

DISSERTATION

Titel der Dissertation

Concurrent Large-Scale Network Data Analysis in High-Speed Mobile Networks

verfasst von

Dipl.-Inform. Univ. Arian Bär

angestrebter akademischer Grad

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2015

Studienkennzahl lt. Studienblatt:	A 786 880
Dissertationsgebiet lt. Studienblatt:	Informatik
Betreut von:	ao. Univ.-Prof. tit. Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Contents

1. General Introduction	1
1.1. Contributions	2
1.2. Thesis Organization	2
I. Data Stream Warehouse Systems	5
2. Introduction	7
2.1. Organization of Part I	8
2.2. Database Principles	9
3. A Novel Data Stream Warehouse Design	13
3.1. Related Work	14
3.2. TicketDB Architecture	15
3.2.1. Data Processing in TicketDB	17
3.3. TicketDB — Lessons Learned	22
3.4. DBStream Architecture	23
3.5. Materialized View Generation	25
3.5.1. Definitions	25
3.5.2. Continuous Execution Language (CEL)	27
3.5.3. Incremental Data Processing	31
3.6. Summary	34
4. Experimental Data Stream Warehouse Performance Analysis	35
4.1. Related Work	35
4.2. Comparing TicketDB with Hadoop	36
4.2.1. Network Data Description	37
4.2.2. Hadoop Benchmark Design	38
4.2.3. Job Implementation in MapReduce	39
4.2.4. Performance Evaluation	40

4.2.5. Lessons learned	42
4.3. Comparing DBStream with Spark	44
4.3.1. Spark Introduction	44
4.3.2. System Setup and Datasets	45
4.3.3. Job Definition	45
4.3.4. Benchmark Implementation	46
4.3.5. System comparison	51
4.4. Summary	53
5. Cache-Oblivious Scheduling of Shared Workloads	55
5.1. Related Work	56
5.2. Motivating Example	57
5.3. Challenges and Contributions	58
5.4. Problem Statement	60
5.5. Scheduling Algorithms	64
5.5.1. Preliminaries	65
5.5.2. Optimal Algorithm Based on A* Search	68
5.5.3. Baseline Algorithm	69
5.5.4. Greedy Algorithm	69
5.5.5. Heuristic Algorithm	70
5.6. Experimental Evaluation	72
5.6.1. Experimental Setup	73
5.6.2. Scheduling Algorithm Comparison	74
5.6.3. Scalability Comparison	77
5.6.4. PostgreSQL Experiments	78
5.7. Summary	83
II. Network Monitoring Applications	85
6. Introduction	87
6.1. Organization of Part II	88
6.2. Network Monitoring Principles	88
7. DBStream Application Overview	93
7.1. HTTP Classification	93
7.1.1. HTTPTag - Host Name to Tag Matching	94

7.1.2. Long Term Service Usage Patterns	95
7.2. Tracking Anomalies in the Akamai CDN	95
7.2.1. Fixed-line Network Data	96
7.2.2. CDN Tracking using DBStream	97
7.3. Operating DBStream at Scale	99
7.4. Summary	101
8. Botnet Detection	103
8.1. Related Work	103
8.2. Definitions	104
8.3. DNS Failure Graphs	106
8.3.1. Initial Data Processing	108
8.3.2. Labeling Unproductive DNS Traffic	108
8.3.3. Identifying Groups of Suspicious Clients	109
8.4. Tracking Clusters of Malicious Hosts	112
8.4.1. Dissecting Identified Clusters	114
8.4.2. Long-term Botnet Tracking	115
8.5. Summary	116
9. M2M Traffic Classification	117
9.1. Related Work	118
9.2. Motivation	118
9.3. Feature Extraction and Selection	120
9.4. Online M2M Classification	124
9.4.1. DBStream Weka Integration	124
9.4.2. Aggregating Sessions per Device	124
9.4.3. Establishing Ground Truth	125
9.5. Classification Performance Evaluation	126
9.6. Summary	130
10. Conclusion and Future Research Directions	133
10.1. Statements of the Thesis	135
10.2. Vision	136
Bibliography	137
Curriculum Vitae	153

Abstract

Mobile computer networks have become an ubiquitous infrastructure comparable in importance to the road system or the power grid. For the operators of such networks it is fundamental to ensure high quality and react to outages and possible threats as soon as possible. To guarantee proper network operation, a high level view as well as the ability to drill down specific problems to find their root cause is needed. The main objective of this thesis is to design and evaluate a data processing system to accomplish this goal.

Traditional database and data warehouse system are still the most commonly adopted solution for storing and retrieving data today. While in database systems items may change frequently, deeper data analysis is done in data warehouse systems refreshed in time scales of days, weeks or even months. More recently, stream processing systems have been introduced. Those system focus on processing data as they are produced and provide results in real-time. For network monitoring applications, features of both types of systems are required. In fact, detection and diagnosis of network incidents requires accessing and processing current and historical information, whereas effective troubleshooting requires near real-time alarming. The main focus of this thesis is how those two processing requirements can be effectively and efficiently combined in a single computing system.

The solution proposed by this thesis is DBStream, a novel data stream warehouse system. In DBStream a continuous stream of data from a network monitoring system is split into small batches and stored in the system. The DBStream user registers queries which are continuously executed over each batch as soon as importing has finished. Query results are stored back into the system and are immediately available for further queries, live graphical visualization or can be exported to external tools for further elaboration.

In the first part of this thesis we review the design process, as well as the lessons we learned during the design of the DBStream system. We also investigate the performance of our approach by a comprehensive comparison to other competitive systems. Furthermore, we give a theoretical formulation of the properties of the query language and exploit it to achieve enhanced performance through cache-oblivious scheduling.

In the second part of this thesis, we show several applications from the network monitoring domain. Two of them are presented in full detail. The first one tackles the detection of malicious mobile botnets; the second application deals with the classification of machine-to-machine devices in mobile networks. Especially the second application clearly shows that DBStream enables network experts to focus on the task at hand rather than on the functional problem of how to process the data.

Although, in this thesis, the focus is on network traffic monitoring and analysis applications, the developed solutions can be generalized to other domains. Network data are historic by nature, that is, all reported data items refer to maybe only a few Milli seconds old, but past events. Once such a data item has been created it never changes, only new items are appended over time. Data with similar properties are collected in other application fields such as wireless sensor networks, smart grids and intelligent transportation systems. In this sense it is possible to adopt the proposed solutions to other application domains.

Zusammenfassung

Mobilfunknetzwerke sind heutzutage eine allgegenwärtige Infrastruktur, welche in ihrer Bedeutung mit dem Straßen- oder Stromnetz vergleichbar sind. Für die Betreiber solcher Netze ist es von größter Wichtigkeit, hohe Qualität sicherzustellen und auf Ausfälle und mögliche Gefahren so schnell wie möglich reagieren zu können. Um einen ordnungsgemäßen Netzbetrieb zu gewährleisten, sind eine gute Gesamtübersicht, sowie die Möglichkeit, spezielle Probleme bis ins Detail zu begutachten, um deren Ursache zu ergründen, notwendig. Das Hauptziel dieser Arbeit ist die Entwicklung und Evaluierung eines Datenverarbeitungssystems, welches in der Lage ist, oben genannte Aufgaben zu erfüllen.

Traditionelle Datenbank- und Data-Warehouse-Systeme sind nach wie vor die am häufigsten gewählte Lösung zum Speichern und Abrufen von Daten. Während Datenbanksysteme für einfache Anfragen und sich häufig ändernde Einträge verwendet werden, werden komplexe Datenanalysen in Data-Warehouse-Systemen durchgeführt, welche typischerweise in Zeitintervallen von Tagen, Wochen oder sogar Monaten aktualisiert werden. Datenstromsysteme sind eine neuere Entwicklung in diesem Forschungsbereich. Diese Systeme verarbeiten Daten direkt nach der Erzeugung und liefern Ergebnisse in Echtzeit. Für Netzwerk-Monitoring-Anwendungen sind Eigenschaften von Data-Warehouse-Systemen und Datenstromsystemen erforderlich. Tatsächlich ist es nicht nur wichtig, auf Netzwerkzwischenfälle möglichst schnell, in nahezu Echtzeit, reagieren zu können, sondern auch diese zu einem späteren Zeitpunkt im Detail analysieren zu können. Der Schwerpunkt dieser Arbeit ist es, ein System zu entwerfen und zu evaluieren, welches diese zwei Datenverarbeitungsanforderungen effektiv und effizient in einem einzigen Computersystem kombiniert.

Die in dieser Arbeit vorgeschlagene Lösung ist DBStream, ein neuartiges Data-Stream-Warehouse-System. In DBStream wird ein kontinuierlicher Datenstrom aus einem Netzwerk-Monitoring-System in kleine Partitionen aufgeteilt und gespeichert. Die DBStream-Benutzer hinterlegen Datenverarbeitungsaufgaben, die kontinuierlich über jede neue Partition ausgeführt werden, sobald deren Speicherung abgeschlossen ist. Abfrageergebnisse werden zurück in das System gespeichert und stehen für weitere Abfragen sofort zur

Verfügung. Diese können für grafische Visualisierungen verwendet werden oder an externe Tools für die weitere Ausarbeitung exportiert werden.

Im ersten Teil dieser Arbeit untersuchen wir den Entwurfsprozess und präsentieren die Erkenntnisse die wir aus der Gestaltung von DBStream gezogen haben. Zudem vergleichen wir die Leistung von DBStream mit modernen Konkurrenzsystemen wie z. B. Hadoop oder Spark. Darüber hinaus geben wir eine theoretische Formulierung der DBStream Abfragesprache an und nutzen diese, um eine Leistungssteigerung durch eine verbesserte Zeitablaufplanung zu erzielen.

Im zweiten Teil dieser Arbeit werden mehrere Netzwerk-Monitoring-Anwendungen vorgestellt, von welchen wir zwei im Detail beschreiben. Die erste Anwendung befasst sich mit der Erkennung von böartigen Botnetzen, die zweite mit der Klassifizierung von Machine-to-Machine Geräten in Mobilfunknetzwerken. Vor allem bei der zweiten Anwendung wird deutlich, dass DBStream Netzwerkexperten dabei unterstützen kann, sich auf die anstehende Aufgabe zu konzentrieren und sich nicht mit dem Problem der Optimierung der Datenverarbeitung beschäftigen zu müssen.

Obwohl der Schwerpunkt in dieser Arbeit auf Netzwerk-Monitoring-Anwendungen liegt, können die entwickelten Lösungen auch auf andere Bereiche angewendet werden. Netzwerkdaten sind von historischer Natur, das heißt, alle Daten beziehen sich auf vergangene Ereignisse. Sobald ein solches Datum erstellt wurde, ändert es sich nicht mehr, es werden ausschließlich neue Daten erzeugt und an die bestehenden angehängt. Daten mit ähnlichen Eigenschaften kommen auch in anderen Anwendungsgebieten vor, wie z. B. bei drahtlosen Sensornetzen, intelligenten Stromnetzen und intelligenten Transportsysteme. Deshalb sind wir überzeugt, dass die vorgeschlagenen Lösungen auch in diesen anderen Anwendungsgebieten erfolgreich einsetzbar sind.

Acknowledgments

I would like to express my gratitude to all people I worked with and who supported me during the period of my thesis and without whom this thesis would not have been possible. First I would like to thank my FTW advisor Pedro Casas for his support and helpful comments on network monitoring and machine learning topics and especially his positive attitude towards research, life and the universe in general. Many thanks also to Prof. Lukasz Golab from the University of Waterloo, Ontario, Canada who advised me on data processing related topics and supported me in many tasks and gave me tips on how to perform and present research. I also would like to thank my supervisor Prof. Erich Schikuta from the University of Vienna for his advice and very helpful tips on how to conduct successful research.

I also want to express my gratitude to the my current and former colleges at FTW: Thomas Paulin, Fabio Ricciato, Stefan Ruehrup, Peter Romirer-Maierhofer, Alessandro D'Alconzo, Tobias Witek, Eduard Hasenleitner, Mirko Schiavone, Pierdomenico Fiadino, Dimitra Paraskevopoulou and Rene Pilz. The helpful environment they created and their support made this thesis possible.

Special thanks also to Alessandro Finamore, Marco Mellia and Ignazio Bermudaz from Politecnico di Torino, for the fruitful discussions and the nice time I had when I visited them in Turin.

Most importantly I want to thank my girlfriend Qi Wang for her support and understanding during the whole time of my PhD.

Finally, I want to thank my family for their support although I am far away and can only visit them at times. I also want to thank Matthias Elter for his efforts in proofreading the thesis.

Acronyms

3G	Third Generation
3GPP	3rd Generation Partnership Project
ACID	Atomicity, Consistency, Isolation, Durability
AVS	Adult Video Service
BTS	Base Transceiver Station
C&C	Command and Control
CDH	Cloudera Distribution Including Apache Hadoop
CDN	Content Delivery Network
CEL	Continuous Execution Language
CIR	Cluster Internal Ratio
CN	Core Network
CSG	Candidate Search Graph
CT	Continuous Table
DAG	Directed Acyclic Graph
DBMS	Database Management System
DBW	Directed Bandwidth
DCR	Domain Cluster Ratio
DDoS	Distributed Denial of Service
DGA	Domain Generation Algorithm
DLA	Directed optimal Linear Arrangement
DNS	Domain Name System
DSL	Domain Specific Language
DSMS	Data Stream Management System

DSW	Data Stream Warehouse
DPI	Deep Packet Inspection
DWH	Data Warehouse
eCDF	Empirical Distribution Function
ELKI	Environment for Developing KDD-Applications Supported by Index-Structures
ETL	Extraction Transformation and Load
FP	False Positive
FPR	False Positive Ratio
GGSN	Gateway GPRS Support Node
GPS	Global Positioning System
GTP	GPRS Tunneling Protocol
LAN	Local Area Network
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoE	Internet of Everything
IoT	Internet of Things
ISP	Internet Service Provider
M2M	Machine-to-Machine
MAWI	Measurement and Analysis on the WIDE Internet
METAWIN	Measurement and Traffic Analysis in Wireless Networks
MIB	Management Information Base
ML	Machine Learning

MSID	Mobile Station Identifier
MS	Mobile Station
mTAN	mobile Transaction Authentication Number
MTRAC	M2M TRAffic Classification
NAS	Network-Attached Storage
NoSQL	Not only SQL
NTMA	Network Traffic Monitoring and Analysis
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
OS	Operating System
P2P	Peer-to-Peer
POS	point-of-sale
RAN	Radio Access Network
RDBMS	Relational Database Management System
RNC	Radio Network Controller
RTT	Round-Trip Time
SD	Standard Deviation
SDA	Simple Daily Aggregation
SGSN	Serving GPRS Support Node
SNMP	Simple Network Management Protocol
SLD	Second Level Domain
SPEC	Standard Performance Evaluation Corporation
SQL	Structured Query Language
SVM	Support Vector Machine

TAC	Type Allocation Code
TBA	Threshold Based Aggregation
TLD	Top Level Domain
TMB	Total Maximum Bandwidth
TP	True Positive
TPR	True Positive Ratio
TPC	Transaction Processing Performance Council
TCP	Transmission Control Protocol
UDF	User Defined Function
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
VNI	Visual Networking Index
VP	Vantage Point
WTMB	Weighted Total Maximum Bandwidth

List of Figures

2.1. A standard deployment of DBStream in an Internet Service Provider (ISP) network. DBStream is a data repository capable of processing data streams coming from a wide variety of sources.	8
3.1. TicketDB architecture design.	17
3.2. Time partitioning of continuous tables.	18
3.3. Optional hash partitioning of continuous tables.	18
3.4. Data processing flow in TicketDB.	20
3.5. General overview of the DBStream architecture.	24
3.6. Multiple input window definitions possible in DBStreams Continuous Execution Language (CEL).	29
3.7. Rolling average over the last 3 minutes, updated every minute.	31
3.8. Complex data processing flow for an incremental query.	31
4.1. An illustration of the <i>in-reducer grouping</i> technique.	40
4.2. Summary of job durations in both parallel systems.	41
4.3. Job inter-dependencies for the DBStream job implementation.	47
4.4. DBStream task execution over time for FIFO and Shared scheduling.	49
4.5. Performance numbers for different setups using Spark.	50
4.6. DBStream performance compared to PostgreSQL.	51
4.7. Comparison of DBStream and Spark.	52
5.1. A precedence graph corresponding to two base tables (0, 1) and four materialized views (2, 3, 4, 5).	59
5.2. A simple and an optimized ordering of the tasks from Figure 5.1.	60
5.3. An example comparing Directed Bandwidth (DBW) with Total Maximum Bandwidth (TMB).	63
5.4. An example comparing TMB with Weighted Total Maximum Bandwidth (WTMB) assuming we know the size of the output of each task.	64

5.5. Candidate search graph for the workload whose precedence graph was shown in Figure 5.1.	66
5.6. Visualization of a run of the Heuristic algorithm on the candidate search graph created from the precedence Directed Acyclic Graph (DAG) in Figure 5.1.	71
5.7. Precedence graph for the 7 selected TPC-DS queries and tables in <i>tpc-ds-7q</i>	72
5.8. Comparison of a) TMB costs, b) WTMB costs assuming the algorithms are optimizing for TMB and c) WTMB costs assuming the algorithms are optimizing for WTMB.	76
5.9. Performance analysis executing the same workload with different schedules and increasing cache size.	79
5.10. Disk read I/O for the <i>tpc-ds-7q</i> workload scheduled by the Heuristic and the Baseline algorithms.	80
5.11. Cache usage over time for <i>tpc-ds-63q</i>	82
6.1. Simplified overview of a Third Generation (3G) network including a monitoring system like e.g. METAWIN and the data export to DBStream.	89
7.1. Long term service usage patterns of frequently used anti virus services and popular video streaming services.	94
7.2. Tstat plus DBStream.	96
7.3. Evolution of number of connections served by Akamai CDN (top) and difference of number of connections served in consecutive 5 minutes time windows (bottom).	98
7.4. Evolution of server elaboration time percentiles for the preferred data center in log-scale.	99
8.1. General overview of the whole botnet detection system.	106
8.2. Visual representation of the effect of the Hamming distance based reordering algorithm.	111
8.3. Number of Mobile Station Identifiers (MSIDs) vs. time for clusters A and B.	113
9.1. Example features derived for M2M and non M2M devices.	121
9.2. Example of extended features extracted for M2M and non M2M sessions.	122
9.3. Accuracy, True Positive Ratio (TPR) and False Positive Ratio (FPR) for J48 algorithm with different session aggregations.	127
9.4. FPR per day for selected classifiers.	128

9.5. Comparison of our different session aggregation approaches Simple Daily Aggregation (SDA) and Threshold Based Aggregation (TBA) using a J48 classifier	129
9.6. Fraction of M2M devices	130
9.7. Cumulative ratio of classified devices	130

1. General Introduction

Since the introduction of computer networks in general and the Internet more specifically, networked computer systems have become more and more important to modern society. Today's Internet is a highly complex, distributed system, spanning the globe and reaching even into outer space to the International Space Station. Human communication relies to a large extent on emails, (mobile) phone calls and social media. It has become normal to buy electronics, cloth or even cars, book flights and make bank transfers over the Internet. The stock market exchanges large amounts of stocks via interconnected high frequency trading systems. This shows that computer networks have become a corner stone of today's modern society.

Network operators are responsible for the proper functioning of those highly complex networks. They face the challenge to detect and react very quickly to network anomalies, security breaches and, at the same time, plan ahead to adopt their networks to novel usage patterns.

Network monitoring systems are designed to support operators in this task. The above challenges pose several requirements to such systems. A system should be: (i) able to store data over extended time periods, (ii) make results available quickly in the order of minutes or even seconds and (iii) network experts should be able to easily specify and extend typical analysis tasks. Whereas, many isolated systems and approaches have been proposed to capture and analyze network data [113, 81, 57, 60], there is still a clear lack of comprehensive approaches, integrating, combining and post processing data from multiple sources.

In this thesis, we propose the DBStream system, a novel Data Stream Warehouse (DSW) based on traditional database techniques, as a solution for comprehensive network monitoring. We show that DBStream is performance-wise at least on par with the most recent large-scale data processing frameworks such as Hadoop and Spark. Finally, we present several novel network monitoring applications we designed and implemented and show how we can make the best use of the DBStream system.

1. General Introduction

1.1. Contributions

The major contributions of this thesis can be summarized as follows.

- We introduce the high performance network data analysis system TicketDB and compare its performance to Hadoop.
- We show which parts of TicketDB can be improved to develop a system of higher performance and flexibility.
- We introduce the DBStream system along with its flexible and easy to use processing language CEL.
- We compare the performance of DBStream to modern large-scale data processing frameworks and locate performance bottlenecks in those systems.
- We define the problem of cache-oblivious scheduling and present and evaluate novel scheduling algorithms to efficiently produce schedules of higher performance.
- We give an overview of several Network Traffic Monitoring and Analysis (NTMA) applications and report on operating DBStream at large-scale.
- We present a novel botnet detection approach based on Domain Name System (DNS) failure graphs extracted from mobile network data.
- We introduce the M2M TRAffic Classification (MTRAC) approach, able to detect Machine-to-Machine (M2M) devices using machine learning on coarse-grained traffic descriptors.

1.2. Thesis Organization

The main organizational element of this thesis is its separation into two parts. This separation is a result of the discussion of two different research domains. In Part [I](#), database and DSW topics are introduced and discussed in detail. Part [II](#) describes the application of the principles introduced in the first part onto typical NTMA applications.

The first part is subdivided into the following chapters. Chapter [2](#) starts with an introduction into database and DSW related topics, relevant for this thesis. In Chapter [3](#), the design and implementation of the early TicketDB system and its evolved predecessor DBStream along with its SQL-based processing language CEL are described in detail.

A performance evaluation of the presented systems, in comparison with other state-of-the-art large-scale data processing frameworks, like e.g. Hadoop and Spark, is given in Chapter 4. The first part is ended by a theoretical study of multiple scheduling algorithms to solve the problem of efficiently ordering a shared workload, optimizing for cache reuse, which is presented in Chapter 5.

In the second part of this thesis, applications of the approaches presented in the first part to NTMA problems are given. Chapter 6 gives an introduction into terms and definitions commonly used for describing and analyzing mobile networks. We continue with a general overview of several applications of TicketDB and DBStream and statistics from operating DBStream at scale in Chapter 7. In Chapter 8 an approach for the detection and tracking of networks of malicious hosts, called botnets, is presented. In the last chapter, Chapter 9, a continuous machine learning approach for the classification of M2M traffic is presented.

The thesis is ended by a comprehensive conclusion and future work given in Chapter 10.

Please note, that the related work is presented at the beginning of each chapter in a separate section.

Part I.

Data Stream Warehouse Systems

2. Introduction

The complexity of large-scale, Internet-like networks is constantly increasing. With more and more Internet services, the massive adoption of Content Delivery Networks (CDNs) and Cloud solutions for content hosting and delivery, and the continuous growth of bandwidth-hungry video-streaming services, network and data center infrastructures are becoming extremely difficult to understand and track. NTMA has taken an important role to understand the functioning of the network, especially to get a broader and clearer visibility of unexpected events. One of the major challenges faced by large-scale NTMA applications is the processing and analysis of big amounts of *heterogeneous* and *fast* network monitoring data. By nature, this data is heterogeneous as it consists of multiple types of measurements coming from different kinds of logging systems. In addition, network monitoring data comes in form of high-speed data *streams*, which need to be rapidly and continuously processed and analyzed.

A variety of methodologies and tools have been devised to passively monitor network traffic. Although all these solutions are capable of extracting large amounts of detailed information from live networks, they lack the ability to integrate multiple heterogeneous data sources. What is needed is a flexible system able to store and post-process the data from those rich and heterogeneous sources in order to understand the complex dynamics of nowadays Internet. Such a system should be capable of handling different types of NTMA applications, from high velocity, near real-time data processing applications such as service performance tracking, anomaly detection and alerting, to more complex high volume analysis tasks involving the processing of large amounts of stored historical data. The Data Stream Warehouse (DSW) paradigm [67] provides the means to handle both types of monitoring applications within a single system, combining the real-time capabilities of Data Stream Management System (DSMS) with the deep analytic processing of traditional data warehouses, storing long-term historical data.

The main research question to be addressed by this thesis is whether it is possible to design a system which is able to handle the challenges posed above. An overview of the final system we present in this thesis, called DBStream, is given in Figure 2.1. It is shown how data from multiple vantage points in the network is collected, merged with

2. Introduction

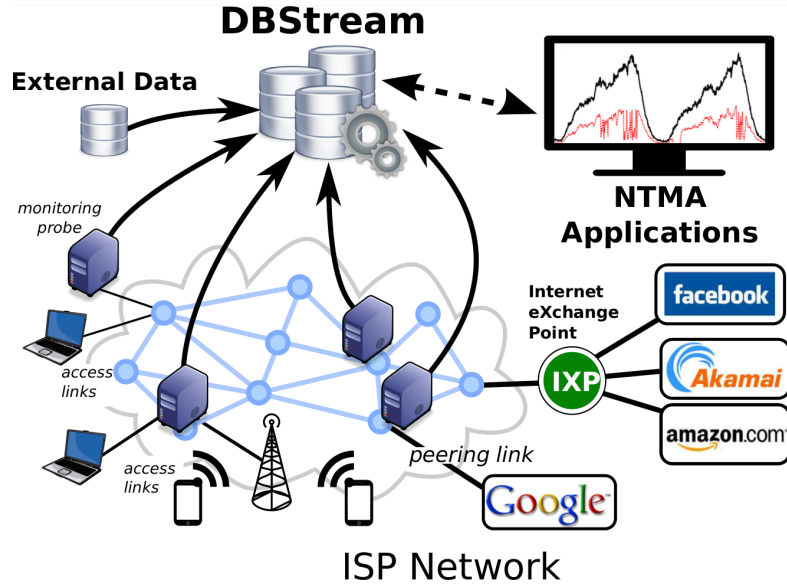


Figure 2.1.: A standard deployment of DBStream in an Internet Service Provider (ISP) network. DBStream is a data repository capable of processing data streams coming from a wide variety of sources.

external information and finally made available to external applications for visualization purposes.

2.1. Organization of Part I

The remainder of this part is organized as follows. The next Section 2.2 gives a short introduction into the basic principles of database systems, as well as Data Warehouse (DWH), DSMS and DSW systems.

Chapter 3 is split into two major sections. In Section 3.2 the predecessor system of DBStream called TicketDB is presented. It follows Section 3.3, which presents the lessons learned from designing and implementing such a system as well as some of the issues with the design of TicketDB. Next, Section 3.4 describes the design of the final DBStream system.

In Chapter 4, performance results for both systems are provided and compared to MapReduce based data processing solutions. Section 4.2 compares the early TicketDB system with an enhanced version of the MapReduce system Hadoop, given processing jobs typically found in NTMA. The performance of DBStream is compared to the large-scale data processing framework Spark in Section 4.3.

Finally, in Chapter 5 a theoretical approach to improve the performance of shared workload systems is presented. This approach applies cache-oblivious techniques to the domain of precedence constraint shared workload scheduling and is evaluated using standard database benchmarks from the TPC-DS benchmark [93].

2.2. Database Principles

In this section, we give a short overview of the most basic principles of database systems. This section is intended to increase the readability of Part I of this thesis. We begin by an introduction of Database Management Systems (DBMSs) and Relational Database Management Systems (RDBMSs), which we generally refer to simply as "databases" in the follow-up of this thesis. Next, we describe the concept of DWHs. Then we give a short description of DSMSs. We conclude this section with an introduction to DSWs, which combine some of the principles of DSMSs and DWHs.

An overview of Not only SQL (NoSQL) systems, like e.g. Hadoop, can be found in Section 3.1. We refer the interested reader to [106] and [107] for a good overview of the inner workings of distributed parallel database systems.

