_k capture the fact that each parameter may have its own rate at which the residuals become uncorrelated. We will describe how these constants are set and give an example momentarily. GRS has the following attractive features:

- Unlike conventional regression models, GRS enables us to capture the probability density function $\texttt{pdf}_{\hat{y}(X)}(p)$ based on the samples collected through experiments conducted so far. We prove that GRS helps even further by enabling us to derive a closed form for *EIP(X)* from Equation 5.3.

- We will prove empirically using real and synthetic data that GRS is powerful enough to capture the response surfaces that arise in parameter tuning. (Gaussian processes have been used to great success on complex tasks like simulation of fire evolution and aircraft flight [68].)

- As we show momentarily, GRS enables us to naturally balance the twin tasks of *exploration* (understanding the surface) and *exploitation* (going after known high-performance regions) that arise in parameter tuning. It is nontrivial to achieve this balance, and many previous techniques [29, 76] lack it. Furthermore, GRS enables easy update as well as validation.

Lemma 5.4.3. Prediction using GRS: *Suppose a GRS is generated from n collected samples $\langle X^{(i)}, y^{(i)} \rangle$, $1 \leq i \leq n$. For any X, the GRS generates an estimate of $y(X)$ that is normally distributed with mean $u(X)$ and variance $v^2(X)$ where:*

$$u(X) = \vec{f}^t(X)\vec{\beta} + \vec{c}^t(X)\mathbf{C}^{-1}(\vec{y} - \mathbf{F}\vec{\beta}) \tag{5.5}$$

$$v^2(X) = \alpha^2[1 - \vec{c}^t(X)\mathbf{C}^{-1}\vec{c}(X)] \tag{5.6}$$

$\vec{c}(X) = [corr(X, X^{(1)}), \ldots, corr(X, X^{(n)})]^t$, $\mathbf{C}$ is an $n \times n$ matrix with element i, j equal to $corr(X^{(i)}, X^{(j)})$, $1 \leq i, j \leq n$, $\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^t$, and $\mathbf{F}$ is an $n \times h$ matrix with the ith row composed of $\vec{f}^t(X^{(i)})$.$\square$

Proof: Recall that the joint distribution of $y(X)$ and $Y^n = [y(X_1), y(X_2), \ldots, y(X_n)]^t$ is a $(1 + n)$-dimensional Gaussian distribution

$$\begin{pmatrix} y(X) \\ Y^n \end{pmatrix} \sim N_{1+n}\left[\begin{pmatrix} \vec{f}^t(X) \\ F \end{pmatrix}\beta, \alpha^2 \begin{pmatrix} 1 & \vec{c}^t(X) \\ \vec{c}(X) & \mathbf{C} \end{pmatrix}\right]$$

The conditional distribution of $y(X)$ given Y^n is still a Gaussian distribution with mean and variance as expressed in Equation (5.5) and (5.6) [68]:

$$(y(X)|Y^n = [y_1, y_2, \ldots y_n]^t) \sim N\left[\hat{y}(X), v^2(X)\right]$$

□

Note that $\vec{f}^t(X)\vec{\beta}$ in Equation 5.5 is simply a plug in of X into the regression model from Definition 5.4.2. The second term in Equation 5.5 is an adjustment of the prediction based on the errors (residuals) seen at the sampled settings, i.e., $y^{(i)} - \vec{f}^t(X^{(i)})\vec{\beta}$, $1 \leq i \leq n$. Intuitively, the prediction at X can be seen as a weighted sum of the values $y^{(i)}$ observed through experiments; where the weights are determined by the correlation function from Definition 5.4.2. Since the correlation function weighs nearby settings more than distant settings, the prediction at X is affected more by y values observed at the nearby settings.

Also note that the variance at X—which is the *uncertainty* in the GRS's estimate $\hat{y}(X)$ at X—depends on the distance between X and the settings $X^{(i)}$ where experiments were done to collect samples. Intuitively, if X is close to one or more settings $X^{(i)}$ where we have collected samples, then we will have more confidence in the prediction than the case where X is far away from all settings where experiments were done. Thus, GRS captures the uncertainty in estimated values in an intuitive fashion.

Lemma 5.4.3 gives us the necessary building blocks to compute the expected improvements from experiments that have not been done yet. We first give an example to illustrate the basic ideas of GRS.

Example 5.4.4. The solid (red) line near the top of Figure 5.3 is a true one-dimensional response surface. Suppose five experiments are done, and the collected samples are shown as circles in Figure 5.3. iTuned creates a GRS from these samples. The (green) line marked with "+" symbols represents the predictions $u(X)$ generated by the GRS as per Lemma 5.4.3. The two (black) dotted lines around this line denote the 95% confidence interval, namely, $[u(X) - 2v(X), u(X) + 2v(X)]$.

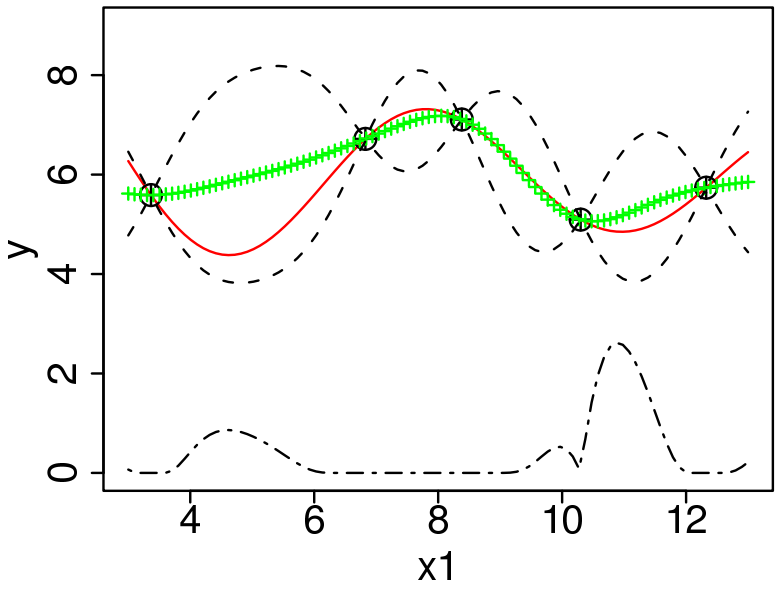

Figure 5.3: Example GRS from five samples

For example, at $x_1 = 8$, the predicted value is 7.2 with confidence interval [6.4, 7.9]. Note that, at all points, the true value (solid line) is within the confidence interval; meaning that the GRS learned from the five samples is a good approximation of the true response surface. Also, note that at points close to the collected samples, the uncertainty in prediction is low. The uncertainty increases as we move further from the collected samples. □

Recall from Lemma 5.4.3 that the estimate of $y(X)$ based on the n collected samples $\langle X^{(i)}, y^{(i)} \rangle$, $1 \leq i \leq n$, is normally distributed with mean $u(X)$ and variance $v^2(X)$. Hence it follows that the probability density function of $\hat{y}(X)$ is:

$$\texttt{pdf}_{\hat{y}(X)}(p) = \frac{1}{\sqrt{2\pi}v(X)} \exp(\frac{-(p - u(X))^2}{2v^2(X)}) \tag{5.7}$$

Theorem 5.4.5. *The expected improvement from conducting an experiment at X is:*

$$\text{EIP}(X) = \int_{p=0}^{p=y(X^\star)} (y(X^\star) - p)\texttt{pdf}_{\hat{y}(X)}(p)dp \tag{5.8}$$

EIP(X) *has the following closed form:*

$$\text{EIP}(X) = v(X)[\mu(X)\Phi(\mu(X)) + \phi(\mu(X))] \tag{5.9}$$

Here, $\mu(X) = \frac{y(X^\star)-u(X)}{v(X)}$. Φ *and* ϕ *are* $N(0,1)$ *normal cumulative distribution and density functions respectively.*

Proof: Substituting Equation 5.1 into Equation 5.3, we have

$$\begin{aligned} EIP(X) &= \int_{p=-\infty}^{p=+\infty} IP(X)\texttt{pdf}_{\hat{y}(X)}(p)dp \\ &= \int_{p=-\infty}^{p=y(X^\star)} (y(X^\star) - p)\texttt{pdf}_{\hat{y}(X)}(p)dp \\ &= \int_{p=-\infty}^{p=y(X^\star)} [y(X^\star) - \hat{y}(X) + \hat{y}(X) - p]\texttt{pdf}_{\hat{y}(X)}(p)dp \end{aligned}$$

Note that $\int_{p=-\infty}^{p=y(X^\star)} [y(X^\star) - \hat{y}(X)]\texttt{pdf}_{\hat{y}(X)}(p)dp$

$$\begin{aligned} &= [y(X^\star) - \hat{y}(X)]\Phi(\frac{y(X^\star) - \hat{y}(X)}{v(X)}) \\ &= v(X)\mu(X)\Phi(\mu(X)) \end{aligned}$$

and $\int_{p=-\infty}^{p=y(X^\star)} [\hat{y}(X) - p]\texttt{pdf}_{\hat{y}(X)}(p)dp$

$$\begin{aligned} &= -\int_{t=-\infty}^{t=\frac{y(X^\star)-\hat{y}(X)}{v(X)}} t * v(X)\phi(t)dt \quad \{\text{let } t = \frac{p - \hat{y}(X)}{v(X)}\} \\ &= v(X)\phi(\mu(X)) \end{aligned}$$

So

$$\begin{aligned} EIP(X) &= v(X)\mu(X)\Phi(\mu(X)) + v(X)\phi(\mu(X)) \\ &= v(X)[\mu(X)\Phi(\mu(X)) + \phi(\mu(X))] \end{aligned}$$

□

Therefore, the next experiment should be run at the setting

$$X_{next} = \arg \max_{X \in DOM} EIP(X)$$

Recall that *DOM* is the set of all the feasible configuration settings. Intuitively, the next experiment to run should be picked from regions where there is high uncertainty, which is expressed as $v(X)$ in (5.9), or the predicted value can improve over the current best setting, which is expressed as $\mu(X)$ in (5.9). In regions where the current GRS from the observed samples is uncertain about its estimate, i.e., where $v(X)$ is high, exploration is preferred to reduce the model uncertainty. At the same time, in regions where it is possible to achieve better performance, i.e. $\mu(X)\Phi(\mu(X)) + \phi(\mu(X))$ is high, the current GRS is used to pick samples around the current good setting $X^\star$ for exploitation. There is a tradeoff between exploration (global search) and exploitation (local search).

Example 5.4.6. The dotted line at the bottom of Figure 5.3 shows *EIP*(X) along the x_1 dimension. (All *EIP* values have been scaled by 40 to make the plot fit in this figure.) There are two peaks in the *EIP* plot. (I) *EIP* values are high around the current best sample (X^* with x_1=10.3), encouraging local search (exploitation) in this region. (II) *EIP* values are also high in the region between x_1=4 and x_1=6 because no samples have been collected near this region; the higher uncertainty mo-

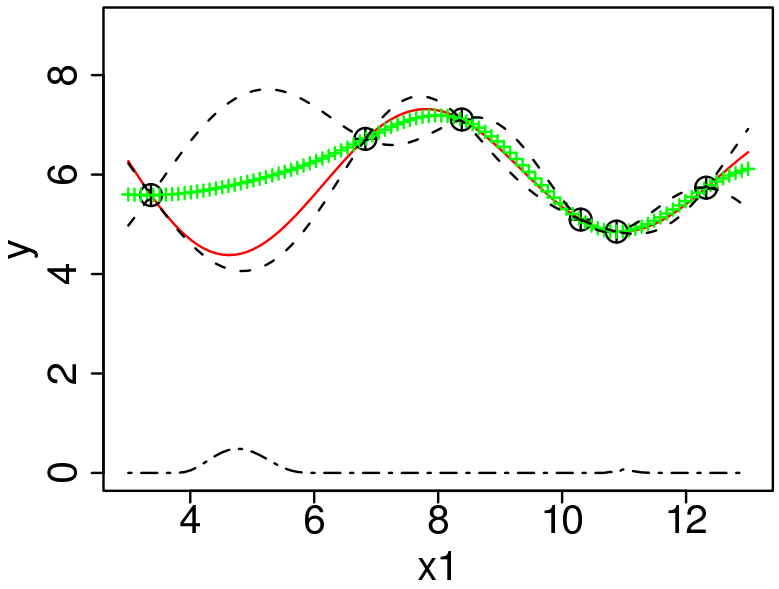

Figure 5.4: Example of EIP computation

tivates exploring this region. Adaptive Sampling with conduct the next experiment at the highest *EIP* point, namely, x_1=10.9. Figure 5.4 shows the new set of samples as well as the new *EIP*(X) after the GRS is updated with the new sample. As expected, *EIP* around x_1=10.9 has reduced. *EIP*(X) now has a maximum value at x_1=4.7 because the uncertainty in this region is still high. Adaptive Sampling will experiment here next, bringing in a sample close to the global optimum at x_1=4.4.

5.4.3 Overall Algorithm and Implementation

Figure 5.2 shows the overall structure of iTuned's Adaptive Sampling algorithm. So far we described how the initialization is done and how *EIP(X)* is derived. We now discuss how iTuned implements the other steps in Figure 5.2.

Finding the Setting that Maximizes EIP: Line 3 in Figure 5.2 requires us to find the setting $X \in DOM$ that has the maximum *EIP*. Since we have a closed form for EIP, it is efficient to evaluate EIP at a given point. In our implementation, we

pick $k = 1000$ settings (using LHS sampling) from the space of feasible settings, compute their EIP values, and pick one that has the maximum value to run the next experiment.

Initializing the GRS and Updating it with New Samples: It follows from Definition 5.4.2 that initializing the GRS with a set of $\langle X^{(i)}, y^{(i)} \rangle$ samples, or updating the GRS with a newly collected sample, involves deriving the best values of the constants α, θ_k, and γ_k, for $1 \leq k \leq d$, based on the current samples. This step can be implemented in different ways. Our current implementation uses the well-known and efficient statistical technique of *maximum likelihood estimation* [88].

When to Stop: When does Adaptive Sampling stop (Line 2 in Figure 5.2)? The easy case is when the user issues an explicit stop command once they are satisfied with the tuned performance. iTuned incorporates a novel stopping condition that can handle the harder cases, namely, when iTuned is invoked (i) in the auto-tuning mode, and (ii) by a nonexpert user.

Intuitively, Adaptive Sampling can stop when the maximum expected improvement over all settings $X \in DOM$ falls below a threshold. However, there is a possible pitfall: if the current GRS does not represent the underlying response surface reasonably well, then the expected improvement values at some settings X may differ from the actual improvement that X gives. iTuned safeguards against this problem by leveraging the properties of a GRS and the statistical testing methodology of *cross validation* [88]. The same technique can be used to find the right number of samples to collect during initialization.

5.5 A Platform for Running Online Experiments

We now consider where and when iTuned will run experiments. There are some simple answers. If parameter tuning is done before the database goes into production use, then the experiments can be done on the production platform itself. If the database is already in production use and serving real users and applications, then experiments could be done on an offline test platform. Previous work on parameter tuning (e.g., [29, 76]) assume that experiments are conducted in one of these settings.

While the two settings above—preproduction database and test database—are practical solutions, there are not sufficient because:

- The workload may change while the database is in production use, necessitating retuning.
- A test database platform may not exist (e.g., in an SMB).
- It can be nontrivial or downright infeasible to replicate the production resources, data, and workload on the test platform.

iTuned leverages a comprehensive solution, called the .eX framework, that addresses concerns like these.[4] The guiding principle behind this solution is: exploit underutilized resources in the production environment for experiments, but never harm the production workload. The two salient features of the solution are:

- **Designated resources:** An interface is proved for users to designate *which* resources can be used for running experiments. Candidate resources include

[4]This solution for running experiments was designed and implemented by Vamsidhar Thummala. It is described here for completeness.

(i) the production database (the default for running experiments), (ii) standby (failover) databases backing up the production database, (iii) test database(s) used by DBAs and developers, and (iv) staging database(s) used for end-to-end testing of changes (e.g., bug fixes) before they are applied to the production database. Resources designated for experiments are collectively called the *workbench*.

- **Policies:** A policy is specified with each resource that dictates *when* the resource can be used for experiments. The default policy associated with each of the above resources is: "if the CPU, memory, and disk utilization of the resource for its *home use* is below 10% (threshold t_1) for the past 10 minutes (threshold t_2), then the resource can be used for experiments." Home use denotes the regular (i.e., nonexperimental) use of the resource. The two thresholds are customizable. Only the default policy is implemented currently, but we are exploring other policies.

iTuned's implementation in Fa consists of a front-end module that interacts with users, and a planner module which plans experiments using Adaptive Sampling. The planned experiments are submitted to the .eX framwork which schedules these experiments on the workbench as per user-specified (or default) policies. Monitoring data needed to enforce policies is obtained through database monitoring tools.

The design of the workbench is based on splitting the functionality of each resource into two: (i) *home use*, where the resource is used directly or indirectly to support the production workload, and (ii) *garage use*, where the resource is used to run experiments. We will describe the home/garage design using the standby database as an example, and then generalize to other resources.

All database systems support one or more hot standby databases whose home

use is to keep up to date with the (primary) production database by applying redo logs shipped from the primary. If the primary fails, a standby will quickly take over as the new primary. Hence, the standby databases run the same hardware and software as the production database. It has been observed that standby databases usually have very low utilization since they only have to apply redo log records. In fact, [43] mentions that enterprises that have 99.999% (five nines) availability typically have standby databases that are idle 99.999% of the time.

Thus, the standby databases are a valuable and underutilized asset that can be used for online experiments without impacting user-facing queries. However, their home use should not be affected, i.e., the recovery time on failure should not have any noticeable increase. The .eX framework achieves this property using two *resource containers*: the home container for home use, and the garage container for running experiments. The current implementation of resource containers uses the *zones* feature in the Solaris OS [73]. CPU, memory, and disk resources can be allocated dynamically to a zone, and the OS provides isolation between resources allocated to different zones. Resource containers can also be implemented using virtual machine technology which is becoming popular [74].

The home container on the standby machine is responsible for applying the redo log records. When the standby machine is not running experiments, the home container runs on it using all available resources; the garage lies idle. The garage container is *booted*—similar to a machine booting, but much faster—only when a policy fires and allows experiments to be scheduled on the standby machine. During an experiment, both the home and the garage containers will be active, with a partitioning of resources as determined by the .eX framework. Figure 5.5 provides an illustration. For example, as per the default policy stated earlier, home and

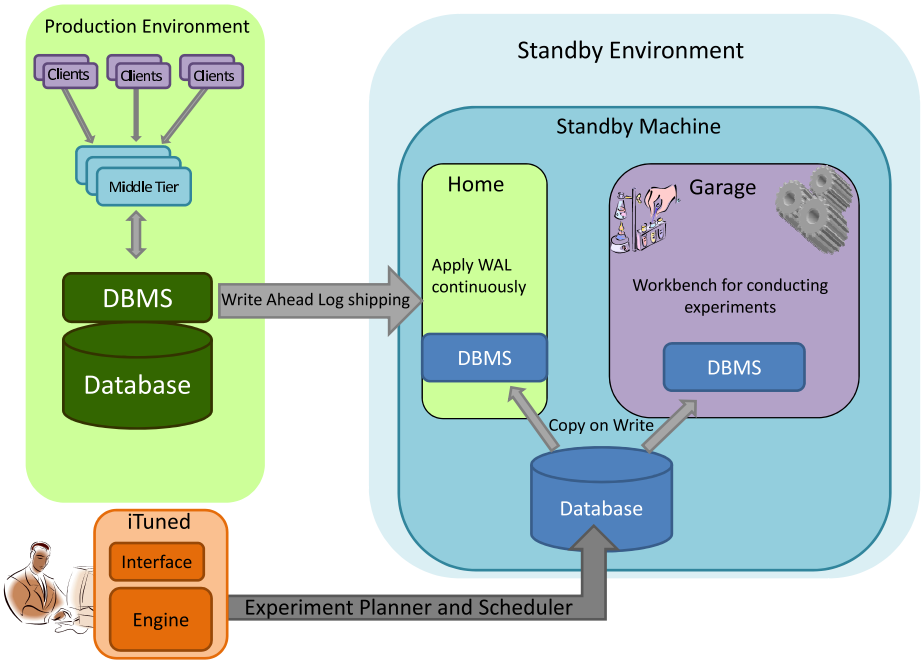

Figure 5.5: The .eX framework in action for standby databases

garage will get 10% and 90%, repectively, of the resources on the machine.

Both the home and the garage containers run a full and exactly the same copy of the database software. However, on booting, the garage is given a *snapshot* of the current data (including physical design) in the database. The garage's snapshot is logically separate from the snapshot used by the home container, but it is physically the same except for *copy-on-write* semantics. Thus, both home and garage have logically-separate copies of the data, but only a single physical copy of the data exists on the standby system when the garage boots. When either container makes an update to the data, a separate copy of the changed part is made that is visible to the updating container only (hence the term copy-on-write). The redos applied by the home container do not affect the garage's snapshot. The current implementation of snapshots and copy-on-write semantics in the .eX framework leverages the Zettabyte File System [73], and is extremely efficient (as we will show

in the empirical evaluation).

The garage is *halted* immediately under three conditions: when experiments are completed or the primary fails or there is a policy violation. All resources are then released to the home container which will continue functioning as a pure standby or take over as the primary as needed. Setting up the garage (including snapshots and resource allocation) takes less than a minute, and tear-down takes even less time. The whole process is so efficient that recovery time is not increased by more than a few seconds.

While the above description focused on the standby resource, the .eX framework applies the same home/garage design to all other resources in the workbench (including the production database). The only difference is that each resource has its own distinct type of home use which is encapsulated cleanly into the corresponding home container. *Thus, the overall approach of iTuned works even in settings where there are no standby or test databases.*

5.6 Improving iTuned's Efficiency

Experiments take time to run. This section describes features that can reduce the time iTuned takes to return good results as well as make iTuned scale to large numbers of parameters. Table 5.2 gives a short summary. The first three features are fully integrated into iTuned, workload compression is currently a simple standalone tool, and the last three features will be implemented in future.

Feature	Description and Use
Sensitivity analysis	Identify and eliminate low-effect parameters
Parallel experiments	Use multiple resources to run parallel expts
Early abort	Identify and stop low-utility expts quickly
Workload compression	Reduce per-experiment running time without reducing overall tuning quality
Semantic knowledge	Exploit *advisory parameters* in database systems
Incremental tuning	Cluster parameters to ensure independent effects across clusters; tune one cluster at a time
Interactive tuning	Get user feedback from intermediate results

Table 5.2: Features that improve iTuned's efficiency

5.6.1 Eliminating Unimportant Parameters Using Sensitivity Analysis

Suppose we have generated a GRS using n samples $\langle X^{(i)}, y^{(i)} \rangle$. Recall that, given any setting X, the GRS can produce a prediction with mean $u(X)$ and variance $v^2(X)$. Using the GRS, we can compute the expected value of y when x_1=v as:

$$E(y|x_1{=}v){=}\frac{1}{M}\int_{dom(x_2)}\cdots\int_{dom(x_d)}\hat{y}(v, x_2, \ldots, x_d)dx_2\cdots dx_d \tag{5.10}$$

where $M = |dom(x_2)| * \ldots * |dom(x_d)|$ and $|dom(x_i)|$ is the length of the domain of x_i. Intuitively, Equation 5.10 averages out the effects of all parameters other than x_1, and $E(y|x_1)$ is a function of x_1 measuring its effect on y. If we consider l equally-spaced values $v_i \in dom(x_1)$, $1 \leq i \leq l$, then we can use Equation 5.10 to compute the expected value of y at each of these l points. A plot of these values, e.g., as shown in Figure 5.3, gives a visual feel of the overall effect of parameter x_1 on y. We term such plots *effect plots*. In addition, we can consider the variance of these values, denoted $V_1 = \texttt{Var}(E(y|x_1))$. Intuitively, if V_1 is low, then y does not vary much as x_1 is changed; hence, the effect of x_1 on y is low. On the other hand,

large V_1 means that y is sensitive to x_1's setting.

Therefore, we define the *main effect* of x_1 as $\frac{V_1}{\texttt{Var}(y)}$ which represents the fraction of the overall variance in y that is explained by the variance seen in $E(y|x_1)$. The main effect of the other parameters $x_2, \ldots, x_d$ is defined in a similar fashion. Any parameter with low main effect can be set to its default value with little negative impact on performance, and need not be considered for tuning.

5.6.2 Running Multiple Experiments in Parallel

If the .eX framework can find enough resources on the workbench, then iTuned can run $k > 1$ experiments in parallel. The batch of experiments from LHS during initialization can be run in parallel. Running k experiments from Adaptive Sampling in parallel is nontrivial because of its sequential nature. A naive approach is to pick the top-k settings that maximize EIP. However, the pitfall is that these k samples may be from the same region (around the current minimum or with high uncertainty), and hence redundant.

We set two criteria for selecting k parallel experiments: (I) Each experiment should improve the current best value (in expectation); (II) The selected experiments should complement each other in improving the GRS's quality. iTuned determines the next k experiments to run in parallel as follows:

1. Select the experiment $X^{(i)}$ that maximizes the current EIP.

2. An important feature of GRS is that the uncertainty in prediction (Equation 5.6) depends only on the X values of collected samples. Thus, after $X^{(i)}$ is selected, we update the uncertainty estimate at each remaining candidate setting. (The predicted value, from Equation 5.5, at each candidate remains unchanged.)

3. We compute the new EIP values with the updated uncertainty term $v(X)$, and pick the next sample $X^{(i+1)}$ that maximizes EIP. The nice property is that $X^{(i+1)}$ will not be clustered with $X^{(i)}$: after $X^{(i)}$ is picked, the uncertainty in the region around $X^{(i)}$ will reduce, therefore EIP will decrease in that region.

4. The above steps are repeated until k experiments are selected.

5.6.3 Early Abort of Low-Utility Experiments

While the exploration aspect of Adaptive Sampling has its advantages, it can cause experiments to be run at poorly-performing settings. Such experiments take a long time to run, and contribute little towards finding good parameter settings. To address this problem, we added a feature to iTuned where an experiment at $X^{(i)}$ is aborted after $\Delta \times t_{min}$ time if the workload running time at $X^{(i)}$ is greater than $\Delta \times t_{min}$. Here, t_{min} is the workload running time at the best setting found so far. Be default, $\Delta = 2$.

5.6.4 Workload Compression

Work on physical design tuning has shown that there is a lot of redundancy in real workloads which can be exploited through workload compression to give 1-2 orders of magnitude reduction in tuning time [22]. [22] proposed an approach where the given workload is partitioned based on distinct query templates, and a representative subset is picked per partition via clustering. To demonstrate the utility of workload compression in iTuned, we came up with a modified approach. We treat a workload as a series of execution of query mixes, where a query mix is a set of queries that run concurrently. An example could be $\langle 3Q_1, 6Q_{18} \rangle$ which denotes three instances of TPC-H query Q_1 running concurrently with six instances of Q_{18}. We partition

the given workload into distinct query mixes, and pick the top-k mixes based on the overall time for which each mix ran in the workload.

5.6.5 Using Database-specific Knowledge

It is common to have database parameters whose settings affect the query execution plan chosen by the optimizer, but do not affect anything else including resource allocation and database configuration. We term such parameters *advisory parameters*. PostgreSQL's *effective_cache_size parameter* (recall Section 5.1) is an example. More common examples include parameters used as inputs to the optimizer's cost model, e.g., the cost of a sequential I/O.

Consider two settings $X^{(i)}$ and $X^{(j)}$ that differ in the settings of advisory parameters only. Despite this difference, suppose the optimizer picks the same set of execution plans for $X^{(i)}$ and $X^{(j)}$. If iTuned "knows" about advisory parameters, then it can avoid running an experiment at $X^{(j)}$ if an experiment has already been done at $X^{(i)}$ (since the same plans would run in the same environment). This optimization is important and frequently applicable because typical databases have a number of advisory parameters, most of which are high-impact because they can change execution plans.

5.6.6 Other Techniques

A number of other features could be added to iTuned. One feature is early aborting of an experiment if during the experiment we see that the eventual performance value will not be better that the best performance found so far. This optimization has to be applied with care because early abortion may not give us a sample to update the GRS.

Another approach for scalability is analyze interactions among the effects of different parameters. Recall that the main effect of parameter x_1 is defined as $\frac{V_1}{\texttt{Var}(y)}$. Similarly, an *interaction effect* between x_1 and x_2 can be defined as $\frac{\texttt{Var}(E(y|x_1,x_2))-V_1-V_2}{\texttt{Var}(y)}$, where:

$$E(y|x_1=v_1,x_2=v_2)=\int_{dom(x_3)}\cdots\int_{dom(x_d)}\hat{y}(v_1,v_2,x_3,\ldots,x_d)dx_3\cdots dx_d$$

Intuitively, the interaction effect between x_1 and x_2 is high if the effect of x_1 on y is very sensitive to x_2's setting. That is, different settings of x_2 cause different effects from x_1. We can identify important interaction effects using the above equation, and then partition the parameters in disjoint groups such that no cross-group interactions exist. iTuned could then take a *divide-and-conquer* approach to parameter tuning, i.e., tuning one group of parameters at a time, probably ranking the groups in some order.

5.7 Empirical Evaluation

Our experimental setup involves a local cluster of machines, each with four 2GHz processors and 3GB memory, running PostgreSQL 8.2 on Solaris 10. One machine runs the production database. The other machines are used as hot standbys, test platforms, or workload generators. Recall from Section 5.5 that the .eX framework used by iTuned can run experiments on the production database, standbys, and test platforms. By default, we use a standby database for experiments.

5.7.1 Methodology and Summary

We first summarize the different types of empirical evaluation conducted and the results obtained.

- Section 5.7.2 breaks down the overhead of various operations in the API provided by the .eX framework for running experiments on demand in a production setting; and shows that this framework is noninvasive and efficient.

- Section 5.7.3 shows real response surfaces that highlight the issues motivating our work, e.g., (i) why database parameter tuning is not easy for the average user; (ii) how parameter effects are highly sensitive to workloads, data properties, and resource allocations; and (iii) why optimizer cost models are insufficient for effective parameter tuning, but it is important to keep the optimizer in the tuning loop.

- Section 5.7.4 presents tuning results for OLAP and OLTP workloads of increasing complexity that show iTuned's ease of use and up to 10x improvements in performance compared to default parameter settings, rule-based tuning based on popular heuristics, and a state-of-the-art automated parameter tuning technique. We show how iTuned can leverage parallelism, early aborts, and workload compression to cut down tuning times drastically with negligible degradation in tuning quality.

- iTuned's performance is consistently good with both PostgreSQL and MySQL databases, demonstrating iTuned's portability.

- Section 5.7.5 shows how iTuned can be useful in other ways apart from recommending good parameter settings, namely, visualizing parameter impact

as well as approximate response surfaces. This information can guide further manual tuning.

The tuning tasks in our empirical evaluation consider up to 30 database configuration parameters. By default, we consider the following 11 parameters for OLAP workloads in PostgreSQL: (P1) shared_buffers, (P2) effective_cache_size, (P3) work_mem, (P4) maintenance_work_mem, (P5) default_statistics_target, (P6) random_page_cost, (P7) cpu_tuple_cost, (P8) cpu_index_tuple_cost, (P9) cpu_operator_cost, (P10) memory allocation, and (P11) CPU allocation.

5.7.2 Performance of the .eX framework for Conducting Experiments

Recall the implementation of the .eX framework from Section 5.5. Table 5.3 shows the various operations in the interface provided by the .eX framework, and the overhead of each operation. The Create Container operation is done once to set up the OS environment for a particular tuning task; so its 10-minute cost is amortized over an entire tuning session. This overhead can be cut down to 17 seconds if the required type of container has already been created for some previous tuning task. Note that all the other operations take on the order of a few seconds. For starting a new experiment, the cost is at most 48 seconds to boot the container and to create a read-write snapshot of the database (for workloads with updates). A container can be halted within 2 seconds, which adds no noticeable overhead if, say, the standby has to take over on a failure of the primary database.

Operation by the .eX framework	Time (sec)	Description
Create Container	610	Create a new garage (one time process)
Clone Container	17	Clone a garage from already existing one
Boot Container	19	Boot garage from halt state
Halt Container	2	Stop garage and release resources
Reboot Container	2	Reboot the garage (required for adding additional resources to a container)
Snapshot-R DB	7	Create read-only snapshot of database
Snapshot-RW DB	29	Create read-write snapshot of database

Table 5.3: Overheads of operations in the .eX framework

5.7.3 Why Parameter Tuning is Nontrivial

The OLAP (Business Intelligence) workloads used in our evaluation were derived from TPC-H running at scale factors (*SF*) of 1 and 10 on PostgreSQL [80]. The physical design of the databases are well tuned, with indexes approximately tripling and doubling the database sizes for SF=1 and SF=10 respectively. Statistics are always up to date. The heavyweight TPC-H queries in our setting include Q1, Q7, Q9, Q13, and Q18.

Figure 5.1 shows a 2D projection of a response surface that we generated by running Q18 on a TPC-H SF=1 database for a number of different settings of the eleven parameters from Section 5.7.1. The database size with indexes is around 4GB. The physical memory (RAM) given to the database is 1GB to create a realistic scenario where the database is 4x the amount of RAM. This complex response surface is the net effect of a number of individual effects:

- Q18 (Large Volume Customer Query) is a complex query that joins the Lineitem, Customer, and Order tables. It also has a subquery over Lineitem (which gets rewritten as a join), so Q18 accesses Lineitem—the biggest table

in TPC-H—twice.

- Different execution plans get picked for Q18 in different regions of the response surface because changes in parameter settings lead to changes in estimated plan costs. These plans differ in operators used, join order, and whether the same or different access paths are used for the two accesses to the Lineitem table.

- Operator behavior can change as we move through the surface. For example, hybrid hash joins in PostgreSQL change from one pass to two passes if the *work_mem* parameter is lower than the memory required for the hash join's build phase.

- Resource interference can happen. For example, if a hybrid hash join in PostgreSQL starts to create temporary files on disk, the accesses go through the OS file cache which competes for RAM with *shared_buffers*. Thus, increasing *shared_buffers* can degrade performance if hybrid hash joins are spilling to disk.

It took us several days of effort, more than a hundred experiments with PostgreSQL, as well as email conversations with PostgreSQL developers to understand the unexpected nature of Figure 5.1. It is unlikely that a non-expert who wants to use a database for some application—say, Peter in Section 5.1—will have the knowledge (or patience) to tune the database like we did. Surfaces like Figure 5.1 show how critical experiments are to understand which of many different effects dominate in a particular setting.

The average running time of a query can change drastically depending on whether it is running alone in the database or it is running in a concurrent mix of queries of

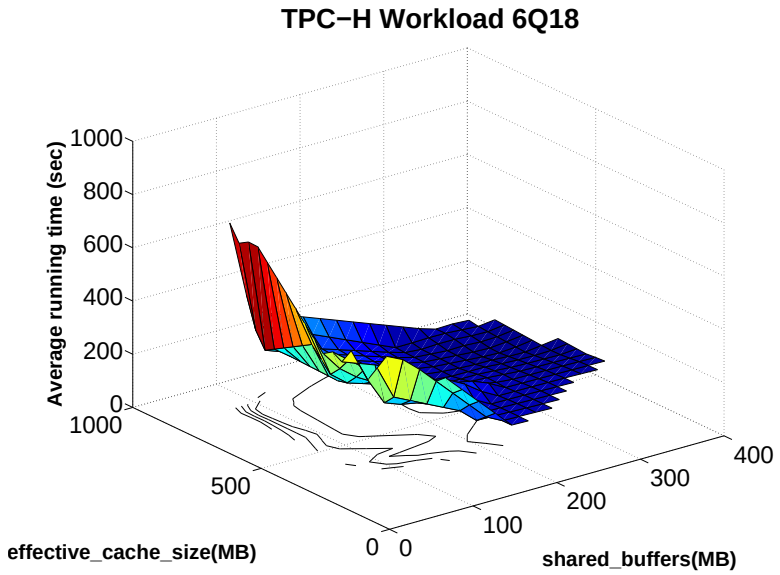

Figure 5.6: Impact of shared_buffers Vs. effective_cache_size for workload W4 (TPC-H SF=10)

the same or different types. For example, consider Q18 running alone or in a mix of six concurrent instances of Q18 (each instance has distinct parameter values). At the default parameter setting of PostgreSQL for TPC-H SF=1, we have observed the average running time of Q18 to change from 46 seconds (when running alone) to 1443 seconds (when running in the mix). For TPC-H SF=10, there was a change from 158 seconds (when running alone) to 578 seconds (when running in the mix).

Two insights come out from the results presented so far. First, query optimizers compute the cost of a plan independent of other plans running concurrently. Thus, optimizer cost models cannot capture the true performance of real workloads which consist of query mixes. Second, it is important to keep the optimizer in the loop while tuning parameter settings because the optimizer can change the plan for a query when we change parameter settings. While keeping the optimizer in the loop is accepted practice for physical design tuning (e.g., [23]), to our knowledge, we are the first to bring out its importance and enable its use in configuration parameter

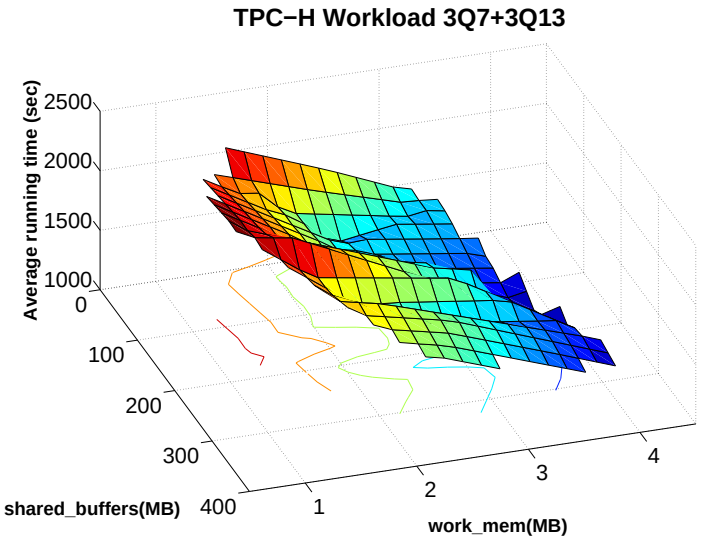

Figure 5.7: Impact of shared_buffers Vs. work_mem for workload W5 (TPC-H SF=10)

tuning.

Figure 5.6 shows a 2D projection of the response surface for Q18 when run in the 6-way mix in PostgreSQL for TPC-H SF=10. The key difference between Figures 5.1 (Q18 alone, TPC-H SF=1) and 5.6 (Q18 in 6-way mix, TPC-H SF=10) is that increasing *shared_buffers* has an overall negative effect in the former case, while the overall effect is positive in the latter. We attribute the marked effect of *shared_buffers* in Figure 5.6 to the increased cache hits across concurrent Q18 instances.

Figures 5.7 and 5.8 show the response surface for a workload where *shared_buffers* has limited impact. The highest impact parameter is *work_mem*. This workload has three instances of Q7 and 3 instances of Q13 running in a 6-way mix in PostgreSQL for TPC-H SF=10. All these results show why users can have a hard time setting database parameters, and why experiments that can bring out the underlying response surfaces are inevitable.

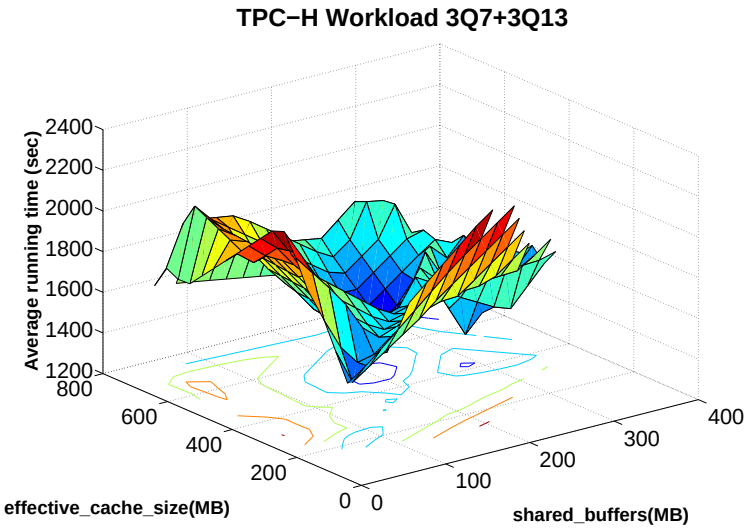

Figure 5.8: Impact of shared_buffers Vs. effective_cache_size for workload W5 (TPC-H SF=10)

5.7.4 Tuning Results

We now present an evaluation of iTuned's effectiveness on different workloads and environments. iTuned should be judged both on its *quality*—how good are the recommended parameter settings?—and *efficiency*—how soon can iTuned generate good recommendations? Our evaluation compares iTuned against:

- **Default** parameter settings that come with the database.
- **Manual rule-based** tuning based on heuristics from database administrators and performance tuning experts. We use an authoritative source for PostgreSQL tuning [66].
- **Smart Hill Climbing (SHC)** is a state-of-art automated parameter tuning technique [89]. It belongs to the hill-climbing family of optimization techniques for complex response surfaces. Like iTuned, SHC plans experiments while balancing exploration and exploitation (Section 5.4.2). But, SHC lacks

key features of iTuned like GRS representation of response surfaces, .eX framework, and efficiency-oriented features like parallelism, early aborts, sensitivity analysis, and workload compression.

- **Approximation to the optimal setting:** Since we do not know the optimal performance in any tuning scenario, we run a large number of experiments offline for each tuning task. We have done at least 100 (often, 1000+) experiments per tuning task over the course of six months. The best performance found is used as an approximation of the optimal. This technique is labeled *Brute Force*.

iTuned and SHC do 20 experiments each by default. iTuned uses the first 10 experiments for initialization. Strictly for the purposes of evaluation, by default iTuned uses only early abort among the efficiency-oriented techniques from Section 5.6.

Figure 5.9 compares the tuning quality of iTuned (I) with Default (D), manual rule-based (M), SHC (S), and Brute Force (B) on a range of TPC-H workloads at SF=1 and SF=10. The performance metric of interest is workload running time; lower is better. The workload running time for D is always shown as 100%, and the times for others are relative. To further judge tuning quality, these figures show the rank of the performance value that each technique finds. Ranks are reported with the prefix R, and are based on the range of performance values observed by Brute Force; lower rank is always better. Figures 5.9 also shows (above I's bar) the total time that iTuned took since invocation to give the recommended setting. Detailed analysis of tuning times is done later in this section.

Figure 5.9 involves 11 distinct workloads, all of which are nontrivial to tune. Workloads W1, W2, and W3 consist of individual TPC-H queries Q1, Q9, and

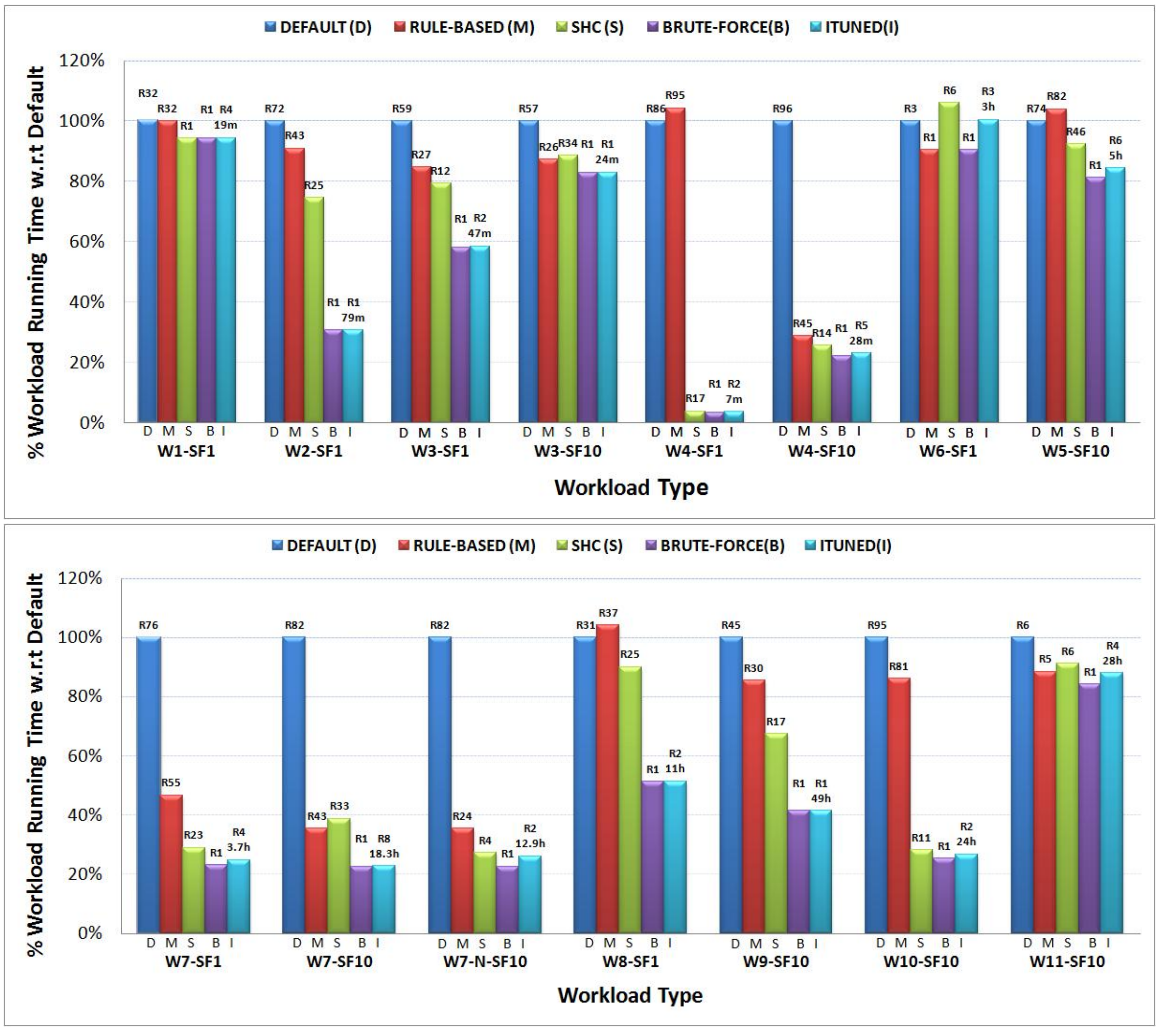

Figure 5.9: Comparison of tuning quality. iTuned's tuning times are shown in minutes (m) or hours (h). Ri denotes Rank i

Q18 respectively running at a *Multi-Programming Level (MPL)* of 1. MPL is the maximum number of concurrent queries. TPC-H queries have input parameters. Throughout our evaluation, we generate each query instance randomly using a TPC-H query generator. Different instances of the same query are distinct with high probability.