Database Management Systems (DBMSs)

The history of database systems goes back to the 1960's, with the first commercial database systems appearing in the late 1960's. A DBMS is a computing system used for accessing, changing and storing data in an organized way, over extended time periods. Data stored in such a system can typically be accessed by a specialized language called *query language*. The authors of [61] give a complete overview of DBMS with a focus on RDBMS. Typically, database systems are expected to provide the following benefits over storing data in regular file systems:

- Allow the creation of databases and their logical organization referred to as *schema*.
- Provide a declarative *query language* for accessing and modifying stored data.
- Enable efficient access to large amounts of data over extended time periods.
- Durability: even in case of a hardware failure the system should be recoverable.
- Isolation: allow data access by multiple users without unexpected interactions.

2. Introduction

Relational Database Management Systems (RDBMSs)

In 1971, Edgar F. Codd changed database systems significantly with his ground breaking paper [38], describing a novel way to logically design database systems. In this work, he introduced the relational model and formulated the mathematical foundation of database systems called *relational algebra*. Even today, this model remains the basis of most commercial and open source database systems. The main novelty are tables, which are a logical, abstract representation of the stored data. Tables help the programmer to access data in an easy, structured way. The application of this model enables the database to store data in different formats without changing the access to the data. The most widely used declarative *query language* based on the relational model is the Structured Query Language (SQL).

Transaction Processing

In a typical database data are accessed by multiple users concurrently. Therefore, a structured approach on handling concurrency is essential. Transaction processing is one approach to solve this issue implemented by many database systems. A transaction is a group of one or more database operations which is executed *atomically* and *isolated* from other transactions. In order to ensure durability, the database system writes all changes separately into a log file before they are applied to the database. In case a hardware failure or programming error occurs during the execution of a transaction, the system is able to revert all already completed steps. Thus, leaving the system always in a consistent state.

In 1983, the authors of [73] introduced the terminology of Atomicity, Consistency, Isolation, Durability (ACID) constraints to database systems. The main achievement of this paper was the introduction of general terms to describe certain transaction related system properties in a unified, implementation independent way. The acronym ACID is a combination of four terms with the following meaning:

- **Atomicity:** each transaction is executed as one atomic operation, which either completes successfully or completely fails. In case of a failure, all steps of the transaction are rolled back and the system is put back into the state before the execution of the transaction started.
- **Consistency:** at the end of a transaction all consistency constraints have to be met. E.g. a bank account balance can not be less than zero.

- **Isolation:** all transactions are executed as if they would be running sequentially and not in parallel. Isolation is typically implemented using locking mechanisms.
- **Durability:** after a transaction has been successfully completed, the effect of this transaction can not be lost.

Data Warehouses (DWHs)

Data Warehouse (DWH) systems are used in large corporations and enterprises for reporting, analytical and predictive purposes. Typically data in DWHs is organized in a *star schema* as presented in [32]. For example, an operational database system may be used to handle sales. This database executes transactions whenever items are bought, new items become available or stock is refilled. From this database, aggregated data, e.g. how many users bought item A today, are transferred to the DWH. This process is referred to as Extraction Transformation and Load (ETL), in which data are pre-aggregated and transformed into the *star schema* of the DWH.

In the *star schema* data are split into *dimension* and *fact* tables and referenced via unique identifiers. In the above example, *dimensional* data would be, e.g. details about the user buying items like the region she is coming from, the phone number or ZIP code. *fact* data would be information such as: when did she buy which item. This information is stored in separate tables, whereas typically *fact* tables are time partitioned in case they are becoming large. Reports are then created by aggregating over a subset of dimensions during query time. Those reports might be used by the management to identify regions where sales decreased and take appropriate counter measures.

Data Stream Management Systems (DSMSs)

In a database system, data are static and are only changed or accessed by queries. DSMS follow a different approach and invert this model. They treat queries as the static part, which are installed into the system. Whereas, data is flowing through the system as a continuous stream, which is presented to each installed query. Typically, data are not stored at all, only the results of the installed queries are reported periodically to external systems.

The first such system introduced in [27] was called Aurora. After which many other systems were designed to follow the same principles. A good overview of DSMS can be found in [69].

Data Stream Warehouses (DSWs)

Data Stream Warehouse (DSW) systems have been proposed recently to handle big and fast data in one single system. This approach can be seen as an extension to DSMS, adding a very long history to such systems. Thus enabling the system to e.g. replay old data to learn new patterns which were not detected with already implemented approaches. Another approach is to extend regular DWHs by the capability to digest and integrate new data very fast into the reports generated by the system. The resulting system is a DWH, extended by streaming capabilities.

DSW systems can be categorized into the following three categories:

DBMS based systems adding new capabilities like the fast import of data arriving at the system and the ability to evaluate queries over data streams. Those systems include DataDepot [66] which is based on a AT&T proprietary DBMS, DejaVu [46], based on MySQL and the TruViso [82] system which is based on PostgreSQL.

DSMS based systems like the Moirae system [13], which is based on Borealis [1] and adds a PostgreSQL based storage layer.

MapReduce based systems which use e.g. Hadoop [140] and add continuous data processing capabilities. Examples of such systems include the Muppet [85] system adding update functionalities to the Map part of Hadoop jobs and the Nova system [104] which is based on Pig latin [105].

The DBStream system presented in this thesis falls into the category of DBMS based systems and utilizes the PostgreSQL database engine.

3. A Novel Data Stream Warehouse Design

In this chapter, we describe the evolution of our approach for network data storage and analysis. The first step in this evolution is a system called TicketDB, which is described in detail in Section 3.2. The main focus of TicketDB is on data imports and parallel queries, enabling very high import and query processing performance. In the following Section 3.3 we describe some of the issues with the TicketDB approach leading to the design of a novel system. This novel system is called DBStream and is described in full detail in Section 3.4. The focus of DBStream lies on the continuous processing of data arriving in time windowed batches. Queries on the imported data continuously transform them into materialized views. With this novel approach we were able to achieve the following benefits:

- Writing stream processing queries is nearly as simple as writing regular SQL.
- DBStream automatically processes data as they arrive and stores the result in materialized views. This approach enables DBStream to answer previously known queries much faster, without processing huge amounts of raw data.
- The windowing mechanism of DBStream allows efficient incremental queries, which are particularly useful for network monitoring.

This chapter is based on the following publications. The TicketDB system was introduced in [15]. The DBStream system was introduced in [16] and its performance was described in more detail in [18]. The author of this thesis was responsible for the design and implementation of the TicketDB and DBStream system as well as the main writing of the papers mentioned above. Fabio Ricciato helped in designing the network data analysis jobs used to evaluate the performance of TicketDB and with the writing of the paper. Antonio Barbuzzi and Pietro Michiardi, helped in comparing TicketDB to Hadoop and implemented all Hadoop related experiments. Pedro Casas, Marco Mellia and Alessandro Finamore, helped in describing the applications of DBStream. Whereas Alessandro Finamore is responsible for the improvement and execution of the Spark related benchmarks. Lukasz Golab helped with the integration of DBStream into the general database context.

3.1. Related Work

The introduction of Big Data processing lead to a new era in the design and development of a large-scale data processing system. Many novel data processing and storage systems have been redesigned from scratch to improve both performance and scalability. Most of them achieve increased performance by re-implementing the data processing engine, relaxing ACID constraints [129] and/or applying novel data processing paradigms. Still, a major limitation of such systems is the inability to cope with continuous analytics. DSMSs, such as Gigascope [40], Borealis [2], Esper [52] or the more recent Streambase system [130], support continuous processing, but they cannot support analytics over historical data, as it is required in NTMA applications.

DSW systems extend traditional databases and DWH with the ability to ingest and process new data in near real-time. DataCell [92] and DataDepot [67] are two examples, as well as the DBStream system presented in this thesis. Another import development are NoSQL systems based on the MapReduce framework made popular in [45]. Those systems use a key-value interface rather than a high level declarative language, like SQL, typically supported by DBMSs. Hadoop [140] and Hive [134] are two popular open source implementations of the MapReduce framework. Dremel [98] is a Google proprietary technology that exploits the MapReduce paradigm and uses a column oriented database to optimize web search. Spark [146] is another interesting system, promising an approx. 100x scale up factor with respect to Hadoop by utilizing an in-memory processing architecture.

MapReduce systems focus on processing data in large batches rather than streams as it is required as well in the case of NTMA purposes. There has been some recent work on enabling real-time analytics in NoSQL systems, such as Muppet [85], SCALLA [91] and Spark Streaming [147]. At the moment, the main focus of Spark Streaming lies on the processing of real-time data, e.g., a stream of twitter feeds. Unfortunately, it is not possible out-of-the-box to perform non real-time processing, where data arrive with delay. Nevertheless, Spark Streaming seems to be an interesting candidate for future network monitoring solutions.

Another class of systems are hybrid systems, which integrate SQL and NoSQL technologies into one system. For instance, HadoopDB [3] connects multiple single-node database systems and utilizes Hadoop for scale-out. Processing can be parallelized using Hadoop MapReduce, but within each MapReduce task, data processing is pushed into the relational database system PostgreSQL, installed on each cluster node. Other approaches include the proposal to combine DSMS with DBMS for the improvement of

online analytics [34, 77].

From a query language point of view, many novel, continuous and streaming processing languages have been proposed [40, 67, 34, 10, 47]. All of them assume that persistent data storage is handled by external systems and queries can only refer to temporary memory-resident state (e.g., a current window of time). In contrast, many DBMSs support the definition of materialized views using SQL queries over large historical tables, but continuous view maintenance over time is limited to simple types of queries such as filters and joins, without the ability to formulate more complex aggregations. DBStream enables users and applications to declaratively specify, using full SQL including User Defined Functions (UDFs), exactly how to update a view when new data arrive at the system. DBStream CEL queries may even refer to previously generated results that are stored in the same view that is currently updated. To the best of our knowledge, this is not *declaratively* supported by any other system. In particular, there has been recent work in the networking community on extending SQL with additional functionalities required for network monitoring. Examples include complex window expressions [25] and sequential patterns [67]. However, none of these proposals include the declarative, incremental processing that is supported by DBStream.

Systems for complex analytics have attracted the attention of the research community as well as many large companies. For example, Facebook [26] and RIPE¹ have already integrated the MapReduce system Hadoop into the core of their end-user service provisioning systems. However, with the exception of the proprietary, closed-source DataDepot system, none of these systems were designed to address continuous data processing, required for NTMA applications. Furthermore, to the best of our knowledge, DBStream is the only available system that supports incremental queries defined through a declarative language. As we will show in this thesis, the continuous analytical capabilities and incremental query processing make DBStream particularly useful for tracking the status of large-scale mobile networks.

3.2. TicketDB Architecture

The constantly increasing data rates of mobile and fixed-line computer networks force monitoring solutions to constantly increase their performance accordingly. Therefore, systems storing and analyzing network data in a centralized location need highly optimized and efficient import and query mechanisms. In this section, we describe the design

¹Hadoop and HBase at RIPE NCC, <http://blog.cloudera.com/blog/2010/11/hadoop-and-hbase-at-ripe-ncc/>

3. A Novel Data Stream Warehouse Design

and implementation of TicketDB, a parallel database system based on the PostgreSQL database engine, tailored to achieve this task.

One important aspect of network monitoring data is its historical nature. Once a certain network event has been recorded and timestamped, it never changes again. Therefore, we consider network data analysis as a typical On-Line Analytical Processing (OLAP) application and adopt widely used OLAP techniques in the design of TicketDB. First, we apply the star schema presented in [32] to split data into *fact* and *dimension* tables (see Section 2.2 for details on the star schema). Second, we utilize a two stage table partitioning approach, splitting data into smaller batches. Data is partitioned by time and each time partition is further split into several hash partitions. The hash partitioning applies a hash function on the primary key column of the table, like e.g. the Mobile Station Identifier (MSID), to assign a data item to a certain partition. This approach allows us to import multiple partitions in parallel, thus speeding up the import process. At query time, data can be processed in parallel on each partition, followed by a final aggregation step.

The overall architecture of TicketDB is presented in Figure 3.1. The current implementation of TicketDB utilizes the *scale-up* approach. That means, TicketDB is installed on a single high performance server, with many CPU cores, a huge amount of RAM and two fiber-channel attached RAID arrays for data storage.

During data import, input data is first split into fact and dimension information. In the next step, time partitioning, as shown in Figure 3.2 is applied, which in the current setup uses one hour partitions. Next, hash partitioning as shown in Figure 3.3 is used to distribute data over several database nodes, each residing on a different RAID array. The TicketDB master node stores a data directory containing information about the placement of hash and time partitions among database nodes. The application of data partitioning has the following benefits:

- Data import can be parallelized.
- A single query, accessing several hours or even days of data can be executed in parallel on multiple partitions at the same time.
- Indices on tables can be created on each partition separately. If a new partition is added to the system a new index is created for that partition and all already existing indices do not have to be changed.
- Indices can be created on each partition separately, in parallel.

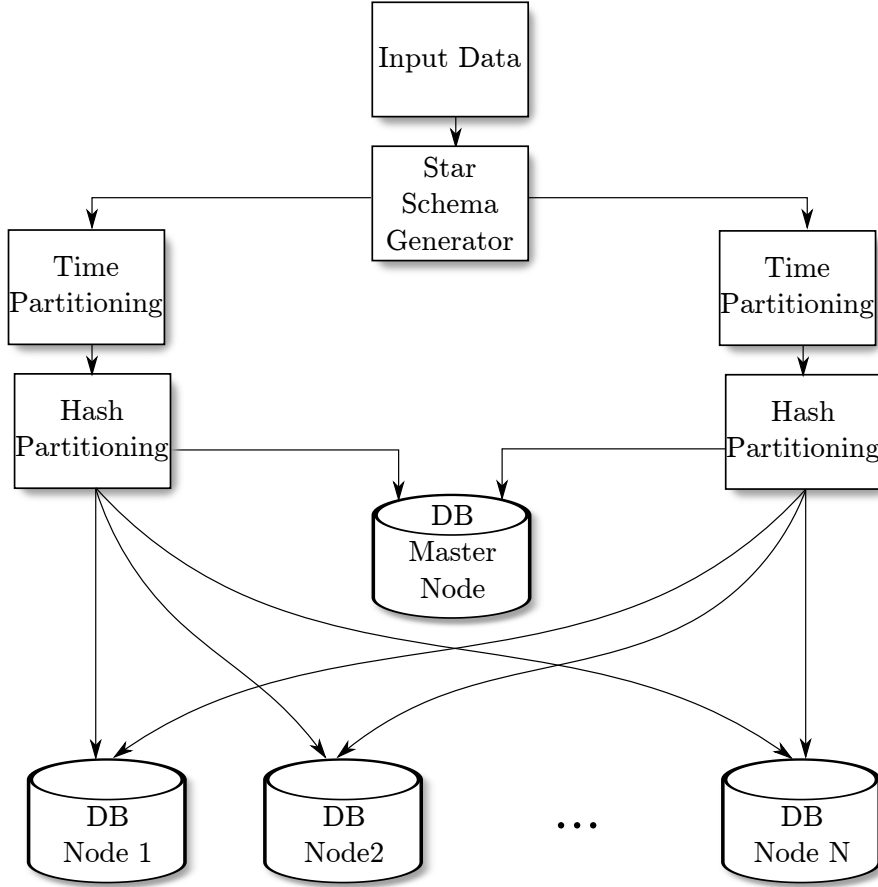


Figure 3.1.: TicketDB architecture design.

- If data needs to be deleted from a table, full partitions can be dropped, which is much more efficient than deleting data from a very large unpartitioned table.

In TicketDB each hash partition of a time partition is stored in a separate PostgreSQL database node. All database nodes are hosted on the same high-end server machine. Although data is stored on two distinct RAID-6 disk arrays and therefore safe towards up to two disk failures per array, our current version does not support replication or hot-standby. In contrast to a cluster system like Hadoop, which can overcome node failures, if the TicketDB server stops working, the whole system becomes unavailable.

3.2.1. Data Processing in TicketDB

In this section we describe how data is imported and queried in TicketDB. We use the following example processing job to demonstrate the inner workings of TicketDB. For

3. A Novel Data Stream Warehouse Design

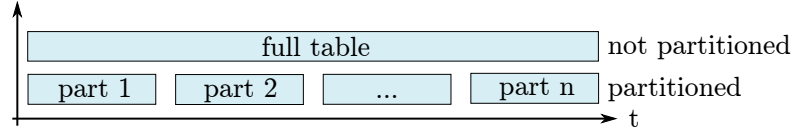


Figure 3.2.: Time partitioning of continuous tables.

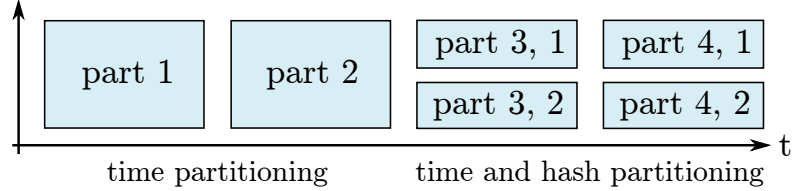


Figure 3.3.: Optional hash partitioning of continuous tables.

every IP address we compute the number of uploaded, downloaded and total exchanged (upload + download) bytes. We refer to this processing job as **Heavy User Job** in the remainder of this section. As input data, we use a dataset extracted from the publicly available Measurement and Analysis on the WIDE Internet (MAWI) archive, captured from a trans-Pacific link between the United States and Japan [35]. Packet payload is omitted from the trace and IP addresses are anonymized. The format of the extracted dataset as well as the assignment of the columns to the dimension and fact tables is shown in Table 3.1.

Name	Target	Data Type
Timestamp	Fact	integer
Source IP Address	Dimension	inet
Destination IP Address	Dimension	inet
Source Port	Dimension	smallint
Destination Port	Dimension	smallint
TCP flags	Fact	smallint
Protocol	Dimension	smallint
Exchanged bytes	Fact	bigint
Exchanged packets	Fact	bigint

Table 3.1.: Data format extracted from MAWI traces.

Data Import

The first step is to import data into TicketDB. Therefore, a highly optimized C program splits data into fact and dimension information, as specified in the target column of

Table 3.1. In addition, unique primary keys are generated to be able to join those two tables later on. Data is first split into time partitions, which then are split into hash partitions before import. All hash partitions of the same time partition are then imported in parallel, resulting in an increased import performance. Another performance optimization we use is that instead of regular `insert` statements to import the data into tables, we import data in batches of 100 thousand rows at once using the much more efficient `copy` command. In this way we are able to increase the import performance by one order of magnitude.

Another important measure to take into account is that in PostgreSQL the physical storage size of partitions plays an important role in the overall system performance. Although we do not know the ultimate reason, we noticed that if a table becomes larger than 1 GB, performance suffers. One reason for this might be that each table is written into one or more files, whereas the maximum size of a file is 1 GB. If a table is larger than 1 GB, it is split into multiple files. This seems to have a negative impact on performance, since more files have to be opened for reading and writing. In addition, if the tables used as hash partitions get bigger the memory available to each database connection has to be increased. Otherwise, more complex operations like, e.g. sort have to be executed on the much slower disk storage. If this is the case, not only the performance of the sort operation is decreased since the slower disk system is used instead of the RAM, but also other concurrent queries are affected since the overall disk I/O is increased. Therefore, we optimize the size of partitions to stay below 1 GB, even in times of high network load, when more data are generated by the network vantage points.

Example Processing Job Implementation

Queries posed to TicketDB can run in parallel on multiple time and hash partitions. We will now explain how to implement such queries and how they are executed inside TicketDB by an illustrative example. Let us assume we want to compute the number of uploaded and downloaded bytes on a given link in one day, as in the **Heavy User Job**. If data is time partitioned per hour and each time partition is further split into two hash partitions, each stored in a separate database node, 48 partitions need to be processed to answer our query.

Figure 3.4 shows an overview of the data flow in TicketDB for only four time partitions. Starting from the top, first a new *outer* database connection is created for each hour which needs to be processed. Then, since each hour is split into two hash partitions per hour, two *inner* database connections are created. In each of the *inner* connections we

3. A Novel Data Stream Warehouse Design

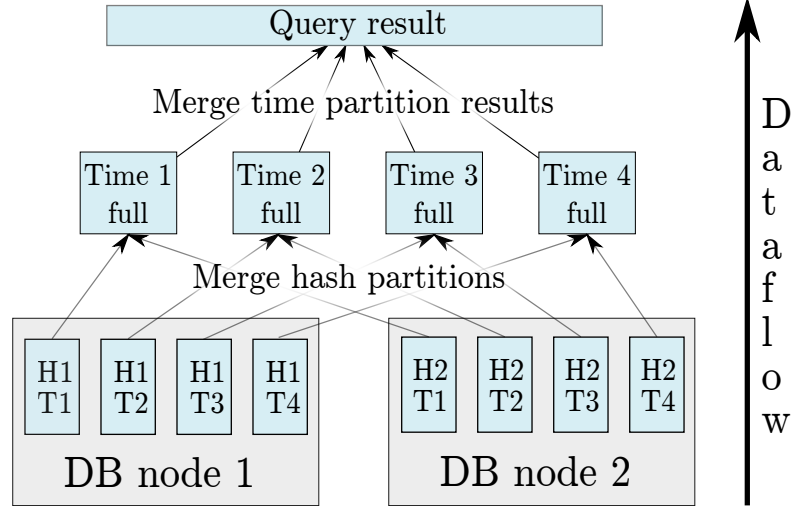


Figure 3.4.: Data processing flow in TicketDB.

compute the sum over the total exchanged packets field per source and destination IP and the results are returned to the *outer* connections. In each *outer* connection, the sum over the results of the *inner* connections is computed. Ultimately, the results from all time partitions are merged into the final query result.

Although the query parallelization is not fully automatized in TicketDB, a query framework supports the user in writing those queries. As also shown in Algorithm 1, two functions help the user to formulate those queries. The first function is called `qf_dist_sql_outer` and handles the creation of the *outer* connections, the second function is called `qf_dist_sql_inner` and handles the creation of the *inner* database connection. It is possible to limit the maximum number of parallel database connections used by a single TicketDB query with a global configuration parameter. To run a query using the query framework, a user has to specify three parts, (i) which query to run on the raw data stored in the hash partitions, (ii) how data between hash partitions is merged and (iii) how data from different time partitions is merged.

We now explain the **Heavy user Job** shown in Algorithm 1 in full detail. In the inner most query, exchanged bytes are aggregated per IP address by joining a partition of the `fact_ip` containing the exchanged bytes, with the `dim_ip` table containing the actual IP addresses. Since there is no information about upload and download, but only exchanged bytes between two IPs, we create this information, by using the **union all** operator over two subqueries. The first subquery counts all exchanged bytes where a IP appears as the source IP as uploaded bytes. The second subquery counts all exchanged bytes as the

Algorithm 1 Query implementation for **Heavy User Job**.

```

1: select * from qf_dist_sql_outer($outer$
2:   insert into heavy_users_TIMESTAMP (
3:     select ts, ip, sum(up) up, sum(down) down,
4:       sum(total) total
5:   from qf_dist_sql_inner($inner$
6:     with t as
7:       (select ts, source_ip, dest_ip, up bytes from
8:         (select timestamp - (timestamp
9:           id, sum(bytes) up
10:        from fact_ip group by ts, id) as f
11:       join dim_ip d on (f.id=d.id))
12:     select ts, ip, sum(up) up, sum(down) down,
13:       sum(tot) tot
14:     from (
15:       select ts, source_ip ip, bytes up, 0 down,
16:         bytes tot from t
17:       union all
18:       select ts, dest_ip ip, 0 up, bytes down,
19:         bytes tot from t
20:     ) foo
21:   group by ts, ip order by ts, tot desc
22:   $inner$, TIMESTAMP)
23:   group by ts, ip)
24: )
25: $outer$, start_timestamp, end_timestamp);

```

downloaded bytes for the destination IP. In addition, the sum of uploaded and downloaded bytes is calculated at the same time and stored in the **total bytes** field. Now, we aggregate over the result of the union using the **group by** operator and as a result get data in the following format: **<ts, IP, uploaded bytes, downloaded bytes, total bytes>**. Since the function **qf_dist_sql_inner** returns the concatenation of all results from each hash partition, we aggregate by timestamp and IP. Finally, in the last step, we aggregate the output of the **qf_dist_sql_outer** function by timestamp and IP and write the results into the output table which we call **heavy_user_TIMESTAMP**. Please note that **TIMESTAMP** will be automatically replace by the actual timestamp this query was started with. If the query covers more time partitions than the maximum amount of available database connections the query is executed in multiple rounds, each running with the maximum number of connections available or the number of remaining time partitions. A further optimization to this approach is to reuse the connection as soon as

3. A Novel Data Stream Warehouse Design

a single query has finished. The current implementation does not include this feature, which is left to future work.

3.3. TicketDB — Lessons Learned

In this section, we describe some limitations and drawbacks of the TicketDB approach, which ultimately lead to the design and implementation of DBStream.

Usability Considerations

Although the query performance, presented in Section 4.2, of TicketDB is very high, for a typical user it is cumbersome and time consuming to write parallel queries like the one presented in Algorithm 1. Most of the people working in network monitoring and network data analysis have an electrical engineering background and are not familiar with the details of SQL. Therefore, we decided to design a language which is flexible and more easy to use.

Another usability problem of TicketDB is that the output of queries can not be stored systematically. Often an analysis should run over extended time periods over historical and newly imported data, which is not directly supported by TicketDB. Lets imagine a network problem was detected by running an analysis on data from the last two weeks. The operator now took a counter measure to fix the problem or improved the performance of the monitored network. Obviously, it is very interesting to run the same analysis again and to compare the results to the previously obtained results. In TicketDB this means that the whole query has to be repeated and extended to the new data.

Performance drawbacks

Running a parallel query over a huge dataset also consumes a huge amount of resources. Therefore, in many cases it is beneficial to keep intermediate results to avoid computing the same intermediate steps over and over again. In TicketDB, handling such situations is not supported by the system.

For instance, if one query was executed over the last two weeks, e.g. from 2014-11-03 17:16 to 2014-11-17 17:16, if the results should be updated at the 2014-11-21 at 14:56, one has to manually calculate the start and end of the new query. In the next step, even if the results of the first query where stored in a table, the results of the new query need to be merged, which results in a huge manual query writing overhead in TicketDB.

Another issue was that the imported data are very big, therefore it is not always feasible to keep them over extended time periods. In contrast, many applications do not need all imported data, but only access a small fraction. Therefore, filtering and aggregating data automatically would provide a natural way of data compression without the need to store excessive amounts of data.

Novel System Requirements

A novel system, replacing TicketDB should have the following properties:

- Queries should be easy to write.
- Queries should still execute fast.
- The whole system should be easy to configure.
- There should be a structured approach on storing intermediate results.
- Time partition handling should be supported by the system as much as possible.

3.4. DBStream Architecture

In this section, we describe the overall architecture of the DBStream system.

In Figure 3.5 a high-level overview of the DBStream architecture is shown. The DBStream system is designed as a set of modules, each executed as a separate operating system process. The most important module is the so called **Scheduler**, it dictates the ordering in which jobs are executed. Data is imported and if necessary cleaned by one or more **Import** modules, signaling the availability of new data to the **Scheduler**. Imported data are stored in time partitioned Continuous Tables (CTs). Jobs registered in the **View Generation** module read data from one or more CTs and write the result into a new CT, which is created automatically if needed. This transformation is achieved by a novel batched data stream processing language called CEL, which is explained in full detail in Section 3.5. The **Retention** module monitors the size of CTs and deletes old data if a certain pre-configured size limit is exceeded. This module is essential for a stable operation of DBStream over extended time periods. Since data are imported continuously, there is a non-negligible risk of filling up all available disk space, which may ultimately lead to a complete stop of the whole system. If configured correctly, the **Retention** module avoids those situations automatically.

3. A Novel Data Stream Warehouse Design

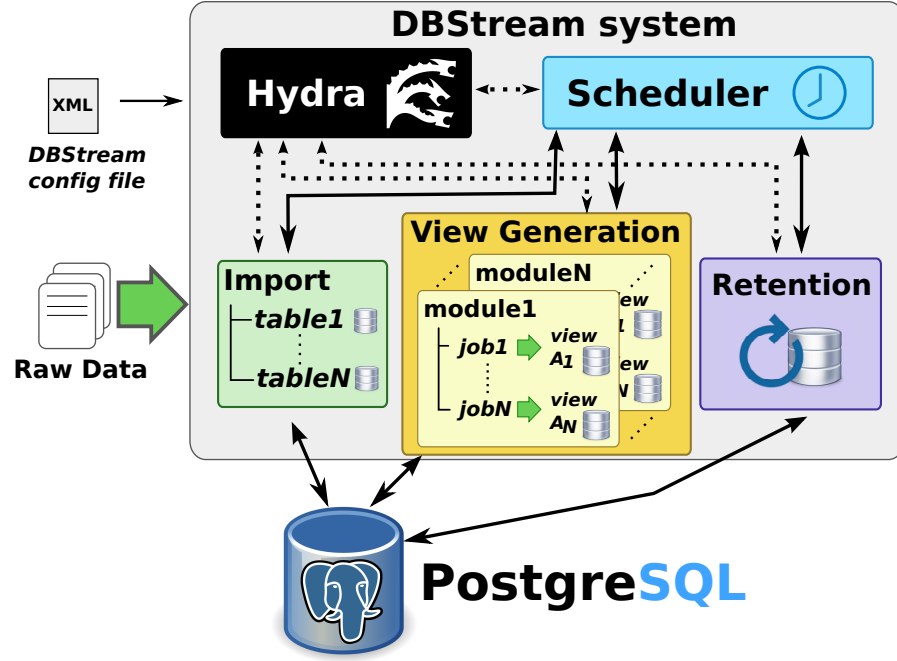


Figure 3.5.: General overview of the DBStream architecture.

The **Scheduler** is the central module of DBStream. All other modules register jobs they want to execute at the scheduler. As soon as the **Scheduler** decides that a certain job can be executed now, it sends a message back to the corresponding module, which now starts the execution of the job immediately. The decision when to execute a job is based on two facts. First, data has to be available for all input time windows, meaning that all *precedence constraints* of a job have to be met. Second, as we will show in detail in Chapter 5, it might not always be efficient to execute a job right away. Therefore, in specific situations, e.g. the maximum number of parallel jobs is exceeded, the **Scheduler** blocks the execution of certain jobs. This is also a counter measure to avoid resource contention and system overload. The point in time until which a job has finished processing is stored in an internal state of a job, which is also persisted on disk in the data dictionary whenever the internal state of the job is advanced. In case the system crashed during the execution of a certain job and is restarted again, the data dictionary is checked until which point in time the job was finished. All intermediate tables which might be created before the crash but did not finish, are deleted. This guaranties the *Atomicity* property of the ACID constraints (please refer to Section 2.2 for details) in DBStream.

As shown in Figure 3.5, all modules mentioned above, are started and monitored by

an application server process called **hydra**. It reads the DBStream configuration file and starts all listed modules. Each module has a standardized interface to provide log information. The **hydra** periodically fetches this log information and makes it available in a centralized location. Another crucial function of the **hydra** is restarting crashed modules. Since modules might depend on external processes potentially running on remote machines, they might crash at unpredictable moments. Therefore, all DBStream modules are designed and implemented such that they can crash at any point in time and leave the whole system in a recoverable state. This provides the guaranties of the *Durability* property of the ACID constraints to DBStream.

The communication with **hydra** as well as the communication between modules, like e.g. between the **Scheduler** and the **View Generation**, is implemented using remote procedure calls over a HTTP interface. Therefore, it is easily possible to distribute certain modules of DBStream over several machines.

3.5. Materialized View Generation

In this section, we detail the materialized view generation module of DBStream, focusing on the novel batched stream processing language CEL.

In DBStream raw data are imported into *base tables*. *Jobs* process data batches from those *base tables* and store the output into *materialized views*. Each *job* can have multiple inputs, which can be either be *base tables* or *materialized views*. From each input, the job fetches a certain time slice, e.g. 5 minutes, which is available to the SQL query inside the *job* like a regular table. Those SQL queries are executed in PostgreSQL and their output is stored as regular time partitioned tables. We refer to a time partitioned *base table* or *materialized view* in DBStream as a Continuous Table (CT), since both are handled in the same way. Please note, that in contrast to a DSMS, in DBStream all data are stored on disk and can be used in processing *jobs* over extended time periods, only limited by the available disk space. For example, a job can compare the current hour of data with the same hour of the day one month ago, without keeping a full month of data in memory.

3.5.1. Definitions

In this section, we introduce several concepts which help to better understand the underlying mechanisms of DBStream and its stream processing language CEL.

A **window** is a time slice of a data stream, e.g. the last 5 minutes from any CT. Since jobs can have more than one input window, one of them needs to be defined as

3. A Novel Data Stream Warehouse Design

the primary window. This primary window is used by the *view generation* module to advance the processing of this job. For example, if a job J has two windows A and B, where A is the primary window and has a size of 1 minute and B has a size of 3 minutes. Then, J is executed every minute, whereas B always contains the last 3 minutes, starting from the end of A and reaching 3 minutes into the past. This concept is visualized in Figure 3.7.

A **job** definition has multiple inputs, one single output, a SQL query and a schema, defining the data types of the output CT. In addition, DBStream keeps track of an internal state of the job which is simply the timestamp until which all processing has finished. For every new time slice of the primary window, a **task** is generated.

A **task** is a SQL query, automatically generated from the job definition, where all inputs are realized as slices of the input CTs. Each task execution advances the internal state of a job by the size of one primary window. For instance, if the primary window A of a task is 1 minute and the internal state of the job is '2014-11-03 11:55:00', the data used to represent this window in the next task of this job starts from '2014-11-03 11:55:00' and ends before '2014-11-03 11:56:00'. If the other window has a length of 3 minutes, it will start at '2014-11-03 11:53:00' and end as well before '2014-11-03 11:56:00'.

DBStream is a DSW and therefore similar to a stream processing system in many ways. In contrast to typical database applications, where time often is modeled as a column with a specific data type, time is an essential part of the architecture of DBStream. Therefore, the exact definition of time is a crucial, determining how the system works and what kind of problems it can solve. The authors of [47] give an interesting overview of different methodologies for time handling. Two of their definitions are very important in understanding the time handling in DBStream. First, the term **system time** is defined to be the wall-clock-time at the system processing the data. Second, the **application time** is used for timestamps which are part of the data processed by the system. Network monitoring systems typically assigning a timestamp to observed events and use their wall-clock-time for this purpose. Since such systems are typically implemented as some sort of stream processing system, and one of the purposes of stream processing systems is to generalize the design of such systems, using the wall-clock-time directly is a reasonable choice. The situation is very different for systems like DBStream. Here, the target is to store and analyze the output of other, typically remote systems which are stream processing systems themselves. Data arriving at DBStream already have a timestamp, assigned by a stream processing system in a higher layer of the processing chain. It would not be very helpful to add another wall-clock-time timestamp to the data. But

Term	Description
window	A time slice of a stream, defined by stream_name (window N [delay M] [primary]) [as window_name] [, ...]
primary	Marks the window along which processing is advanced. primary can only be used once per job.
_STARTTS	Is replace in a query with the start of the primary window.
_ENDTS	Is replace in a query with the end of the primary window.
delay	Can be used to shift a window into the past.
job	Defines how inputs are transformed into the output stream. Its State tracks the application time until which the job has been finished.
task	Concrete unit of work which is executed to advance the state of a job.
application time	Time of the application, contained in the processed data.
system time	Time of the processing system, often referred to as <i>wall-clock time</i> .

Table 3.2.: Definition of the most important terms of CEL.

instead, the time the event was created, already contained in the data, is of importance. Therefore, DBStream always uses the **application time** contained in the data for time windows.

One open issue with the current implementation of DBStream is data arriving at the system not in a monotonically increasing time order. In DBStream, there is no implicit handling of this type of out-of-order data. The rule is that the timestamp provided to DBStream has to be monotonically increasing. Therefore, either data has to be sorted before the import into DBStream, or a new monotonically increasing timestamp has to be generated and attached to the data. The Truviso system detailed in [83] solves the problem of discontinuous, out-of-order data streams in a promising way. Their proposed approach could be applied to DBStream to solve out-of-order data issue.

3.5.2. Continuous Execution Language (CEL)

In this section we describe the novel batched stream processing language CEL introduced in [18]. Table 3.2 gives an overview of the important terms of CEL. We explain CEL by giving several examples taken from network monitoring applications and explain the time windowing mechanisms in full detail.

We start with a very simple example of CEL. Let us assume we want to generate aggregate statistics from a router in the network under study. This router exports data on a per minute basis in the widely adopted NetFlow [37] format. Each row contains information on a per flow basis, where a flow is identified by the 5-tuple of source and destination IP, source and destination port and IP protocol number. In addition, each

3. A Novel Data Stream Warehouse Design

row contains information about the uploaded and downloaded bytes. Our first CEL query will compute the amount of uploaded and downloaded bytes passing through that router on a per hour basis. Such a job is expressed in the following way:

Algorithm 2 Single window CEL job

```
<job inputs="A (window 60min)"
  output="W"
  schema="serial_time int4, total_download int8, total_upload int8" >
  <query>
select __STARTTS, sum(download),
      sum(upload) from A group by __STARTTS
  </query>
</job>
```

In the above example, the `inputs` XML-attribute defines the input windows and the `output` XML-attribute defines the destination Continuous Table (CT) for the result. Here, only a single input window of 60 minutes of the CT A is defined, therefore a new task represented as a SQL query is executed in the underlying DBMS for each full hour of input data. The result of this SQL query is then stored in the CT with the name W. Please note the special keyword `__STARTTS` which can be used in DBStream CEL jobs and is replaced with the start time of the primary window. In this query, one could use `min(serial_time)` as a replacement of `__STARTTS` with the same result. In W timestamps will have a granularity of one hour.

The schema of the output CT W is specified using the `schema` attribute, where the first field always has to be the **application time** of the output stream called `serial_time` in DBStream. The `serial_time` in DBStream is stored as the four byte Unix epoch. The text inside the `query` XML-element specifies the template SQL query which will be executed for every full hour of data of A in this example. All SQL queries which are supported by the underlying DBMS, which is PostgreSQL in the current implementation, are supported inside the `query` XML-element.

The query in the above example, calculates the sum over all uploaded and downloaded bytes in one hour. In the `from` part of the query, the name of the CT A is used, which is a place holder and is replaced by a time slice of stream A of one hour for every executed task. Finally, by using the XML-attribute `index` one can define one or more columns of the output CT for which indices will be created. Those indices are created the same way as in TicketDB, meaning that there is no global index on the whole table, but each partition has its own index.

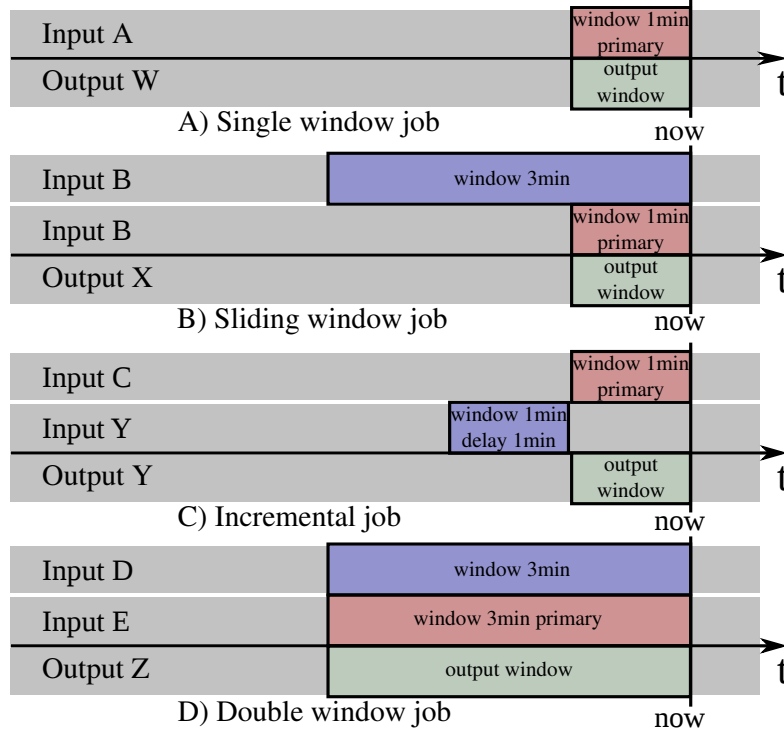


Figure 3.6.: Multiple input window definitions possible in DBStreams Continuous Execution Language (CEL).

In Figure 3.6 an overview of several possible window definitions is shown. CEL does not have an explicit definition for sliding windows, but instead, for each job one single window is marked to be the primary window by specifying the **primary** keyword in its definition. As soon as a task was executed successfully, the internal state of the corresponding job is moved into the future by the size of one primary window. When enough data in all input windows of the next task becomes available, the scheduler executes the next task if a the maximum amount of parallel tasks is not yet reached. Later implementations of the scheduler will include more sophisticated scheduling strategies, based on the work presented in Chapter 5. The second important keyword for window definitions in CEL is **delay**. It can be used to shift a window into the past by a certain amount of time. For example, if the internal state of the job is "2014-11-21 12:20" a window of 1 minute would start at minute "12:19" and end at "12:20". If this window has a delay of 1 minute, given by the following definition (`window 1min delay 1 min`), it would start instead at "12:18" and end at "12:19".

The window definitions visualized in Figure 3.6 have the following properties. Part A) shows the most simple window definition similar to the window definition used in

3. A Novel Data Stream Warehouse Design

Algorithm 2. The single input window is also the primary window of the job. Such jobs are typically used for data projections, transformations and aggregations.

The window definition shown in part B) is an example of a sliding window. Since the primary window has a length of one minute, a task is executed every minute. The second window has a size of 3 minutes. In consecutive task executions, the time slices of the second window will overlap. For instance, if the first task has a primary window starting at "12:19" and ending at "12:20", the 3 minute window will start at "12:17" and end at "12:20". Now the next task of this job, has a primary window starting at "12:20" and ending at "12:21", whereas the 3 minutes window starts at "12:18" and ends at "12:21". An example job implementation is given in Algorithm 3.

The most complex window definition is shown in part C). Here, the primary window is used to fetch data from CT C, whereas the other window is used to make the last minute of the output CT available as an input to the job. Such jobs are very useful whenever state information has to be kept over time. A detailed example of such a job is given in Algorithm 4.

Part D) shows a double window job. Such jobs are typically used to merge information from multiple sources, which provide different parts of the same data, e.g. two monitoring probes, each monitoring a different part of the network. Another typical usage scenario is information enrichment, e.g. one source could contain information about contacted IP addresses and another source contains DNS information, i.e. a mapping of IP addresses to host names. In this situation, a double window job can be used to combine the two information sources and produce a new stream containing contacted host names.

The concept of window definitions of CEL using primary windows is, to the best of our knowledge, a novel feature among stream processing languages. Other stream processing systems use different approaches to define windows and especially sliding windows. For instance, in the well known StreamBase [130] system windows are specified by a size and a slide definition in the following way: [SIZE x ADVANCE y TIME]. Whereas x defines the length of the window and y the amount of time after which new output is generated. The window definitions of StreamBase need the definition of a separate join statement if more than one stream should be used.

Another important improvement of the DBStream system over typical stream processing systems is that data is stored on disk after each task execution. Therefore, the state of all running jobs is always persisted to disk and can be recovered directly after a restart of DBStream. This has three advantages. First, it makes DBStream very robust against hardware failures since no state information is lost in case of a system crash or even a power outage. Second, streams can be replayed starting in the past, only limited

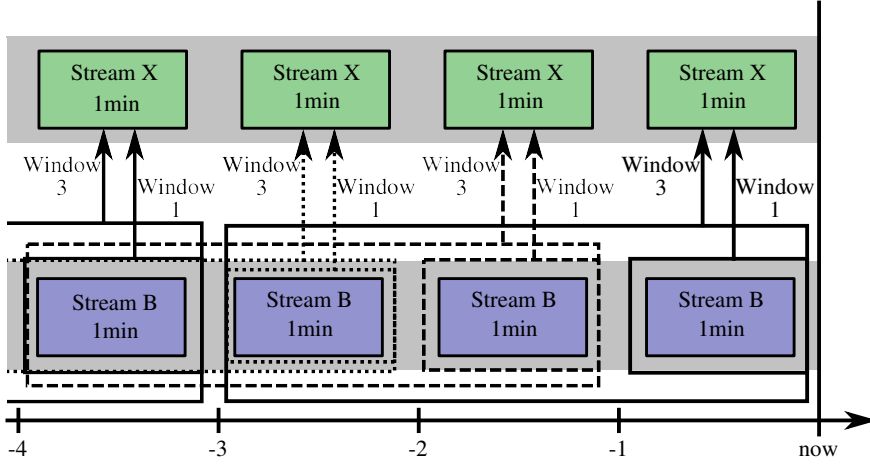


Figure 3.7.: Rolling average over the last 3 minutes, updated every minute.

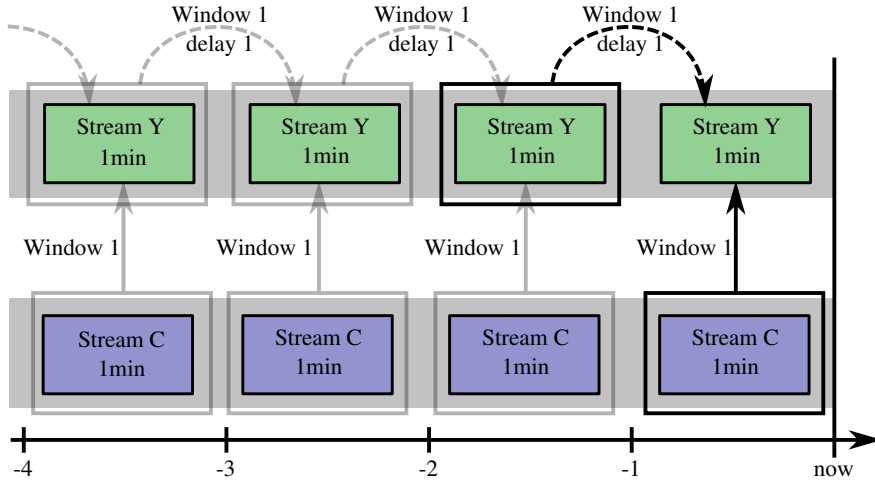


Figure 3.8.: Complex data processing flow for an incremental query.

by the amount of disk space available for a certain CT. Third, DBStream can efficiently process jobs accessing information which is long time in the past, e.g. a job can have the current day and the day same week day one month ago as input.

3.5.3. Incremental Data Processing

In this section, we explain a rolling window job and a complex incremental job in full detail, by giving two exhaustive examples.

3. A Novel Data Stream Warehouse Design

Example - Rolling Window Average

In the first example, we show how a rolling average calculation can be implemented in CEL.

Algorithm 3 Sliding window CEL job

```
<job inputs="B (window 1min primary) as B1, B (window 3min) as B3"
  output="X"
  schema="serial_time int4, avg_download float8, avg_upload float8" >
  <query>
select _STARTTS, avg(download), avg(upload) from B3
  </query>
</job>
```

Algorithm 3 shows a job definition computing the average of uploaded and downloaded bytes over a sliding or rolling of three minutes. The job has two input windows, where the primary input window B1, along which processing is advanced, is one minute long. The sliding or rolling window B3, which is used for the average calculation in the SQL query, is three minutes long and uses the same CT as input. Figure 3.7 visualizes which parts of the input B are used over a period of four task executions.

Example - Rolling Active Set

In this example we show how to incrementally compute the set of IP addresses active over the last hour, updated every minute.

Traditional large-scale batch processing systems as well as stream processing systems offer two different approaches to solve the given problem. One approach is that for every minute, the last hour is queried and the active set of IP addresses is computed. This approach is similar to the previous example and can be useful in certain situations, especially if performance is not crucial on e.g. small amounts of data. Since the same minute of data is scanned over and over again, 60 times in the given example, this approach can be very resource intensive if data are large. Another approach is to keep all unique IP addresses encountered in the last hour along with a timestamp in memory, representing an intermediate state. This approach is very efficient regarding the processing time, but the active set has to stay in memory. In case the system is stopped or crashes due to a hardware failure or power outage, the in-memory state has to be rebuild from past data, which might not be available anymore. In addition, in most Domain Specific Languages (DSLs), unlike in CEL, this type of jobs is not available right away. Typically,

UDFs have to be used to implement such a behavior.

Algorithm 4 Incremental CEL job

```

<job inputs="C (window 1min primary), Y (window 1min delay 1min)"
  output="Y"
  schema="serial_time int4, last int4, ip inet" >
  <query>
select _STARTTS, max(last), ip
  from (
    select _STARTTS as last, ip from C group by 1,2
    union all
    select last, ip from Y where last <= _STARTTS-60 group by 1,2
  ) t group by 1,3
  </query>
</job>

```

In the job implementation shown in Algorithm 4 we show how an incremental job is used to calculate the set of IP addresses active over the last hour, updated every minute. This is achieved by using the past output of the job as an input, delayed by one minute. As we will show in Section 4.3, this approach is much more efficient than traditional approaches. In addition, such a query stores all intermediate state on disk and can therefore be restart at any time by just loading one minute of output data.

The input to this query is C which holds, the IP addresses of active terminals. We now want to transform CT E into a new CT Y which contains for each minute, the distinct set of IP addresses active in the last hour. Therefore, we first add a timestamp called `last` to Y storing the time of the last activity of an IP address. Next, from the current minute of E, we generate a new tuple for each unique IP and set the last activity to the start of the window using the `_STARTTS` keyword. From the previous minute of the output stream Y we select all IP addresses, which were active less than 60 minutes ago. Now, we merge both results using the SQL `UNION ALL` operator and select from the result, for each distinct IP address, the current time, the maximum value of the last activity and the IP itself. This feedback loop allows us to efficiently compute the set of IP addresses active in the last hour every minute, without keeping any explicit state information. The windows used in this computation are shown in Figure 3.8.

3.6. Summary

In this chapter, we presented two system architectures for the processing of network monitoring data. The first system, called TicketDB had a focus on the parallel execution of long running analytical queries. The second system, DBStream is tailored for data stream processing of large shared workloads. We presented the flexible batched data stream processing language CEL and showed how it can be used in a network monitoring environment. In Chapter 4 we analyze the performance of both systems and compare them to modern NoSQL systems. The TicketDB system was used in the Comet project DARWIN3 [132]. The DARWIN3 project focuses on traffic monitoring and network-level measurements in a mobile networks. The DBStream system was the main analysis system used in the Comet project DARWIN4 [133], where it processed multiple terabytes of data every day. The DARWIN4 is a follow-up project to DARWIN3 with a similar objectives, here the focus is more on operational and real-time analysis of data from operational mobile networks. DBStream was used as well at Politecnico di Torino for processing data of the EU FP7 project mPlane [101]. The mPlane project is about the design of a distributed measurement infrastructure to perform active, passive and hybrid measurements. In the future, DBStream will serve as one of the main repositories for network monitoring data in the mPlane project.

In our opinion, the future of large-scale data processing appliances found in typical big data scenarios lies in distributed cluster processing systems like e.g. Spark [146] or the parallel database system Greenplum [110]. Especially, the newest branch of Spark which was called Shark [145] and now became Spark SQL² is very interesting. At the time we were designing DBStream, Spark and Spark SQL were not available yet. Now Spark SQL seems to be usable and a good candidate for replacing PostgreSQL as a data processing engine in DBStream. Another option would be to utilize the parallel database system Greenplum [110] as a data processing back-end. Since Greenplum is based on PostgreSQL, it seems a natural choice for the data processing and storage engine of DBStream.

²<http://spark.apache.org/sql/>

4. Experimental Data Stream Warehouse Performance Analysis

In this chapter we compare the performance of the TicketDB and DBStream system to well-known large-scale batch processing systems such as Hadoop and Spark. We start with an overview of typical benchmarks for Big Data systems in Section 4.1. We continue with Section 4.2, where we give a detailed performance comparison of the TicketDB system, presented in Section 3.2, to the well-known MapReduce system Hadoop. We finalize this chapter with Section 4.3, where we present a comprehensive comparison of the DBStream system, detailed in Section 3.4, to the well-known large-scale data processing engine Spark [146].

4.1. Related Work

Designing and developing a proper benchmark for Big Data solutions is a cumbersome task. In this section, we present an overview of already existing benchmarks and their applicability to the domain of assessing the performance of DSWs. Benchmarking computer systems has a long tradition and many different benchmarks have been developed over time. The two most renown organizations providing benchmarks are the Standard Performance Evaluation Corporation (SPEC) [116] and the Transaction Processing Performance Council (TPC) [93], both founded already in 1988. The benchmarks published by SPEC mainly focus on evaluating CPU and hardware performance, whereas software performance is not very important in the benchmark. Their most famous benchmark is the SPEC CPU2006, which is implemented as a combination of the CINT2006 benchmark, measuring integer performance, and the CFP2006 benchmark, measuring floating point performance.

The focus of the TPC benchmarks lies, as already indicated in the name, on transaction processing and therefore the combination of database software and hardware performance. In the early days of database system the main goal was on increasing the transaction throughput, which is important for On-Line Transaction Processing (OLTP)

4. Experimental Data Stream Warehouse Performance Analysis

workloads. This aspect is covered by the widely used TPC-C and E benchmarks, focusing on OLTP performance. Recently, long running analytical queries, typically used in OLAP workloads became more and more important. Therefore, the TPC-H and TPC-DS benchmarks were developed to measure the performance of decision support system used for executing OLAP workloads. Although the TPC-DS benchmark is the most evolved OLAP benchmark offered by the TPC, providing a good fit for evaluating system performance based on large-scale datasets, it is not very well suited for research purposes. Datasets can be produced from 10 GB of data up to 100 TB. But, in total 99 queries have to be executed and multiple rounds of updates have to be processed to be able to calculate the final benchmark score. The whole process of implementing the full benchmark is very time consuming and is not feasible for scientific performance evaluations, where the focus typically lies on improving only a specific part of the system. However, the datasets and some selected queries can be useful when evaluating system performance also for research purposes.

The recent growth of Big Data processing systems also inspired a growth of benchmarks for such systems. Among them are the Hive Benchmark [78], Gridmix [48], HiBench [75] and the Berkeley Big Data Benchmark [22]. All of which evaluate the performance of large-scale batch processing systems. The Linear Road benchmark [11] evaluates the performance of DSMSs, focusing on the processing of real-time data in second or milliseconds granularities. The main evaluation metric is latency, whereas throughput is not as important.

In DSWs latency is important as well, but typically latencies in the order of minutes are still acceptable. To the best of our knowledge no benchmark focuses on the performance of DSW, where data are imported and processed at the same time. Since this aspect is very important for network data processing, we propose two novel benchmarks for the evaluation of DSWs in the follow-up of this chapter.

4.2. Comparing TicketDB with Hadoop

In this section we compare the performance of the TicketDB system presented in Section 3.2 to the well known large-scale batch processing system Hadoop [140]. To evaluate the performance of the two systems, we define a novel benchmark and extract a dataset from publicly available network traces of a trans-pacific Internet link. On this dataset, in addition to the import, we define four processing jobs representing typical monitoring tasks performed in operational mobile and fixed-line networks. Since the details of the TicketDB system have already been presented in Section 3.2 we focus here on the design

considerations and specific implementation details of the benchmark and dataset. For the details of how the processing jobs were implemented in Hadoop we refer the interested reader to the following paper [15]. We conclude this section by a discussion of the lessons we learned from implementing and running this benchmark in two orthogonal approaches for the analysis of large amounts of data.