Workloads W4, W5, and W6 go one step higher in tuning complexity because they consist of mixes of concurrent queries. W4 (MPL=6) consists of six concur-

rent (and distinct) instances of Q18. W5 (MPL=6) consists of three concurrent instances of Q7 and three concurrent instances of Q13. W6 (MPL=10) consists of five concurrent instances of Q5 and five concurrent instances of Q9.

Workloads W7∼W11 in Figure 5.9 go the final step in tuning complexity by bringing in many more complex query types, much larger numbers of query instances, and different MPLs. W7 (MPL=9) contains 200 query instances comprising queries Q1 and Q18, in the ratio 1:2. W8 (MPL=24) contains 200 query instances comprising TPC-H queries Q2, Q3, Q4, and Q5, in the ratio 3:1:1:1. W9 (MPL=10), W10 (MPL=20), and W11 (MPL=5) contain 100 query instances each with 10, 10, and 15 distinct TPC-H query types respectively in equal ratios. The results for W7-N shown in Figure 5.9 are from tuning 30 parameters.

Figure 5.9 shows that the parameter settings recommended by iTuned consistently outperform the default settings, and is usually significantly better than the settings found by SHC and common tuning rules. iTuned gives 2x-5x improvement in performance in many cases. In fact, iTuned's recommendation is usually close in performance to the approximate optimal setting found (exhaustively) by Brute Force. It is interesting to note that expert tuning rules are more geared towards complex workloads (compare the M bars between the top and bottom halves of of Figure 5.9).

As an example, consider the workload W7-SF10 in Figure 5.9. The default settings give a workload running time of 1085 seconds. Settings based on tuning rules and SHC give running times of 386 and 421 seconds respectively. In comparison, iTuned's best setting after initialization gave a performance of 318 seconds, which was improved to 246 seconds by Adaptive Sampling (77% improvement over default). iTuned's sensitivity analysis found the shared_buffers parameter to have the

most impact on performance. The default setting of 32 MB for shared_buffers is poor. The rule-based setting of 200 MB is better, but iTuned found a setting close to 400 MB where the performance is far better.

Figure 5.9 shows that iTuned takes at the order of tens of hours to find good settings for complex workloads. Table 5.4 gives the absolute tuning values by executing a single instance of workload in seconds. Query mix CM1 and CM2 contains 100 query instances with 10 and 15 distinct TPC-H query types in equal ratios respectively. Reference [53] estimates that configuring large database management systems takes at the order of one to two weeks, so one to two days of time spent parameter tuning is acceptable; especially considering that iTuned gives 2x-5x improvement in performance in many cases. More importantly, Figure 5.10 shows that iTuned's tuning time can be reduced by orders of magnitude using the techniques we proposed in Section 5.6. Early Abort uses $\Delta = 2$ and workload compression picks the top mix in the workload.

For each of the complex workloads from Figure 5.9, we show iTuned's tuning time with and without different techniques. It is clear that these techniques can reduce iTuned's tuning time to at most a few hours. The drop in tuning quality across all these scenarios was never more than 1%. In general, we have found workload compression to be even more effective in parameter tuning than in physical design tuning. Intuitively, parameter settings are less sensitive to which queries get picked in the compressed workload compared to, say, index selection.

Table 5.5 gives a brief summary that shows iTuned's consistent good performance. TPC-W is an e-Commerce benchmark that simulates the activities of a retail website. Our experiments with TPC-W are based on a 48000-transaction workload on a 6GB database. RUBiS [20] is Web service benchmark that imple-

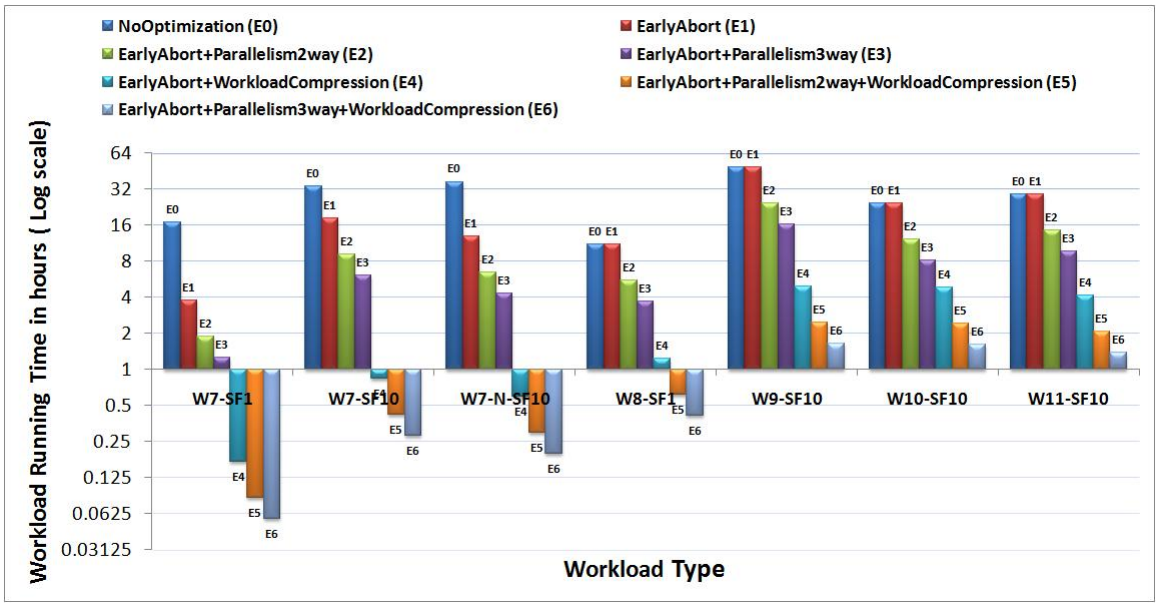

Figure 5.10: Comparison of iTuned's tuning times in the presence of various efficiency-oriented features

ments the core functionality of an auction site like eBay.

5.7.5 Sensitivity Analysis

This section evaluates two important features of iTuned: sensitivity analysis of database parameters and effects plots for visualization; see Section 5.6.1. We use both real workloads and complex synthetic response surfaces in our evaluation. We compare iTuned's performance against SARD [29] which is described in Section 5.3. Recall that, unlike iTuned, SARD is not an end-to-end tuning tool, and can be misled by nonmonotonic effects of parameters.

Our concerns about SARD were validated by a simple evaluation. We chose three popular and hard benchmark functions from the optimization literature: Griewank, Rastrigin, and Rosenbrock [89]. All three functions have a global optimum of 0. We used the functions to generate response surfaces with 20 parameters each. Of

Workload type	Query mix	Default	Rule-based	SHC	Optimal	iTuned
W1-SF1	Q1	107	107	101	101	101
W2-SF1	Q9	310	282	231	95	95
W3-SF1	Q18	315	267	250	183	184
W3-SF10	Q18	158	138	140	131	131
W4-SF1	6Q18	1443	1505	55	51	52
W4-SF10	6Q18	578	167	148	128	133
W5-SF10	3Q7, 3Q13	1167	1057	1237	1057	1173
W6-SF1	5Q1, 5Q9	907	943	838	738	765
W7-SF1	66Q1,134Q18	338	158	98	78	84
W7-SF10	66Q1,134Q18	1085	386	421	246	248
W7-N-SF10	66Q1,134Q18	1085	386	295	246	283
W8-SF1	100Q2,33Q3, 33Q4,34Q5	406	423	366	208	209
W9-SF10	CM1	1451	1240	979	601	601
W10-SF10	CM1	5910	5089	1674	1505	1583
W11-SF10	CM2	1152	1020	1052	971	1014

Table 5.4: Comparison of tuning quality in terms of workload running time after tuning (all the tuning times are shown in seconds.)

these 20 parameters, 5 are important—i.e., they impact the shape of the surface significantly—while the remaining 15 are unimportant. On the Griewank and Rastrigin surfaces—which have significant nonmonotonic behavior—SARD completely failed to identify the unimportant parameters. As iTuned did experiments progressively, it never classified any important parameter as unimportant. By the time fifty experiments were done, iTuned was able to clearly separate the five important parameters from the unimportant ones.

Tables 5.6 and 5.7 gives end-to-end tuning results for three techniques: (i) SARD+AS, where SARD is used to identify the important parameters, and then Adaptive Sampling is started with the samples collected by SARD used for initialization; (ii) SHC (does not do sensitivity analysis), and (iii) iTuned. Note that lower numbers are better in all cases. iTuned clearly outperforms the alternatives.

A very useful feature of iTuned is that it can provide intuitive visualizations of

Workload	Performance Metric	#Parameters	Quality (Rank)	Tuning time (Hours)
TPC-W (MySQL)	Response time	7	R1	3.2
TPC-W (MySQL)	Throughput	7	R4	7.6
TPC-W (PostgreSQL)	Response time	20	R23	2.5
TPC-W (PostgreSQL)	Throughput	20	R8	2.5
RUBiS (MySQL)	Response time	6	R1	6.1
RUBiS (MySQL)	Throughput	6	R2	6.6

Table 5.5: Sample of iTuned's results

Workload	Optimal	SARD+AS	SHC	iTuned
Griewank	0	28.6	28.7	2.0
Rastrigin	0	200.8	209.1	26.1
Rosenbrock	0	40.2	160.5	7.9
W2-SF1	95	240 (R29)	231 (R24)	95 (R1)
W3-SF1	11	43 (R20)	67 (R24)	12 (R4)
W6-SF1	390	450 (R63)	417 (R20)	403 (R5)
W8-SF1	208	208 (R1)	289 (R4)	208 (R1)

Table 5.6: Sensitivity analysis. For W2, W3, W6, W8, rank and performance of best setting (secs) are shown. Lower is better

its current results. Figure 5.11 shows an effect plot (recall Section 5.6.1) generated by iTuned based on 10 experiments for the workload whose surface is shown in Figures 5.7 and 5.8. Figures 5.12 and 5.13 shows the effect plot for workload W4 for SF=1 and SF=10. The parameters P1-P9 correspond to the first nine PostgreSQL parameters listed in Section 5.7.1. Without knowing the actual response surface, a user can quickly grasp the main trends in parameter impact based on the effect plot. Note how the plot mirrors the trends in Figures 5.7 and 5.8.

Workload	Optimal	SARD+AS	SHC	iTuned
Griewank	0	28.6	28.7	2.0
Rastrigin	0	200.8	209.1	26.1
Rosenbrock	0	40.2	160.5	7.9
W2-SF1	95	240 (R29)	231 (R24)	95 (R1)
W3-SF1	11	43 (R20)	67 (R24)	12 (R4)
W6-SF1	390	450 (R63)	417 (R20)	403 (R5)
W8-SF1	208	208 (R1)	289 (R4)	208 (R1)

Table 5.7: Sensitivity analysis. For W2, W3, W6, W8, rank and performance of best setting (secs) are shown. Lower is better

In summary, as few as twenty experiments chosen smartly by iTuned can produce a wealth of information in a reasonable amount of time to aid both naive users and expert DBAs in tuning database configuration parameters.

5.8 Related Work

Databases have fairly mature tools for physical design tuning (e.g., index selection [23]). However, these tools do not address configuration parameter tuning. Furthermore, these tools depend on the cost models in the query optimizer so are limited in that these models do not capture the effects of many parameters.

Surprisingly, very little work has been done on tools for holistic tuning of the many configuration parameters in modern database systems. Most work in this area has either focused on specific classes of parameters (e.g., [75]) or on restricted subproblems of the overall parameter tuning problem (e.g., [29]). IBM DB2 provides an advisor for setting default values for a large number of parameters [53]. DB2's advisor does not generate response surfaces, instead it relies on built-in models of how various parameters affect performance [29]. As we show this chapter, predetermined models may not be accurate in a given setting. SARD (discussed in Section

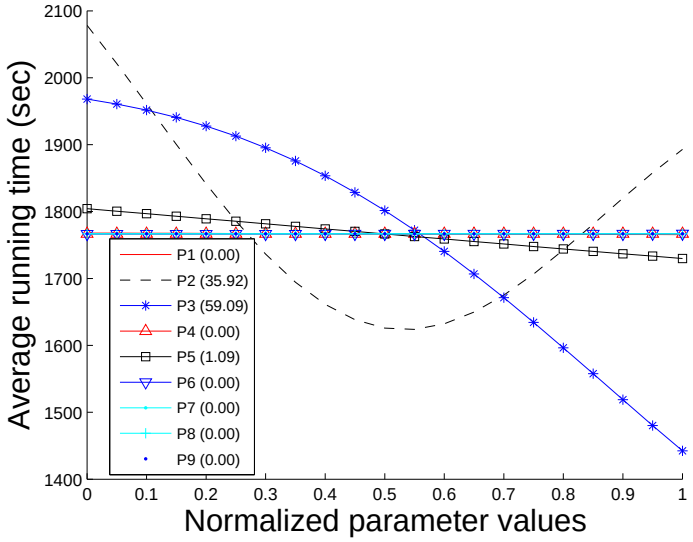

Figure 5.11: Effect plot for workload W5 (SF=10)

5.3) and [76] are also related to iTuned. SARD focuses on ranking parameters in order of impact, and is not an end-to-end tuning tool. Reference [76] proposed techniques to learn a probabilistic model using samples generated from gridding, which was then applied to tune four parameters in Berkeley DB. Gridding becomes very inefficient as the number of parameters increase. Section 5.7 also compared iTuned with a technique based on hill climbing (e.g., [89]) that has been applied to parameter tuning. None of the above techniques have an equivalent component of the .eX framework or the efficiency-oriented features from Section 5.6.

Techniques for tuning specific classes of parameters include solving analytical models [81], using simulations of database performance (e.g., in Oracle database), and control-theoretic approaches for online tuning [75]. These techniques are all based on predefined models of how changes in parameter settings affect performance. Reference [74] proposed techniques to tune the CPU and memory allocations to databases running inside virtual machines. However, the focus was not on planning

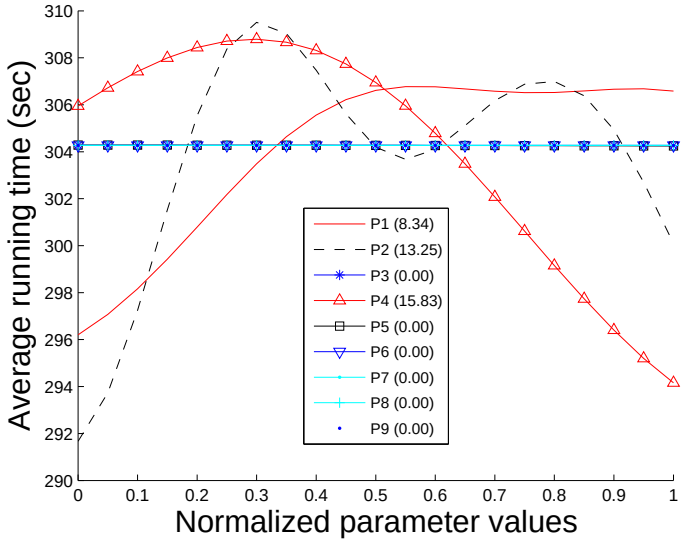

Figure 5.12: Effect plot for workload W4 (SF=1)

experiments to learn the underlying response surfaces. All the above techniques can benefit from the Adaptive Sampling and experiment execution ideas used in iTuned.

Traditional database sampling deals with the problem of sampling from a large dataset, while our approach of Adaptive Sampling is about drawing samples from a response surface that is never materialized fully. Adaptive Sampling shares goals, but not techniques, with conventional database problems like online aggregation [46], acquisitional query processing [55], and sampling for statistics estimation [21]. For example, reference [21] gives a two-phase adaptive method in which the sample size required to reach a desired accuracy is decided based on a first phase of sampling. In contrast, Adaptive Sampling can adapt after each sample is brought in.

Oracle 11g introduced the SQL Performance Analyzer (SPA) to help DBAs measure the impact of database changes like upgrades, parameter changes, schema

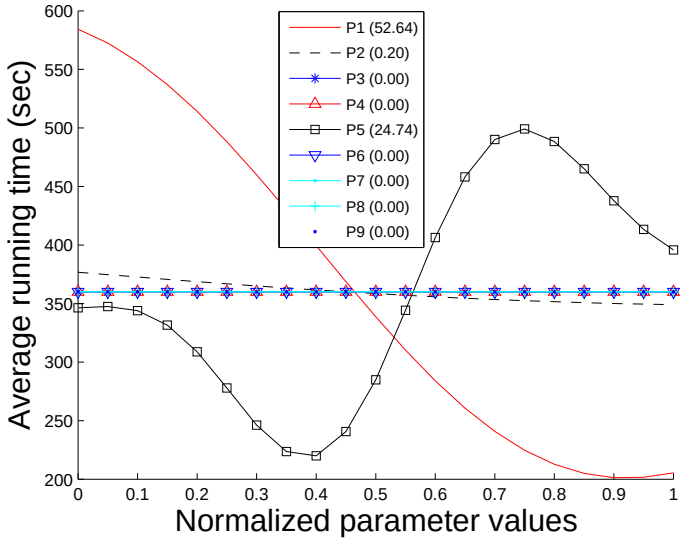

Figure 5.13: Effect plot for workload W4 (SF=10)

changes, and gathering optimizer statistics [91]. (Quoting from [91], "it is almost impossible to predict the impact of such changes on SQL performance before actually trying them.") SPA conducts experiments where SQL statements in the workload are executed with and without applying a change. However, Oracle 11g does not provide an experiment planner that can automatically handle complex tuning tasks like parameter tuning. Finally, experiments are used to collect data in many domains like chemical and mechanical engineering, social science, and computer simulation. While iTuned shares overall guiding principles with experiment planning in these domains, the requirements and algorithms differ.

5.9 Summary

We described the iTuned approach used by Fa to process tuning queries. Specifically, iTuned automates the task of recommending good settings for database configuration parameters. iTuned has three novel features: (i) Adaptive Sampling to proac-

tively bring in appropriate data through planned experiments to find high-impact parameters and high-performance parameter settings, (ii) use of the .eX framework to support online experiments in production database environments through a cycle-stealing paradigm that places near-zero overhead on the production workload, and (iii) portability across different database systems. We showed the effectiveness of iTuned through an extensive evaluation based on different types of workloads, database systems, and usage scenarios.

Chapter 6

Automated Processing of Forecasting Queries

6.1 Motivation

The previous chapters presented techniques for processing diagnosis and tuning queries to resolve system problems that have already happened. Since system slowdowns or outage can lead to financial losses, it is more attractive to provide *proactive system management*— where diagnosis and resolution of potential problems occur before the problems actually happen. To that end, Fa needs the ability to process forecasting queries that ask about future system performance based on historical system data.

Apart from proactive system management, forecasting future events based on historical data is applicable and useful in a range of domains like inventory planning, adaptive query processing, and sensor data management. On-demand computing systems, e.g., Amazon's Elastic Cloud [41], treat physical resources like servers and storage as a part of a shared computing infrastructure, and allocate resources dynamically to support changing application demands. These systems benefit significantly from early and accurate forecasts of workload surges and potential failures [65].

Adaptive query processing [9], where query plans and physical designs adapt to changes in data properties and resource availability, becomes very effective if these changes can be predicted with reasonable accuracy. Forecasting also plays an important role in personal and enterprise-level decision making. For example, inventory planning involves forecasting future sales based on historical data; and day traders rapidly buy and sell stocks based on forecasts of stock performance [93].

In this chapter, we describe how Fa processes declarative forecasting queries posed by users and applications. Fa supports efficient algorithms to generate execution plans for these queries, and returns forecasts and accuracy estimates in real-time. A forecasting query is posed over a multidimensional time-series dataset that represents historical data, as illustrated by the following example query.

Q_1: Select C
From Usage
Forecast 1 day

The full syntax and semantics of forecasting queries will be given in Section 6.2. The From clause in a forecasting query specifies the historical time-series data on which the forecast will be based. The query result will contain forecasts for the attributes listed in the Select clause, with the forecasts given for the timestamp(s) specified in the (new) Forecast clause. By default, this timestamp is specified as an interval, called *lead-time*, relative to the maximum timestamp in the historical data.

The time-series dataset "Usage" used in example query Q_1 is shown in Figure 6.1(a). Usage contains daily observations from Day 5 to Day 17 of the bandwidth used on three links in an Internet Service Provider's network. Since Q_1 specifies a lead-time of 1 day, Q_1's result will be a forecast of attribute C for Day 18. Q_1 is

Day	A	B	C
5	36	24	17
6	12	46	16
7	36	46	17
8	12	47	69
9	35	25	17
10	35	46	68
11	13	46	16
12	13	46	68
13	35	46	68
14	36	46	16
15	35	25	16
16	13	47	68
17	12	25	16

Day	A	B	C	A_{-1}	B_{-1}	C_{-1}	C_1
5	36	24	17	--	--	--	16
6	12	46	16	36	24	17	17
7	36	46	17	12	46	16	69
8	12	47	69	36	46	17	17
9	35	25	17	12	47	69	68
10	35	46	68	35	25	17	16
11	13	46	16	35	46	68	68
12	13	46	68	13	46	16	68
13	35	46	68	13	46	68	16
14	36	46	16	35	46	68	16
15	35	25	16	36	46	16	68
16	13	47	68	35	25	16	16
17	12	25	16	13	47	68	?

Day	A_{-2}	B_{-1}	C_1
5	--	--	16
6	--	24	17
7	36	46	69
8	12	46	17
9	36	47	68
10	12	25	16
11	35	46	68
12	35	46	68
13	13	46	16
14	13	46	16
15	35	46	68
16	36	25	16
17	35	47	?

Figure 6.1: Example datasets. (a) Usage; (b) and (c) are transformed versions of Usage

a *one-time* query posed over a fixed dataset, similar to a conventional SQL query posed over relations in a database system. Our next example forecasting query Q_2 is posed as a *continuous* query over a windowed data stream.

Q_2: Select cpu_util, num_io, resp_time

From PerfMetricStream [Range 300 minutes]

Forecast 15 minutes, 30 minutes

Q_2 is expressed in the CQL continuous query language [7] extended with the Forecast clause. Here, "PerfMetricStream" is a continuous stream of performance metrics collected once every minute from a production database server. At timestamp τ (in minutes), Q_2 asks for forecasts of CPU utilization, number of I/Os, and response time for all timestamps in $[\tau + 15, \tau + 30]$. This forecast should be based on the window of data in PerfMetricStream over the most recent 300 minutes, i.e., tuples with timestamps in $[\tau - 299, \tau]$.

The problem we address in this chapter is how to process forecasting queries automatically and efficiently in order to generate the most accurate answers possible based on patterns in the historical data; also giving accuracy estimates for forecasts.

A forecasting query can be processed using an execution plan that builds and uses a statistical model from the historical data. The model captures the relationship between the value we want to forecast and the recent data available to make the forecast. Before building the model, the plan may apply a series of transformations to the historical data.