This section is based on the publication [15] which was done together with Fabio Ricciato, Pietro Michiardi and Antonio Barbuzzi. The part describing the benchmark implementation on the Hadoop system is the work of Pietro Michiardi and Antonio Barbuzzi and is included in this thesis for completeness reasons, to better understand the outcomes of the benchmark comparison with the TicketDB system. Fabio Ricciato helped with the design of the benchmark jobs from the network monitoring domain and improved the writing of the paper significantly.

4.2.1. Network Data Description

The input to our benchmark jobs are textual CSV files extracted from network traces from the publicly available MAWI archive. Those traces were originally captured from a trans-pacific link between the United States and Japan [35]. In particular, we use traces from sample-point F collected from 2009/03/30 to 2009/04/02. Packet payload is omitted and IP addresses are anonymized in the original trace. From this data we extract CSV files consisting of one record per packet. Each record has a size of approximately 64 Bytes and includes information like the timestamp, source and destination IP addresses, ports, flags, used protocol and packet length. The original trace consists of 432 GB raw data, resulting in several CSV files of roughly 100 GB in total. For a detailed description how this dataset is organized in TicketDB, please refer to Section 3.2.1.

Benchmark Jobs

We now specify four benchmark jobs representative for typical analysis tasks executed by network experts. In each job we compute statistics per hour, day and for the whole trace duration.

- **User Ranking (J1).** For every IP address, we compute the number of uploaded, downloaded and total exchanged (upload + download) bytes.
- **Frequent IP/Port by Heavy Users (J2).** Heavy users are defined as the top 10 IP addresses from job J1 in terms of total exchanged bytes. In this job we compute the top IP address/Port pairs for all heavy users.

4. Experimental Data Stream Warehouse Performance Analysis

- **Service Ranking (J3).** In this job, we split the set of all IP addresses found in the dataset arbitrarily into two subsets, $\mathcal{A}$ (servers) and $\mathcal{B}$ (clients). For every IP address in $\mathcal{A}$, this job computes the total number of connection requests arriving from IP addresses in $\mathcal{B}$.
- **Unanswered TCP Flows (J4).** For every IP address, this job computes the number of unanswered TCP flows. A TCP flow is considered *answered* if the initial SYN packet is acknowledged by a SYN-ACK within 3 seconds.

4.2.2. Hadoop Benchmark Design

MapReduce, popularized by Google with their work in [44], consists of two parts. A programming model and an execution framework that is deployed on a cluster of machines connected by a Local Area Network (LAN).

In MapReduce, input data are appropriately partitioned and typically stored on a distributed file system such as the Hadoop Distributed File System (HDFS). The execution of a job is split into three phases. The first phase is called the **Map** phase, in which a function specified by the user is applied on each entry in the dataset. The **Map** phase can be highly parallelized, in theory each entry can be "mapped" in a separate thread. The produced intermediate results are then stored locally on the file system of the node running the **Map** function. Before data is written on disk it is sorted and again partitioned appropriately. The second phase is called **Shuffle** and routes the intermediate data produced in the first phase to their corresponding **Reduce** nodes. The **Reduce** phase is the final step in the job execution of MapReduce. Here, the intermediate data received from one or more mappers is sorted and aggregated. The final data is then stored back into the distributed file system, from where it is available for users, other applications or further MapReduce jobs. The implementation of more complicated data analysis jobs typically requires the sequential execution of multiple MapReduce steps.

We now describe the main factors which need to be considered to optimize the performance of a job in terms of its makespan¹. For the data analysis jobs presented later in this chapter, disk and network I/O turned out to be the most resource consuming. Therefore, the task of the job designer is to carefully assign the right amount of memory to mappers and reducers, in order to minimize disk and network I/O. Another optimization approach is to introduce an optional **Combiner** phase which can be used to aggregate intermediate data before it is sent to the reducer nodes. This additional **Combiner**

¹The makespan is the wall clock time it takes to finish processing the whole job. It does not include any measurements of how many parallel workers are used to finish the job.

phase can help to reduce network I/O drastically. The optimization of jobs requires many manual steps and good knowledge of the underlying cluster infrastructure. The size of intermediate data, the number of available CPU cores, the total amount of available memory as well as the speed of the network connecting the cluster are parameters which need to be considered carefully.

4.2.3. Job Implementation in MapReduce

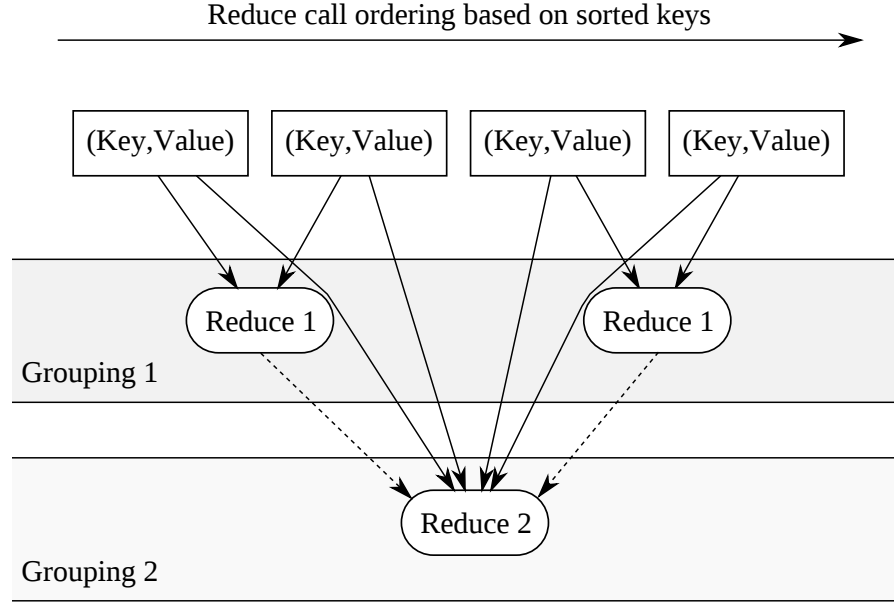
As most general purpose data processing frameworks also MapReduce offers multiple possible implementations of a single processing task. During the design and implementation of the benchmark, we evaluated multiple different implementations of the jobs of which we only present the best performing implementations in this section.

One commonality among all our jobs is that they aggregate data on different time scales, i.e. per hour, per day and per week. In the most simplest implementation, one can just launch one MapReduce job for each time granularity. But, since this would lead to multiple scans of the same input data unnecessary I/O overhead is the result. As we will show, it is possible, by carefully specifying the **Reduce**, **Grouping**, **Sorting** and **Partitioning** phases, to reduce the number of scans on the input data and therefore increase the performance of the processing jobs.

In this section, we introduce a novel design pattern that we call *in-reducer grouping*. This design pattern is somehow similar to *grouping sets* or the *ROLLUP* and *CUBE* operators found in commercial parallel SQL database systems, but applied to MapReduce in our implementation.

In Hadoop, the **Grouping** phase is used to prepare the intermediate data before they are send to the reducers. Therefore, key/value pairs are grouped by the key. Then, the reducers receive sets of (**key**, **list<value>**) from which the final results are computed. Normally, the **Grouping** phase is implicit in Hadoop and executed automatically by the framework.

The main idea of our approach is to move the definition and execution of the **Grouping** from the framework level to the user-code level and execute multiple grouping stages with only one pass over the data. Figure 4.1 gives a general overview of our *in-reducer grouping* technique. It is shown how a single reduce node is executing the adapted **Reduce** phase. In our approach, the key/value pairs are sorted and processed by the reduce function in sort order. This enables us to process the same input item with multiple reduce functions. In Figure 4.1 we show the use of two reduce functions named **Reduce 1** and **Reduce 2**. Please note, that that *in-reducer grouping* approach benefits the most from hierarchical grouping stages, where all grouping stages share the same grouping

Figure 4.1.: An illustration of the *in-reducer grouping* technique.

key. In this case, higher grouping levels are exact inclusion of lower grouping levels, leading to the highest performance increases of our approach. For instance in our later job executions we aggregate data per hour, per day and per week. In this special case we can directly use the output of one reduce phase as the input to the next, indicated by the dotted lines in Figure 4.1.

Although we are aware that other high-level languages for Hadoop like e.g. Pig Latin [105], Hive QL [134] and JAQL [24] exist, at the time of executing the presented benchmark none of those were offering a performance comparable to a plain MapReduce implementation. An extensive comparison of those data processing language in terms of expressiveness as well as processing performance is given in [127]. Please note, that none of those languages is able to handle *in-reducer groupings* as presented above out-of-the-box.

4.2.4. Performance Evaluation

In the following, we present a performance evaluation, comparing the two parallel approaches TicketDB and Hadoop MapReduce. This comparison should not be understood as a general comparison of parallel database systems against MapReduce systems such as Hadoop. Instead, we would like to understand the performance differences of certain parts of the system, as well as their impact on overall performance, given a network monitoring setting.

4.2. Comparing TicketDB with Hadoop

The Hadoop platform is designed to run on a cluster of machines, which are interconnected via a fast network. In contrast, the TicketDB system, in its current implementation, only supports single node parallel installations. Therefore, the choice of comparable hardware was very tough. Finally, we decided to run DBStream on a single high-performance server machine and Hadoop on a set of 11 cluster nodes which have similar hardware costs as the single server machine. Both deployments had a cost of approximately 15 thousand € at the time the experiments were executed. Please note that those costs only include the hardware purchase, whereas other costs like energy, cooling or maintenance overhead are not included in the given costs. In addition, scaling the performance of each system does result in different cost scalings. Whereas MapReduce systems are supposed to scale linearly in terms of performance (although we will show this is not always the case in Section 4.3) and money, scaling up a single high performance server machine can be very expensive. The authors of [99] compare the scale-out and scale-up paradigms in terms of performance and costs. They conclude that for highly parallel workloads, like e.g. web search, scale-out outperforms scale-up in terms of hardware costs, but they also note that management and maintenance costs increase linearly with the number of nodes in the cluster.

The TicketDB system was installed on a single high-performance server machine, hosting two hexa-core Intel XEON X5670 CPUs at 2.93 GHz with 48 GB of RAM installed. As a storage subsystem two fiber channel attached storages with twelve 750 GB disks each, running RAID6 were used. The Hadoop system was installed on 11 Amazon EC2 nodes running Hadoop 0.21. All EC2 nodes were interconnected via a 10 GBit Ethernet and had 7.5 GB of available RAM, two virtual CPU cores and 850 GB of disk space.

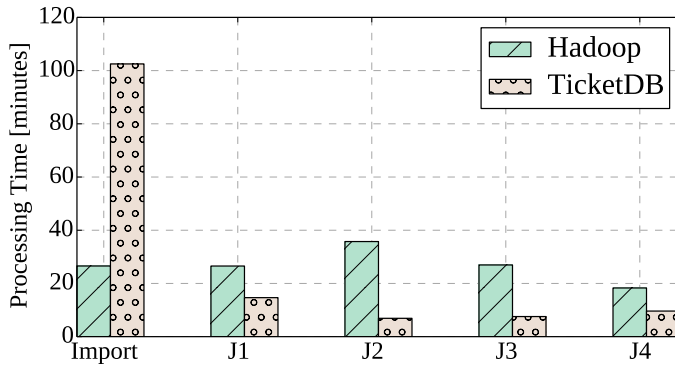


Figure 4.2.: Summary of job durations in both parallel systems.

4. Experimental Data Stream Warehouse Performance Analysis

In Figure 4.2 the processing times of the import and each individual job are shown. During the run of the experiments, we also measured the maximum aggregated I/O throughput of both systems. The Hadoop cluster achieved a maximum I/O read of approx. 200 MB/s. In contrast, the TicketDB system was able to achieve up to approx. 650 MB/s. This difference partially explains the different job executions timings. Please also note, that given the setup of the Amazon EC2 cluster it is possible that multiple virtual machines are launched on the same physical machine during the run of the benchmark. This could lead to performance degradation. The authors of [76] report a high variability in the processing time of consecutive executions of the same task in the Amazon EC2 cloud. In order to avoid such issues we were executing the same job 10 times and report only the average score in Figure 4.2.

For an overview of the TicketDB system, including an description of how jobs were implemented please refer to Section 3.2. In the TicketDB system, input data is only scanned completely once during the import. Especially the split into dimension and fact data takes a considerable amount of time, thus decreasing the import performance as compared to Hadoop, where importing data is just a copy to the HDFS. After this initial scan, data are stored in the PostgreSQL format in fact and dimension tables. The execution of J2 can benefit hugely from this split, since only the smaller dimension table has to be accessed. However, the other jobs need still to access and join dimension and fact data, but are still much faster than their counterparts in Hadoop.

Although, the import takes longer time in the TicketDB system, all executed jobs are up to a factor of 6 times faster than in the optimized version of Hadoop used for the experiments.

4.2.5. Lessons learned

In this section we compared two different approaches to large-scale network data analysis in terms of performance. We compared the TicketDB system presented in detail in Section 3.2 to an alternative system based on Hadoop MapReduce. Our experiences can be summarized as follows.

In the **TicketDB** case, we designed a system from scratch: thus, a major development effort has been put in building key components such as data management and partitioning. Although the SQL statements implementing the sample analysis jobs considered in this section are not at all trivial, they are much more concise than their MapReduce counterparts.

In the **Hadoop** case, major effort was put into the design and implementation of the MapReduce jobs. This also involved "hacking" the underlying framework for more

computationally efficient job execution. For complex operations, the same job could be implemented in several ways: our results show that the naive approach of concatenating simple jobs was far from being efficient. Instead, a common design pattern, we called *in-reducer grouping*, produced the best performing implementations for the given jobs. This involved using MapReduce as system to partition and route data to complex **Reduce** functions. Moreover, a substantial amount of work has been devoted to the optimization of Hadoop, by appropriately tuning its several parameters to minimize I/O operations, which are the main cause for the decreased performance of the jobs.

In conclusion, this performance comparison has shed light on why large-scale network analytic applications require a deep understand of both system and algorithmic aspects of data processing. We showed that although MapReduce has the potential of decoupling system design from the actual data analysis tasks, in reality the boundary between these tasks is often very small, especially in case of optimizations. We also showed that, despite the lack of open-source implementations of parallel databases, the design of the key component of a shared-nothing architecture is not a daunting task. However, the expressiveness of standard query languages such as SQL can not be immediately exploited, since schema design and query optimization is tightly coupled with data partitioning.

4.3. Comparing DBStream with Spark

In this Section, we compare the performance of the DBStream system presented in Section 3.4 to the state-of-the-art Big Data framework Spark.

Please note that this Section is largely based on the publication [18]. The design and implementation of the benchmark was done by the author of this thesis. Whereas an optimization and the execution of the benchmark on the Spark cluster were done by people from Politecnico di Torino.

4.3.1. Spark Introduction

Spark is an open-source MapReduce solution proposed by the UC Berkley AMPLab. It exploits Resilient Distributed Datasets (RDDs), i.e., a distributed data abstraction which allows in-memory operations on large clusters in a fault-tolerant manner [146]. This approach has been demonstrated to be particularly efficient [22] enabling both iterative and interactive applications in Scala, Java and Python. Moreover, an application does not strictly require the presence of a Hadoop cluster to take advantage of Spark. In fact, the system offers a resource manager and supports different data access mechanisms. However, it is most commonly used in combination with Hadoop and the Hadoop Distributed File System (HDFS).

In this context, it is also important to mention Spark Streaming [147], a recent evolution of Spark, enabling real-time analysis through the processing of small time batches. Of particular interest are the system primitives for defining sliding windows and develop continuous queries similarly to what was discussed in Section 3.5.2. Unfortunately, the time batches used in Spark are defined by the system time and not by the application time embedded in the data. Therefore, Spark Streaming targets mainly real-time analysis scenarios but offers limited support for processing already collected data. On the one hand, the purpose of NTMA is to perform analysis in real-time, on the other hand it is common that monitoring solutions generate high precision timestamps already during the generation of the data. Therefore, the post-processing of already collected data for NTMA applications targeted in this thesis is not possible with Spark Streaming. Recent discussions on the mailing list of Spark Streaming suggest some possible workarounds². Although, very interesting, we leave the evaluation of Spark Streaming in the context of network monitoring applications as future work.

²<http://apache-spark-user-list.1001560.n3.nabble.com/window-analysis-with-Spark-and-Spark-streaming-td8806.html#a9185>

4.3.2. System Setup and Datasets

We installed Spark on a set of eleven machines of the following identical hardware: a 6 core XEON E5 2640, 32 GB of RAM and a 5 disks of 3 TB each. One of those eleven machines has been dedicated to DBStream, recombining 4 of the available disks in a RAID10. We use PostgreSQL in version 9.2.4 as the underlying DBMS. The remaining 10 machines compose a Hadoop cluster. The cluster runs Cloudera Distribution Including Apache Hadoop (CDH) 4.6 with the MapReduce v1 Job Tracker enabled. On the cluster we also installed Spark v1.0.0 but we were only able to use the standalone resource manager³.

All machines are located within the same rack connected through a 1Gb/s switch. The rack also contains a 40 TB Network-Attached Storage (NAS) used to collect historical data. In particular, in this work we use four, 5 day-long datasets, each collected at a different vantage point in a real ISP network from the 3. to the 7. of February 2014. Each vantage point is instrumented with Tstat [57] to produce per-flow text log files from monitoring the traffic of more than 20,000 ADLS households. For each TCP connection, Tstat reports more than 100 network statistics and generates a new log file each hour. Overall, each of the four dataset corresponds to approx. 160 GB of raw data, about 5 times the memory available on a single cluster node. In total, the four datasets sum up to approx. 650 GB, which is about twice as large as the total amount of memory available in the whole cluster.

4.3.3. Job Definition

Based on our experience in the design of network monitoring applications and benchmarks for large-scale data processing systems, we define a set of 7 jobs that are representative of the daily operations we perform on our production Hadoop cluster. We chose those jobs also based on the experience we gathered in designing and executing

³Apparently the implementation of Yarn provided in CDH 4.6 has some incompatibilities with Spark. These seem to be solved in CDH 5 providing Yarn by default and a package for Spark v1.0.0. Unfortunately, testing such configuration requires an upgrade of the operating system of each cluster node, i.e., a sensible modification for our production environment.

4. Experimental Data Stream Warehouse Performance Analysis

the benchmark for TicketDB and Hadoop discussed in Section 4.2.1.

Import imports the data into the system from the NAS, where raw data is stored in files of one hour each.

J1 for every 10 minutes i) map each destination IP address to its organization name through the Maxmind Orgname⁴ database and ii) for each found organization, compute aggregated traffic statistics, i.e. min/max/avg Round Trip Time, number of distinct server IP addresses, total number of uploaded/downloaded bytes.

J2 for every hour, i) compute the organization name-IP mapping as in J1, ii) collect all data having organization names related to the Akamai CDN, and iii) compute some statistics, i.e. min/max/average Round-Trip Time (RTT), aggregated for the whole Akamai service.

J3 for every hour, i) compute the organization name-IP mapping as in J1, and ii) select the top 10 organization names having the highest number of distinct IP addresses connecting to them.

J4 for every hour, i) transform the destination IP address into a /24 subnet, and ii) select the top 10 /24 subnets having the highest number of flows.

J5 for every minute, for each source IP address, compute the total number of uploaded/downloaded bytes and the number of flows.

J6 for every minute, i) find the set of distinct destination IP addresses, and ii) use it to update the set of IP addresses that were active over the past 60 minutes.

J7 for every minute, i) compute the total uploaded/downloaded bytes for each source IP address, and ii) compute the average over the past 60 minutes.

Overall, these jobs define network statistics related to CDNs and other organizations (J1 to J4), statistics related to the monitored households (J5) and two incremental queries (J6 and J7) computing aggregated statistics over rolling sets of IP addresses.

4.3.4. Benchmark Implementation

Each analysis engine has different peculiarities, properties and tuning options. In addition, different implementations are possible for the specified benchmark jobs. In this

⁴The Maxmind Orgname database provides a mapping between IPs and Organization names see www.maxmind.com.

section, we provide an implementation which we consider reasonable and discuss possible modifications that could affect overall performance.

DBStream Benchmark Implementation

All queries are implemented in the Continuous Execution Language (CEL) of DBStream, described in Section 3.5. The fact that the output of a job is stored on disk and can be used as input to another job is exploited to achieve increased processing performance. Figure 4.3 shows the resulting job dependencies, where the nodes represent the jobs and an arrow from e.g. job J1 to J2 means that the output of J1 is used as input to J2. The number next to each arrow indicates the size of the input window in minutes. For example, in order to compute the results of J6 we first gather the set of active IP addresses per minute in J6 prepare. Then, J6 uses J6 prepare and its own past output as input for the computation of the next output time window. This is indicated by the reflexive arrow starting from and going back into J6.

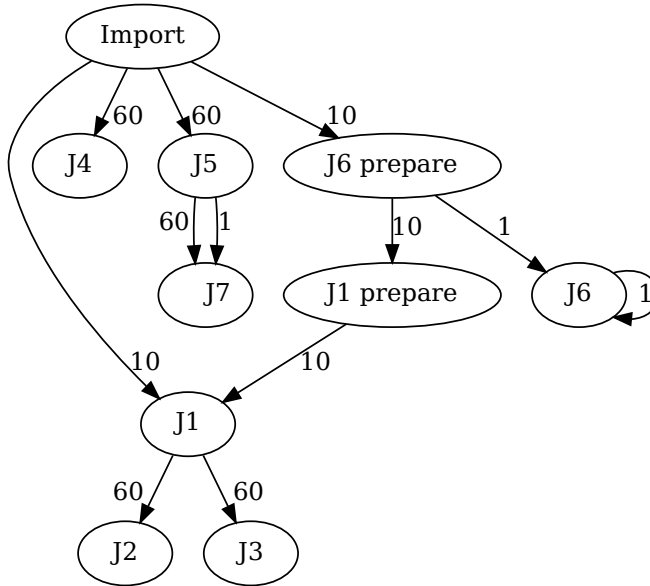


Figure 4.3.: Job inter-dependencies for the DBStream job implementation.

The main bottle neck of the DBStream system in this setup is the disk sub-system. Therefore, we were trying to minimize the amount disk I/O by intelligent scheduling. Typically tasks are scheduled in FIFO order in DBStream. Since we set the number of parallel tasks to 64, FIFO effectively results in all tasks being executed as soon as the input data is ready. The effect of the FIFO scheduling is shown in Figure 4.4 (top),

4. *Experimental Data Stream Warehouse Performance Analysis*

where each point in the plot corresponds to the execution of one window. The x-axis of this figure corresponds to the time after the start of the experiment at which a certain task finished execution. The y-axis corresponds to the time when the data item was created by the vantage point, normalized to the start of the whole dataset. Since some jobs process faster than others, in the FIFO case, the time distance between those jobs increases over the run of the experiment. The first step is the data import, which not only puts data onto the disk, but also into the disk cache of the Operating System (OS). As soon as the difference in progress of different jobs needing the same input gets too big, the data of the input drops out of the cache and has to be read from disk again. This increases the I/O overhead and, at the same time, decreases the overall system performance of DBStream.

Let us explain the underlying processes in more detail. For example, let's imagine the import has a fixed size of 8 GB per hour of data and takes 1 minute to finish processing. Let's assume as well, that there are only two jobs A and B defined on top of the import and A needs 1 minute to process and B 2 minutes. Now we start the experiment and the first job which can execute is the import which needs one minute to finish. Then, both jobs A and B start to process hour 1 and the import starts processing hour 2. All those three tasks are executed in parallel in DBStream. After another minute has past, the import has finished hour 2 and starts processing hour 3. Job A has finished hour 1 and now starts processing hour 2. Since the import and job A have the same processing time, their progress will only get, in the worst case, one hour apart from each other. One hour of data corresponds to 8 GB and let's assume that there are 16 GB of RAM available for disk cache. After the import has finished processing one hour, this hour automatically is available in the cache from where it can be used by job A. Since the import and job A never get more than one hour apart from each other, no data has to be read from disk twice. In contrast, for job B, the situation is very different. Since the processing time of job B is 2 minutes and therefore twice as long as the import, for every imported hour, the time difference between job B and the import increases by one hour. Therefore as soon as this difference gets longer than 2 hours, imported data does not fit in the disk cache anymore and has to be fetched from disk again when it is needed by job B. This results in an increased I/O overhead since data need to be read from disk multiple times, thus decreasing the performance of the whole system. In the following Chapter 5, we discuss how to schedule processing jobs in order to reduce the maximum amount of needed cache. Although the approach presented there is already able to improve the performance for the single threaded case it is not yet ready to be applied in DBStream.

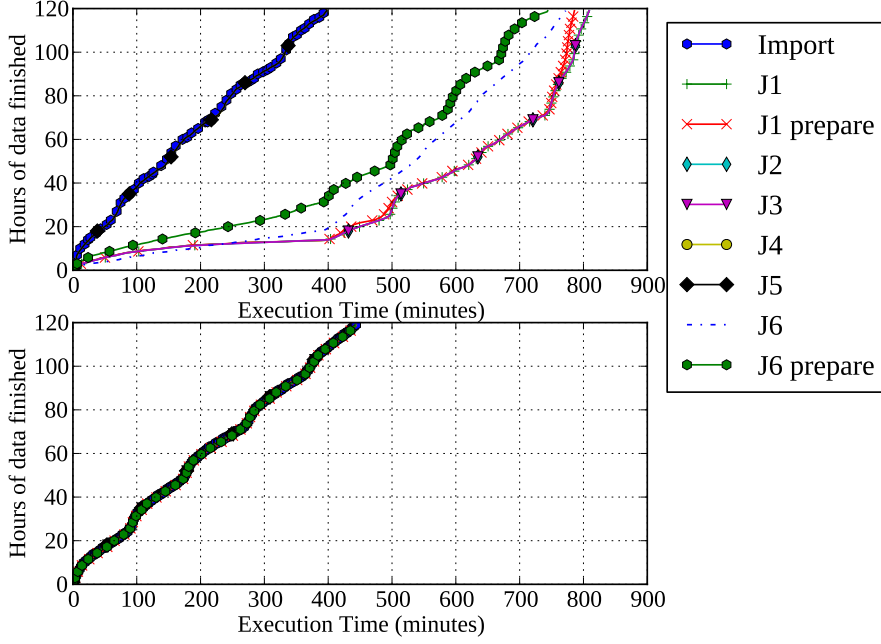


Figure 4.4.: DBStream task execution over time for FIFO and Shared scheduling.

Figure 4.4 (bottom) shows execution of the same set of jobs using a "shared" scheduling strategy. In the "shared" case, a new hour is only imported if the difference in time between the imported hour t_i and the hour t_j for which all jobs have finished processing is smaller than x .

$$t_i - t_j < x \quad (4.1)$$

In this case, data stays in the OS cache and fewer IO operations are needed to complete the experiment. By setting $x = 1$, we are able to reduce the execution time of 4 Vantage Points (VPs) by a factor of 45% from 808 minutes to 446 minutes.

Spark Benchmark Implementation

Each job is implemented as a separate Spark application using Scala. The program receives a list of files located on HDFS as input and processes them sequentially. The jobs J1 to J5 have a straight forward implementation, since they do not have strong data dependencies and data are already split per hour. However, the two incremental queries are more complex to implement. In fact, we were forced to implement the logic to store and update the time windows from scratch. For this purpose, we consider a very simple

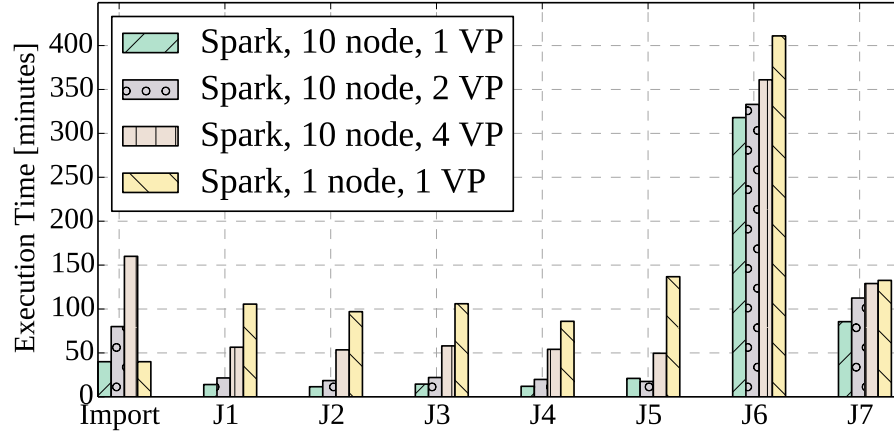


Figure 4.5.: Performance numbers for different setups using Spark.

approach composed of two nested loops. We first loop over the list of files creating an RDD collecting per-minute time windows. Then, we loop over each new time window, re-aggregating it with an array of data storing the previous 60 minute time windows. After applying several optimizations, like e.g. changing the internal storage of IP addresses, we were able to reduce the processing time for J6 from 366 to 318 minutes for the one VP case. Typical optimizations used for such a problem like skip lists or complex tree structures are hard to parallelize and would not be a fair comparison to a declarative language like CEL.