Let us consider a plan p to process our example query Q_1. Plan p first transforms the Usage dataset to generate the dataset shown in Figure 6.1(b). A tuple with timestamp τ in this transformed dataset contains the values of attributes A, B, and C for timestamps $\tau - 1$ and τ from Usage, as well as the value of C for timestamp $\tau+1$. These figures use the notation X_δ to denote the attribute whose value at time τ is the value of attribute $X \in$ Usage at time $\tau + \delta$. Using the tuples from Day 6 to Day 16 in the transformed dataset as training samples, plan p builds a *Multivariate Linear Regression* model (MLR) [93] that can estimate the value of attribute C_3 from the values of attributes A, B, C, A_{-1}, B_{-1}, and C_{-1}. The MLR model built is:

$$C_1 = -0.7A - 1.04B - 0.43C - A_{-1} + 1.05B_{-1} - 0.32C_{-1} + 114.4$$

Once this model has been built, it can be used to compute Q_1's result, which is the "?" in Figure 6.1(b) because C_1 for Day 17 is equal to C for Day 18 given our transformation. By substituting $A = 12$, $B = 25$, $C = 16$, $A_{-1} = 13$, $B_{-1} = 47$, and $C_{-1} = 68$ from Day 17 (Figure 6.1(b)) into the MLR model, we get the forecast 87.78.

However, plan p has some severe shortcomings that illustrate the challenges in accurate and efficient forecasting:

(1) It is nontrivial to pick the best set of transformations to apply before building

the model. For example, if plan p had performed the appropriate attribute creation and removal to generate the transformed dataset shown in Figure 6.1(c), then the MLR built from this data forecasts 64.10 as Q_1's result. 64.10 is more accurate than 87.78; notice from the Usage data that when A_{-2} and B_{-1} are around 35 and 47 respectively, then C_1 is around 68.

(2) Linear regression may fail to capture complex data patterns needed for accurate forecasting. For example, by building and using a *Bayesian Network* synopsis on the dataset in Figure 6.1(c), the forecast can be improved to 68.5 (see Example 6.3.2 in Section 6.3). This observation raises a challenging question: how can we pick the best statistical model to use in a forecasting plan?

(3) For high-dimensional datasets, most statistical models (including MLR) have very high model-building times, and often their accuracy degrades as well. This fact is problematic when forecasts are needed for real-time decisions, particularly in high-speed streaming applications. Furthermore, plans must adapt as old patterns disappear and new patterns emerge in time-varying streams.

6.2 Abstraction of Forecasting Queries

Consider multidimensional time-series datasets in Fa that have a relational schema $\Gamma, X_1, X_2, \ldots, X_n$. Γ is a *timestamp* attribute with values drawn from a discrete, ordered domain $dom(\Gamma)$. A dataset can contain at most one tuple for any timestamp $\tau \in dom(\Gamma)$. Each X_i is a time series. We use $X_i(\tau)$ to denote X_i's value at time τ.

We will defer the discussion of continuous forecasting queries over windowed

data streams to Section 6.9. A one-time forecasting query over a fixed dataset has the general form:

Select *AttrList*

From D

Forecast [`absolute`] L [, [`absolute`] L']

$D(\Gamma, X_1, \ldots, X_n)$ is a time-series dataset, *AttrList* is a subset of $X_1, \ldots, X_n$, and L, L' are intervals or timestamps from $dom(\Gamma)$. Terms enclosed within "[" and "]" are optional.

Before we consider the general case, we will first explain the semantics of a simple forecasting query "Select X_i From D Forecast L," which we denote as *Forecast*(D,X_i,L). Let D consist of m tuples with respective timestamps $\tau_1 < \tau_2 < \cdots < \tau_m$. Then, the result of *Forecast*(D, X_i, L) is the two-tuple $\langle X_i(\tau_m + L), acc \rangle$; $X_i(\tau_m + L)$ is the forecast and acc is the estimated accuracy of this forecast. Intuitively, the result of *Forecast*(D, X_i, L) is the forecast of attribute X_i for a timestamp that is L time units after the maximum timestamp in D; hence, L is called the lead-time of the forecast.

The extension to forecasting multiple attributes is straightforward. For example, the result of "Select X_i, X_j From D Forecast L," is a four-tuple $\langle X_i(\tau_m + L), acc_i, X_j(\tau_m + L), acc_j \rangle$, where acc_i and acc_j are the accuracy estimates of the $X_i(\tau_m + L)$ and $X_j(\tau_m + L)$ forecasts.

If the query specifies two lead-times L and L', then the query is called a *range forecasting query*, as opposed to a *point forecasting query* that specifies a single lead-time. The result of a range forecasting query contains a forecast and accuracy estimate for each specified attribute for each timestamp in the $[L, L']$ range. If the keyword `absolute` is specified before a lead-time L, then L is a treated as an

absolute timestamp (e.g., March 1, 2007) rather than as an interval relative to the maximum timestamp in D.

Problem Setting: In this chapter we consider point forecasting queries of the form *Forecast*(D, X_i, L) only, since the main technical contributions we want to present are the algorithms for plan-selection and adaptive query processing. (Range forecasting is not a straightforward extension of point forecasting.) Also, we have chosen to output forecasting query results as forecasts and accuracy estimates. Other options exist, e.g., outputting a probability distribution over possible values (which some of our plans can do).

6.3 Execution Plans

A plan for a forecasting query contains three types of logical operators—*transformers*, *predictors*, and *builders*—and a summary data structure called *synopsis*.

- A *transformer* $T(D)$ takes a dataset D as input, and outputs a new dataset D' that may have a different schema from D.
- A *synopsis* $Syn(\{Y_1, \ldots, Y_N\}, Z)$ captures the relationship between attribute Z and attributes $Y_1, \ldots, Y_N$, such that a *predictor* $P(Syn, u)$ can use Syn to estimate the value of Z in a tuple u from the known values of $Y_1, \ldots, Y_N$ in u. Z is called Syn's *output attribute*, and $Y_1, \ldots, Y_N$ are called Syn's *input attributes*.
- A *builder* $B(D, Z)$ takes a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$ as input and generates a synopsis $Syn(\{Y_1, \ldots, Y_N\}, Z)$.

Next, we give two example physical implementations each for the logical entities defined above.

Project transformer: A project transformer π_{list} retains attributes in the input that are part of the attribute list *list*, and drops all other attributes in the input dataset; so it is similar to a duplicate-preserving project in SQL.

Shift transformer: Shift(X_j,δ), where $1 \leq j \leq n$ and δ is an interval from *dom*(Γ), takes a dataset $D(\Gamma, X_1, \ldots, X_n)$ as input, and outputs dataset $D'(\Gamma, X_1, \ldots, X_n, X')$ where the newly-added attribute $X'(\tau) = X_j(\tau+\delta)$. When δ is positive (negative), then X' is copy of X_j that is shifted backward (forward) in time.

Example 6.3.1. *The dataset in Figure 6.1(c) was computed from the Usage(A, B, C) dataset in Figure 6.1(a) by applying the transformers Shift($A, -2$), Shift($B, -1$), Shift($C, 1$), and the transformer π_{A_{-2},B_{-2},C_1} in sequence.*

Multivariate Linear Regression (MLR): An *MLR synopsis* with input attributes $Y_1, \ldots, Y_N$ and output attribute Z estimates the value of Z as a linear combination of the Y_j values [88]. Mathematically:

$$Z = c + \sum_{j=1}^{N} \alpha_j Y_j \tag{6.1}$$

The *MLR-builder* uses a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$ to compute the *regression coefficients* α_j and the constant c in Equation 6.1. Note that Equation 6.1 is actually a system of linear equations, one equation for each tuple in D. The MLR-builder computes the least-squares solution of this system of equations, namely, the values of α_js and c that minimize the sum of $(Z(\tau) - \hat{Z}(\tau))^2$ over all the tuples in D [93].

Here, $Z(\tau)$ and $\hat{Z}(\tau)$ are respectively the actual and estimated values of Z in the tuple with timestamp τ in D. Once all α_js and c have been computed, the *MLR-predictor* uses Equation 6.1 to estimate Z in a tuple given the values of attributes $Y_1, \ldots, Y_N$.

Bayesian Networks (BN): A *BN synopsis* is a summary structure that can represent the joint probability distribution $Prob(Y_1, \ldots, Y_N, Z)$ of a set of random variables $Y_1, \ldots, Y_N, Z$ [88]. A BN for variables $Y_1, \ldots, Y_N, Z$ is a directed acyclic graph (DAG) with $N + 1$ vertices corresponding to the $N + 1$ variables. Vertex X in the BN is associated with a *conditional probability table* that captures $Prob(X|Parents(X))$, namely, the conditional probability distribution of X given the values of X's parents in the DAG. The DAG structure and conditional probability tables in the BN satisfy the following equation for all $(Y_1 = y_1, \ldots, Y_N = y_N, Z = z)$ [88]:

$$Prob(y_1, \ldots, y_N, z) = \prod_{i=1}^{N} Prob(Y_i = y_i|Parents(Y_i)) \times Prob(Z = z|Parents(Z))$$

Given a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$, the *BN-builder* finds the DAG structure and conditional probability tables that approximate the above equation most closely for the tuples in D. Since this problem is NP-hard, the BN-builder uses heuristic search over the space of DAG structures for $Y_1, \ldots, Y_N, Z$ [88].

The *BN-predictor* uses the synopsis generated by the BN-builder from $D(\Gamma, Y_1, \ldots, Y_N, Z)$ to estimate the unknown value of Z in a tuple u from the known values of $u.Y_1, \ldots, u.Y_N$. The BN-predictor first uses the synopsis to infer the distribution

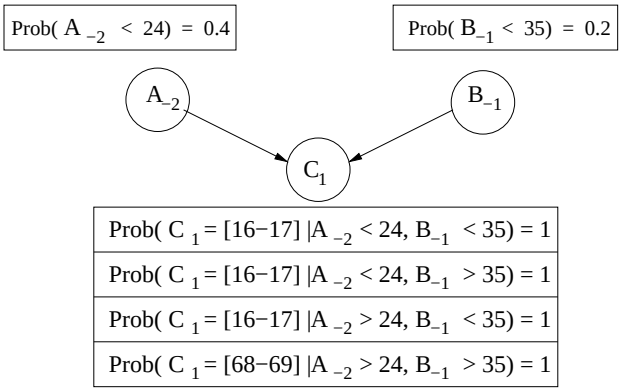

Figure 6.2: BN synopsis built from data in Fig. 6.1(c)

$Prob(u.Z = z|u.Y_j = y_j, 1 \leq j \leq N)$. The exact value of $u.Z$ is then estimated from this distribution, e.g., by picking the expected value.

Example 6.3.2. *Figure 6.2 shows the BN synopsis built by the BN-builder from the transformed Usage*(A, B, C) *dataset in Figure 6.1(c). To compute example query* Q_1*'s result, the BN-predictor will use the synopsis to compute* $Prob(C_1 \in [16-17]|A_{-2} = 35, B_{-1} = 47)$ *and* $Prob(C_1 \in [68-69]|A_{-2} = 35, B_{-1} = 47)$*, which are 0 and 1 respectively; hence the forecast* 68.5 *will be output.*

6.3.1 Initial Plan Space Considered (Φ)

There is a wealth of possible synopses, builders, predictors, and transformers from the statistical machine-learning literature [88]. For clarity of presentation, we begin by considering a focused physical plan space Φ. Section 6.7 describes how our algorithms can be applied to larger plan spaces.

An execution plan $p \in \Phi$ for a *Forecast*(D, X_i, L) query first applies a sequence of transformers to D, then uses a builder to generate a synopsis from the transformed dataset, and finally uses a predictor to make a forecast based on the synopsis and the transformed dataset. For example, the transformers in Example 6.3.1, followed

by the BN synopsis built by the BN-builder and used to make the forecast by the BN-predictor in Example 6.3.2, is one complete plan in Φ.

Transformers: The transformers in a plan $p \in \Phi$ are limited to Shift and π. From working with a number of transformers for adding, dropping, and mapping attributes (Section 6.7), we have found that Shift and π play a critical role in presenting the input data in ways that a synopsis built from the data can capture patterns useful for forecasting. Shift creates relevant new attributes—not present in the original input dataset—to include in a synopsis. π eliminates irrelevant and redundant attributes that harm forecasting if included in synopses: (i) these attributes can reduce forecasting accuracy by obscuring relevant patterns, and (ii) these attributes can increase the time required to build synopses.

Synopses, Builders, and Predictors: A plan $p \in \Phi$ contains one of five popular synopses—along with builder and predictor for that synopsis—from the statistical machine-learning literature: Multivariate Linear Regression (MLR), Bayesian Networks (BN), *Classification and Regression Trees* (*CART*), *Support Vector Machines* (*SVM*), and *Random Forests* (*RF*) [88]. The choice of which synopses to include in Φ was guided by some very recent studies that compare various synopses. MLR and BN are described in Section 6.3. The following describes the synopses and their respective builders and predictors. It is not necessary to understand the specific details of these synopses to understand our contributions.

Classification and Regression Tree (CART)

CART is a technique that builds a classification tree for predicting a categorical attribute and builds a regression tree for predicting a continuous-valued attribute

[88].

Given a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$, the *CART-builder* learns a CART T from D that represents the important interactions between the attribute Z to be predicted and the other attributes Y_i.

When making a prediction of the unknown value of Z in a tuple u, the *CART-predictor* determines a path in T from the root to a certain leaf node $node_k$ based on the known values of $u.Y_1$, ..., $u.Y_N$. The value of Z is finally derived from the value distribution in $node_k$.

Support Vector Machines (SVM)

SVM is a popular technique for classification. It maps training data into a higher dimensional space using a kernel function, and determines a linear hyperplane to separate the training data with maximal margin in the higher dimensional space for the purpose of classification [49].

Given a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$, the *SVM-builder* learns an SVM S from D with the type of kernel *Radial Basis Function (RBF)*. There are two parameters to configure for the SVM-builder, namely, *cost* and *gamma*. These two parameters are found to be sensitive to prediction accuracy, and take much time to tune [49].

When making a prediction of the unknown value of Z in a tuple u, the *SVM-predictor* maps a vector of the known values of $u.Y_1$, ..., $u.Y_N$ into a new vector V in the higher dimensional space using the specified kernel function of RBF, and predicts the value of Z based on the position of V relative to the separating hyperplane in S.

Random Forests (RF)

RF represents *ensemble learning* [88] that is used widely in machine learning today. RF constructs an ensemble of tree predictors $h(\mathbf{y};\theta_k)$ $(k = 1, \ldots, K)$ where $\mathbf{y}$ is a vector of values for the observable attributes and θ_k $(k = 1, \ldots, K)$ are independent and identically distributed random vectors [13]. Random forests have several nice properties for prediction. For example, it is excellent in accuracy among current prediction techniques, and it can avoid overfitting [18].

Given a dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$, *RF-builder* determines each tree predictor $h(\mathbf{y};\theta_k)$ from the randomly selected attributes in D. The RF-builder needs to configure two parameters, the number of tree predictors in the RF and the number of attributes in each tree [13].

When making a prediction of the unknown value of Z in a tuple u, the *RF-predictor* gets a prediction from each tree predictor based on the known values of $u.Y_1$, ..., $u.Y_N$, and combines these predictions for the final prediction using weighted averaging or voting strategies [13].

6.4 Plan-Selection Preliminaries

In this section we describe the important characteristics of the problem of selecting a good plan from Φ for a *Forecast*(D, X_i, L) query. We describe the structure of plans from Φ, estimate the total size of the plan space, and show the difficulty in estimating the forecasting accuracy of a plan.

Observation 6.4.1. (Plan structure) *Let $D(\Gamma, X_1, \ldots, X_n)$ be a time-series dataset. A plan $p \in \Phi$ for a* Forecast(D, X_i, L) *query can be represented as $\langle Y_1, \ldots, Y_N$, type, $Z\rangle$ where:*

- *type* $\in$ {*MLR,BN,CART,SVM,RF*} *is p's synopsis type. The synopsis type uniquely determines both p's builder and p's predictor.*

- $Y_1, \ldots, Y_N$ *are attributes such that each* $Y_i = X_j(\tau + \delta)$ *for some* j *and* δ, $1 \leq j \leq n$ *and* $\delta \leq 0$. $Y_1, \ldots, Y_N$ *are the input attributes of p's synopsis.*

- *Attribute* $Z = X_i(\tau + L)$ *is the output attribute of p's synopsis.*

This observation can be justified formally based on our assumptions about Φ and the definitions of synopses, builders, predictors, and transformers. Because of space constraints, we provide an intuitive explanation only.

Recall that plan p for *Forecast*(D, X_i, L) will use a synopsis to estimate the query result, namely, the value of $X_i(\tau_m + L)$, where τ_m is the maximum timestamp in D. p's synopsis has access only to data in D up to timestamp τ_m in order to estimate $X_i(\tau_m + L)$. Furthermore, ϕ has only the Shift transformer to create new attributes apart from attributes $X_1, \ldots, X_n$. With these restrictions, the input attributes in p's synopsis can only be $X_j(\tau + \delta)$, $1 \leq j \leq n$ and $\delta \leq 0$, and the output attribute is $X_i(\tau + L)$.

Observation 6.4.2. (Size of plan space) *Suppose we restrict the choice of Shift transformers to derive input attributes for synopses in a plan to Shift*(X_j, δ), $1 \leq j \leq n$ *and* $-\Delta \leq \delta \leq 0$. *Then, the number of unique plans in* Φ, $|\Phi| = 5 \times 2^{(n+1)\Delta}$.

In one of our application domains, $n = 252$ and $\Delta = 90$, so $|\Phi| = 5 \times 2^{22,770}$!

6.4.1 Estimating Forecasting Accuracy of a Plan

The *forecasting accuracy* (accuracy) of a plan $p = \langle Y_1, \ldots, Y_N, \textit{type}, Z \rangle$ is the accuracy with which p's synopsis can estimate the true value of Z given the values

of $Y_1, \ldots, Y_N$. The goal of plan selection for a Q =*Forecast*(D, X_i, L) query is to find a plan that has accuracy close to the best among all plans for Q in Φ. To achieve this goal, we need a way to compute the accuracy of a given plan.

The preferred technique in statistical machine-learning to estimate the accuracy of a synopsis is called *K-fold cross-validation* (*K-CV*) [88]. K-CV can estimate the accuracy of a plan p that builds its synopsis from the dataset $D(\Gamma, Y_1, \ldots, Y_N, Z)$. K-CV partitions D into K (nonoverlapping) partitions, denoted $D_1, \ldots, D_K$. (Typically, $K = 10$.) Let $D'_i = D - D_i$. For $i \in [1, K]$, K-CV builds a synopsis $Syn_i(\{Y_1, \ldots, Y_N\}, Z)$ using the tuples in D'_i, and uses this synopsis to estimate the value of $u.Z$ for each tuple $u \in D_i$. This computation will generate a pair $\langle a_j, e_j \rangle$ for each of the m tuples in D, where a_j is the tuple's actual value of Z and e_j is the estimated value. Any desired accuracy metric can be computed from these pairs, e.g., *root mean squared error* $= \sqrt{\sum_{j=1}^{m} \frac{(a_j - e_j)^2}{m}}$, giving an accuracy estimate for p.

Observation 6.4.3. (Estimating accuracy) *K-CV is a robust technique to estimate the accuracy of a plan $p = \langle Y_1, \ldots, Y_N, type, Z \rangle$ without knowing the actual query result. However, K-CV builds a synopsis of type type K times, and uses these synopses to estimate Z for each input tuple.*

In effect, K-CV is computationally expensive. However, discussions with researchers in statistical machine-learning revealed that there are no known techniques to estimate the accuracy of an MLR, BN, CART, SVM, or RF synopsis fairly-accurately on a dataset D without actually building the synopsis on D. (In Section 6.6.6, we report experimental results related to this point.) Therefore, K-CV is as good a technique as any to estimate the accuracy of a plan, and we will use it for that purpose.

Algorithm *Forecasting Plan Search (FPS)*
Input: *Forecast*$(D(\Gamma, X_1, \ldots, X_n), X_i, L)$ query
Output: Forecasts and accuracy estimates are output as FPS runs. FPS is terminated when a forecast with satisfactory accuracy is obtained, or when lead-time runs out;

1. $l = 0$; /* current iteration number of outer loop */
BEGIN OUTER LOOP
2. Attribute set $Attrs = \{\}$; /* initialize to an empty set */
3. $l = l + 1$; /* consider Δ more shifts than in last iteration */
4. **FOR** $j \in [1, n]$ and $\delta \in [-l\Delta, 0]$
   Add to *Attrs* the attribute created by Shift(j, δ) on D;
5. Generate attribute $Z = X_i(\tau + L)$ using Shift(X_i, L) on D;
6. Let the attributes in *Attrs* be $Y_1, \ldots, Y_q$. Rank $Y_1, \ldots, Y_q$ in decreasing order of relevance to Z; /* Section 6.5.2 */

/* Traverse the ranked list of attributes from start to end */
   BEGIN INNER LOOP
7. Pick next chunk of attributes from list; /* Section 6.5.3 */
8. Decide whether a complete plan should be generated using the current chunk of attributes; /* Section 6.5.5 */
9. **IF Yes**, find the best plan p that builds a synopsis using attributes in the chunk. If p has the best accuracy among all plans considered so far, output the value forecast by p, as well as p's accuracy estimate; /* Section 6.5.4 */
   END INNER LOOP
END OUTER LOOP

Figure 6.3: Plan selection and execution algorithm for one-time forecasting queries

6.5 Processing One-Time Queries

The observations in Section 6.4 motivate our algorithm, called *Forecasting Plan Search (FPS)*, to process a one-time forecasting query *Forecast*(D, X_i, L) query.

6.5.1 Overview of FPS

Figures 6.3 and 6.4 give an overview of FPS which runs as a two-way nested for loop. For simplicity of presentation, Figure 6.3 does not show how computation can be shared within and across the loops of FPS; sharing and other scalability issues

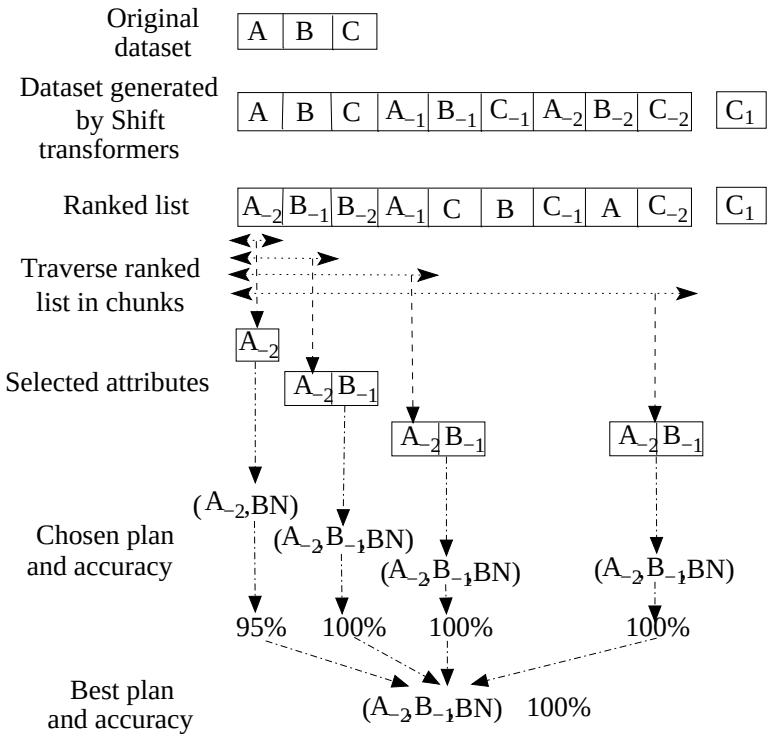

Figure 6.4: Pictorial view of FPS processing Example query Q_1 from Section 6.1

are discussed in Section 6.5.6.

FPS's outer loop enumerates a large number of attributes of the form $X_j(\tau+\delta)$, $1 \leq j \leq n$ and $-l\Delta \leq \delta \leq 0$, where $\Delta \geq 0$ is a user-defined constant. The value of l is increased across iterations of the loop so that more and more attributes will be enumerated progressively. FPS aims to pick subsets of attributes from the enumerated set such that synopses built from these subsets give good accuracy. Intuitively, a combination of a highly-predictive attribute subset $Y_1, \ldots, Y_N$ and an appropriate synopsis type *type* gives a plan $p = \langle Y_1, \ldots, Y_N, type, Z \rangle$ for estimating $Z = X_i(\tau + L)$ with good accuracy.

As illustrated in Figure 6.4 and Line 6 of Figure 6.3, FPS ranks the enumerated attributes in decreasing order of relevance to attribute Z. Techniques for ranking are described in Section 6.5.2. In its inner loop, FPS traverses the ranked list from

highly relevant to less relevant attributes, and extracts one chunk of attributes at a time. Techniques for traversing the ranked list are described in Section 6.5.3.

For each chunk C, FPS decides whether or not to derive a complete plan from C. Section 6.5.5 describes how this decision is made, and the implications of making a wrong decision. If the decision is to derive a plan, then FPS finds a highly-predictive subset of attributes $Y_1, \ldots, Y_N$ from C, and an appropriate synopsis type *type*, to produce the best plan $p = \langle Y_1, \ldots, Y_N, type, Z \rangle$ possible from C. These techniques are described in Section 6.5.4. Recall from Section 6.4.1 that FPS has to build p's synopsis in order to estimate p's accuracy. Once the synopsis is built, the extra cost to use the synopsis to compute the value forecast by p is small.

Whenever FPS finds a plan p whose accuracy estimate is better the accuracy estimates of all plans produced so far, FPS outputs the value forecast by p as well as p's accuracy estimate. Thus, FPS produces more and more accurate forecasts in a progressive fashion, similar to online aggregation techniques proposed for long-running SQL queries [46]. FPS runs until it is terminated by the application/user who issued the query, e.g., a user may terminate FPS when she is satisfied with the current accuracy estimate.

We now present the options we considered to implement each step in FPS. These options are evaluated experimentally in Section 6.6, and a concrete instantiation of FPS is presented in Section 6.7.

6.5.2 Ranking Attributes

Line 6 in Figure 6.3 ranks the attributes enumerated by FPS's outer loop in decreasing order of relevance to the output attribute Z. Intuitively, attributes more relevant to Z are more likely to give good forecasts when included in a synopsis to forecast

Z. Such relevance can be computed in one of two ways: (i) *correlation-based*, where attributes are ranked based on their correlation with Z; and (ii) *time-based*, where attributes whose values are more recent are ranked higher.

Correlation-based Ranking: There exist two general approaches to measure correlation between attributes Y and Z, one based on linear-correlation theory and the other based on information theory [94]. *Linear correlation coefficient* (LCC) is a popular measure of linear correlation [88].

$$LCC(Y,Z) = LCC(Z,Y) = \frac{\sum_i (y_i - \overline{Y})(z_i - \overline{Z})}{\sqrt{\sum_i (y_i - \overline{Y})^2}\sqrt{\sum_i (z_i - \overline{Z})^2}}$$

Each (y_i, z_i) is a pair of (Y, Z) values in a tuple in the input dataset, and $\overline{Y}$ and $\overline{Z}$ denote the respective means. LCC is efficient to compute, but it may not capture nonlinear correlation.

Information gain of Z given Y, denoted $IG(Z,Y)$, is an information-theoretic measure of correlation between Y and Z computed as the amount by which the *entropy* of Z decreases after observing the value of Y. $IG(Z,Y) = H(Z) - H(Z|Y) = H(Y) - H(Y|Z) = IG(Y,Z)$, where H denotes entropy, e.g., $H(Y) = -\sum_i Prob(y_i) \log_2(Prob(y_i))$.

$$IG(Z,Y) \;=\; IG(Y,Z) = -\sum_i Prob(z_i)\log_2(Prob(z_i)) + \sum_j Prob(y_j)\sum_i Prob(z_i|y_j)\log_2(Prob(z_i|y_j))$$

Information gain is biased towards attributes with many distinct values, so a normalized variant called *symmetrical uncertainty* (SU) is often used to measure cor-

relation.

$$SU(Y,Z) = SU(Z,Y) = 2\left[\frac{IG(Y,Z)}{H(Y)+H(Z)}\right] \tag{6.2}$$

Time-based Ranking: Time-based ranking is based on the notion that an attribute whose value was collected closer in time to τ is more predictive of $Z(\tau)$ than an attribute whose value was collected earlier in time. For example, consider attributes Y_1 and Y_2 created from the input dataset D by Shift(X_1, δ_1) and Shift(X_2, δ_2) transformers respectively. Because of the shift, the value of $Y_1(\tau)$ ($Y_2(\tau)$) comes from a tuple in D with timestamp $\tau + \delta_1$ ($\tau + \delta_2$). Let $\delta_1 < \delta_2 \leq 0$. Then, $Y_2(\tau)$ is more relevant to $Z(\tau)$ than $Y_1(\tau)$.

6.5.3 Traversal of the Ranked List in Chunks

Single Vs. Multiple Chunks: FPS traverses the ranked list in multiple overlapping chunks $C_1 \subset C_2 \subset \cdots \subset C_k$ that all start from the beginning of the list (see Figure 6.4). If all the attributes are considered as a single chunk (i.e., if $k = 1$), then the time for attribute selection and synopsis learning can be high because these algorithms are nonlinear in the number of attributes. Overlapping chunks enable FPS to always include the highly relevant attributes that appear at the beginning of the list. The potential disadvantage of overlapping chunks is the inefficiency caused by repetition of computation when the same attributes are considered multiple times. However, as we will show in Section 6.5.6, FPS can share computation across overlapping chunks, so efficiency is not compromised.

Fixed-size Vs. Variable-size Increments: Successive overlapping chunks $C_1 \subset C_2 \subset \cdots \subset C_k$ can be chosen with fixed-size increments or variable-size increments. An example of consecutive chunk sizes with fixed-size increments is $|C_1| = 10, |C_2| = 20, |C_3| = 30, |C_4| = 40$ and so on (arithmetic progression), while an example with variable-size increments is $|C_1| = 10, |C_2| = 20, |C_3| = 40, |C_4| = 80$ and so on (geometric progression).

6.5.4 Generating a Plan from a Chunk

At Line 9 in Figure 6.3, we need to generate a complete plan from a chunk of attributes C. This plan $\langle \Omega, type, Z \rangle$ should contain attributes $\Omega \subseteq C$ and a synopsis type *type* that together give the best accuracy in estimating the output attribute Z among all choices for $(\Omega, type)$. We break this problem into two subproblems: (i) finding an attribute subset $\Omega \subseteq C$ that is highly predictive of Z, and (ii) finding a good synopsis type for Ω.

Selecting a Predictive Attribute Subset

The problem of selecting a predictive attribute subset $\Omega \subseteq C$ can be attacked as a search problem where each state in the search space represents a distinct subset of C [45]. Since the space is exponential in the number of attributes, heuristic search techniques can be used. The search technique needs to be combined with an estimator that can quantify the predictive ability of a subset of attributes. We consider three methods for attribute selection:

1. Wrapper, described in [88], estimates the predictive ability of an attribute subset Ω by actually building a synopsis with Ω as the input attribute set, and using K-CV to estimate accuracy. Wrapper uses *Best First Search* (*BFS*) [88] as

its heuristic search technique. BFS starts with an empty set of attributes and enumerates all possible single attribute expansions. The subset with the highest accuracy estimate is chosen and expanded in the same manner by adding single attributes. If no improvement results from expanding the best subset, up to k (usually, $k = 5$) next best subsets are considered. The best subset found overall is returned when the search terminates. Building synopses for all enumerated subsets makes Wrapper good at finding predictive subsets, but computationally expensive.

2. Correlation-based Feature Selection (CFS), described in [45], is based on the heuristic that a highly-predictive attribute subset Ω is composed of attributes that are highly correlated with the (output) attribute Z, yet the attributes in Ω are uncorrelated with each other [45]. This heuristic gives the following estimator, called *CFS Score*, to evaluate the ability of a subset Ω, containing k (input) attributes $Y_1, \ldots, Y_k$, to predict the (output) attribute Z (Equation 6.2 defines SU):

$$\textit{CFS Score}(\Omega) = \frac{\sum_{i=1}^{k} SU(Y_i, Z)}{\sqrt{k + \sum_{i=1}^{k} \sum_{j \neq i, j=1}^{k} SU(Y_i, Y_j)}} \tag{6.3}$$

The numerator in Equation 6.3 is proportional to the input-output attribute correlation, so it captures how predictive Ω is of Z. The denominator is proportional to the input-input attribute correlation within Ω, so it captures the amount of redundancy among attributes in Ω. CFS uses BFS to find the attribute subset $\Omega \subseteq C$ with the highest CFS Score.

3. Fast Correlation-based Filter (FCBF), described in [94], is based on the same heuristic as CFS, but it uses an efficient deterministic algorithm to eliminate attributes in C that are either uncorrelated with the output attribute Z, or redundant when considered along with other attributes in C that have higher correlation with Z. While CFS's processing-time is usually proportional to n^2 for n attributes,

FCBF's processing-time is proportional to $n \log n$.

Selecting a Synopsis Type

To complete a plan from an attribute subset Ω, we need to pick the synopsis type that gives the maximum accuracy in estimating Z using Ω. Recall from Section 6.4.1 that there are no known techniques to compute the accuracy of a synopsis without actually building the synopsis. One simple, but expensive, strategy in this setting is to build all five synopsis types, and to pick the best one. As an alternative, we tried to determine experimentally whether one or more synopsis types in BN, CART, MLR, SVM, and RF compare well in general to the other types in terms of both the accuracy achieved and the time to build. The results are very encouraging, and reported in Section 6.6.5.

6.5.5 Decision to Generate a Plan or Not

At Line 8 in Figure 6.3, we need to decide whether to generate a complete plan using the current chunk of attributes C. Generating a complete plan p from C involves an attribute selection and building one or more synopses, so it is desirable to avoid or reduce this cost if p is unlikely to be better than the current best plan. To make this decision efficiently, we need an estimator that gives a reasonably-accurate and efficiently-computable estimate of the *best accuracy possible* from chunk C. We can *prune* C if the estimated best accuracy from C is not significantly higher than the accuracy of the current best plan. We considered two options:

- **No Pruning**, where a plan is generated for all chunks.
- **Pruning(k)**, where chunk C is pruned if the best CFS Score (Equation 6.3) among attribute subsets in C is worse than the k-th best CFS score among

all chunks so far.

6.5.6 Sharing Computation in FPS

Recall from Section 6.5.3 that FPS traverses the ranked list of attributes in overlapping chunks $C_1 \subset C_2 \subset \cdots \subset C_k$ starting from the beginning of the list. The version of FPS described in Figure 6.3—which we call *FPS-full*—performs attribute selection and synopsis building from scratch for each chunk where a complete plan is generated.

An incremental version of FPS, called *FPS-incr*, shares (or reuses) computation across chunks by generating a plan from the attributes $\Theta \cup \{C_i - C_{i-1}\}$ at the ith chunk, instead of generating the plan from C_i. Here, $C_i - C_{i-1}$ is the set of attributes in C_i that are not in C_{i-1}, and $\Theta \subseteq C_{i-1}$ enables reuse of computation done for previous chunks. We consider two options for choosing Θ:

$$\Theta = \begin{cases} \Theta_{i-1} & (\textit{FPS-incr-cum}) \\ \Theta_{j'}, \quad j' = \text{argmax}_{1 \le j \le i-1}\, acc_j & (\textit{FPS-incr-best}) \end{cases}$$

Here, $\Theta_j \subseteq C_j$ is the attribute subset chosen in the plan for chunk C_j (Section 6.5.4), with corresponding accuracy acc_j. Intuitively, Θ in FPS-incr-cum tracks the cumulative attribute subset selected so far, while FPS-incr-best uses the best attribute subset among all chunks so far.

Note that sharing can be done across chunks in a ranked list—like we described above—as well as across the much larger overlapping chunks of attributes considered by successive iterations of FPS's outer loop; our description applies to this case as well.

Parameter/Option name	Default
Forecast lead time (Sec. 6.2)	25
Δ in FPS (Fig. 6.3)	90
Ranking attributes (Sec. 6.5.2)	Correlation-based (LCC)
Ranked-list traversal (Sec. 6.5.3)	Variable-sized (10×2^i)
Attribute selection (Sec. 6.5.4)	FCBF
Synopsis type (Sec. 6.5.4)	BN
Generate plan or not? (Sec. 6.5.5)	No pruning
Sharing across chunks (Sec. 6.5.6)	FPS-incr-cum

Table 6.1: Defaults for experiments

6.6 Experimental Evaluation

Our algorithms for processing one-time and continuous forecasting queries have been implemented in Fa. The implementation of all synopses is based on the open-source *WEKA* toolkit [88]. Table 6.1 indicates the experimental defaults.

6.6.1 Datasets, Queries, and Balanced Accuracy

Since Fa's target domain is system and database monitoring, the datasets, forecasting queries, and accuracy metrics used in our experimental evaluation are drawn primarily from this domain. We used real, testbed, and synthetic datasets; described next and in Table 6.2.

Real datasets: We consider three real datasets: *Aging-real*, *FIFA-real*, and *Motes-real*. Aging-real is a record of OS-level data collected over a continuous period of two months from nine production servers in the Duke EE departmental cluster; [84] gives a detailed analysis. The predictive patterns in this dataset include the effects of *software aging*—progressive degradation in performance due to, e.g., memory leaks, unreleased file locks, and fragmented storage space—causing transient system failures. We use Aging-real to process queries that forecast with lead-time L

Name	n	m	I	Description
1. Aging-real	44	4820	15	OS-level data collected for 55 days from nine Duke EE departmental servers
2. FIFA-real	124	2068	60	Load metrics collected for 92 days from the 1998 Soccer World Cup Web-site
3. Periodic-small-tb	196	4315	1	10-minute period; query parameter values varied to vary transaction response time
4. Periodic-large-tb	252	6177	1	90-minute period; resource contention created by varying number of DBMS threads
5. Aging-fixed-tb	252	1499	1	non-periodic; resource contention caused by an aging [84] CPU-intensive thread
6. Multi-large-tb	252	5000	1	non-periodic; resource contention caused by both CPU and disk contention
7. Multi-small-tb	252	719	1	16-minute period; resource contention caused by both CPU and disk contention
8. Aging-variant-syn	3	7230	1	non-periodic; rate of aging is not fixed like in Aging-fixed-tb
9. Complex-syn	6	14760	1	non-periodic; pattern simulating a problem that affects response time with a lag
10. Complex-noisy-syn	3	8162	1	non-periodic; the pattern is similar to that in Complex-syn, with white noise added
11. Motes-real	48	7712	1	sensor measurements collected from wireless sensors [30]

Table 6.2: Datasets used in experiments; n, m, and I are the number of attributes, tuples, and measurement interval (minutes) respectively; real, tb, and syn represent real, testbed, and synthetic datasets respectively

whether a server's average response time will exceed a specified threshold.

The Motes-real dataset is a trace of readings collected from 54 sensors in the Intel Research lab at Berkeley. The sensors collect readings of attributes like humidity, light, temperature, and voltage. This dataset contains 8 days of readings [30]. In our experiments, we consider queries that forecast the humidity and light attributes.

FIFA-real is derived from a 92-day log for the 1998 FIFA Soccer World Cup Website; [8] gives a detailed analysis. We use this dataset to process queries that forecast

with lead-time L whether the Web-site load will exceed a specified threshold. The load on the FIFA Web-site is characteristic of most popular Web-sites—with periodic segments, high burstiness at small time scales, but more predictability at larger time scales, and occasional traffic spikes.

Testbed datasets: These datasets contain OS, DBMS, and transaction-level performance metrics collected from a MySQL DBMS running an OLTP workload on a monitoring testbed we have developed [33]. Table 6.2 gives brief descriptions. These datasets are used to process queries that forecast with lead-time L whether the average transaction response time will exceed a specified threshold. Our testbed datasets cover (i) periodic workloads—with different period lengths and complexity of pattern—(ii) aging behavior—both periodic and non-periodic, fixed and varying rate of aging—and (iii) multiple performance problems (e.g., both CPU and I/O contention) happening in overlapping and nonoverlapping ways.

Synthetic datasets: We also generated synthetic time-series datasets using Matlab to study the robustness of our algorithms; see Table 6.2.

Balanced Accuracy (BA): Notice that the queries we consider in our experimental evaluation forecast whether the system will experience a performance problem L time units from now. The accuracy metric preferred for such queries in the system management domain is called *Balanced Accuracy (BA)* [65]. BA is computed using K-CV (Section 6.4.1) as: BA = 0.5(1 - FP) + 0.5(1 - FN). Here, FP is the ratio of false positives (predicting a problem when there is none) in the ⟨actual value, estimated value⟩ pairs. Similarly, FN is the ratio of false negatives (failing to predict an actual problem).

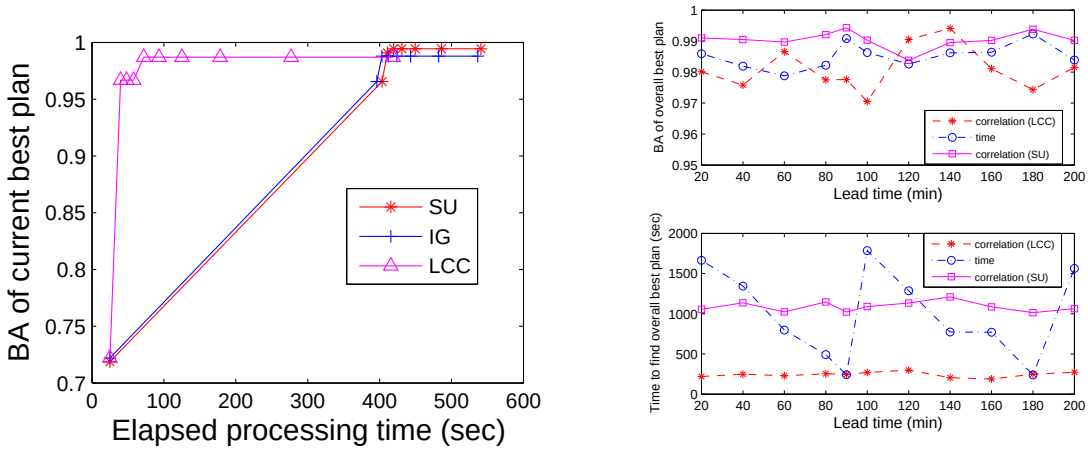

Figure 6.5: (a) Comparing correlation metrics, (b) Correlation Vs. time-based ranking. These two experiments use the Periodic-large-tb dataset.

Graphs: The majority of our graphs track the progress of FPS over time; the X axis shows the elapsed time since the query was submitted, and the Y axis shows the best BA among plans generated so far. Fast convergence to the maximum BA possible indicates good performance. BA is always ≥ 0.5, so the minimum Y value in these graphs is 0.5.

6.6.2 Ranking Attributes

Recall from Section 6.5.2 that FPS can choose to do time-based ranking of enumerated attributes, or correlation-based ranking using LCC, IG, or SU. Figures 6.5(a) and 6.5(b) show the general trend that we have seen across all our datasets and queries:

- FPS using LCC converges much faster and to almost the same BA as SU and IG. The worst results for LCC were for the synthetic Complex-syn dataset where we forced the predictive attributes to have nonlinear correlation with the output attribute Z. Even in this case, LCC converged to the same BA as

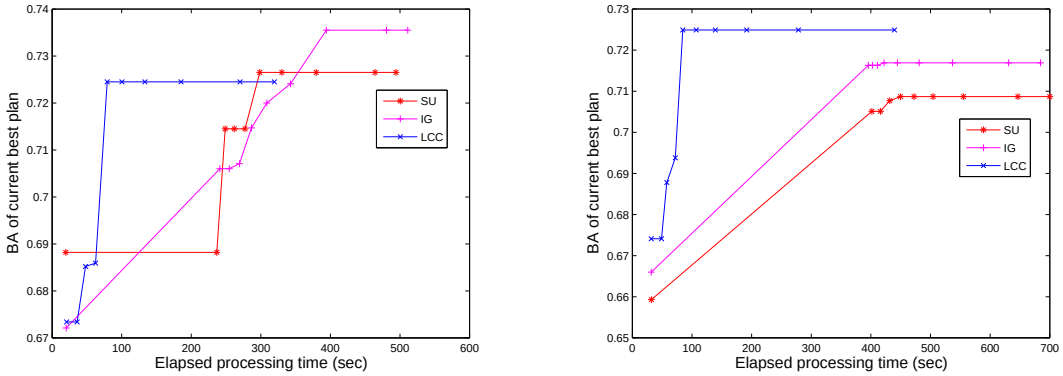

Figure 6.6: Comparing correlation metrics: (a) Aging-real, (b) Multi-large-tb

IG and SU, with 20% more time than SU and 14% more time than IG.

- Correlation-based ranking using LCC is more robust than time-based ranking. Note the sensitivity of time-based ranking to the lead-time in Figure 6.5(b): time-based ranking will converge quickly only when the predictive patterns are "close by" in time, which *depends on the lead-time* for Periodic-large-tb.

Figure 6.6(a) and 6.6(b) show that FPS using LCC metric to rank attributes converges faster than that using IG or SU metrics while achieving comparable prediction accuracies. Figure 6.7(a) shows that IG and SU are more accurate and robust ranking metrics than LCC in terms of the predictability of an attribute. Please note that Figure 6.7(a) uses the Complex-syn dataset in which there is strong nonlinear correlation between the attribute to be predicted and the other attributes. As the LCC metric only captures linear correlations between two attributes, it is hard to get a good ranking of the attributes in the Complex-syn dataset using the LCC metric.