Figure 4.5 shows the results we were able to achieve running Spark on our cluster of 10 nodes. For the jobs J1 to J5, Spark offers great performance and the whole cluster is perfectly able to parallelize processing, resulting in very high performance. However, the jobs J6 and J7 are not processed very fast. Especially J6 can not be parallelized very well, which is mainly influenced by two factors:

1. Since data have to be synchronized and merged in one single location after each minute job parallelization suffers.
2. Distinct sets have to be computed for every minute in the data. In typical MapReduce applications the number of data items is brought down significantly by each reduce stage. In contrast, in the given application the number of data items is not reduced much since the set of used IP addresses stays rather stable over consecutive minutes. Therefore, huge amounts of data have to be moved between nodes in the reduce phase. We think the network interconnection between the nodes gets overloaded causing the bottle neck of J6.

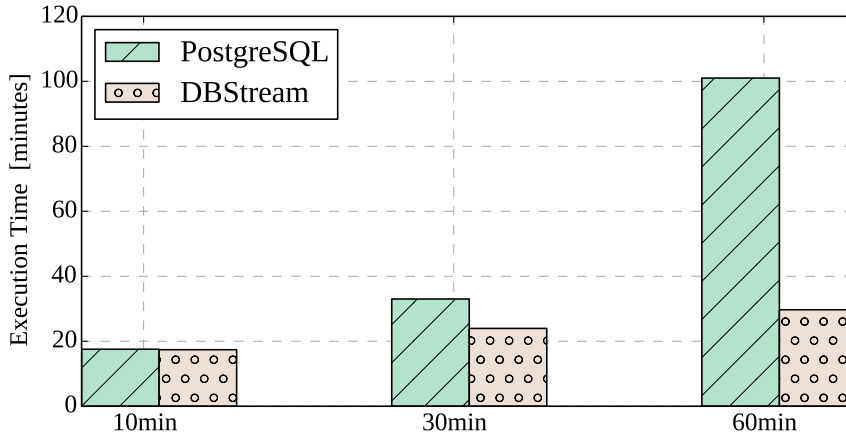


Figure 4.6.: DBStream performance compared to PostgreSQL.

Also for J7 the computation has to be synchronized for every minute. Although also this job does not parallelize very well, the effect is not as strong as for J6. In this case, the amount of data which needs to be exchanged between nodes is much smaller, since the output for every minute is only one single number.

4.3.5. System comparison

In our first experiment, we show the efficiency of incrementally computing the set of active IP addresses as compared to re-computing the result from scratch for each individual minute. The results of this comparison are visualized in Figure 4.6. We use the DBStream job presented in Section 3.5.3, calculating the set of IPs active over a moving window (of e.g. 60 minutes) and compare its performance to a regular PostgreSQL implementation of the same job. We fix the primary window to be one minute long and vary the size of the secondary, sliding window, evaluating three variants: finding all the active IP addresses within the past 10 minutes, 30 minutes and 60 minutes. Intuitively, as the length of the sliding window referenced by the query increases, we expect the performance advantage of incremental processing to increase. For example, in case of a 60 minute window, computing a new partition of the view from scratch requires accessing 60 separate partitions of the base table. In contrast, incremental processing requires accessing only two partitions: the previous result and the new window of base data. The results of this experiment, shown in Figure 4.6, confirm the hypothesis. The provided numbers correspond to the execution time on one day of data. For a 10 minute time window, the incremental approach shows only a very slight advantage over the regular

4. Experimental Data Stream Warehouse Performance Analysis

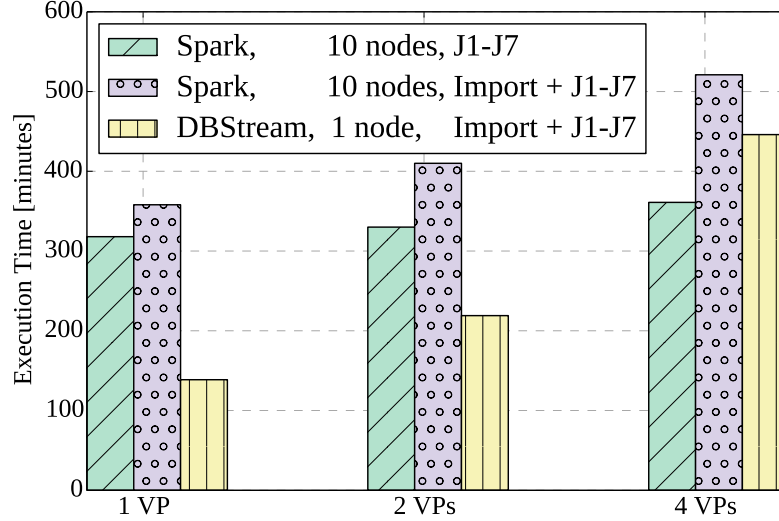


Figure 4.7.: Comparison of DBStream and Spark.

PostgreSQL approach. However, for 30 minute time windows, the incremental approach is noticeably faster, and for the 60 minute time window, the incremental approach is over three times faster than the regular approach.

Finally, in Figure 4.7 we compare the performance of Spark and DBStream in terms of *makespan*. In DBStream, the total execution time is measured from the start of the import of the first hour of data until all jobs finished processing the last hour of data. For Spark all jobs were started at the same time in parallel. We report the total execution time of the job finishing last, which was J6 in this experiment. Since for Spark, the import is done before the jobs start processing the data, we also report the job processing time plus the time it takes to import the data separately.

For DBStream, the execution time increases nearly linearly with the number of VP and therefore the amount of data to process. In contrast, for Spark the main factor is the execution time of J6. The total execution time does not increase much with more VPs, since multiple instances of J6 run in parallel. Therefore, Spark is able to utilize its parallel nature better the more jobs are running, whereas DBStream shows better performance for incremental jobs. For the one VP case, Spark, running on a 10 node cluster, takes 2.6 times longer than DBStream, running on a single node of the same hardware, to finish importing and processing the data.

4.4. Summary

In this chapter, we analyzed the performance of the two data stream warehouse systems we proposed in this thesis. The TicketDB system showed a very good performance as compared to the widely adopted Hadoop system. DBStream was compared to the novel high performance large-scale data processing engine Spark and was able to be on par with a cluster of 10 nodes for certain selected queries.

In the future, we plan to extend DBStream and further increase its performance. Especially, we want to design and implement an extended version of the scheduling presented in Section 5 and apply it to the DBStream system. In addition, we also plan to design an open source benchmark for DSW systems, using either a synthetic data generator or publicly available data. This novel benchmark would be good contribution to the research community since there is a clear lack of benchmarks focusing on fast Big Data applications.

5. Cache-Oblivious Scheduling of Shared Workloads

Shared workload optimization is feasible if the set of tasks to be executed is known in advance, as is the case in updating a set of materialized views or executing an Extraction Transformation and Load (ETL) workflow. In this chapter, we consider data-intensive workloads with precedence constraints arising from data dependencies. While there has been previous work on identifying common subexpressions and task re-ordering to enable shared scans, in this chapter we solve the problem of scheduling shared data-intensive workloads in a cache-oblivious way. Our solution relies on a novel formulation of precedence constrained scheduling with the additional constraint that once a data item is in the cache, all tasks that require this item should be executed as soon as possible thereafter. We give an optimal algorithm using A* search over the space of possible schedules. In addition, we propose efficient and effective heuristics that obtain nearly-optimal schedules in much less time. We present experimental results on real-world data warehouse workloads and the TPC-DS benchmark to validate our claims.

Several data management scenarios exist in which the workload consists of a set of tasks known in advance. For example, ETL processing involves executing a predefined workload of operations that pre-process data before inserting them into the database. Another example is data stream processing and publish-subscribe systems, in which a predefined set of queries is continuously executed on incoming data. Also, in data warehouses, materialized view maintenance is often done periodically, in which all views are updated together.

Previous work has recognized optimization opportunities in these scenarios, referred to as shared workloads, including scan sharing, shared query plans and evaluating common sub-expressions only once [111]. In this chapter, we address the following problem: given a shared workload, even after identifying common sub-expressions and shared scan opportunities, it is still not clear *what is an optimal ordering of tasks that minimizes cache misses?* Furthermore, since we may not know the exact amount of cache that is available to the data management system at a given time, we want to generate the task

5. Cache-Oblivious Scheduling of Shared Workloads

ordering in a *cache-oblivious way*, i.e., in a way that exploits caching without knowing the cache size.

Throughout this chapter, we will use the term “cached results” in a general sense. Depending on the application, this could refer to the disk-RAM hierarchy or the RAM-cache hierarchy.

This chapter is based on the publication [19], which will be presented in April 2015 at 31st IEEE International Conference on Data Engineering (ICDE) 2015. The paper is a joint work done together with Lukasz Golab, Stefan Ruehrup, Mirko Schiavone and Pedro Casas. Lukasz Golab helped with the introduction and forming and advancing the main idea of this work. Stefan Ruehrup helped with the theoretical parts of the paper and the development of the WTMB cost function. Mirko Schiavone is responsible for running most of the experiments on the database machine. Last but not least, Pedro Casas provided general guidance and helped with the introduction as well as the motivation of this work.

5.1. Related Work

The work presented in this chapter is related to three research areas: cache-oblivious algorithms, shared workload optimization and scheduling.

Cache-oblivious algorithms have been studied for over a decade, beginning with Frigo et al. [59]. The idea is to make nearly optimal use of the cache without knowing its size. A common approach has been to divide-and-conquer a given problem so that at some level the resulting sub-problems are small enough to fit in the cache, regardless of the size of the cache. There has been some very recent work on cache-oblivious scheduling [5], but only for the special case of a chain of streaming operators, which is not directly applicable to our problem of scheduling a DAG of update tasks.

In **shared workload optimization**, there has been work on efficiently refreshing a data warehouse consisting of a collection of materialized views. One set of optimizations focuses on sharing work among similar views; examples include finding common sub-expressions among similar views [90, 100] and choosing an optimal view graph when there are many possible source views to choose from [58]. There is also similar work in shared data processing systems, e.g., computing a global query plan [12, 64], sharing work across MapReduce jobs [103] and enabling shared table scans [65, 94, 148]. In this chapter, we address a complementary issue: once a shared query plan is found, we order the execution of the tasks to maximize the chances of reusing cached results.

The other set of optimizations addresses ordering of view updates and, to the best of

our knowledge, there is no prior work on cache-oblivious ordering. Labio et al. considered select-project-join views and ordered the updates according to whichever view has the smallest delta [84]. Golab et al. proposed a scheduling algorithm for view refresh in an online data warehouse, which orders jobs by their improvement in freshness divided by their processing time [68]. Their algorithm is for an online setting, where new data arrive asynchronously and the full workload is not known upfront, therefore work sharing may not be feasible.

Similarly, there has been work on query re-ordering to maximize sharing opportunities. Agrawal et al. addressed the problem of scheduling file scans given an expected frequency of queries accessing each file [6]. Ahmad et al. [7] studied query re-ordering to take advantage of positive interactions among queries. Wolf et al. [142] considers reordering MapReduce jobs to enable shared scans. In our solution, we also reorder tasks to take advantage of sharing, but in a cache-oblivious way. Gupta et al. studied the problem of choosing which common sub-expressions to materialize in order to speed up the evaluation of a sequence of queries, given a fixed-sized cache to store the subexpressions [71]. In contrast, our solution does not require the knowledge of the cache size.

Finally, from a **scheduling** point of view, as we mentioned earlier, previous work on precedence constrained scheduling [33], directed optimal linear arrangement and directed bandwidth [41] cannot model our objective of minimizing the distance between related tasks that require the same data item. Scheduling with sequence-dependent setup times [9] is also related to our approach. In this problem, the execution time of each task includes some setup time that depends on all the tasks that have been executed up to now, plus the actual task processing time. However, this work mostly considers production scheduling where physical objects are involved, leading to different assumptions and objectives.

5.2. Motivating Example

While the solution presented in this chapter is applicable to any data-intensive shared workload (i.e., where data I/O is the bottleneck, not CPU), our motivation for studying cache-oblivious task ordering comes from Data Stream Warehouses (DSWs) such as DataDepot [66] and the DBStream system presented in Section 3.4. DSWs are a combination of traditional data warehouse systems and stream engines. They support very large fact tables, materialized view hierarchies and complex analytics; however, in contrast to traditional data warehouses that are usually updated once a day or once a

5. Cache-Oblivious Scheduling of Shared Workloads

week, DSWs are refreshed more often (e.g, every 5 minutes) to enable near real-time processing and queries over historical data. Example applications include network, data center or infrastructure monitoring, data analysis for intelligent transportation systems and smart grid monitoring.

Since the “claim to fame” of DSW systems is their ability to ingest new data and refresh materialized views frequently, view maintenance must be performed efficiently. The system must finish propagating one batch of new data throughout the view hierarchy before the next batch arrives. Otherwise, at best a backlog of buffered data will build up, at worst data will be lost and is not available for future analysis.

For example, consider the simple view hierarchy shown in Figure 5.1, with 0 and 1 being base tables and the other nodes corresponding to materialized views. Note that some views (e.g., 2 and 4) are computed directly over base tables while others are computed over other views (e.g., 5). This is an example of a predefined workload that a DSW might repeatedly execute when a batch of new data for tables 0 and 1 arrives.

The view hierarchy forms a precedence graph. When a batch of new data arrives for table 1, we first insert it into table 1 and then we can use it to update views 2 and 4. Since view 5 needs view 2 as an input, it can only start processing after view 2 was updated. Thus, a legal ordering of view updates must satisfy the given precedence constraints i.e., we cannot update view 5 if we have not yet updated view 2.

However, different legal orderings may lead to different cache performance. For example, right after updating table 1 with new data, that new batch of data is likely to be in the cache. Therefore, we should then update the views 2 and 4 while the new batch of data is still in the cache. On the other hand, if we update table 0 in between views 2 and 4, then the new batch of data from table 1 is more likely to be evicted and will have to be reloaded before updating view 4. Put another way, we would need a larger cache to avoid cache misses.

5.3. Challenges and Contributions

Even in the simple example shown in Figure 5.1, it is not obvious which ordering minimizes cache misses. It makes sense to update views 2 and 4 immediately after updating table 1, but should we update view 2 before 4 or vice versa? As the number of tasks in the workload and their data dependencies increase, so does the complexity of choosing an efficient ordering. Furthermore, in practice we usually do not know exactly how much cache is available for a given task at a given time. For instance, in a DSW, view updates compete for resources with ad-hoc queries.

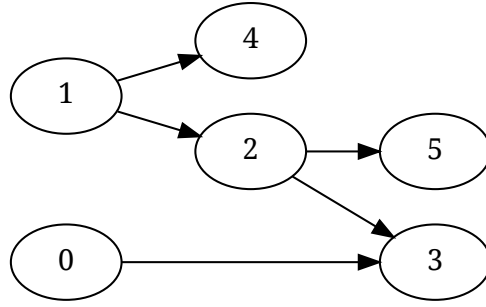


Figure 5.1.: A precedence graph corresponding to two base tables (0, 1) and four materialized views (2, 3, 4, 5).

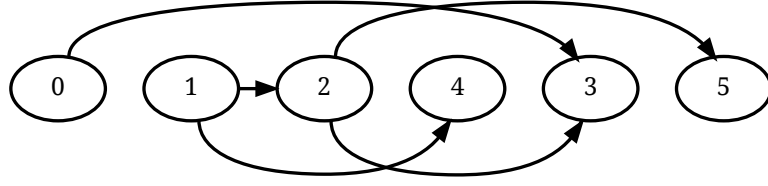
The intuition behind our solution is simple: tasks that require a certain data item should be scheduled as soon as possible after this item is placed in the cache. Otherwise, other tasks that require other data items will be scheduled, increasing the likelihood that the original data item will be evicted from the cache. In other words, we need to minimize the amount of time a data item (e.g., a new batch of data loaded into a materialized view) spends in the cache until all the subsequent tasks that need it have been executed.

For example, Figure 5.2 illustrates two possible legal orderings of the five tasks from Figure 5.1, obtained by linearizing the view precedence graph. For each node that includes at least one outgoing edge (e.g., each task that produces data required by other task(s)), we can compute how long these data must remain in the cache. At the top of the figure, the “distance” between table 0 and view 3, which requires data from table 0, is four, i.e., three other tasks will run in between. For view 2, the maximum distance is three, since both view 3 and view 5 need data from view 2 and a total of three view updates will run from the time view 2 data are inserted into the cache until the updates of view 3 and view 5 are completed. On the other hand, the ordering shown at the bottom of the figure has a distance of only one between table 0 and view 3—they are executed one after the other and data from table 0 is more likely to still be in the cache at the time of execution of view 3. The idea behind our approach is to minimize the distance between related tasks and therefore decrease the possibility of cache misses, without having to know the cache size (our notion of distance will be formalized in Section 5.4).

The specific contributions of the work presented in this chapter are as follows.

1. We formalize our objective of minimizing the distances between related tasks as minimizing the WTMB cost of the resulting schedule, which extends two related

Simple solution:



Optimized solution:

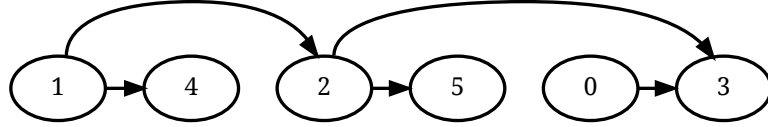


Figure 5.2.: A simple and an optimized ordering of the tasks from Figure 5.1.

classical problems: directed bandwidth and directed optimal linear arrangement [41, 62] (details in Section 5.4).

2. We transform the problem into a shortest path problem and give an algorithm for finding an optimal ordering that uses A* search to efficiently examine the space of all possible orderings.
3. Since the optimal A*-based algorithm is infeasible in practice for all except for the simplest workloads, we propose two heuristics that search a small subspace of possible orderings and return good orderings in much less time.
4. We experimentally show that the proposed heuristics obtain nearly-optimal answers on small problem instances, where it is feasible to compute an optimal solution. Further more, we show the effectiveness and efficiency advantages of the heuristic against a baseline algorithm using real DSW workloads and the TPC-DS decision support benchmark.

5.4. Problem Statement

The general problem we investigate in this paper is the scheduling of tasks with precedence constraints corresponding to data dependencies. Precedence constraints impose a partial order on the tasks. This partial order is given as input in the form of a DAG $G = (V, E)$, where each node $v \in V$ represent a task and each directed edge $e = (u, v) \in E$ is a precedence constraint, which requires that task u has to be scheduled before task v . Optionally, the input may include the size of the output of each

task, which will be discussed later in this section. In addition to satisfying the given precedence constraints, we will impose optimization goals on the generated ordering to minimize cache misses.

G may consist of a number of connected components, e.g., a view hierarchy that uses some set of base tables and another view hierarchy that is sourced from a different set of base tables. Since inter-dependencies and therefore cache optimization opportunities only exist in each connected component, we can deal with each connected component separately and thus we assume from now on that G is connected.

We assume the tasks are data-intensive. That is, the bottleneck is loading the data into the cache rather than the subsequent processing; otherwise, even having an unlimited cache would not help much. In addition, we assume a *cache-oblivious* setting, in which we do not know the size or granularity of the cache. Further more, we assume that the tasks belonging to a given connected component are to be scheduled serially on a single machine, although separate connected components may be scheduled in parallel. We defer a full treatment of multi-threaded scheduling in our context, as well as handling task priorities, to future work.

Let $\sigma : V \rightarrow \{0, 1, \dots, |V|\}$ be a schedule function that orders the tasks (i.e., the nodes in the precedence graph) in a given workload. The *precedence constrained scheduling* problem as formulated in [62] asks whether a schedule can meet a deadline. On a single-processor system, the problem of scheduling tasks with precedence constraints, without taking caching into account, is solvable in polynomial time [62]. However, real systems benefit from caching: the result of a preceding task u can be retrieved from the cache for task v , if the cache has enough capacity to keep the results of u despite other tasks that are scheduled between u and v . Therefore, minimizing the distance between u and v in the schedule σ , which is expressed by $|\sigma(u) - \sigma(v)|$, increases the likelihood of a cache hit.

There are two classical problems that express related objectives: (1) Directed Bandwidth (DBW), which aims to construct a schedule with a bound on the maximum distance of an edge in the precedence graph and (2) Directed optimal Linear Arrangement (DLA), which aims to construct a schedule with a bound on the sum of the distances for all edges:

Directed Bandwidth (DBW) (GT41 in [62], GT43 in [41]): Given a graph $G = (V, E)$ and a positive integer K , is there a schedule function $\sigma : V \rightarrow \{1, \dots, |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\max |\sigma(v) - \sigma(u)| \leq K ? \quad (5.1)$$

5. Cache-Oblivious Scheduling of Shared Workloads

Directed optimal Linear Arrangement (DLA) (GT42 in [62], cf. GT44 in [41]): Given a graph $G = (V, E)$ and a positive integer K , is there a schedule function $\sigma : V \rightarrow \{1, \dots, |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{(u,v) \in E} |\sigma(v) - \sigma(u)| \leq K? \quad (5.2)$$

Both of the above problems are NP-complete [63, 109]. Note that the problems are defined as decision problems, for which the corresponding optimization problems can be shown to be equally complex.

In our context, solving the DBW problem only optimizes for the single longest edge in the entire workload and does not take any other data dependencies into account. The DLA approach is not suitable in the context of caching as it was originally intended for scheduling of manufacturing workloads, in which a task produces multiple items, one for each subsequent tasks it is connected to. For example, recall Figure 5.2 and note the edges from task 2 to tasks 3 and 5. DLA counts both of these edges, effectively assuming that two copies of the output of task 2 need to be stored. However, *we are only interested in the longest edge from a task to any subsequent task that depends on it*, as that determines how long the data generated by the initial task need to stay in the cache.

Based on the above observations, we formulate a new problem, *TMB*, that reflects our objective, which is a combination of DBW and DLA:

Total Maximum Bandwidth (TMB): Given a graph $G = (V, E)$ and a positive integer K , is there a schedule function $\sigma : V \rightarrow \{1, \dots, |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{u \in V} \max_{\{v | (u,v) \in E\}} |\sigma(v) - \sigma(u)| \leq K? \quad (5.3)$$

If the input also includes the size of the output of each task u , call it ω_u , then we can extend TMB to the *Weighted Total Maximum Bandwidth (WTMB)* by optimizing for the weighted distance of the longest edge from any task to a dependent task. For WTMB, the optimization problem becomes:

$$\sum_{u \in V} \omega_u \max_{\{v | (u,v) \in E\}} |\sigma(v) - \sigma(u)| \leq K? \quad (5.4)$$

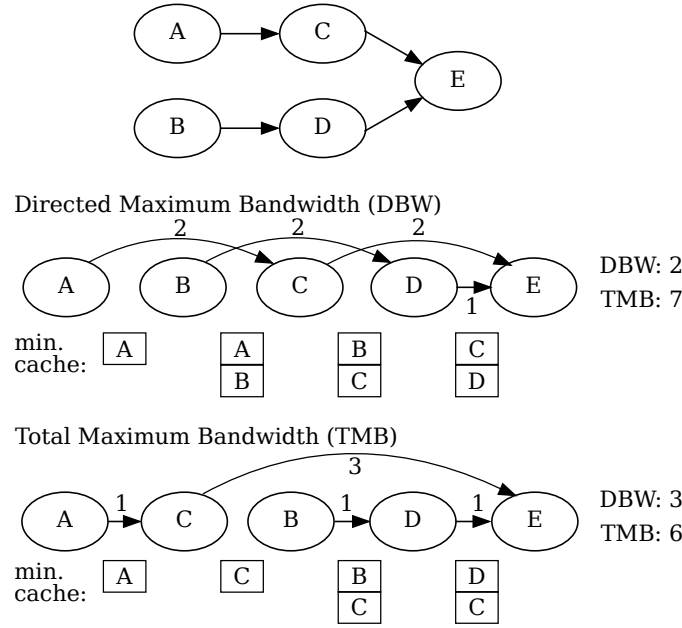


Figure 5.3.: An example comparing DBW with TMB.

Examples

We close this section with two examples, one comparing the TMB and DBW cost functions and the other comparing TMB with WTMB.

Figure 5.3 shows a precedence graph for five tasks (top), followed by two possible schedules (bottom). At the bottom of each schedule, we show the minimal cache contents at every step to avoid cache misses (labeled min. cache). The first schedule has an optimal DBW cost of only 2, the length of the longest edge. Its TMB cost is 7, which is the distance between A and C (of 2) plus the distance between B and D (of 2) plus the distance between C and E (of 2) plus the distance between D and E (of 1). To understand the cache contents illustrated below the schedule, note that when A is done, its output should be cached to be later used efficiently by C. Then, when B is done, its output is cached to be used by D. When C is done, its output is cached for use by E, but the output of A is no longer needed. When D is done, its output and C's output are cached for use by E and B's output is no longer needed.

The lower schedule in Figure 5.3 has an optimal TMB cost of 6, but its DBW cost is higher than that of the first schedule—here, the longest edge has a length of 3. This schedule requires less cache over time to avoid cache misses since only one item (namely C) has to be stored in the second step.

5. Cache-Oblivious Scheduling of Shared Workloads

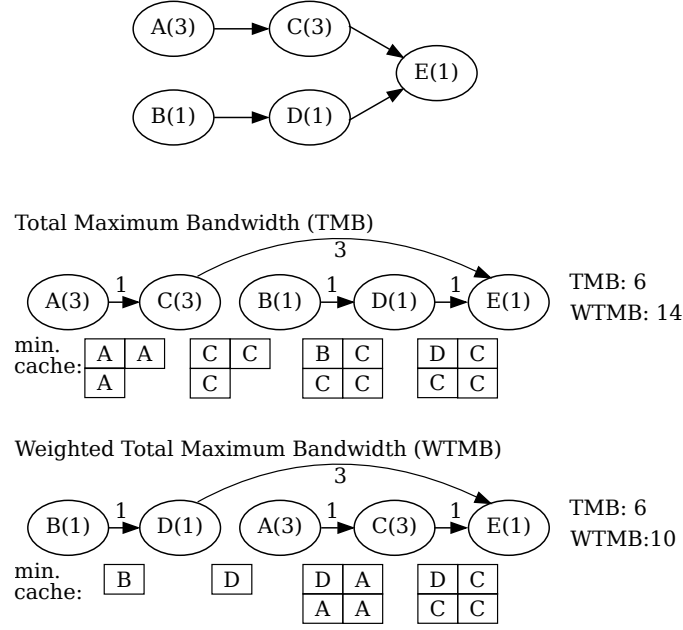


Figure 5.4.: An example comparing TMB with WTMB assuming we know the size of the output of each task.