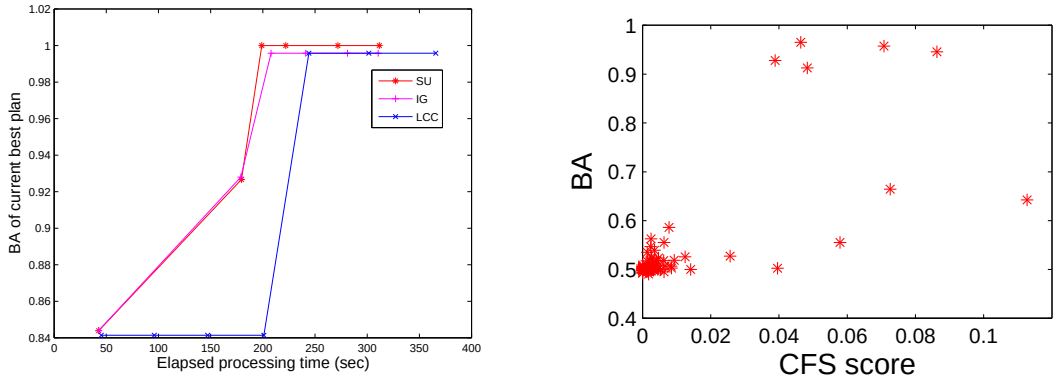

Figure 6.7: (a) Comparing correlation metrics (Complex-syn), (b) CFS-Score Vs. BA for random subsets(Periodic-large-tb)

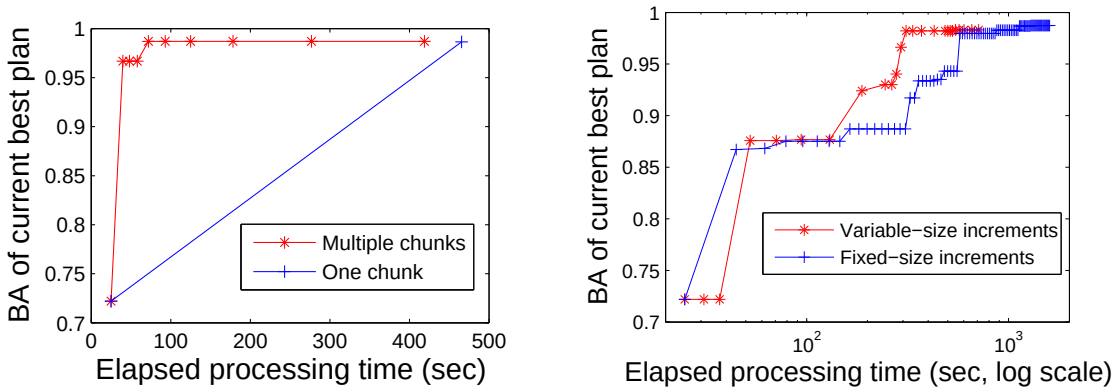

Figure 6.8: (a) Importance of chunk-based traversal, (b) Choosing chunk sizes (Δ=30 in Fig. 6.3). These two experiments use the Periodic-large-tb dataset.

6.6.3 Traversal of Ranked List

The trend in Figure 6.8(a) was consistent across all our datasets: it is significantly better to consider attributes in the ranked list in multiple overlapping chunks rather than considering all attributes all-at-once. The main reason is that the time for attribute selection and synopsis learning is nonlinear in the number of attributes. With multiple overlapping chunks, the chunks are smaller since the effects of computation sharing from Section 6.5.6 kick in.

We have found variable-size traversal (Section 6.5.3)—with chunk sizes increasing in a geometric progression 10×2^i up to a maximum size, then switching to an arithmetic progression with the last increment—to be more effective than fixed-size traversal:

- If ranking correctly brings the most predictive attributes to the beginning of the ranked list, then FPS gets close to the best accuracy from the first few chunks. Here, fixed-size and variable-size traversal are comparable.

- If the predictive attributes are not at the beginning of the ranked list—e.g., because the use of LCC for ranking failed to capture nonlinear correlation, or Δ was too low—then FPS may not get good accuracy until the middle/last chunks are considered or until more attributes are enumerated; variable-size traversal can get there sooner. This effect is clear in Figure 6.8(b) which considers a case where $\Delta = 30$ was low, so multiple iterations of the outer loop in Figure 6.3 were required to get to the best BA possible. Note the log scale on the X axis.

6.6.4 Selecting a Predictive Attribute Subset

Figure 6.9(a) represents the consistent trend we observed: (i) The running times of the attribute-selection techniques are in the order FCBF $<$ CFS $\ll$ Wrapper, with Wrapper being about an order of magnitude slower than FCBF and CFS, without any significant advantage in accuracy; (ii) FCBF tends to perform better on average than CFS, but other features of FPS—mainly, sharing of computation (Section 6.5.6)—blur the difference between FCBF and CFS. (*Principal Component Analysis* also performs worse than FCBF/CFS.)

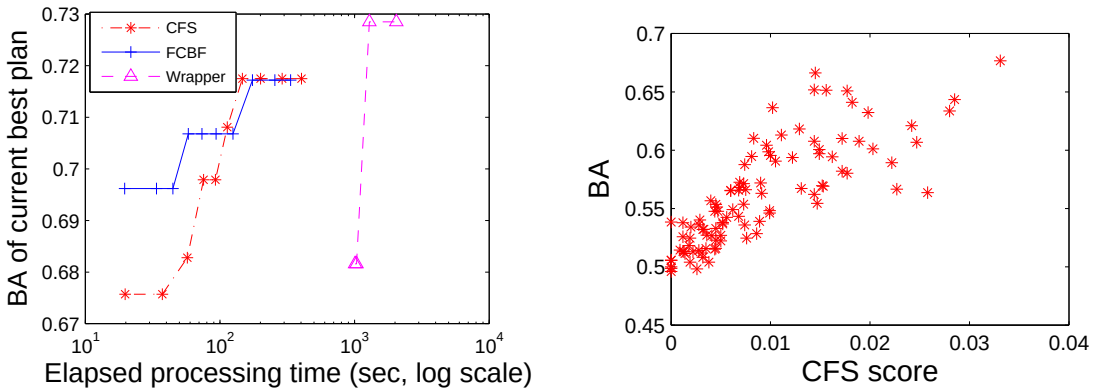

Figure 6.9: (a) Comparing attribute-selection techniques (Aging-real), (b) CFS-Score Vs. BA for random subsets (Multi-large-tb)

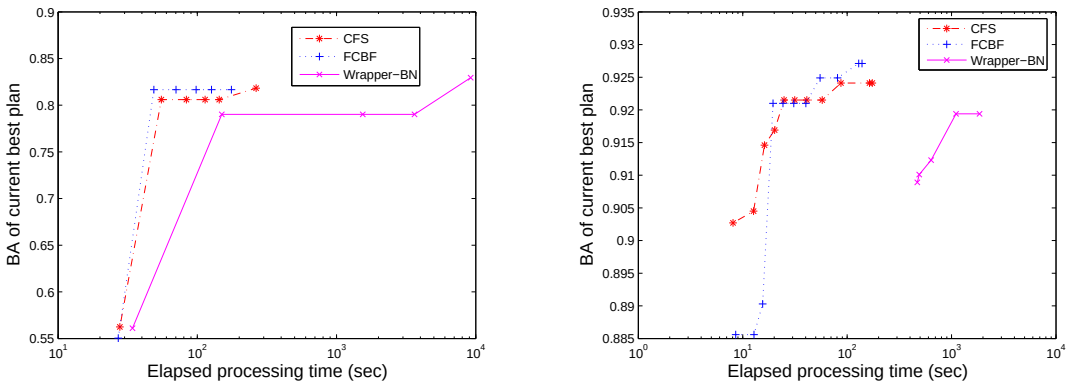

Figure 6.10: Comparing attribute-selection techniques: (a) Aging-variant-syn, (b) Aging-fixed-tb

Figure 6.10 and Figure 6.11 show that the increasing order in running times of the attribute selection techniques is FCBF < CFS << Wrapper (please note that the X-axis is in log scale). At the same time, FPS using FCBF and CFS can get comparable accuracies that are no worse than the accuracy achieved by FPS using Wrapper.

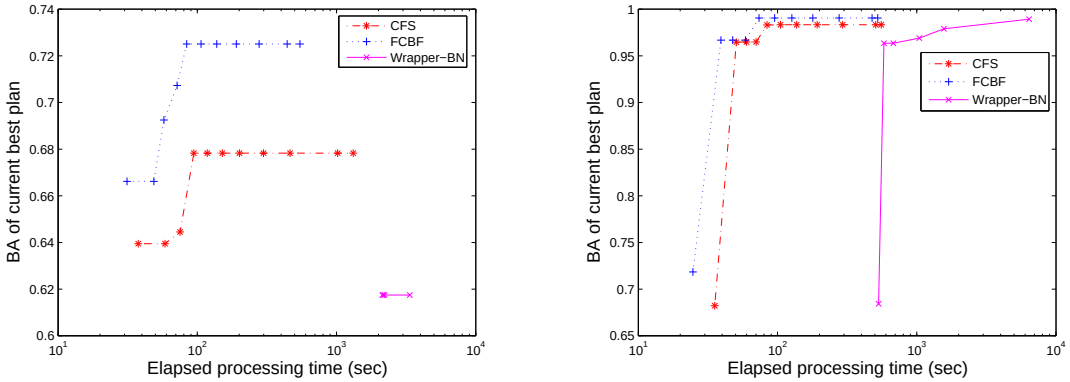

Figure 6.11: Comparing attribute-selection techniques: (a) Multi-large-tb, (b) Periodic-large-tb

6.6.5 Selecting a Synopsis

Table 6.3 shows the time to produce the best plan, and the corresponding BA, for FPS when using BN, CART, MLR, and SVM synopses. As a comparison point, we provide the best performance of RF synopses which were found to be one of the best synopses available today in a recent comprehensive study [18]. (More detailed comparisons with RFs are provided in Section 6.6.7.)

- The running times for SVM and RF are consistently worse than for BN/CART/MLR. The missing data points for SVM and RF are cases where Java runs out of heap space on our machine before the algorithms converge; showing the high memory overhead of these algorithms.

- BNs are competitive with all other synopses both in terms of accuracy and running time; we attribute this performance to the fact that once the right transformations are made, a reasonably-sophisticated synopsis can attain close to the best accuracy possible.

	FPS (BN)		FPS (CART)		FPS (MLR)		FPS (SVM)		RF	
Dataset	BA	Time	BA	Time	BA	Time	BA	Time	BA	Time
Aging-real	0.71	62.4	0.71	134.9	0.64	35.8	0.51	1948.7		
FIFA-real	0.87	28.8	0.85	36.6	0.84	201.4				
Periodic-small-tb	0.84	44.9	0.85	249.3	0.80	130.3			0.86	22339.7
Periodic-large-tb	0.98	71.98	0.98	167.7	0.97	92.6				
Aging-fixed-tb	0.91	27.5	0.93	56.8	0.89	145.3				
Multi-large-tb	0.72	99.7	0.73	197	0.71	100.8				
Multi-small-tb	0.91	53.2	0.91	49.7	0.85	19.1	0.86	482.3	0.91	933.2
Aging-variant-syn	0.82	14.2	0.81	109.4	0.80	24.2			0.85	3200.1
Complex-syn	0.99	130.1	0.99	506.4	0.99	134.7				

Table 6.3: Synopsis comparisons. The time in seconds to produce the best plan, and the corresponding BA, are shown. The missing data points are cases where Java ran out of heap space before convergence.

Figure 6.12 and Figure 6.13 show plots of the experimental results in Table 6.3. SVM and RF are not included in these figures because their running times are at least an order of magnitude longer than that of BN, MLR, and CART. Note that the running time in Table 6.3 corresponds to the time in seconds needed to reach a BA close to (within 1%) the maximum BA possible in each case; so the time in Table 6.3 do not always correspond to the maximum BA in Figure 6.12 and Figure 6.13. Recall that Fa's goal is to produce a reasonably-accurate plan quickly for a given query, balancing the tradeoff between accuracy and plan-selection time.

Figure 6.14 shows the experimental results with the Motes-real dataset.

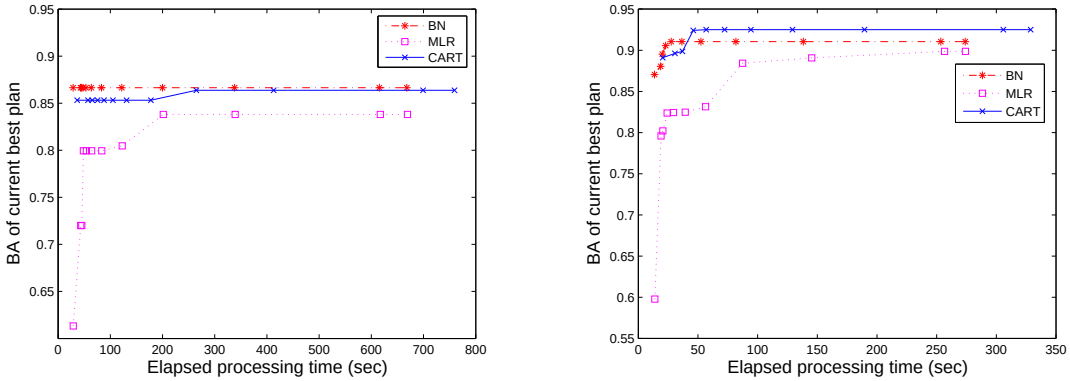

Figure 6.12: Comparing synopses: (a) FIFA-real, (b) Aging-fixed-tb

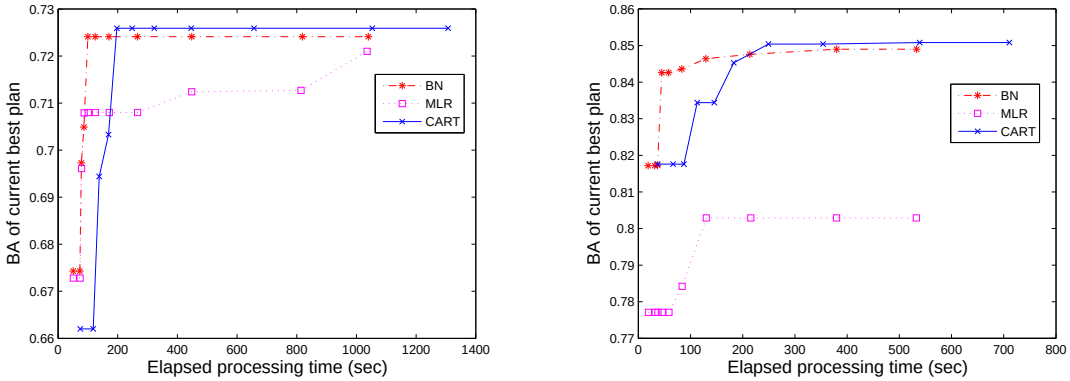

Figure 6.13: Comparing synopses: (a) Multi-large-tb, (b) Periodic-small-tb

6.6.6 Decision to Generate a Plan or Not

Our attempts to use the best CFS Score from a set of attributes Ω as an estimator of the best BA from Ω gave mixed results. Figures 6.9(b) and 6.7(b) plot the CFS Score and corresponding BA for a large set of randomly chosen attribute subsets from Multi-large-tb and Periodic-large-tb: CFS Score seems to be a reasonable indicator of BA for Multi-large-tb (note the almost linear relationship), but not so for Periodic-large-tb (note the points where CFS Score is reasonably high, but BA is far below the best). Figure 6.15(a) shows the performance of CFS-Score-based pruning, for k=1 and k=5, for Periodic-large-tb. While pruning does sometimes reduce the

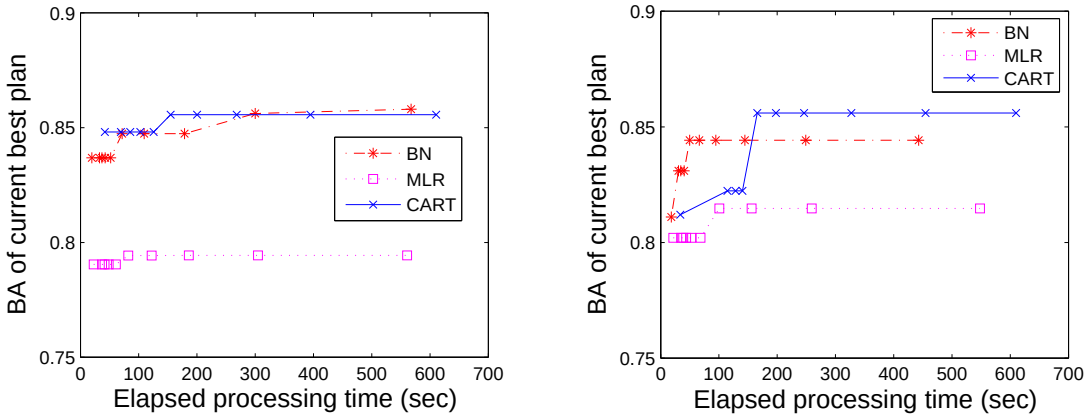

Figure 6.14: Comparing synopses: (a) Forecasting humidity in Motes-real, (b) Forecasting light in Motes-real

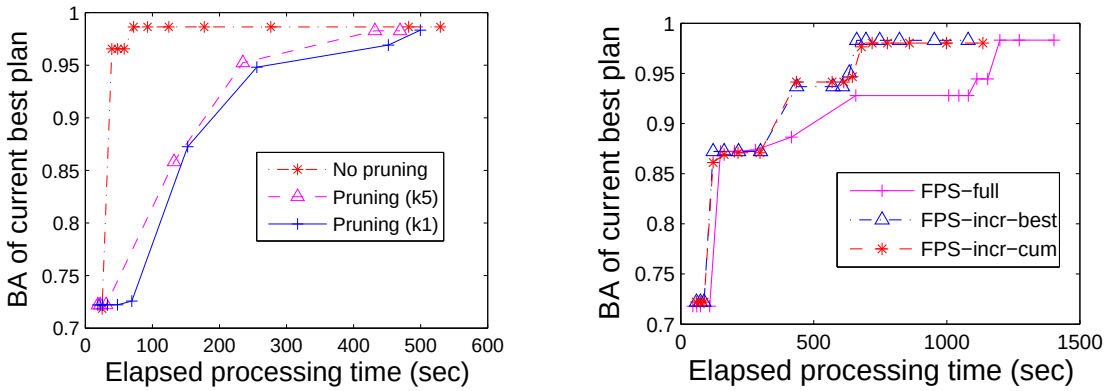

Figure 6.15: (a) CFS-Score-based pruning (Periodic-large-tb), (b) Improvements with sharing (Periodic-large-tb)

total processing-time considerably, it does not translate into faster convergence; in fact, convergence can be delayed significantly as seen in Figure 6.15(a).

Figure 6.16 and Figure 6.17 show that in most cases CFS-Score-based pruning does not help reducing the time to convergence, although the prediction accuracy is sacrificed.

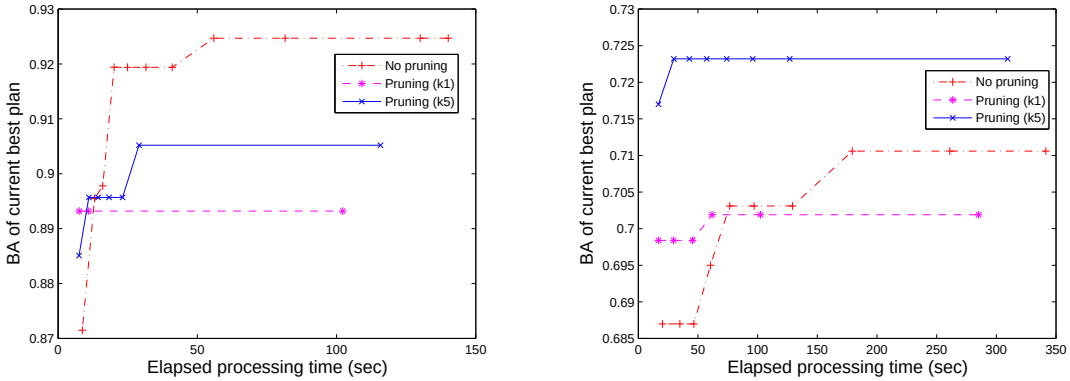

Figure 6.16: CFS-Score-based pruning: (a) Aging-fixed-tb, (b) Aging-real

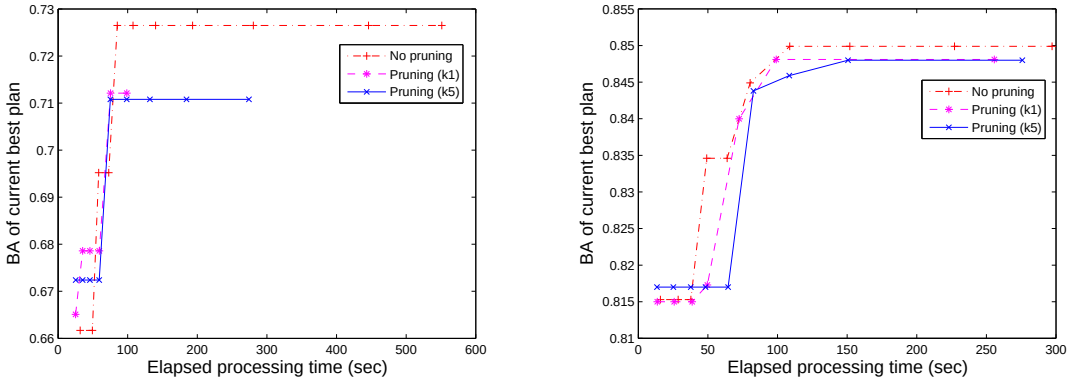

Figure 6.17: CFS-Score-based pruning: (a) Multi-large-tb, (b) Periodic-small-tb

6.6.7 Effect of Sharing

Figure 6.15(b) shows the consistent trend we observed across all datasets: computation sharing improves the speed of convergence of FPS significantly, without any adverse effect on accuracy. Note that sharing can be done across chunks in a ranked list as well as across the much larger chunks of attributes considered by successive iterations of FPS's outer loop. To show both effects, we set $\Delta = 30$ instead of the default 90 in Figure 6.15(b), so multiple iterations of the outer loop are required to converge. We found no significant difference between FPS-incr-cum and FPS-incr-best.

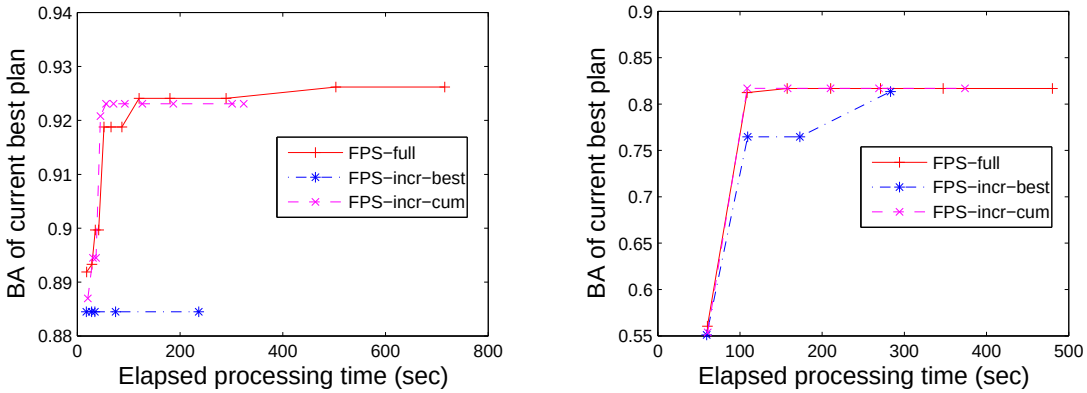

Figure 6.18: Improvements with sharing: (a) Aging-fixed-tb, (b) Aging-variant-syn

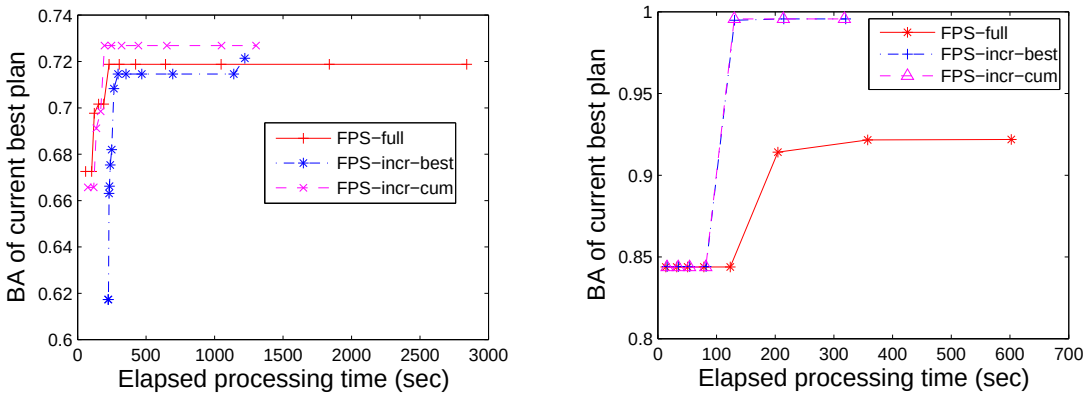

Figure 6.19: Improvements with sharing: (a) Multi-large-tb, (b) Complex-syn

Figure 6.18 and Figure 6.19 show that the computation sharing within chunks and across chunks can reduce the time to convergence without any significant loss in accuracy.