Figure 5.4 compares two schedules in terms of TMB and WTMB costs for the same precedence graph as in Figure 5.3. But here we also know the sizes of the tasks. The notation A(3) indicates that the size of the output of A is 3. Both schedules have the same TMB cost of 6, but the second one has a lower WTMB cost of 10 (which is the distance between B and D of 1, plus the distance between D and E of 3, plus the distance between A and C multiplied by the size of A of 3, plus the distance between C and E multiplied by the size of E of 3). The minimal cache contents are shown below as before. Note that the output of A and C now has size 3 and therefore both take up 3 cache slots, as illustrated. The bottom schedule has a lower WTMB cost and requires less cache over time and is therefore more likely to avoid cache misses.

5.5. Scheduling Algorithms

In this section, we present algorithms that take a precedence graph as input and output a schedule optimized for TMB or WTBM if the task output sizes are known, utilizing Equations 5.3 or 5.4, respectively. In Section 5.5.1, we define the concepts and subroutines that will be used by the algorithms. We then present an optimal algorithm based on A*-search of the complete space of possible schedules in Section 5.5.2. Followed

by three approximate algorithms that examine a subset of possible schedules: a simple breadth-first baseline approach, presented in Section 5.5.3, and present two heuristics, a greedy algorithm that always chooses a task whose distance to its predecessor is the smallest in Section 5.5.4 and an algorithm that applies a dept-first heuristic to choose efficient schedules in Section 5.5.5.

5.5.1. Preliminaries

First, we define the Candidate Search Graph (CSG), $\overline{G} = (\overline{V}, \overline{E})$, in which the sequence of edge labels along every path from the start to the sink is a feasible schedule that obeys the precedence constraints encoded by the given precedence graph $G = (V, E)$ ¹. For example, the CSG corresponding to the precedence graph from Figure 5.1 is shown in Figure 5.5. Each node $\overline{v} \in \overline{V}$ denotes the schedulable tasks at that point in the schedule, i.e., those which can now be executed because all their precedence constraints have been met. Each edge $(\overline{u}, \overline{v}) \in \overline{E}$ is labeled with the name of the task that is to be executed at that step. The start node at the top of Figure 5.5 contains tasks 0 and 1, which must run before any other tasks. If we follow the right edge, labeled 1, the schedulable tasks are now 0, 2 and 4 and so on.

We can construct $\overline{G}$ from G in a straightforward way. In the first step, the source node in $\overline{G}$ contains all the root nodes in G , i.e., the tasks without any predecessors. In each subsequent step, we create edges for all tasks contained in the labels of nodes created in the previous step, labeled with the task number. For each of the created edges, we create a new node in $\overline{G}$ that contains the tasks that are now schedulable, only if such a node has not already been created. Finally, if no more nodes are schedulable, the edges are connected to the sink node, labeled with $\emptyset$.

Since we create $\overline{G}$ on-the-fly, the following definitions are based on a partial schedule. Let s be a possibly partial schedule of $|s|$ tasks. Let $get_cands(G, s) := \{v : (u, v) \in E \text{ and } u \in s \text{ and } v \notin s\}$. That is, get_cands returns the set of schedulable tasks that can now be appended to s assuming that all the tasks in s have already been executed. Furthermore, for a task u , let $successors(u)$ be the set of tasks that depend on u , i.e., $successors(u) = \{v : (u, v) \in E(G)\}$.

Finally, we define a $wmb_cost(s, G)$ function that evaluates the WTMB cost of a possibly partial schedule s according to Equation 5.4². Let σ be the ordering function

¹A similar search graph was used in [137] in the context of the direct optimal linear arrangement problem.

²Please note that if the TMB cost needs to be calculated all weights ω_u are just set to 1.

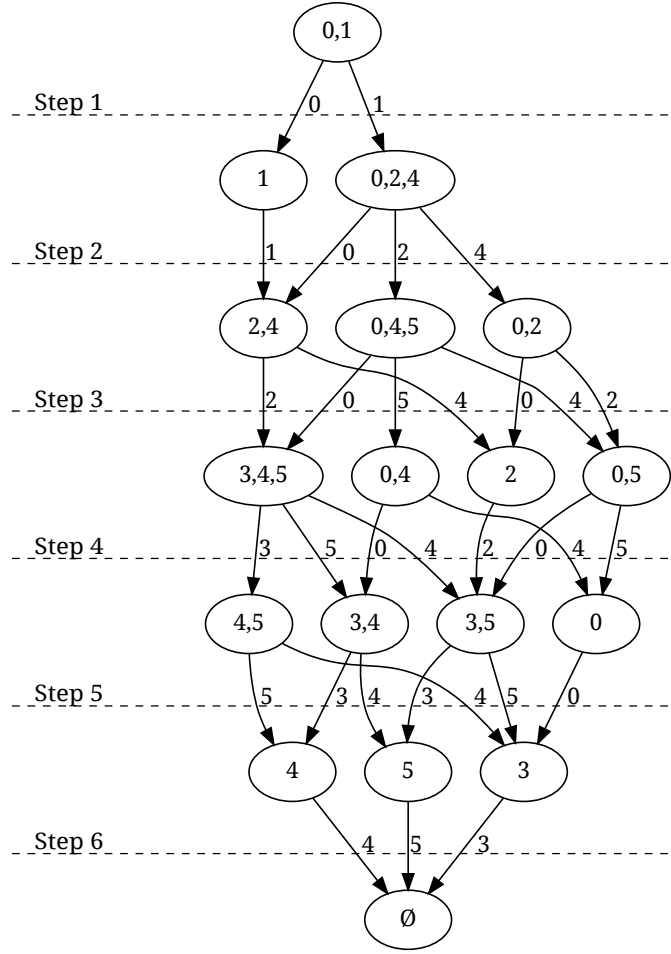


Figure 5.5.: Candidate search graph for the workload whose precedence graph was shown in Figure 5.1.

of s (recall Section 5.4). For each task u in s , we compute $w_{tmb_cost}(s, G)$ as follows.

1. If u has no successors, do nothing;
2. else if all of u 's successors are already in s , add to the total cost the distance between u 's last successor and u , i.e., $\max_{v \in \text{successors}(u)} |\sigma(v) - \sigma(u)|$, multiplied by ω_u .
3. else (if not all of u 's successors are in s), add to the total cost the quantity $|s| + 1 - \sigma(u)$, which is a lower bound on the distance between u 's last successor (which has not yet been scheduled) and u . (again, multiplied by ω_u if given and 1 otherwise).

Algorithm 5 shows the pseudo code for the WTMB cost function. Each task in the

Algorithm 5 (Weighted) `wmbb_cost`

```

1: cost = 0 // the overall cost of the schedule
2: for task in s do
3:   stepCost = 0
4:   maxOutPos = 0 // position of furthest successor
5:   outTasksDone = 0
6:   for outTask in successors(task) do
7:     if outTask in s then
8:       outTasksDone++
9:       maxOutPos = max( $\sigma$ [outTask], maxOutPos)
10:    end if
11:  end for
12:   $\ell$  = len(successors(task))
13:  if  $\ell$  == 0 then
14:    do nothing // no successors
15:  else if outTasksDone ==  $\ell$  then
16:    stepCost = maxOutPos -  $\sigma$ (task)
17:  else
18:    stepCost = len(s) -  $\sigma$ (task)
19:  end if
20:  cost +=  $\omega$ (task) * stepCost
21: end for
22: return cost

```

possibly partial schedule s is considered sequentially. Lines 6 through 11 count how many of the current task's successors are in s and record the position in s of the furthest successor of the current task. Note the use of the position function σ to find the position of `outTask` in s . Line 12 counts the total number of successors of a given task. If this number is 0, the current task does not incur any WTMB cost. Otherwise, if all the successors have already been scheduled, we can precisely compute the WTMB cost in line 16, which is simply the difference in the position of the furthest successor and the task itself. Otherwise, line 18 computes a lower bound on the given task's WTMB cost. In line 20, we add the cost of the current task to the total cost of the schedule. Note the ω function that determines the output sizes for WTMB. In case of TMB, ω_u is simply 1 for every task.

For example, consider the partial schedule $s = \langle 0, 1, 2, 4 \rangle$ for the precedence graph from Figure 5.1. For task zero, the cost is four since its successor, task 3, appears four positions later in the sequence. For task 1, the cost is two since its successor, task 4, appears two positions later. The cost for task 2 is two, which is a lower bound for its true cost, since its successor, task 3, has not been scheduled yet. Finally, the cost for

task 4 is zero since it has no successors. Thus, $wmb_cost(s, G) = 4 + 2 + 2 + 0 = 8$.

5.5.2. Optimal Algorithm Based on A* Search

We begin with an optimal algorithm based on A* search that considers every possible schedule and selects an optimal one with the lowest wmb_cost . As we will experimentally show in Section 5.6, this algorithm is not feasible in practice for non-trivial problem instances because the number of possible schedules considered by the A* algorithm can be prohibitively large in real-world scenarios.

A* search finds a least-cost path between two nodes, in our case the start node and the sink node of the CSG. Please note that a least-cost path in the CSG corresponds to a least-cost schedule. For each node x , the cost function used by A* includes two parts: $g(x)$, which is the cost of the path from the start node to x and $h(x)$ which is a heuristic function that approximates, but must not overestimate, the cost of the path from x to the sink node. In our problem, $g(x)$ is simply the $wmb_cost(s, G)$ function, where s is the schedule corresponding to the path from the start node to x and G is the precedence graph. The more interesting part is the definition of $h(x)$.

To solve our problem, we define $h(x)$ as the sum of the outgoing edges present in the precedence graph for each task that has not yet been scheduled along the path from the start node to x . To understand why this is an admissible function for A* search (i.e., one that does not overestimate the remaining cost of the path), note that if a task node has an outgoing edge in the precedence graph, then there is a successor task that must be scheduled after that node. Thus, a lower bound on the total maximum bandwidth cost for the given task is the number of its outgoing edges in the precedence graph, i.e., the number of its successors. This lower bound occurs if all the successors are scheduled immediately after the given task. If any other task is scheduled before the last successor, the cost can only increase.

For example, consider the partial schedule $s = \langle 0, 1, 4 \rangle$ based on Figure 5.1. The $g(x)$ function of the node in the CSG corresponding to this partial schedule is simply $wmb_cost(s, G)$, which is 5 (three for task 0 plus two for task 1, since not all of their successors have been scheduled and zero for task 4 because it does not have any successors). To compute $h(x)$, note that tasks 2, 3 and 5 are yet to be scheduled. The sum of the outgoing edges of these three nodes in the given precedence graph is two, which gives us $h(x)$. Thus, the total cost of s as computed by A* search is $g(x) + h(x) = 7$. It is easy to verify that no complete schedule with $s = \langle 0, 1, 4 \rangle$ as its prefix can have a wmb_cost of less than 7.

5.5.3. Baseline Algorithm

We now present the first of three algorithms that consider a subset of the possible schedules and therefore are faster than the A*-based algorithm, but are not guaranteed to find a good solution. The first such algorithm is the simplest and fastest approach we refer to as *Baseline*: at every step, it schedules the currently-schedulable tasks in random order, executes *get_cands*(G, s) and schedules the next layer of the precedence graph. Thus, using the precedence graph from Figure 5.1 as input, in the first step, Baseline executes tasks 0 and 1 in random order, then tasks 2 and 4 in random order and then tasks 3 and 5, again in random order. The running time of Baseline corresponds to that of breadth-first-search, which is $\mathcal{O}(|V| + |E|)$.

5.5.4. Greedy Algorithm

The next algorithm is the standard greedy heuristic applied to our problem: at every step, it chooses a schedulable task that yields the lowest *wmb_cost*(s, G) when added to the current partial schedule s . Ties are broken randomly.

Using the precedence graph from Figure 5.1 as input, the greedy heuristic first decides between tasks 0 and 1. For both $s = \langle 0 \rangle$ and $s = \langle 1 \rangle$, *wmb_cost*(s, G) = 1 since not all of 0's or 1's successors, respectively, have been scheduled. Suppose the tie-break results in task 1 being scheduled first. In the next step, the schedulable tasks are still 0, 2 and 4. For $s = \langle 1, 0 \rangle$, *wmb_cost*(s, G) is two. For $s = \langle 1, 2 \rangle$, *wmb_cost*(s, G) is two due to task 1 plus one due to task 2, which gives three. For $s = \langle 1, 4 \rangle$, *wmb_cost*(s, G) is two due to task 1, plus zero since task 4 has no successors. Thus, the greedy algorithm randomly chooses between task 0 and 4 to after task 1. We omit the remaining steps for brevity.

We now analyze the runtime complexity of the greedy algorithm. It uses the *get_cands* function to retrieve the set of currently schedulable tasks. However, since each step of the algorithm adds one task to the schedule, only the successors of this new task need to be added to the schedulable set. This gives $\mathcal{O}(|V| + |E|)$ for all *get_cands* calls over all the iterations.

The runtime is dominated by calling *wmb_cost* for the considered schedules, which requires looping over all the outgoing edges of the tasks already in the schedule. This gives $\mathcal{O}(|E|)$ per call. Since the algorithm iterates $|V|$ times and, clearly, at every iteration there are no more than $|V|$ schedulable tasks, for which *wmb_cost* is evaluated, the overall complexity of the greedy algorithm is $\mathcal{O}(|V|^2|E|)$.

5.5.5. Heuristic Algorithm

Our final algorithm is called *Heuristic*. In contrast to the greedy algorithm, which only examines the *wtmb_cost* of adding every schedulable task to the current schedule in each iteration, the heuristic algorithm computes a complete feasible schedule for each schedulable task in every iteration and chooses the task with the lowest cost complete schedule. However, to keep the running time manageable, the heuristic algorithm cannot explore every possible feasible schedule, as done by the A* algorithm. Instead, the complete schedules for each schedulable task are heuristically computed via deepest-first traversal, as explained below.

First, the heuristic algorithm pre-processes the precedence graph G by adding depth information to each node, corresponding to the distance to the furthest ancestor. For instance, in Figure 5.1, the depth of tasks 0 and 1 is zero, the depth of tasks 2 and 4 is one, the depth of task 5 is two and the depth of task 3 is also two, its distance to task 0 is one, but the distance to its other ancestor, task 1, is two.

Next, we illustrate what happens in the first iteration using the graph from Figure 5.1 as input. Initially, the only schedulable tasks are 0 and 1. We need to build complete schedules starting with 0 and 1, respectively, compute their *wtmb_cost* and choose the task whose complete schedule has the lowest *wtmb_cost*. Again, ties are broken randomly.

The complete depth-first schedule that starts with task 1 is computed as follows. After task 1 has been scheduled, the schedulable tasks are 0, 2 and 4, of which both 2 and 4 have the largest depth. Let us assume task 2 is chosen next. The schedulable tasks then become 0, 4 and 5, of which 5 is chosen because its depth is the largest. With the partial schedule now $\langle 1, 2, 5 \rangle$, the schedulable tasks are 0 and 4. We choose 4 and finally 0 and then 3. This gives the complete schedule $s = \langle 1, 2, 5, 4, 0, 3 \rangle$. Its *wtmb_cost* is three for task 1, four for task 2 and one for task 0, which gives eight.

Similarly, the complete depth-first schedule that starts with task 0 is computed as follows. After task 0 has been scheduled, the only schedulable task is 1, so we choose it. Next, we have a choice between tasks 2 and 4, both of which have the same depth, so let us say we choose task 2. Then, the schedulable tasks are 3, 4 and 5, of which 3 and 5 have the highest depth, so let us say we choose task 3. This leaves tasks 4 and 5 and we choose 5 first because its depth is higher. This gives a complete schedule of $\langle 0, 1, 2, 3, 5, 4 \rangle$, whose *wtmb_cost* is three for task 0, four for task 1 and two for task 2, which is nine in total.

Thus, at the end of the first iteration, the Heuristic algorithm chooses task 1 and the second iteration begins. Note that the complete schedules calculated in the first iteration are now discarded and new complete schedules will be built in the second iteration, all

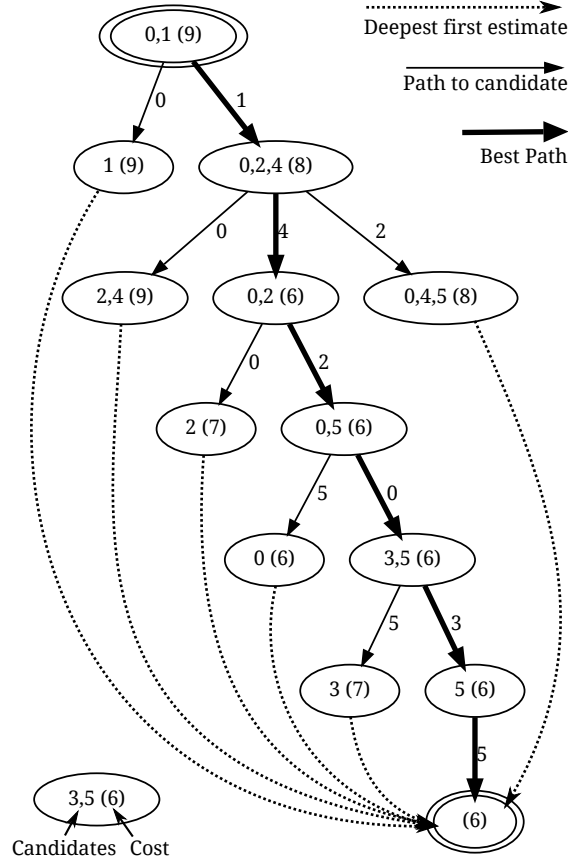


Figure 5.6.: Visualization of a run of the Heuristic algorithm on the candidate search graph created from the precedence DAG in Figure 5.1.

of which will have task 1 scheduled first.

Figure 5.6 summarizes the way in which the heuristic algorithm traverses the candidate search graph using the graph from Figure 5.1 as input. As we describe above, in the first iteration, two complete schedules are built, one starting with task 0 and one starting with task 1. The latter is chosen by the heuristic algorithm, indicated by the bold arrow. The *wtm_b*_cost is also shown in the figure; note the cost of nine if we choose task 0, versus the cost of eight if we choose task 1. In the second iteration, the heuristic algorithm considers tasks 0, 2 and 4 and computes the corresponding three depth-first schedules. Choosing task 4 next is the best option. After task 4 has been selected, the algorithm computes two new depth-first complete schedules corresponding to adding tasks 0 and 2, respectively, to the existing partial schedule of $\langle 1, 4 \rangle$. Adding task 2 is cheaper, as shown in Figure 5.6. The complete schedule generated by the heuristic algorithm is indicated by the bold arrows: $\langle 1, 4, 2, 0, 3, 5 \rangle$ and has a *wtm_b*_cost of six.

5. Cache-Oblivious Scheduling of Shared Workloads

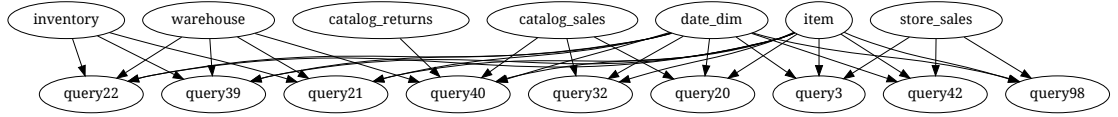


Figure 5.7.: Precedence graph for the 7 selected TPC-DS queries and tables in *tpc-ds-7q*.

The intuition behind computing complete schedules in a depth-first manner is to schedule successors right after their ancestors; notice that when a task with a higher depth than the previous task is chosen, these two tasks should be very close together in the topological sort of the precedence graph. However, any other heuristic for building possible complete schedules for a given schedule prefix is compatible with the framework we have described in this section.

Finally, we discuss the time complexity of the heuristic algorithm. Pre-processing the precedence graph to compute depth information, i.e. longest paths to a root in G , can be done via a linear-time shortest-path algorithm on G with negative edge weights. This is only possible because G is a DAG, where after edge weight negation no negative cycles are possible. Now, one iteration of the algorithm involves computing multiple complete schedules in a deepest-first manner. For each such complete schedule, exactly one task is moved from the schedulable set to the actual schedule and the successors of this task are added to the schedulable set. Therefore, each node and edge in G need to be visited only once. If the set of schedulable tasks is maintained in a data structure such as a binary heap that allows retrieval and deletion of the minimum-depth node and insertion in $\mathcal{O}(\log |V|)$, computing one complete schedule requires $\mathcal{O}(|E| + |V| \log |V|)$.

The overall heuristic algorithm iterates $|V|$ times. In each iteration, there are at most $|V|$ schedulable tasks, each of which requires a complete schedule to be built, at a cost of $\mathcal{O}(|E| + |V| \log |V|)$ and its *wtm_cost* must be computed, but the former dominates the runtime. This gives the overall runtime complexity of the heuristic algorithm as $\mathcal{O}(|V| \cdot |V| \cdot (|E| + |V| \log |V|)) = \mathcal{O}(|E| \cdot |V|^2 + |V|^3 \log |V|)$.

5.6. Experimental Evaluation

This section presents our experimental findings on the effectiveness and efficiency for both TMB and WTMB of the algorithms we presented in Section 5.5 (A*, Baseline, Greedy and Heuristic). We start with a description of our data sets and experimental environment (Section 5.6.1), followed by the results.

In Section 5.6.2, we experiment with different precedence graphs as inputs and we

report the TMB/WTMB scores obtained by each algorithm as well as the time it took to generate the schedules. In general, we find that Heuristic obtains nearly-optimal schedules, but is slower than Greedy and Baseline (but still much faster than A*). Furthermore, both Greedy and Heuristic generate significantly better schedules than Baseline. In Section 5.6.3, we run the proposed algorithms (except A*) on very large random precedence graphs to see if they can efficiently compute schedules for complex workloads. We found that Heuristic does not scale as well as Baseline and Greedy, but can still handle large precedence graphs. In Section 5.6.4, we execute various workloads in PostgreSQL and show the real-world performance improvements due to our scheduling algorithms. Again, Heuristic and Greedy outperform Baseline.

5.6.1. Experimental Setup

We used a dual CPU Xeon E5-2630 server machine, with 64 GB of RAM and a 10-disk RAID10 storage subsystem. As a database we use PostgreSQL in version 9.2.4. We implemented all the algorithms presented in Section 5.5 in the Go language³.

The `pgfincore`⁴ library is used to advise the operating system to drop tables from the disk cache. We use this functionality to evict tables from the cache when they are no longer needed. We will explicitly state whenever we make use of this function. The purpose of using this function is to reduce the amount of used cache to the minimum number of items which are still needed by pending tasks.

We use the following groups of data sets. The number of nodes and edges in the corresponding precedence graphs are shown in Table 5.1 under the columns labeled $|V|$ and $|E|$. Below, we also provide the depth of each precedence graph, measured as the number of nodes on the longest path. Further details on the precedence graphs, as well as the source code of our algorithms and the random graph generator, may be found at <https://github.com/arbaer/schedule>.

- *running* corresponds to the running example from Figure 5.1 with a depth of 3.
- *test1*, *test2*, *test3* and *test5* correspond to small hand-crafted workloads with various features. Their depths are three, four, five and four, respectively; however, *test5* additionally contains a node with a very high fan-out.
- *realworld1* and *realworld2* are two network monitoring workloads from data warehouses we were operating at the time of writing using the DBStream DSW presented in Section 3.4. The tasks are base tables and materialized view updates.

³<http://golang.org>

⁴<http://pgfoundry.org/projects/pgfincore/>

5. Cache-Oblivious Scheduling of Shared Workloads

realworld1 has a depth of 5 and contains two nodes with a high fan-out, whereas *realworld2*'s depth is 6, but the overall fan-out is lower.

- *tpc-ds-scan*, *tpc-ds-7q*, *tpc-ds-11q* and *tpc-ds-63q* are based on the TPC-DS decision support benchmark [136]. TPC-DS contains 24 base tables (7 fact tables and 17 dimension tables) and 99 predefined queries over the base tables. The number of tables required by a query ranges from one to 13, with an average of 4. We generated two versions of the benchmark: one with a scale factor of 10 and one with 100. All TPC-based precedence graphs have a depth of 2 since they consist of a layer of queries accessing a set of tables.

The *tpc-ds-scan* workload consists of scan queries over the largest base tables: *catalog_sales*, *web_sales* and *store_sales*. These three tables account for 8.4 GB at a scale factor of 10. The other three workloads contain 7, 11 and 63 queries from TPC-DS; the corresponding precedence graphs get progressively more complex. We do not use all 99 queries as not all of them are data-intensive and benefit from caching. As an example, we show the smallest *tpc-ds-7q* precedence graph in Figure 5.7.

- In one experiment, we also use large randomly-generated precedence graphs to test the scalability of our algorithms. These will be described later.

We use the small *running* and *test* workloads to test how close the solutions obtained by the heuristics are to an optimal solution; these are the only workloads on which it was feasible to run the A* algorithm. The *realworld* and TPC-DS workloads show that Greedy and Heuristic are scalable and outperform Baseline. We have the task output sizes ω_u for the *realworld* and *tpc-ds* workloads, which we use to test the WTMB version of our problem.

5.6.2. Scheduling Algorithm Comparison

In the first set of experiments, we generate schedules using all algorithms presented in Section 5.5 and report how long it takes to create the schedules and the TMB cost of the schedules, even though we know the output sizes for some workloads, we ignore them for now and will consider WTMB shortly. In this experiment, we do not actually run the workloads in a database system. Table 5.1 shows the results. Each row corresponds to a different workload (we omit *tpc-ds-scan* as it is only relevant to the PostgreSQL experiments later in this section). For Baseline, Greedy and Heuristic, we report the mean TMB score and the Standard Deviation (SD) over 100 runs, since these algorithms

Table 5.1.: Performance comparison of the implemented algorithms. Times marked with * indicate that the experiment was stopped after 1 hour of wall clock time.

Graph	$ V $	$ E $	Baseline			Greedy		
			TMB		time	TMB		time
			Mean	SD		Mean	SD	
running	6	5	9	0.00	0.010s	8	0.00	0.012s
test1	5	4	6	0.00	0.010s	5	0.00	0.013s
test2	11	10	16.16	0.64	0.010s	16.16	1.32	0.011s
test3	22	21	65.54	3.69	0.010s	47.52	6.62	0.012s
test5	25	30	91.58	12.96	0.010s	66.68	5.91	0.014s
realworld1	43	48	269.1	11.8	0.048s	107.2	16.9	0.022s
realworld2	57	69	297.6	24.6	0.028s	120.4	10.3	0.023s
tpc-ds-7q	14	25	62.0	2.9	0.017s	52.8	3.7	0.019s
tpc-ds-11q	31	70	362.4	13.3	0.028s	311.7	15.3	0.020s
tpc-ds-63q	85	310	1488.2	60.2	0.099s	1162.5	78.7	0.046s

Graph	$ V $	$ E $	Heuristic				A*		
			TMB		time	nodes	TMB	time	nodes
			Mean	SD			opt		
running	6	5	6	0.00	0.016s	18	6	0.019s	33
test1	5	4	5	0.00	0.012s	13	5	0.012s	21
test2	11	10	14	0.00	0.017s	34	14	0.031s	336
test3	22	21	37.44	0.50	0.038s	90	35	1.451s	12.4K
test5	25	30	48.40	0.49	0.053s	123	46	37.771s	325K
realworld1	43	48	80.0	1.8	0.075s	163	–	1h*	25.0M
realworld2	57	69	119.2	11.8	0.184s	616	–	1h*	29.5M
tpc-ds-7q	14	25	43.9	2.8	0.018s	55	40	0.27s	56.6K
tpc-ds-11q	31	70	220.5	11.8	0.054s	313	–	1h*	43.0M
tpc-ds-63q	85	310	853.2	12.2	1.505s	1598	–	1h*	16.2M

break ties randomly and therefore may return different schedules for the same input. Bold TMB numbers indicate the best results. For Heuristic and A*, we also count the number of node visits during their execution, the number of node visits for Baseline and Greedy is small and therefore not reported. Note that A* finished running within one hour only on small problem instances and the number of nodes it visits is generally very large.

To summarize the results so far: for the workloads where A* was able to finish, we see that Heuristic gives nearly-optimal schedules. Greedy also works well for some of the smaller problem instances. Baseline gives the most expensive schedules. On the other hand, Baseline and Greedy are extremely fast, Heuristic is still very fast and A* is the

5. Cache-Oblivious Scheduling of Shared Workloads

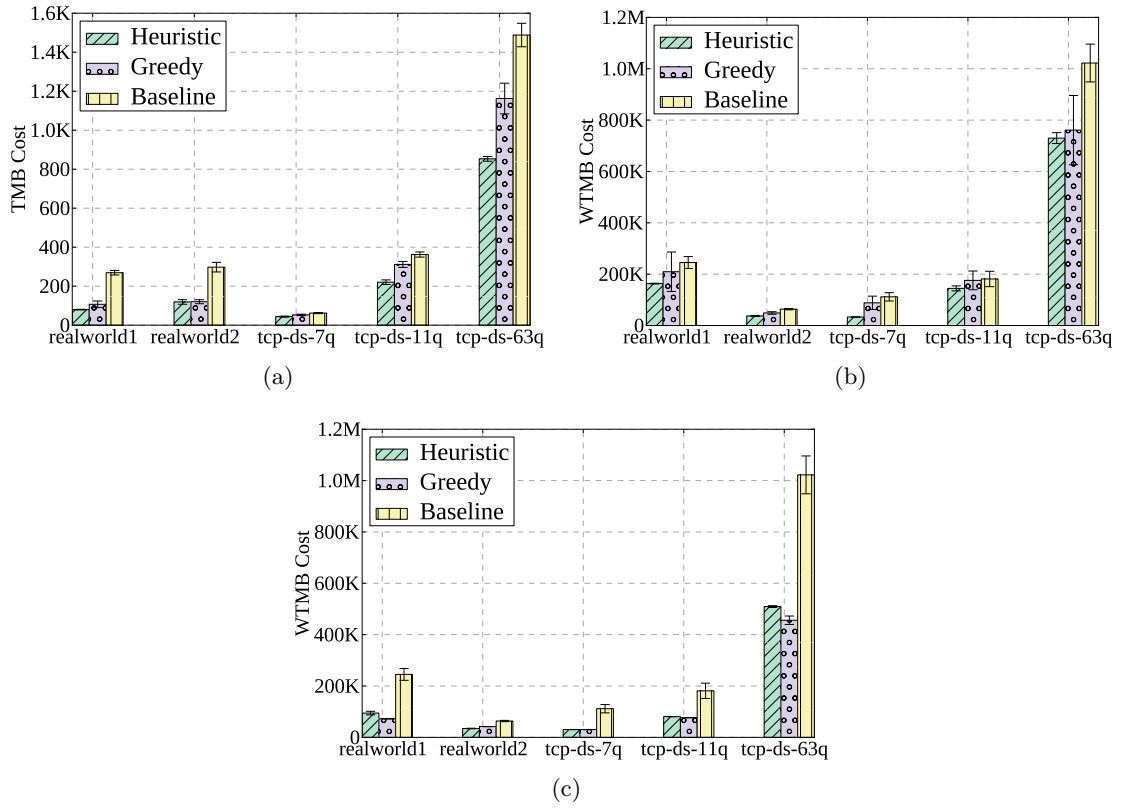


Figure 5.8.: Comparison of a) TMB costs, b) WTMB costs assuming the algorithms are optimizing for TMB and c) WTMB costs assuming the algorithms are optimizing for WTMB.

slowest.

Figure 5.8 compares the TMB and WTMB costs of the schedules returned by Heuristic, Greedy and Baseline for the workloads that come with output sizes, namely real-world and TPC-DS, these workloads are too large for A* to handle in reasonable time. The average costs and error bars are included, based on 100 runs of each algorithm.

Figure 5.8a starts with the TMB costs that were already reported in Table 5.1, indicating that both Greedy and Heuristic are significant improvements over Baseline and that Heuristic is the overall winner, although it takes longer to compute the schedules.

In Figure 5.8b, we show the WTMB costs of the schedules from Figure 5.8a, i.e., we have the algorithms optimize for TMB as before, but we report the WTMB score of the resulting schedules by incorporating output sizes. Heuristic continues to give the best and most stable results — note the wide error bars for Baseline and Greedy. In particular, different runs of Greedy may give widely different TMB results. For instance,

Table 5.2.: Scalability results.

Graph	$ V $	$ E $	Baseline	Greedy	Heuristic
rand1	100	200	0.02s	0.04s	0.84s
rand2	200	400	0.03s	0.30s	10.41s
rand3	300	600	0.04s	0.95s	40.01s
rand4	400	800	0.10s	2.45s	2m57s
rand5	500	1K	0.09s	4.59s	5m43s
rand6	1K	2K	0.34s	34.20s	85m46s
rand7	2K	4K	0.71s	4m22s	–
rand8	4K	8K	2.75s	31m47s	–

if there are several base tables with various sizes but same TMB costs, Greedy randomly chooses a table, regardless of the size of the table. Those results indicate that even if the table sizes are unknown, the our Heuristic approach is able to identify schedules of better performance.

Note that the y-axis scales of Figure 5.8a and Figure 5.8b are different. TMB effectively assumes that each output size is one, whereas the WTMB scores are much higher because they reflect the true sizes of the inputs.

Figure 5.8c shows the WTMB scores assuming the algorithms know the output sizes and are actually optimizing for WTMB, not TMB. Comparing to Figure 5.8b, which has the same y-axis scale, knowing the output sizes clearly helps to lower the WTMB score of the resulting schedules for Greedy and Heuristic. Interestingly, Greedy slightly outperforms Heuristic in this experiment, meaning that greedily selecting tasks with the lowest WTMB scores is a good strategy and there are no more ties that Greedy has to break randomly, unless the output sizes are exactly the same. We hypothesize that Heuristic could be tuned for the WTMB problem, e.g., by incorporating output sizes in the deepest-first schedule generation, but even now it is not much worse than Greedy.

5.6.3. Scalability Comparison

In this experiment, we randomly generate very large precedence graphs and measure the running time of Baseline, Greedy and Heuristic. Table 5.2 reports the number of nodes and edges for each random graph and the running times of our algorithms. Baseline and Greedy are very simple algorithms and scale extremely well. Heuristic does not scale as well, but can still handle graphs with up to 1000 nodes and edges in reasonable time.

5.6.4. PostgreSQL Experiments

In this set of experiments, we execute various workloads under various schedules in the PostgreSQL database to measure the real-world performance improvements of our techniques. Here, we focus on the disk-RAM hierarchy.

Experiment 1

We start by running the simple *tpc-ds-scan* workload of three queries that scan three base TPC-DS tables:

```
Q1: select count(*) from catalog_sales;
Q2: select count(*) from web_sales;
Q3: select count(*) from store_sales;
```

In PostgreSQL, these queries result in full table scans, resulting in an I/O intensive workload. In one schedule, we also use the following three functions calls to drop tables from the cache:

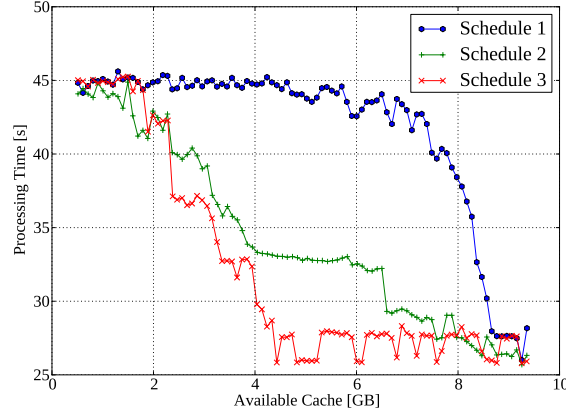
```
X1: select drop_table_cache(catalog_sales);
X2: select drop_table_cache(catalog_sales);
X3: select drop_table_cache(catalog_sales);
```

We create three schedules, *S1*, *S2* and *S3*, executing each query three times to demonstrate the differences in running time. In schedule *S3*, we also actively evict tables from the cache when they are not needed any more, indicated by the operations X1, X2 and X3.

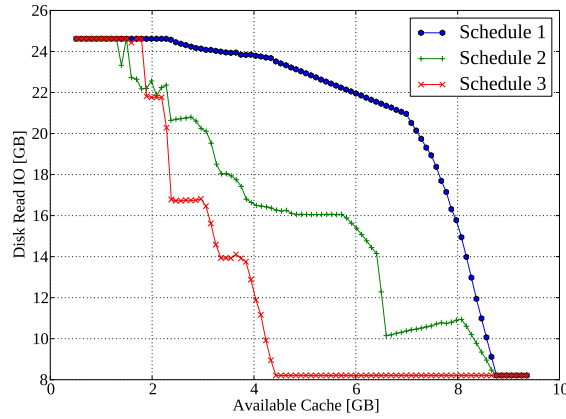
```
Schedule 1: Q1,Q2,Q3,      Q1,Q2,Q3,      Q1,Q2,Q3
Schedule 2: Q1,Q1,Q1,      Q2,Q2,Q2,      Q3,Q3,Q3
Schedule 3: Q1,Q1,Q1,X1,    Q2,Q2,Q2,X2,    Q3,Q3,Q3,X3
```

In Figure 5.9a and 5.9b, we show the processing time and read I/O, respectively, of the three schedules under varying amounts of available cache (RAM), ranging from 500MB to 10GB in steps of 100MB. We control this by running a program that allocates and fills a specific amount of memory, making that amount unavailable for caching. In each experimental iteration, we first execute one schedule, force the operating system to drop all caches and then execute the next schedule.

Figure 5.9a reports the processing times. *Schedule 1* is clearly the least efficient, except either no data fit into the cache, under 2GB, or all data fit into the cache, over 8.4GB.



(a) Processing time



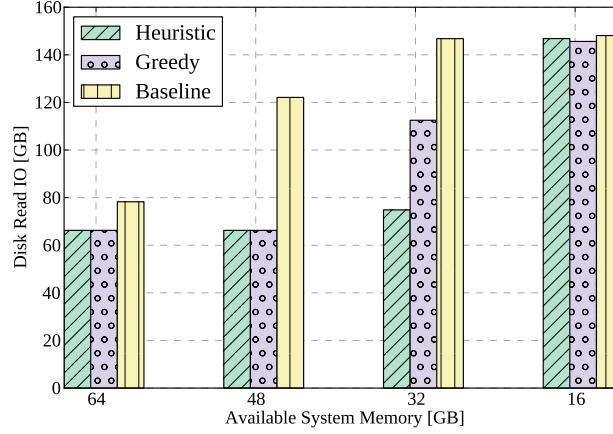
(b) Disk I/O

Figure 5.9.: Performance analysis executing the same workload with different schedules and increasing cache size.

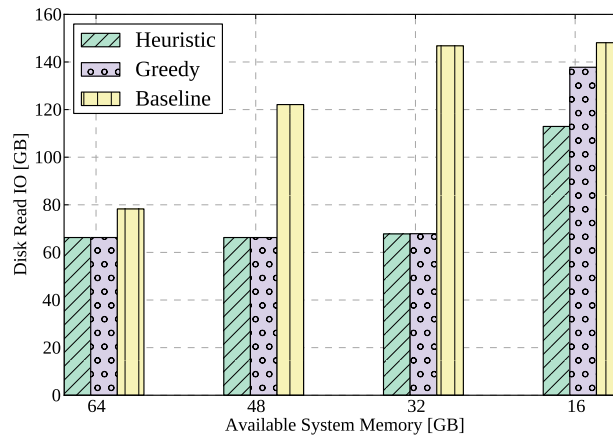
Schedule 2 performs much better and *Schedule 3* achieves even better performance since tables that are not required any more are explicitly removed from the cache, simulating an optimal cache eviction strategy. The largest difference occurs at 4.5 GB of free RAM since now the biggest of the three tables fits entirely into the disk cache. At this point, *Schedule 1* finishes in 45 seconds while *Schedule 2* and *Schedule 3* in 33 seconds and 26 seconds, respectively. The resulting performance increase of *Schedule 3* over *Schedule 1* is 73 percent.

Figure 5.9b illustrates the amount of disk read I/O during the run of this experiment under the same available cache conditions as before. These results indicate that there is a correlation between the amount of disk read I/O during the execution of a schedule and its processing time. *Schedule 1* needs to fit nearly all the tables into the cache before

5. Cache-Oblivious Scheduling of Shared Workloads



(a) Algorithms **not** considering table sizes.



(b) Algorithms considering sizes.

Figure 5.10.: Disk read I/O for the `tpc-ds-7q` workload scheduled by the Heuristic and the Baseline algorithms.

larger amounts of data can be reused. In *Schedule 2*, much more data can be reused through the cache, but even for larger amounts of available cache, between 4.5 and 8.4 GB, some data has to be fetched multiple times from disk. Finally, in *Schedule 3* since an optimal cache eviction strategy is applied, as soon as the biggest table fits entirely into the cache, data are only fetched once from disk.

We conclude from this experiment that changing the execution order of a workload can reduce the amount of disk I/O if not all the data fit into the cache, which also influences the execution time of a workload if it is I/O bound.

Experiment 2

Next, we show that the reduced amounts of disk read I/O are reproducible with queries from the TPC-DS benchmark. We use the *tpc-ds-7q* workload, consisting of 7 data-intensive queries and execute them on TPC-DS tables generated using a scale factor of 100. This scale factor results approximately in 100GB of data. During the execution of a schedule, whenever a table is no longer needed in that schedule, it is evicted from the cache using the `drop_table_cache()` function, simulating an optimal cache eviction strategy.

We create schedules using Baseline, Greedy and Heuristic, each *not* considering the sizes of the outputs (tables), i.e., optimizing for TMB, not WTMB. We run the experiment four times and reduce the total amount of available system memory from 64 GB to 16 GB in steps of 16 GB, simulating machines with different amounts of available memory. We do this in the same manner as in the previous experiment.

Figure 5.10a shows the results, with disk read I/O on the y-axis. For 64 and 48 GB, only the Baseline schedule shows increased amounts of disk I/O. At 32 GB of available RAM, Heuristic performs better than Baseline. Although Greedy is also better than Baseline, it does not perform as well as Heuristic. Finally, for the 16 GB case all schedules perform nearly the same.

Figure 5.10b shows the results of a similar experiment, but one in which we also give the algorithms the table sizes, allowing the algorithms to optimize for WTMB.

The main difference compared to Figure 5.10a are the reduced I/O reads for the 32 GB run for Greedy and Heuristic. Even in the 16 GB case, these algorithms perform better than Baseline.

If we extrapolate the memory needs linearly, the Greedy and Heuristic schedules would reduce the memory needs from 640 GB of the Baseline schedule to only 320 GB for an execution without additional read I/O using the next larger TPC-DS scale factor of 1000.

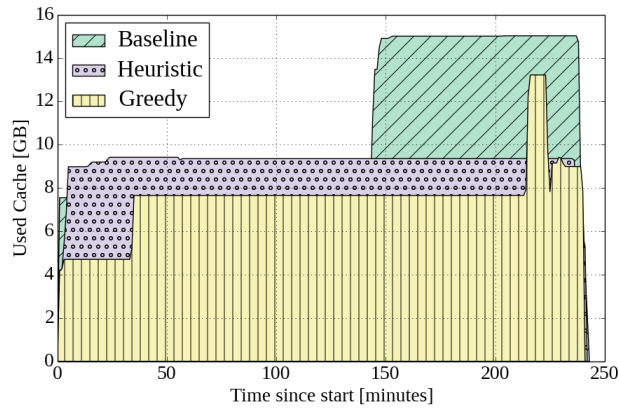
Experiment 3

In the last experiment, we study the effect of different schedules on cache usage over time. We use the *tpc-ds-63q* workload, which contains the 63 most data-intensive queries. We use a TPC-DS scale factor of 10. Only the raw data are imported into the database, without creating any further auxiliary data structures such as indices. We do not optimize the queries for processing time since our focus is on I/O optimization via scheduling by minimizing the amount of needed cache. Before each experimental run, we first clear

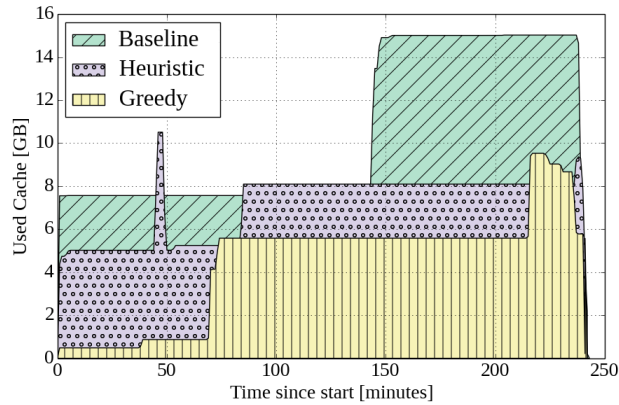
5. Cache-Oblivious Scheduling of Shared Workloads

the cache and then execute the *tpc-ds-63q* workload in the order specified by the scheduling algorithm. Every second, we sample the disk cache usage. As soon as a table is no longer needed by any other query in the schedule, we remove it from the cache using the `drop_table_cache()` function. Note that we never actively load any data into the cache, but instead tables are loaded automatically when they are accessed by a query.

Figure 5.11a shows the cache usage (on the y-axis) as a function of time since the start of the experiment (x-axis) for Baseline, Greedy and Heuristic, optimizing for TMB, i.e., the algorithms do not know the table sizes. Figure 5.11b shows the results for WTMB. In both cases, the cache usage over time is greatly reduced by Greedy and Heuristic as compared to Baseline.



(a) Scheduled **without** considering table sizes.



(b) Scheduled considering table sizes.

Figure 5.11.: Cache usage over time for *tpc-ds-63q*.

Finally, we compare the algorithms by their areas under the cache usage curves. If $c_s(t)$ denotes the cache usage of schedule s at any given point in time t , we can formulate

the average cache usage for the total execution time ΔT as $\overline{cu}_s$ as

$$\overline{cu}_s = \frac{\int_{t_0}^{t_0+\Delta T} c_s(t) dt}{\Delta T}. \quad (5.5)$$

Table 5.3.: Average and Maximum cache usage for different schedules of the tpc-ds-63q workload.

Algorithm	Avg [GB]	Decr.	Max [GB]	Decr.
Baseline	10.439	100%	15.025	100%
Greedy TMB	7.514	71%	13.217	88%
Greedy WTMB	4.430	42%	9.529	63%
Heuristic TMB	9.190	88%	9.586	63%
Heuristic WTMB	7.113	68%	10.511	70%

The resulting cache usages are shown in Table 5.3. The schedule of the Greedy algorithm needs on average only 42% of RAM of the Baseline schedule. This leaves either more RAM for other concurrently running applications or results in less read I/O if RAM is smaller than the workload.

5.7. Summary

In this section, we formulated and solved a novel problem in the context of shared data-intensive workload scheduling. Given a set of tasks with data dependencies, such as those corresponding to updating a DAG of materialized views, we computed an ordering of the tasks that 1) obeys the precedence constraints encoded by the data dependencies and 2) minimizes cache misses in a cache-oblivious way. Our solution relies on scheduling all tasks that need a particular data item as soon as possible after this item is available in the cache. We gave an optimal algorithm based on A* search and efficient heuristics that find nearly-optimal orderings in a fraction of the time needed to run the A* algorithm.

In the future, we plan to extend the presented approach to recurring shared workloads and parallel execution. More in detail, we think it is possible to extend this approach to recurring workloads, where the same DAG of jobs is executed over and over again. This is typically the case in systems like DBStream, presented in Section 3.4. For example, the same set of queries might be executed for each new batch of arriving data which might happen as often as once per minute. What we have in mind is an approach where multiple DAGs of sequential time batches are executed in parallel. In addition, by introducing parallelization borders we can keep the cache-oblivious property presented in this chapter.

Part II.

Network Monitoring Applications

6. Introduction

In recent years cellular network and especially mobile Internet usage has increased at a rapid pace. Just a couple of years ago mobile communication was restricted to phone calls and text messages. This usage pattern has changed drastically and nowadays people use their smart phones mainly for tasks like e.g. browsing the Internet, reading email, watching videos, social media, shopping and securing banking transactions via mobile Transaction Authentication Numbers (mTANs). In addition, novel mobile devices like e.g. M2M or wearable devices introduce unprecedented usage patterns to mobile networks. Therefore, it is very important for mobile ISPs to constantly adapt and improve their network and, if possible, protect their customers from cyber threats.

The vast amount of important information stored on smart phones and their increasing involvement in payments and banking transactions makes it very interesting for cyber criminals to steal that information or even get control over the whole mobile device. The diversity of OSs of such devices has undergone a stringent consolidation. In 2013, 78.5% of all sold smart phones were running the Android OS and only about 15.6% iOS¹. This makes finding exploits for a single mobile OS, particularly Android, more valuable since a large quantity of smart phone users might be affected. Apart from personal information fraud and identity theft, devices remotely controlled by cyber criminals via malicious software might take part in so called Distributed Denial of Service (DDoS) attacks. During such an attack, many infected devices e.g. jointly sent requests to a particular website or service over and over again in order to overload it and ultimately render it unavailable. From a network perspective, each device involved in such an attack may use a lot of bandwidth. This can cause local or even network-wide problems caused by overloaded links or other equipment.

Another important, fast growing share of mobile communication is caused by Machine-to-Machine (M2M) devices. Those devices include telemeters (e.g. used in trucks to track tolling), point-of-sale (POS) terminals (e.g. used in restaurants for paying by debit card directly at the table), but also include smart meters, health monitoring devices and many other applications. M2M devices are part of the so called Internet of Things (IoT) which

¹Please refer to <http://www.gartner.com/newsroom/id/2665715> for details.

6. Introduction

was recently extended to the even broader definition of the Internet of Everything (IoE). According to Ciscos Visual Networking Index (VNI) [131], mobile data traffic of M2M devices has the highest yearly growth rate of 113% among all mobile device types. Therefore, many operators are interested in detecting and monitoring the novel traffic patterns introduced by such devices over extended time periods to be able to plan and adapt their networks accordingly. Another interest is the development of novel business models, such as those presented in [139], tailored to suit the demands and needs of M2M devices.

6.1. Organization of Part II

In the second part of this thesis, we present several NTMA applications realized with the systems presented in first part. In the following Section 6.2, we give an overview of the mobile network monitoring principles underlying the presented NTMA applications. In Chapter 7 we present a general overview of several DBStream applications and statistics form operating DBStream in the DARWIN4 [133] project for more than one year. In Chapter 8 we present an application of the field of mobile Internet security dealing with the detection of malicious botnets. We present the identification and tracking of several clusters of malicious hosts found in the live network over an extended time period. The second application is presented in Chapter 9. Here, the focus is on the identification of M2M devices in a mobile network, based on coarse-grained traffic statistics.

6.2. Network Monitoring Principles

In this section we introduce the principles of mobile networks and Network Traffic Monitoring and Analysis (NTMA). Although 4G networks are widely used already in countries like the US and South Korea, in this thesis we focus on studying data from a 3G network. The reason is simply that the used monitoring system offers mainly data from the 3G part of the mobile network.

In the following, we give a short overview of the inner workings of a 3G network and explain certain network monitoring methodologies. At the end of this section, we present the network monitoring system METAWIN [115], which was the source of most of the analyzed data.

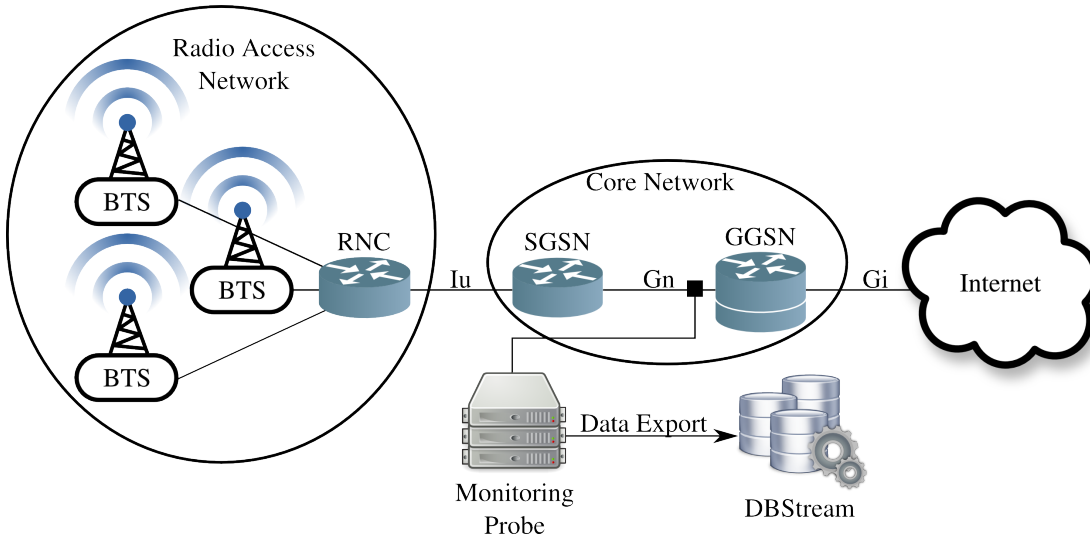


Figure 6.1.: Simplified overview of a 3G network including a monitoring system like e.g. METAWIN and the data export to DBStream.

UMTS Data Plane Overview

In this section, we present a simplified overview of a 3G network. The focus is on the building blocks of the network, which are necessary to understand the applications presented later on. Please refer to [14] for a full overview of 3G networks with a special focus on the mobile Internet part .

A simplified view of a 3G network is shown in Figure 6.1. A 3G or Universal Mobile Telecommunications System (UMTS) network as defined by the 3rd Generation Partnership Project (3GPP) is divided into two parts. The outer part is called the Radio Access Network (RAN) and the inner part is called the Core Network (CN). In the RAN, Mobile Stations (MSs), like e.g. smart phones, tablets, Universal Serial Bus (USB) sticks offering 3G connectivity, etc., are connected to antennas referred to as Base Transceiver Stations (BTSs). Multiple such BTS are then connected to one Radio Network Controller (RNC). Again multiple RNC are connected via the so-called *Iu* interface to the first component of the CN, which is the Serving GPRS Support Node (SGSN). The SGSN is responsible for handling MS mobility and multiple such SGSN forward data to the Gateway GPRS Support Node (GGSN) over the *Gn* interface. The GGSN is the last part of a 3G network and keeps track of users connected to the network. From here data is forwarded over another interface called *Gi* to the Internet. When a user connects to the 3G network, the GGSN keeps track of all information regarding that

6. Introduction

user in a so-called PDP-Context. The whole network forms a star like structure which concentrates towards the GGSN.

As shown in Figure 6.1, in our setup the network monitoring system is connected to the *Gn* interface. At this interface data from large parts of the network are concentrated, making it a suitable vantage location for network wide analysis. From the monitoring system, data are extracted and exported towards the DBStream system. In the whole system only anonymized data are used.

The 3GPP specified a novel protocol for the handling of mobility inside a mobile network. This protocol is called GPRS Tunneling Protocol (GTP) and is used in the mobile part of the network, from the MS until the GGSN. GTP contains several protocol layers below the traditional user IP layer. Those layers were introduced specifically to handle mobility, since typical Internet protocols like TCP/IP are not designed for moving devices. GTP enables MS to move from one geographical location to another, connecting and disconnecting from several BTS on the way, whilst they are constantly connected to the GGSN and therefore have constant Internet access. Since GTP is a tunneling protocol, user IP traffic is encapsulated into additional GTP packets. Therefore, legacy IP based network monitoring solutions can not be applied directly. All GTP and in some cases even lower protocol layers have to be parsed, extracted and recombined in mobile network monitoring software before the application of traditional IP based network monitoring approaches is feasible.