6.7 Making FPS Concrete

The trends observed in our experimental evaluation point to reasonable defaults for each step of FPS in Figure 6.3, to find a good plan quickly from Φ for a one-time

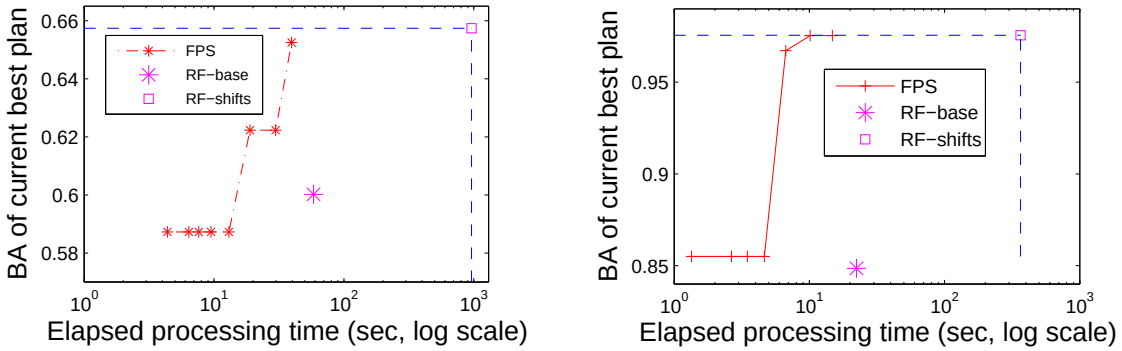

Figure 6.20: FPS Vs. State-of-the-art synopsis (RF): (a)Aging-real, (b) Complex-syn

Forecast(D, X_i, L) query:

- Use LCC for ranking attributes.
- Traverse the ranked list in multiple overlapping chunks, with increasing increments in chunk-size up to a point.
- Generate a good plan from each chunk considered.
- Use FCBF or CFS for attribute selection.
- Build BN synopses only.
- Use FPS-incr-cum for computation sharing.

A recent comprehensive study [18] found RFs to be one of the best synopses available today. Figure 6.20, Figure 6.21 and Table 6.4 compare the above concrete instantiation of FPS with:

- *RF-base*, which builds an RF synopsis on the original input dataset. Intuitively, RF-base is similar to applying today's most-recommended synopsis on the input data. Note that RF-base does not consider transformations.

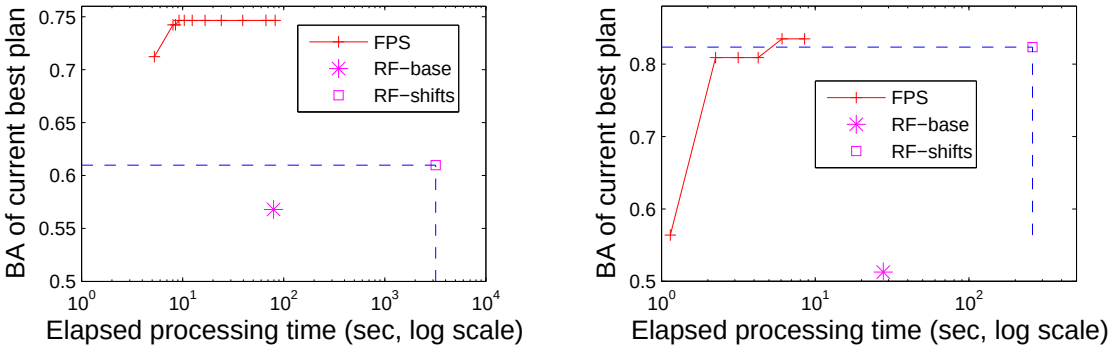

Figure 6.21: FPS Vs. State-of-the-art synopsis (RF): (a) Multi-large-tb, (b) Aging-variant-syn

	FPS (BN)		RF-shifts		RF-base	
Dataset	**BA**	**Time**	**BA**	**Time**	**BA**	**Time**
FIFA-real	0.86	10.8	0.79	2357.9	0.76	112
Aging-fixed-tb	0.91	13.6	0.91	1660.6	0.91	45.1
Periodic-large-tb	1	13.9	0.52	2839.8	0.52	69.1

Table 6.4: FPS Vs. RF; The datasets had to be scaled down to get RF to run within reasonable time

- *RF-shifts*, which builds an RF synopsis on the input after applying all Shift(j, δ) transformers, $1 \leq j \leq n$ and $-\Delta \leq \delta \leq 0$.

These results in Figure 6.20, Figure 6.21 and Table 6.4 demonstrate FPS's superiority over RF-base and RF-shifts in finding a good plan quickly for one-time forecasting queries. Furthermore, FPS's ranking-based approach and scalable attribute-selection algorithms make it robust to noisy and redundant attributes in the input.

In Section 6.8, we extend FPS to consider a much larger space of transformers than Φ. This extension is based on two main observations:

1. FPS handles a large space of possible shift transformations by first applying all these transformations to create the corresponding attributes, and then applying efficient attribute traversal and selection techniques to find good

transformations quickly. The same technique can be used to incorporate more transformers, e.g., we consider *log*, *wavelet*, and *difference* transformers.

2. FPS can first apply simple transformers (e.g., Shift, π) and synopses (e.g., BN) to identify the small subset Ω of original and new attributes that contribute to good plans. Then, more complex transformers and synopses can be applied only on Ω.

6.8 More Transformations

In addition to *Shift* and π, we consider more transformers such as *Wavelet*, *Log*, and *Difference* to enhance *FPS*'s ability to find a good execution plan quickly. Figure 6.22 shows the extension of *FPS* with the *Wavelet* transformer; Line 1 is a new line. The other transformers can be applied in a similar way. All the experiments in this section use the Complex-noisy-syn dataset.

6.8.1 Wavelet Transformer

The transformer *Wavelet(X, level)* applies wavelet transform to a time series X, and generates $level + 1$ time series, among which one is an approximation series (low-frequency component of X) and the others are detail series (high-frequency component of X) [64].

Notice that *Wavelet* transformers have the potential to improve overall accuracy and reduce the time to convergence. We found MLR synopses to get the most benefit from *Wavelet* transformers, as seen in Figure 6.23.

Algorithm *Extended Forecasting Plan Search*
Input: *Forecast*$(D(\Gamma, X_1, \ldots, X_n), X_i, L)$ query
Output: Forecasts and accuracy estimates are output as FPS runs. FPS is terminated when a forecast with satisfactory accuracy is obtained, or when lead-time runs out;
/* the newly added line to *FPS* */
1. Apply *Wavelet* transformers to the attributes $X_1, \ldots, X_n$ in D with $level = 2$, and generate a new dataset D';
2. $l = 0$; /* current iteration number of outer loop */
BEGIN OUTER LOOP
3. Attribute set $Attrs = \{\}$; /* initialize to an empty set */
4. $l = l + 1$; /* consider Δ more shifts than in last iteration */
5. **FOR** $j \in [1, n]$ and $\delta \in [-l\Delta, 0]$
   Add to *Attrs* the attribute created by Shift(j, δ) on D';
6. Generate attribute $Z = X_i(\tau + L)$ using Shift(X_i, L) on D';
7. Let the attributes in *Attrs* be $Y_1, \ldots, Y_q$. Rank $Y_1, \ldots, Y_q$ in decreasing order of relevance to Z; /* Section 6.5.2 */
/* Traverse the ranked list of attributes from start to end */
   BEGIN INNER LOOP
8. Pick next chunk of attributes from list; /* Section 6.5.3 */
9. Decide whether a complete plan should be generated using the current chunk of attributes; /* Section 6.5.5 */
10. **IF Yes**, find the best plan p that builds a synopsis using attributes in the chunk. If p has the best accuracy among all plans considered so far, output the value forecast by p, as well as p's accuracy estimate; /* Section 6.5.4 */
   END INNER LOOP
END OUTER LOOP

Figure 6.22: Extended Plan Selection Algorithm

6.8.2 Log Transformer

The transformer *Log(X)* transforms a time series X into a new time series X', where $X'(i) = log(X(i))$ $(1 \leq i \leq m)$.

The experimental results shown in Figure 6.24 demonstrate that all the three synopses benefit from the *Log* transformers, because the *Log* transformers can help reduce the nonlinearity in the patterns useful for forecasting, which makes the patterns easy to be captured by the synopses.

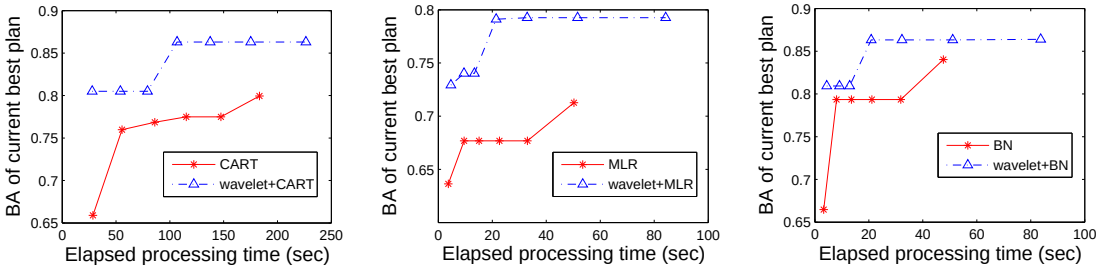

Figure 6.23: Using Wavelet transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses

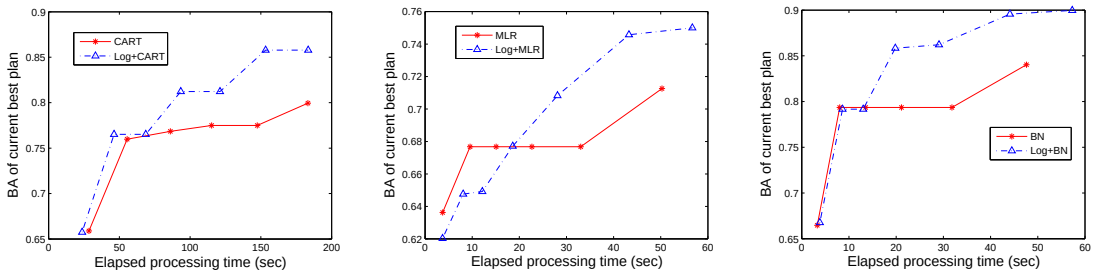

Figure 6.24: Using Log transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses

6.8.3 Difference Transformer

The transformer *Difference(X)* transforms a time series X into a new time series X', where $X'(i) = X(i) - X(i-1)$ $(2 \leq i \leq m)$.

The experimental results shown in Figure 6.25 indicate that the extended FPS incorporating *Difference* transformers can gain some improvement in the prediction accuracy with BN, MLR, and CART synopses, because the *Difference* transformers can bring out temporal trends in the time series that contribute to making accurate prediction.

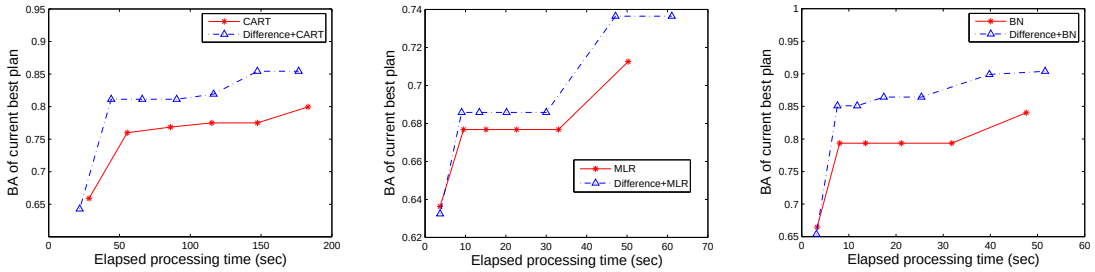

Figure 6.25: Using Difference transformers in extended FPS (a) with CART, (b) with MLR, (c) with BN synopses

6.9 Continuous Forecasting Queries

So far we considered one-time forecasting of the form *Forecast*(D, X_i, L) where the dataset D is fixed for the duration of the query. We now extend our techniques to handle continuous queries of the form *Forecast*$(S[W], X_i, L)$ where S is a data stream of relational tuples, and W is a sliding window specification over this stream.

Semantics: The semantics of a *Forecast*$(S[W], X_i, L)$ query is a straightforward extension of the one-time semantics from Section 6.2. (This design principle comes from the CQL continuous query language [7] on which our extensions are based; see example query Q_2 in Section 6.1.) The result of a *Forecast*$(S[W], X_i, L)$ query at time τ is the same as the result of *Forecast*(D, X_i, L), where D is the time-series dataset containing the window W of data in stream S at τ. As time advances, this window of data shifts, and a continuous stream of forecasts will be produced in the result of *Forecast*$(S[W], X_i, L)$.

Example 6.9.1. *Consider the example continuous query* Forecast*(Usage[Range 5 days],C,1 day) that treats the Usage data in Figure 6.1(a) as a stream. CQL syntax is used to specify a sliding window containing tuples in the last 5 days. Thus, for example, on Day 10, a forecast will be output for C for Day 11, based on the data (window) for Days 6-10.*

6.9.1 FPS-Adaptive (FPS-A)

We could process a *Forecast*$(S[W], X_i, L)$ query by running FPS on the initial window of data to get the best plan p, and then keep producing new forecasts using p as the window of data changes. However, this technique will produce inaccurate forecasts as the predictive patterns in the data change over time. Another option is to rerun FPS from scratch to generate the current best plan whenever the window changes; which is inefficient.

A good algorithm for processing *Forecast*$(S[W], X_i, L)$ should use the same plan that FPS would find for each window of data, but incur minimal cost to maintain this plan as the window slides. FPS-A (FPS-Adaptive) is our attempt at such an algorithm. FPS-A exploits the structure (Figure 6.4) and defaults (Section 6.7) we established for FPS. FPS-A works as follows:

- The ranked list of all enumerated attributes is maintained efficiently. Because FPS uses LCC for ranking, FPS-A gets two useful properties. First, LCC can be updated incrementally, and batched updates make this overhead very low. Second, recent work [97] shows how tens of thousands of LCCs can be maintained in real-time; we haven't used this work yet since batched updates give us the desired performance.

- If there are significant changes to the ranked list, then the overall "plan structure" is updated efficiently (e.g., the best attribute subset for a chunk may have changed, giving a new best plan).

At all points of time, FPS-A maintains a *reference rank list* R^r composed of k chunks $C_1^r \subset C_2^r \subset \cdots \subset C_k^r$, the best plans $p_1^r, \ldots, p_k^r$ for these chunks, and the overall best plan p_{best}^r. The current p_{best}^r is used for producing forecasts and accuracy

Algorithm *Fa's Adaptive Maintenance of Plans for Forecasting Queries (FPS-A)*
Input: A β-sized batch of tuples inserted/deleted from the sliding window for a *Forecast*$(S[W], X_i, L)$ query. R^r, $C_1^r, \ldots, C_k^r$, $p_1^r, \ldots, p_k^r$, p_{best}^r are the current reference values;
Output: The reference values will be updated if required;

1. Do batched update of all LCC values to get the new ranked list R and its chunks $C_1, \ldots, C_k$;
2. For attribute Y, let $LCC(Y,Z)^r$ and $LCC(Y,Z)$ be the reference and current LCC between Y and $Z = X_i(\tau + L)$;
3. For attribute Y, $Level(Y)^r = i$ if $Y \in C_i^r$ and $Y \notin C_{i+1}^r$;
4. For attribute Y, $Level(Y) = i$ if $Y \in C_i$ and $Y \notin C_{i+1}$;

// Attributes that gained LCC significantly and moved up levels

5. $Gainers$ = Set of Y such that $Level(Y)^r > Level(Y)$ AND $|LCC(Y,Z)^r - LCC(Y,Z)| > \alpha$);

// Attributes that lost LCC significantly and moved down levels

6. $Losers$ = Set of Y such that $Level(Y)^r < Level(Y)$ AND $|LCC(Y,Z)^r - LCC(Y,Z)| > \alpha$);
7. **IF** ($|Gainers| = 0$ AND $|Losers| = 0$)
8. No changes are required for reference values. Exit;
9. **FOR** (i going from 1 to number of chunks k) {
10. **IF** (there exists a $Y \in Losers$ such that $Y \in p_i^r$) {
11. Regenerate p_i^r using the attribute subset in p_{i-1}^r and the attributes in $C_i - C_{i-1}$, and using BN synopsis;
12. } /* END IF */
13. **ELSE** {
14. $NewAttrs$ = Set of attributes $Y \in Gainers$ such that $Y \in C_i$ and $Y \notin p_i^r$;
15. If $|NewAttrs| > 0$, then regenerate p_i^r using $NewAttrs$ and current attribute subset in p_i^r, and BN synopsis;
16. } /* END ELSE */
17. } /* END FOR */
18. Update the reference rank list, chunks, and best plan;

Figure 6.26: FPS-Adaptive (FPS-A)

estimates at any point of time. The reference values are initialized by running FPS on the initial window of tuples. Figure 6.26 shows how FPS-A maintains these values as the window slides over time. FPS-A uses two user-defined thresholds: α for detecting significant changes in LCC values, and β that determines the batch size for updating LCC values. The notation $Level(Y)^r$ is used to denote the number of the largest chunk to which attribute Y belongs in R^r. That is, $Level(Y)^r = i$ if $Y \in C_i^r$ and $Y \notin C_{i+1}^r$.

After updating LCC scores, FPS-A computes two sets of attributes: *Gainers* (*Losers*) that moved up (down) one or more levels in the ranked list, and whose LCC values changed significantly. If there are no *Gainers* or *Losers*, then the current reference values are fine as is. If one or more attributes in the attribute subset of plan p_i^r for the ith chunk are in *Losers*, then p_i^r may have become suboptimal. So, FPS-A regenerates p_i^r using the best attribute subset for the previous chunk and the attributes at this level (recall Section 6.5.6). Similarly, if new attributes have moved into the ith chunk, then p_i^r can be updated efficiently to (possibly) include these attributes. FPS-A builds BN synopses (only), and updates the synopsis in the current best plan after every batch of tuples are processed. While this synopsis can be updated incrementally, we have got equally efficient performance from simply rebuilding the synopsis on the few (< 10) attributes chosen by attribute selection.

Notice that FPS-A reacts to changes in the ranked list only when attributes transition across levels. (The threshold α filters out spurious transitions at chunk boundaries.) This feature makes FPS-A react aggressively to changes at the head of the ranked list—where attributes have a higher chance of contributing to the best plan—and be lukewarm to changes in the tail. FPS-A gets this feature from the fact that successive overlapping chunk sizes in FPS increase in a geometric progression.

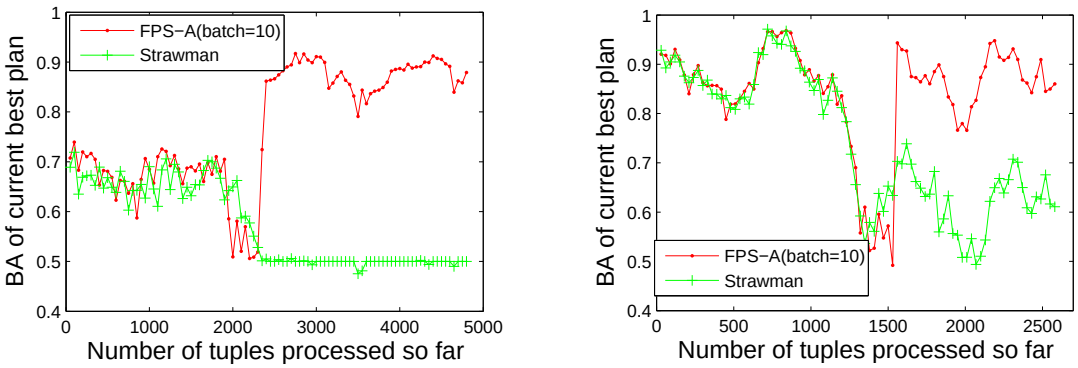

Figure 6.27: Adaptivity and convergence properties

We now report experiments where we evaluate FPS-A on the three metrics for adaptive query processing defined in [9]: (i) speed of adaptivity (how quickly can FPS-A adapt to changes in stream properties?), (ii) convergence properties (when stream properties stabilize, can FPS-A produce the accuracy that FPS gives for the stable properties?), and (iii) run-time overhead (how much extra overhead does FPS-A incur in settings where stream properties are stable, and in settings where properties change over time?).

We consider continuous queries with a 200-minute window on the input stream. $\alpha = 0.1$ for FPS-A. Figures 6.27(a) and (b) show FPS-A's speed of adaptivity and convergence properties for two input streams. The input stream used in Figure 6.27(a) concatenates the Aging-fixed-tb and Multi-large-tb datasets from Table 6.2 to create a scenario where predictive patterns change because of a change in the workload on the monitored system. The second stream is more adversarial where we randomly permute the attributes in the Multi-small-tb dataset over time.

We compare FPS-A's performance with that of *Strawman* which runs FPS on the initial window of data to choose the initial best plan. Like FPS-A, Strawman updates the BN synopsis in the current best plan after every batch of tuples have been processed. However, unlike FPS-A, Strawman never updates the set of attributes

β	Without property changes			With property changes		
	FPS-A	**Strawman**	%	**FPS-A**	**Strawman**	%
5	32.02	32.83	2.53	33.41	36.69	9.82
10	61.17	62.24	1.74	66.37	60.73	-8.5
15	89.5	90.99	1.56	81.87	88.33	7.89
20	116.27	118.17	1.64	95.50	112.82	18.13
25	129.34	132.10	2.13	108.54	126.24	16.30

Table 6.5: Run-time overhead. Values for FPS-A and Strawman are tuples processed per second. % is FPS-A's degradation relative to Strawman

selected. The X-axis in Figure 6.27 shows the number of tuples processed so far, and the Y axis show the estimated accuracy of the current best plan. Note that FPS-A adapts fairly quickly in Figure 6.27(a) when the stream properties change $X = 2000$, but Strawman does not because the set of predictive attributes has changed. Comparing the accuracies before and after the change in Figure 6.27(a) with the respective individual best accuracies for the Aging-fixed-tb and Multi-large-tb datasets in Table 6.3, shows that FPS-A indeed finds plans comparable to FPS. The behavior in Figure 6.27(b) is similar.

Table 6.5 shows the extra overhead for FPS-A over Strawman for the adversarial stream. When there are no changes in stream properties, FPS-A's extra overhead is limited to LCC maintenance; which is around 2%. Even when stream properties change and FPS-A has to update attribute subsets, the worst-case overhead is $< 20\%$. Note that FPS-A's overhead can be lower than that of Strawman if FPS-A finds a smaller attribute subset.

6.10 Related Work

Forecasting based on historical data is useful in many domains. The motivation of making systems and DBMSs easier to manage (ideally self-managing) has driven

recent interest in forecasting. References [65, 67] show how various synopses can successfully forecast performance problems and system failures in real enterprise environments. Transformations are selected manually in [65, 67]. *Muscles* [93] uses MLR synopses to process continuous forecasting queries by maintaining the synopses incrementally over streams. Unlike FPS-A, transformations are not maintained over time in *Muscles*. Our work differs from previous work on forecasting in two fundamental ways: (i) we consider declaratively-specified forecasting queries, and develop automated algorithms for choosing plans composed of transformation, prediction, and synopsis-building operators; (ii) our algorithms balance accuracy against the time to generate forecasts.

The design of FPS has been influenced by some recent studies that compare various synopses and transformations. Reference [18] reports a large-scale empirical comparison of the accuracy (processing-time is not considered) of ten synopses, including regression, BN, CART, SVM, and RF. This study finds RF to be one of the best synopses available. Reference [87] is a similar study, but with a much smaller scope—only BN and CART synopses are considered. However, reference [87] evaluates processing-time and considers some attribute-selection algorithms. Reference [45] is a comprehensive evaluation of attribute-selection algorithms, with findings similar to ours. None of this work considers algorithms for learning a good combination of synopses and transformations.

Synopses are used widely in database and data stream systems, e.g., for approximate query answering, acquisitional query processing, prediction of completion times of business intelligence workloads, and query optimization, e.g., [4, 55]. This class of work focuses on conventional SQL/XML-style queries. Synopses are being applied to system management, e.g., [65, 67]. Self-tuning capabilities of commer-

cial DBMSs have been enhanced significantly, e.g., [72]. Currently, all these systems take a predominantly reactive approach to tuning and diagnosis, which can be made proactive with Fa's accurate and automated forecasting. Our work can benefit from recent work on maintaining synopses and correlation metrics over high-speed data streams.. For example, [97] develops techniques to maintain a large number of LCCs in real-time.

There is work in the data-mining literature that is related to Fa, e.g., work on extracting rules from time-series data [28], mining sequential patterns [3], and inter-transaction association rules [42]. The focus here is on finding interesting/surprising *local* relationships or patterns in the data. Some of these patterns can be useful for forecasting.

6.11 Summary

In this chapter, we described how users and applications can pose declarative forecasting queries in Fa — both one-time and continuous queries — and get forecasts in real-time along with accuracy estimates. We described the FPS algorithm to process one-time forecasting queries using plans composed of operators for transforming data, building statistical models from data, and doing inference using these models. We also described the FPS-A algorithm that adapts plans for continuous forecasting queries based on the time-varying properties of input data streams. Our experimental evaluation demonstrates the effectiveness and scalability of both algorithms.

Fa combines data-management techniques with statistics and machine-learning to make three important contributions: (i) automated algorithms for choosing forecasting plans composed of transformation, prediction, and synopsis-learning opera-

tors; (ii) plan-selection algorithms that balance result accuracy against the time to generate results; and (iii) extensive evaluation of forecasting plans using synthetic, testbed, and real datasets; showing 10x improvement over using (just) state-of-the-art synopses.

Chapter 7

Conclusions

Networked computing systems are increasingly hard to manage due to their scale, complexity, and dynamic characteristics. Manual management of these systems based on rules-of-thumb or custom scripts is often tedious and error-prone. Recent progress in monitoring tools enables system administrators to collect fine-grained data about system activity to improve system efficiency and to avoid costly system downtime. However, the monitoring data collected from production systems is massive in size and noisy; which makes it hard for system administrators to fully utilize this data for effective system management.

This dissertation described the Fa data-management platform where system administrators can express observational queries and system-management queries over the monitoring data in a declarative manner. For a given query, Fa automatically finds a reasonably-good execution plan quickly and executes the plan to generate query result with confidence estimates and supporting evidence about the result.

With Fa, the tedious task of maintaining system performance can be simplified as a sequence of management queries: (i) Detect or predict system performance problems with forecasting queries; (ii) Find the cause of performance problems with diagnosis queries; (iii) Recommend changes to system configurations to resolve the diagnosed problems with tuning queries, and validate the recommendation on the

production system.

Fa processes diagnosis queries in two phases:

(i) A database of problem signatures is constructed from numeric and categorical system monitoring data to distill the essential properties of already diagnosed problems. In Phase I of diagnosis query processing, the problem to diagnose is matched against the signature database, with confidence estimates about the matches. High-confidence matches are valuable since they enable leveraging past diagnostic information associated with the matched signatures. Empirical evaluation validated that our signature construction and matching techniques are robust to noise that is common in system data from production systems; the drop in accuracy of matching undiagnosed problems with the signature database is slower than competing techniques as noise in system data increases. Also the confidence estimates about matches are reliable — diagnosis based on high-confidence matches of signatures is often more accurate than that based on low-confidence matches.

(ii) If there is no high-confidence match from Phase I for a given problem represented in the diagnosis query, which may indicate rare or new types of problems, it is necessary to trigger Phase II of diagnosis query processing for further investigation. We presented an anomaly-based clustering technique that groups data about normal system activity to build system baselines, based on how it deviates from the problem data in diagnosis. The deviation is characterized by a few succinct attribute sets, although the system data is of high dimensionality, to pinpoint the cause of problem. Empirical evaluation validated that anomaly-based clustering outperforms conventional clustering techniques such as K-means and hierarchical clustering in terms of

both efficiency and accuracy of diagnosis query results.

To decide whether to trust the query result from Phase I (if not, it needs to trigger the more expensive Phase II to continue query processing), we presented a technique that automatically sets the threshold on the confidence of matches with the signature database, based on user expectation of accuracy from Phase I of diagnosis query processing.

Since Phase II has a harder task to solve than Phase I for diagnosis query processing, Phase II requires more human efforts but may have a less accurate query result than Phase I. However, Phase I requires a considerable number of instances of diagnosed problem types to construct a high-quality signature database. The available system data may not meet this requirement. We presented an active-learning technique that formulates annotation queries asking for diagnosis results of some carefully picked undiagnosed problems. Once the picked problems are diagnosed, they are used to update the signature database. Empirical evaluation validated that the active-learning technique is able to maximize the value of human efforts involved to process annotation queries while improving the quality of the signature database to a desired level.

Fa employs an experiment-driven approach, called iTuned, to process tuning queries in the context of database configuration parameters. Gaussian process regression models are learned based on system monitoring data (e.g., from experiments) to represent the response surface of a performance metric with regard to system configuration parameters. Confidence estimates are produced around the predicted performance at each hypothetical setting. iTuned has two interleaved phases: (i) a planning phase that employs a novel Adaptive Sampling algorithm to plan experiments, and (ii) an execution phase that conducts the planned ex-

periments with the .eX framework. Adaptive Sampling estimates the utility of candidate experiments using the response surface constructed from system monitoring data collected so far; and picks the experiment with the maximum expected utility to conduct next. With a sensitivity analysis technique, iTuned is able to visualize and rank configuration parameters in terms of their performance impact, thus quickly eliminating parameters with minimal performance impact to simplify the tuning query. Furthermore, iTuned can quickly identify regions of potentially high-performance settings. Empirical evaluation validated that the Adaptive Sampling technique balances well between *exploration* of under-sampled regions and *exploitation* of promising regions in the response surface. iTuned supports conducting the planned experiments in a production environment through a cycle-stealing paradigm while ensuring near-zero overhead on the production workload. To make Fa a practical platform for processing tuning queries, iTuned has incorporated several scalability features, including planning and conducting parallel experiments, early stopping of low-utility experiments, and workload compression. Empirical evaluation validated that iTuned outperforms existing tuning techniques, in terms of both the time needed for processing the tuning query and the system performance achieved from the result of the tuning query.

Rather than being reactive, Fa enables proactive system management by supporting forecasting queries over system data that predict future system performance and identify potential system problems automatically. For one-time forecasting queries, we presented a plan selection algorithm that automatically searches in the space of execution plans. Each execution plan consists of a sequence of data transformation operators and synopsis learning and prediction operators. The plan selection algorithm converges to fairly-accurate plans quickly by running as few plans

as possible. For continuous queries over data streams, we presented an adaptive plan selection algorithm that automatically adapts to the time-varying properties of system data. Empirical evaluation validated that the plan selection algorithms for processing both one-time and continuous queries strike a good balance between maximizing the accuracy of the forecasting query result and minimizing the time required to process the forecasting query.

We have evaluated the Fa platform with monitoring data collected from database-backed multitier services, and with synthetic data that models the noisy nature of monitoring data from production systems. The evaluation showed that Fa's query plan selection and execution strategies provide actionable information for system management automatically, accurately, and efficiently. Critical features like reliable confidence estimates, robustness to noise, and providing supporting evidence for query results make Fa a practical and useful platform.

Chapter 8

Future Work

To improve the applicability and practicality of Fa as a platform to simplify system management, we think the following extensions are worth exploring:

- There is room for further improvements to various components of the current Fa platform. The query interface (see Figure 2.4) can be extended to handle new requirements that arise from system management tasks. For instance, the tuning queries can be adapted to consider more than one high-level performance metric for optimization. Furthermore, the query processing techniques in Fa can be improved in terms of both the accuracy of query results and the efficiency to produce the query results.

- Our evaluation of Fa's query processing techniques was performed predominantly with database-backed Web service systems. It would be valuable to evaluate Fa with other types of computing systems or even a system from other domains (e.g., forecasting for financial services) to make the Fa platform have wider applicability.

- In addition to the structured system data (in the format of time-series) we consider in this dissertation, it is important to leverage extensive semi-structured and unstructured system data to improve the accuracy of Fa's query process-

ing techniques. The new data types require new data representation models in the architecture of Fa (see Figure 2.4). Also, system data typically comes from distributed monitoring points, raising the need for a data representation model that facilitates distributed processing of system-management queries.

- Fa's query processor needs an explicit interface for inputting domain knowledge. System administrators may provide domain knowledge with regard to a specific management task, e.g., hints or rules to consider a particular synopsis type or parameter settings for processing a forecasting query.

- To push one step closer to realizing the vision of autonomic computing, it is necessary to add a layer in Fa to support policy-based system management. This layer is on top of the current query interface, and allows users to express high-level management policies to meet the requirements of SLOs. This layer needs to contain a core component to decide: (i) Which system-management queries to be issued through the query interface? And (ii) how to consume the query results returned through the query interface to decide the next action?

8.1 Multi-Goal Experimental Design

Fa's techniques for processing tuning queries assume that only one high-level performance metric is of interest to optimize. For instance, a tuning query may only aim to maximize the system throughput. A more complex case is to optimize one performance metric while satisfying constraints on other metrics. For instance, system administrators may need to maximize system throughput while keeping average request response time under a certain threshold. Fa's Adaptive Sampling algorithm for processing tuning queries needs to be adapted to balance exploration

and exploitation of response surfaces in the multi-goal optimization setting.

8.2 Evaluation with Other Systems

Fa's query processing techniques were mainly evaluated with data collected from three-tier Web service systems. Since Fa does not make any assumption about the managed system, Fa is expected to be generally applicable for managing other computing systems. It would be valuable to evaluate Fa's query processing techniques on a broad range of systems. Such evaluation will motivate Fa to deal with other system-management challenges and new requirements on management-query types and their processing techniques.

8.3 System Data Complexity

Fa's current query processing techniques target structured system data in the format of time-series. There is an abundance of semi-structured data such as stack traces [14] and execution traces [78], and unstructured system data such as bug reports and error messages. It is an important next step to incorporate the rich semi-structured and unstructured system data into the Fa platform to get a holistic view of system behavior, thus improving the quality of Fa's query results. At the same time, it is important to balance the benefit of collecting more system data and the overhead associated with collecting more data. This balance needs to be adjusted dynamically by Fa based on system states and the system-management queries to process. It is a great challenge to represent and query heterogenous (i.e., structured, semi-structured, and unstructured) system data in a seamless and efficient way. Moreover, system data often arrives as data streams in a distributed

fashion; so online and distributed processing of system-management queries is desired but very challenging: (i) How to automatically find and adapt a good execution plan for a given system-management query as the managed system evolves? And (ii) how to execute the plan efficiently in a distributed fashion with reasonably-accurate query results?

8.4 Incorporating Domain Knowledge

Domain knowledge is an important source of information for system management. Fa needs to provide an interface for taking domain knowledge. There are several ways to incorporate domain knowledge into query processing; for example, one way is to ask for feedback from administrators. However, the efficiency of query processing is limited to be at the human timescale instead of at the machine timescale. Another way to incorporate domain knowledge is through data representations such as synopsis. For example, a synopsis represented as a Bayesian network can incorporate domain knowledge by specifying which system components should be connected or disconnected during the process of learning model structure [88]. Synopsis learning enhanced by domain knowledge often requires less data and has better prediction accuracy [39]. As another example, a synopsis represented as queueing networks [54] can be constructed based on the knowledge of the internals of a system component.

8.5 Policy-based System Management

Fa's capability of processing system-management queries automatically and efficiently provides a basis to support policy-based system management, which can further simplify the job of system administrators. Fa needs to provide a simple and

intuitive interface for specifying management policies on top of the current query interface. Fa also needs a core component to make action decisions. The life-cycle of system management starts with an administrator-specified policy, e.g., the average request response time must be less than 5 seconds. System-management queries are automatically submitted through Fa's query interface for problem detection and forecasting, diagnosis, and tuning. The decision-making component picks an action that may apply the recommended tuning strategy to the managed system through system knobs, e.g., changing system configuration parameters online or micro-rebooting a system component to resolve a diagnosed problem. Depending on the actions supported by the managed system, the decision-making component can structure the applicable actions at different system levels. For instance, if micro-rebooting a system component does not solve the diagnosed problem, the next action is to restart the entire system. More importantly, when there are multiple applicable actions, Fa needs a heuristic search algorithm to find the most effective and cost-efficient action. This search algorithm requires a model to estimate the performance impact of each action and a cost model to estimate the cost associated with the action. The decision-making component then closes the loop of performance management with support of forecasting queries, diagnosis queries, and tuning queries. A more visionary step is to employ policy-based system management to make systems self-healing.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. of the 2005 Conf. on Innovative Data Systems Research*, Asilomar, CA, January 2005.

[2] C. C. Aggarwal. A framework for change diagnosis of data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 575–586, 2003.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Intl. Conf. on Data Engineering*, 1995.

[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] E. L. Allwein, R. E. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. In *Proc. of the Intl. Conf. on Machine Learning*, 2000.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, page 665, June 2003.

[7] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[8] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup Web site. Technical report, HPL-1999-35R1, 1999.

[9] S. Babu. *Adaptive Query Processing for Data Stream Management Systems*. PhD thesis, Stanford University, Sept. 2005.

[10] S. Babu, S. Duan, and K. Munagala. Processing diagnosis queries: A principled and scalable approach (poster). In *Proc. of the Intl. Conf. on Data Engineering*, 2008.

[11] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proc. of the Intl. Conf. on Autonomic Computing*, 2005.

[12] N. Bolshakova and F. Azuaje. Cluster validation techniques for genome expression data. *Signal Processing*, 83(4), 2003.

[13] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[14] M. Brodie, S. Ma, G. M. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Proc. of the Intl. Conf. on Autonomic Computing*, 2005.

[15] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for Internet applications. In *Proc. of IEEE Workshop on Internet Applications*, 2003.

[16] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proc. of Symp. on Operating Systems Design and Implementation*, pages 31–44, 2004.

[17] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX 2004 Annual Technical Conference*, pages 15–28, 2004.

[18] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proc. of the Intl. Conf. on Machine Learning*, Jun 2006.

[19] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *Proc. of the Intl. Conf. on Neural Information Processing Systems*, 2000.

[20] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proc. of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2002.

[21] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[22] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing SQL workloads. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[23] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1998.

[24] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. of the Intl. Conf. on Autonomic Computing*, 2004.

[25] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of the Proc. of Symp. on Operating Systems Design and Implementation*, 2004.

[26] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proc. of the ACM Symp. Operating Systems Principles*, 2005.

[27] B. Cook. Towards self-healing multitier web services. M.S. thesis, Duke University, 2007.

[28] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *Knowledge Discovery and Data Mining*, 1998.

[29] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *Proc. of the Intl. Workshop on Self Managing Database Systems*, 2008.

[30] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *VLDB Journal*, 2005.

[31] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in Oracle. In *Proc. of the Conf. on Innovative Data Systems Research*, 2005.

[32] C. Domeniconi, D. Gunopulos, S. Ma, B. Yan, M. Al-Razgan, and D. Papadopoulos. Locally adaptive metrics for clustering high dimensional data. *Data Mining and Knowledge Discovery*, 14(1), 2007.

[33] S. Duan and S. Babu. Proactive identification of performance problems. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2006. Demonstration.

[34] S. Duan and S. Babu. Processing forecasting queries. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2007.

[35] S. Duan and S. Babu. Empirical comparison of techniques for automated failure diagnosis. In *Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, Jun 2008.

[36] S. Duan and S. Babu. Guided problem diagnosis through active learning. In *Proc. of the Intl. Conf. on Autonomic Computing*, 2008.

[37] S. Duan and S. Babu. Automated diagnosis of system failures with Fa. In *Proc. of the Intl. Conf. on Data Engineering*, 2009. Demonstration.

[38] S. Duan and S. Babu. Fa: a system for automating failure diagnosis. In *Proc. of the Intl. Conf. on Data Engineering*, 2009.

[39] S. Duan, P. Franklin, V. Thummala, D. Zhao, and S. Babu. Shaman: A self-healing database system (demo proposal). In *Proc. of the Intl. Conf. on Data Engineering*, 2009.

[40] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[41] Amazon's ec2. `http://www.amazon.com/gp/browse.html?node=201590011`.

[42] L. Feng, H. Lu, J. X. Yu, and J. Han. Mining inter-transaction associations with templates. In *Proc. of the Intl. Conf. on Information and Knowledge Management*, 1999.

[43] R. G. Freeman and A. Nanda. *Oracle Database 11g New Features*. McGraw-Hill Osborne Media, 2007.

[44] Google platform. `http://en.wikipedia.org/wiki/Google_platform`.

[45] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowledge and Data Engineering*, 15(3), Nov 2003.

[46] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1997.

[47] C. R. Hicks and K. V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 1999.

[48] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Corp., 2001. `http://www.research.ibm.com/autonomic`.

[49] C.-W. Hsu, C.-C. Chang, and C.-J. Lin. A practical guide to support vector classification. `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[50] K. Jayaswal. *Administering Data Centers: Servers, Storage, and Voice over IP*. Wiley, 2005.

[51] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[52] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2004.

[53] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration of IBM DB2 universal database. In *IBM Perf. Technical Report*, Jan. 2002.

[54] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[55] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2003.

[56] K. Munagala, R. Tibshirani, and P. O. Brown. Cancer characterization and feature set extraction by discriminative margin clustering. *BMC Bioinformatics*, 2004.

[57] Mysql open source database. `http://www.mysql.com`.

[58] Oprofile: a continuous system-wide profiler for linux. `http://oprofile.sourceforge.net`.

[59] Hp openview. `http://docs.hp.com/en/netsys.html`.

[60] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[61] T. Osugi. Exploration-based active machine learning. M.S. thesis, University of Nebraska, 2005.

[62] E. Parzen. On estimation of a probability density function and mode. *Ann. Math. Stat.*, 33:1065–1076, 1962.

[63] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science, UCB//CSD-02-1175, 2002.

[64] R. Polikar. the wavelet tutorial. `http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html`.

[65] R. Powers, M. Goldszmidt, and I. Cohen. Short term performance forecasting in enterprise systems. In *Proc. of the ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2005.

[66] PostgreSQL performance optimization. `http://wiki.postgresql.org/wiki/Performance_Optimization`.

[67] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proc. of the ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2003.

[68] T. J. Santner, B. J. Williams, and W. Notz. *The Design and Analysis of Computer Experiments*. Springer, first edition, July 2003.

[69] Performance monitoring tools for Linux. `http://perso.wanadoo.fr/sebastien.godard`.

[70] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[71] *Business Internet Group*. The black Friday report on Web application integrity. San Francisco, CA, 2003.

[72] Data engineering workgroup on self-managing database systems. `http://db.uwaterloo.ca/tcde-smdb`.

[73] *Dtrace, ZFS, and Zones in Solaris*. `http://www.sun.com/software/solaris`.

[74] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2008.

[75] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2006.

[76] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. Using probabilistic reasoning to automate software tuning. In *Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, 2004.

[77] Systemtap: Linux kernel instrumentation. `http://sourceware.org/systemtap`.

[78] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev.*, 34(1):3–14, 2006.

[79] Tivoli software. `http://www-306.ibm.com/software/tivoli`.

[80] TPC-H and TPC-W Benchmarks from the Transaction Processing Council. `http://www.tpc.org`.

[81] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. *ACM Transactions on Storage*, 4(1), 2008.

[82] I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun. Support vector machine learning for interdependent and structured output spaces. In *Proc. of the Intl. Conf. on Machine Learning*, 2004.

[83] UCI machine learning repository. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[84] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.

[85] Vtune performance analyzers. `http://www.intel.com/vtune`.

[86] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2002.

[87] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *Computer Communication Review*, 36(5), 2006.

[88] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, 2005.

[89] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proc. of the Intl. World Wide Web Conf.*, 2004.

[90] W. Xu, P. Bodik, and D. Patterson. A flexible architecture for statistical learning and data mining from system log streams. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, Nov. 2004.

[91] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL performance analyzer. *IEEE Data Engineering Bulletin*, 31, 2008.

[92] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communication Magazine*, 1996.

[93] B.-K. Yi, N. Sidiropoulos, T. Johnson, A. Biliris, H. Jagadish, and C. Faloutsos. Online data mining for co-evolving time sequences. In *Proc. of the Intl. Conf. on Data Engineering*, 2000.

[94] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proc. of the Intl. Conf. on Machine Learning*, 2003.

[95] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *Proc. of the European Conference on Computer Systems*, 2006.

[96] X. Zhu and X. Wu. Class noise vs. attribute noise: A quantitative study. *Artif. Intell. Rev.*, 22(3):177–210, 2004.

[97] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of the Intl. Conf. on Very Large Data Bases*, 2002.

Biography

Songyun Duan was born in Jiangsu province, China in November, 1978. From 1997 to 2001, he attended Nanjing University of Posts and Telecommunications and got his Bachelor's degree in computer science. From 2001 to 2004, he attended Tsinghua University and got his Masters degree in computer science. He started graduate studies at Duke University in 2004 to pursue his Ph.D. degree in computer science.

His research interests lie in database systems, machine learning, automatic computing, and semantic web techniques. His work includes developing a data-management platform to assist system administrators to manage large-scale and complex systems, modeling of completion times of business intelligence workloads, and analyzing Web logs to extract valuable information for online advertising. He has published over ten peer-reviewed papers in leading conferences and workshops in database and systems research.