Network Monitoring

In general, network monitoring data can be collected by multiple methods, yielding multiple levels of aggregated data. The highest level of aggregated network data can be collected directly from routers and other networking devices using the Simple Network Management Protocol (SNMP). With SNMP, so called Management Information Bases (MIBs) can be polled, containing very basic information like incoming and outgoing bytes for a link, aggregated over time in between two polling intervals. More detailed information can be accessed by flow based analysis. A flow is identified by the five tuple $\langle \textit{Source IP}, \textit{Destination IP}, \textit{Source Port}, \textit{Destination Port}, \textit{Protocol} \rangle$. Most routers are able to send flow data in the NetFlow [37] format containing a flow identifier and some pre-aggregated statistics about the flow. The most detailed information can be generated by looking into the contents of each packet flowing through the network, this technique is called Deep Packet Inspection (DPI). Many techniques and tools were developed using packet information at this granularity [117, 126].

Different kinds of network monitoring applications exist, based on the analysis of net-

work data depending on the interests and problems arising at the operator of a network. Network monitoring can generally be separated into the following categories: anomaly detection, traffic classification, service performance monitoring and intrusion detection. A PhD thesis [118] recently published at the University of Vienna, gives a broad overview of anomaly detection techniques and introduces novel techniques improving the state of the art. The anomaly detection approaches used there are based on a detailed analysis of the underlying feature distributions introduced in [43] and further refined in [39].

Other methods include the precise measurement of latencies in mobile networks. Whereas network-wide Transmission Control Protocol (TCP) RTTs were studied in [121]. Detailed studies of one-way-delays are provided in [119] and [120]. A comparison between the delays achieved in 3G and 4G mobile networks was conducted in [87].

A good overview of traffic classification approaches based on machine learning is given in [141]. Whereas, in [95] an algorithm to detect changes in the service performance is given. Intrusion detection is typically based on signature matching, for which the tool SNORT [117] is widely used.

The aforementioned approaches are typical tasks answered by network monitoring applications. A centralized easy to use DSW as proposed in this thesis can help in implementing those as well as to identify new threats to the network online, as they occur. The NTMA applications presented in the Chapters 7, 8 and 9 are examples of NTMA performed in a mobile and fixed-line networks.

The Online Monitoring System METAWIN

In this section, we give a brief introduction of the network monitoring system Measurement and Traffic Analysis in Wireless Networks (METAWIN) presented in [115], tailored for passively monitoring 2G and 3G networks. The METAWIN system was developed at FTW in a series of application oriented projects called METAWIN and DARWIN, before the start of this thesis. Although the METAWIN system can be used to monitor all links in the core of a 3G network, we focus in this section only on the monitoring of the Gn interface. Since, the data used for the applications presented later on, stems from this interface.

In the METAWIN system, data is first captured at a monitoring probe, running the LINUX operating system, equipped with one or more Endace capture cards [49]. Those capture cards are able to capture traffic at a very high line rate and provide the option to use Global Positioning System (GPS) clock synchronization to achieve timestamp errors as low as 100ns or better. The METAWIN system is able to parse all packets including user-data and signaling frames. For privacy reasons, IMSI and MSISDN information

6. Introduction

is anonymized by the system using irreversible hash functions and all user payload is stripped away. Since the output of the anonymization is still unique per MS it is possible to generate statistics per MS in a privacy preserving way.

In the next step, but still on the monitoring probe, aggregations are generated for certain protocols, like e.g. DNS or Hypertext Transfer Protocol (HTTP). Those aggregations are generated at line rate without applying any packet sampling, but outputs might only be generated at certain time intervals. The aggregation results are then stored on RAID arrays for optimized I/O as well as data safety. The TicketDB, as well as the DBStream system fetch those stored results and import them into PostgreSQL databases stored on a separate server.

7. DBStream Application Overview

In this chapter, we give an overview of multiple DBStream applications and discuss statistics from operating DBStream for more than one year during the DARWIN4 [133] project. In Section 7.1, we present HTTPTag, a traffic classification system for analyzing applications and services running on top of HTTP. In the following Section 7.2, we study an anomaly found while analyzing the inner workings of a widely used CDN. Finally, in Section 7.3 we report several statistics from running DBStream in the network monitoring project DARWIN4 [133]. In this project, DBStream was used as the central analysis system. In the following chapters, we present two more NTMA applications.

Although we will present four DBStream applications in detail in this thesis, there exist still many other applications, which we shortly summarize in the following. The authors of [135] describe DBStream as part of the general architecture for network monitoring envisioned in the mPlane FP7 project. In [55] several performance impairments of the CDNs hosting Facebook and Youtube are analyzed using DBStream. An early study of performance degradations in Youtube is given in [28] which was later further detailed in [29]. In addition to HTTPTag presented in Section 7.1, more detailed studies of HTTP traffic have been conducted using TicketDB in [31] and [30]. Most recently a novel approach for the characterization of M2M traffic was conducted [122] and the well-known Whatsapp chat service was studied in [56].

7.1. HTTP Classification

In nowadays Internet, HTTP traffic is responsible for the largest share of traffic. The raise of social networks like Facebook and especially the delivery of video streams via HTTP have increased its traffic volume drastically. A recent study of residential customers [96] found that 75% of all Internet traffic is transferred via HTTP. For mobile ISPs it is crucial to understand the composition of the traffic flowing through their network, to be able to plan and adapt their infrastructure accordingly. Therefore, analyzing HTTP traffic is an important task for mobile ISPs.

In this section, we present HTTPTag, a flexible online traffic classification system

7. DBStream Application Overview

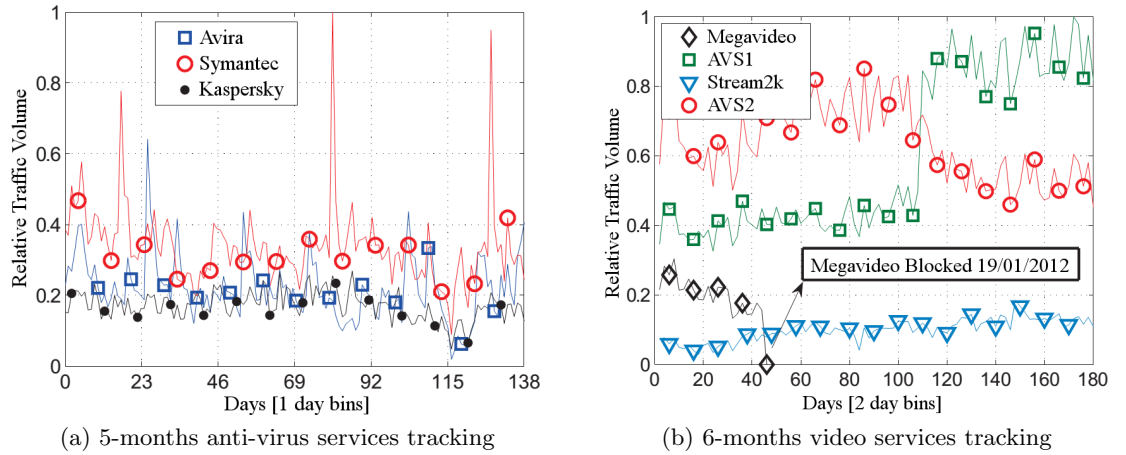


Figure 7.1.: Long term service usage patterns of frequently used anti virus services and popular video streaming services.

for analyzing applications running on top of HTTP. HTTPTag was introduced in [54]. Pierdomenico Fiadino, who is the main author of this work was responsible for designing the HTTP specific part of HTTPTag. The author of this thesis, was mainly responsible for implementing and running HTTPTag on top of TicketDB and porting it to DBStream later. Pedro Casas, is responsible for generally steering the work and many writing improvements.

7.1.1. HTTPTag - Host Name to Tag Matching

The field of traffic classification is very broad and was intensively studied during the last decade. Multiple different approaches and techniques have been evaluated in the research community to achieve this task. The authors of [42] give a comprehensive overview of traffic classification approaches. The most recent approaches, including [102] and [70] as well as the MTRAC approach presented in Chapter 9 apply machine learning techniques for traffic classification purposes. Traditional traffic classification approaches utilize TCP or User Datagram Protocol (UDP) ports to infer the application of the underlying service. Other approaches, like e.g. Tstat [57], use DPI to identify the application of a traffic flow. In contrast, HTTPTag characterizes HTTP traffic based on the host name included in the HTTP header and is therefore similar to the approaches presented in [50] and [51].

The main goal of HTTPTag is to identify how HTTP traffic is distributed among Internet services. In the first step, traffic is captured at the Gn interface (please refer to

Section 6.2 for details), where HTTP packets are identified on-the-fly by the METAWIN system [115]. In TicketDB, the hostname of each HTTP flow is matched against a table of manually created patterns. Those patterns are simple regular expressions, where for each service, multiple regular expressions might be used. For example, for Facebook we use the following two regular expressions: `^www\.facebook\.com$` and `^www\.fbcdn\.net$` since both hostnames are associated with Facebook. In the case of Google, we use a more complex regular expression: `^www\.google\.[a-z]{2,3}$` to be able to match all different country domains (like e.g. `www.google.at`) and other domains (like e.g. `www.google.com`). The inferred service, along with the sum of all bytes exchanged in the HTTP connections is aggregated and stored in TicketDB, first per hour and finally on a daily bases. Those results are stored over extended time periods and are used for live visualization in a web interface.

7.1.2. Long Term Service Usage Patterns

Figure 7.1 shows the results of a long term application of HTTPTag to an operational mobile network. In Figure 7.1a we visualize the tracking of anti virus services over a time period of 138 days. Especially for Symantec, there are three clearly visible spikes, which are caused by updates of the anti virus software. In Figure 7.1b, we show the traffic evolution of several video streaming and Adult Video Services (AVSs), tracked over a time period of 180 days. Please note that Megavideo was blocked starting from the 2012-02-19, this is clearly visible as the traffic volume drops to zero.

The given results show the evolution of different types of services found in a mobile network. ISPs might use this information to optimize the traffic on their core links by applying certain content caching strategies. From the long term evolution of traffic volumes, they might be able to infer when it will become necessary to upgrade a certain link in their network.

7.2. Tracking Anomalies in the Akamai CDN

In today's Internet infrastructure Content Delivery Networks (CDNs) play a very important role. By deploying servers in multiple locations across the Internet, content can be served to end-users with low latency, high availability and high performance. However, CDNs pose novel challenges to ISPs, since changes in the server allocation policy can cause sudden changes to the traffic patterns observed by ISPs. This impacts traffic engineering and, in the worst case, may result in a reduced end-user quality of experience. Therefore, ISPs need advanced monitoring solutions to track traffic served by CDNs.

7. DBStream Application Overview

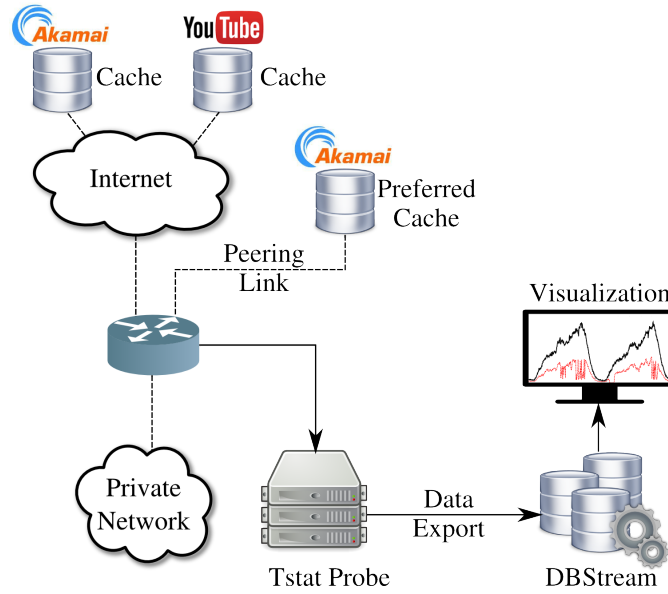


Figure 7.2.: Tstat plus DBStream.

Akamai Technologies¹ is one of the leading CDN providers, delivering approx. 180 TB of HTTP content every 60 seconds².

In this study, we use data from a fixed-line instead of a mobile network, showing that DBStream can also be successfully used in this scenario. Since the Akamai CDN is also widely used among mobile applications and in fact, many fixed-line and mobile applications like e.g. Facebook use the same infrastructure, we think that the approach presented in this section can also be applied to mobile networks.

This work was originally published as a technical report [17]. The author of this thesis, was mainly responsible for implementing and running the DBStream related tasks and investigating the anomaly. Alessandro Finamore and Ignacio Bermudez are responsible for designing and implementing statistical methods to study the found anomaly. Marco Mellia, Pedro Casas and Lukasz Golab are mainly responsible for steering the idea and the resulting work.

7.2.1. Fixed-line Network Data

For this study, we designed and implemented a network analysis system consisting of Tstat, used to capture network traffic, and DBStream, used to analyze the provided

¹<http://www.akamai.com/>

²<http://www.akamai.com/60seconds>

network statistics in greater detail. An overview of this monitoring system is shown in Figure 7.2. Tstat [57] is an open-source packet analyzer capable of monitoring network links of up to several Gb/s relying only on commodity hardware. TCP connections are rebuild in real-time by passively inspecting packets exchanged by networked devices. For each monitored flow, more than 100 different statistics are reported. In our approach, Tstat acts as the probe and generates network monitoring data. This data are first stored locally, on the Tstat probes, and are then transfered to and imported into DBStream. For this study, we used two passive Tstat probes capturing two different vantage points installed in PoPs of a major European fixed-line ISP.

DBStream was installed on a single server with 32 GB of RAM, a single XEON E5 2640 CPU, running at 2.5 GHz, and four 2TB disks (7200 RPM) combined to a RAID 10. We imported the organization database of MaxMind³ into DBStream. The MaxMind database is a commercial product providing for IP address ranges the names of the organizations registering them. It corresponds to a large table in the format `<IP address range, Organization name>` and is later used to map/filter flows by the organization name hosting the destination servers.

We use a one-week long dataset, collected starting from May 13th, 2012. The total dataset corresponds to approx. 496 GB of data (approx. 703 GB after importing it into DBStream), containing 1.052 billion TCP flows. Importing the whole one week long dataset into DBStream took about 14 hours. Therefore, it takes about two hours to import a single day, meaning that this rather small DBStream installation could handle an up to 12 times larger amount of traffic in an online setting.

7.2.2. CDN Tracking using DBStream

We now present an overview of the most important results found while studying the Akamai CDN using DBStream. In the first step, we implemented a DBStream job, using the MaxMind database to extract all flows that contain the string "akamai" in their organization name. In addition, we specifically track a single /25 subnet of the Akamai IP range, which we found to host a large cache, geographically very close to the vantage point under study. Indeed, the servers of this Akamai cache are located behind a network link which is the result of a direct peering agreement between Akamai and the local ISP. In the following, we refer to this subnet as the *preferred* cache.

The top of Figure 7.3 shows the evolution of the number of connections served by the Akamai CDN on Monday and Tuesday. During the peak hours, the *preferred* cache serves

³<http://www.maxmind.com/en/organization>

7. DBStream Application Overview

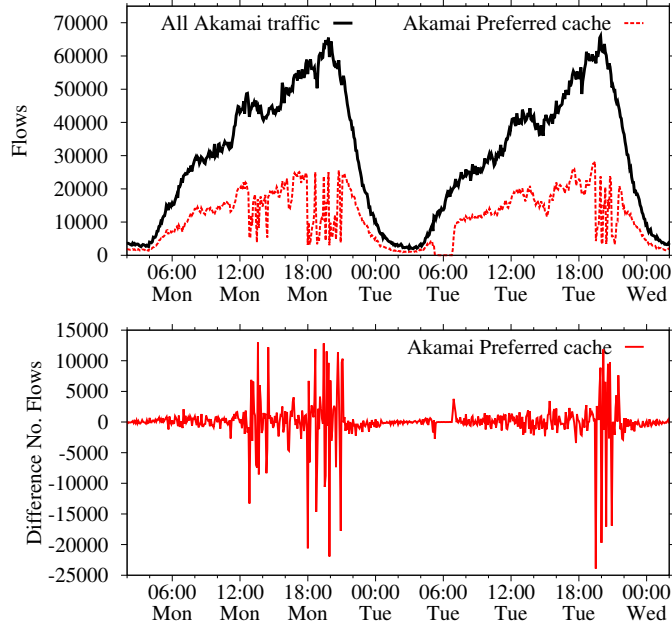


Figure 7.3.: Evolution of number of connections served by Akamai CDN (top) and difference of number of connections served in consecutive 5 minutes time windows (bottom).

about 30% of all traffic exchanged between Akamai and the customers of the studied ISP network. It is clearly visible, that during this time, several notches in the number of flows of the *preferred* cache can be observed. These seem to be caused by the server selection policies of Akamai, shifting traffic between the *preferred* and other Akamai caches. In the bottom part of Figure 7.3 we visualize the evolution of the difference in number of connections served by the *preferred* and other caches in two consecutive 5 minute time windows. Both figures have been obtained using incremental queries as described in detail in Section 3.5.3. We found this characteristic being present for the whole week but we did not observed any clear indication of periodicity.

User Performance Impact

We now investigate the impact of the observed traffic shifts between different Akamai caches on the end-user performance. The top of Figure 7.4 shows the evolution of the 5th, 25th, 50th, and 75th percentile of the elaboration time for the considered time period. We define the elaboration time to be: the time between the clients first packet containing any payload, and the servers first packet containing any payload. In the

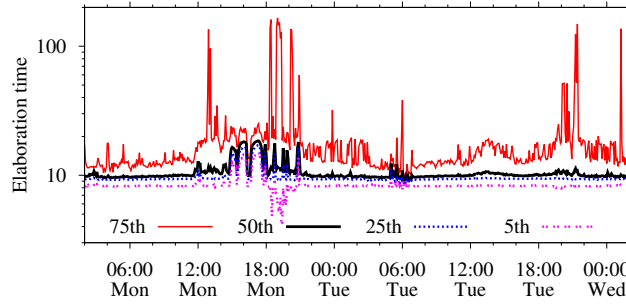


Figure 7.4.: Evolution of server elaboration time percentiles for the preferred data center in log-scale.

case of HTTP, these correspond to the time between the HTTP-request and the HTTP-response. In this case we consider 5 minute long time windows to retrieve accurate percentile estimations. The results show a severe impairment of the elaboration time during the traffic shift occurring on Monday. In particular, the 50-percentile grows from about 10 ms to about 20 ms before and during the shifts around 18:00. This suggests that the CDN server re-allocation could be triggered by some performance issues inside the cache. However, the same effect can not be seen on Tuesday, when no clear impact on performance is visible during the corresponding traffic shift.

Although, in the presented study, we can not give any final conclusion on how much the quality of experience of the customers was affected by the observed traffic shifts and the resulting elaboration time increases, we can conclude the following:

- During the peak hours, the CDN server allocation policy redistributes traffic between the *preferred* cache and nodes in other data centers.
- This clearly has a negative effect on the elaboration time.
- The network path might have an impact on the captured latency oscillations, which can not be identified by our approach.

Although we were not able to find the ultimate cause for the found anomaly, we showed with this case study that DBStream can be successfully applied on fixed-line network data.

7.3. Operating DBStream at Scale

In this section, we present several statistics gathered from the DBStream installation operated in the DARWIN4 [133] project. Unfortunately, due to NDA constraints with

7. DBStream Application Overview

our operator, we are not allowed to present any further details about the applications running on top of DBStream. Therefore, we just present statistics about the general performance of DBStream rather than results from specific applications. To this date, the DBStream installation operated in DARWIN4 [133] was the largest installation.

DBStream was installed on a high performance server machine, hosting four AMD 6380 CPUs, running at 2.5 GHz. Each CPU houses 16 cores, resulting in a total of 64 cores. In total, we installed 256 GB of RAM. The disk subsystem in it's final state consists of four fiber-channel attached RAID arrays, each with 12x 2TB disks forming a RAID10. In addition, the 24 internal disks are split into two disks for the OS, running a RAID1, the other 22x 2TB disks form a large RAID6. All disks except those used for the OS are PostgreSQL tables spaces and are used by DBStream.

In total, DBStream was operated for **385 days**. On the final day of operation, the table partitioning resulted in **984 thousand** tables. This is a very high number of tables, considering that most databases use several hundreds of tables and a typical database administrator know most of them by name. Those tables stored a total of **67 TB** of network monitoring data and processing results on the final day. Please note that this statistic as well as number of table are only a snapshot and include only those tables and amounts of data which were not already deleted by the retention module of DBStream.

In total, over the whole run time, **9.482 million** DBStream tasks (each corresponding a PostgreSQL query) were executed. Put another way, in average, each **3.5 seconds** a new task, updating a time window, was executed. Those tasks produced a total of **1.999 trillion** result rows. Please note that in the current version of DBStream it is not possible to track the number of rows imported into DBStream, therefore this number should be considered as a lower bound and the actual number might be more than twice as high. Last but not least, the processing time of all queries together sums up to **1.212 billion** seconds or **38.45 years**. In the Darwin project, many different data sources and therefore many different imports were used. As soon as an import reaches real-time processing, it e.g. takes 10 minutes to import 10 minutes of data. Thus, 38.45 years is not equivalent to 38.45 years of CPU time, but still in average **36.5** long running analytical queries were executed concurrently.

In the DARWIN4 [133] project, DBStream could successfully be used to analyze multiple network anomalies and run many continuous analysis tasks. In fact, DBStream was used as the main analysis system in this project. Although, not impossible, it would have been much more cumbersome and time consuming to implement all those applications without DBStream.

7.4. Summary

In this chapter, we gave an overview of several applications from the network monitoring domain, realized with TicketDB and DBStream. The HTTPTag approach, which was first designed and implemented in TicketDB and later ported to DBStream, demonstrated the ability of those systems to monitor the trends of applications and services implemented on top of HTTP. In the other application, we showed how DBStream was used to detect an anomaly in the Akamai CDN and evaluated the impact on the end users of the network under study. Finally, we presented a summary of operating DBStream for more than one year as the central analysis system in the DARWIN4 [133] project. By showing several high level statistics, we gave insights at the large scale at which DBStream can be successfully operated.

In the future, DBStream will serve as a repository for network monitoring data in the FP7 project mPlane [101]. There, it will receive and store a continuous stream of data from Tstat [57] probes. This data will then be processed by several analysis modules implemented in CEL and made available for visualization in other external systems. In addition, we will investigate the applicability of DBStream to the data from industry 4.0 approaches as well as other data domains where sensor or other measurements need to be continuously be analyzed.

8. Botnet Detection

In the last years, botnets have become one of the major sources of cyber-crime activities carried out via the public Internet. Typically, botnets serve a number of different malicious activities such as DDoS attacks, email spam and phishing attacks. In this chapter we validate the DNS failure graph approach introduced in [79]. We apply a modified and improved version of this approach to an operational 3G network with an user basis at least one order of magnitude larger as in [79]. Since the monitoring system METAWIN offers anonymized Mobile Station (MS) identifiers, we are able to track the identified botnets over a period of several weeks. Our approach identifies several non overlapping groups of malware infected MS which are part of botnets. In Section 8.4, we present the malicious activities of the groups of MS participating in the most suspicious activities. We end this chapter with a description of how the accuracy of our detection approach could be improved in the future by correlating the knowledge obtained by applying our method in different networks.

This chapter is based on the publication [20]. The writing of this paper was mainly done by the author of this thesis. Antonio Paciello was implementing and running most of the experiments as part of his master thesis. Peter Romirer-Maierhofer helped in improving the writing and rewrote some parts of the original paper.

8.1. Related Work

The authors of [79] detect groups of malicious hosts by the analysis of DNS failure graphs. They decompose the bipartite graph of host names interacting with failed DNS requests on a daily basis. By the application of a complex graph decomposition algorithm based on tri-nonnegative matrix factorization introduced in [80] they are able to extract dense subgraphs from the DNS failure graph. With their approach, they are able to track the largest group of malicious hosts over a period of two weeks.

Although our approach is largely based on [79] we extend the state of the art in two ways:

1. The authors of [79] apply their approach to data captured at a campus network

8. Botnet Detection

of approx. 20k hosts. Whereas we apply our extended approach to data captured from an operational 3G network with an at least two orders of magnitude larger user population. This required highly optimized algorithms as well as the use of a high performance monitoring infrastructure.

2. Since we rely on the METAWIN monitoring system which provides anonymized stable MS identifiers we are able to implement a novel cluster tracking method not suffering from the effects of IP churn, detailed in [144].

Other related approaches can be summarized as follows. The authors of [97] introduce an approach for the identification of malicious botnets using Domain Generation Algorithms (DGAs) to hide suspicious activities. They analyze DNS queries for non-existing domain names, short NXDOMAIN queries, in two ways: First, they exploit string-based characteristics to cluster NXDOMAIN names into groups. Second, they filter false positive clusters by the application of a supervised classifier. Finally, they construct a DGA classifier able to identify known, as well as previously unknown botnet clusters. In addition, they show the effectiveness of their approach by the application to a dataset of domain names and are able to reveal several botnets. Their work is different from our work in the following regard. Whereas we rely on graph characteristics of MS requesting NXDOMAIN names, their work is based on string similarity of the requested names.

The authors of [36] present a botnet detection system called BotGAD. In their work, they investigate host group activity based on the extraction of certain statistical features from DNS traffic, like e.g. the number of DNS queries sent per host. They show that their approach is able to detect botnets such as the Storm Peer-to-Peer (P2P) botnet. Their investigation of host group activity is similar to the work presented in this chapter. However, we focus on a graph clustering-technique and mainly rely on unproductive, NXDOMAIN DNS traffic.

Although the approaches presented in this section already exploit host group activities to identify malicious botnets, to the best of our knowledge we are the first to use stable MS identifiers. This enables us to track botnets over extended time periods of several weeks or even months. In addition, we show that the presented approach can also be applied in operational 3G networks with a much larger user basis.

8.2. Definitions

A botnet is a set of hosts that is controlled by a central malicious server instance — typically called *botmaster* or Command and Control (C&C) server — in order to perform

different types of distributed malicious activities such as sending spam email, participating in DDoS attacks or stealing private information. Since we apply our botnet detection approach to a 3G network, we use the term Mobile Station (MS) instead of *host* in the follow-up of this chapter. Once a MS becomes infected by the botnet-specific malware it is controllable by the botmaster, typically without any knowledge of the owner of the infected MS. The main challenge of a botnet operator is to hide any information exchange which could be associated with malicious activities from both, the owner of the MS, as well as the operator of the network serving the infected MS. Accordingly, the traffic between the C&C server and its botnet clients is hidden in order to avoid the detection and removal from the botnet, by e.g. blocking the MS or the removal of the malicious software from the infected client.

One technique for hiding botnet-related traffic is called domain flux [128]. It is based on a Domain Generation Algorithm (DGA) used for generating pseudo-random domain names valid only for a short period at a specific point in time. The botnet owner registers the pseudo random domain name shortly before it will be used by the botnet. Therefore, the domain name of the C&C server can be changed over time and is hard to block or remove from the DNS sub-system. The algorithm for detecting malicious botnets presented in this chapter is specifically tailored for the identification of botnets based on domain flux.

With the recent spread of smartphones and tablet computers the problem of user equipment security has become highly relevant also for operators of mobile networks. For instance, the authors in [8] present a study of possible attack scenarios specifically affecting mobile computing devices. Mobile devices are very attractive victims for potential botnet operators, since such devices tend to be operated in an always-on manner and hence provide continuous availability of distributed computational capacity and network bandwidth. Accordingly, we believe that studying botnets in mobile networks has become an issue of critical importance.

In this chapter, we perform a validation of the basic principles presented in [79], and show that their approach is applicable also in our scenario of an operational 3G network. Based on the findings established in [79], we advance the proposed methods by *i.*) introducing stable user identifiers allowing the long-term tracking of infected hosts, and *ii.*) adapting the proposed methods for the application within a significantly larger network scenario.

8.3. DNS Failure Graphs

In this chapter we analyze DNS traffic captured at the Gn link of an operational mobile network using the METAWIN system detailed in Section 6.2. In contrast to other approaches, based on DNS queries reaching only a single DNS server, we are able to capture all DNS queries, including those sent to provider-internal as well as external DNS servers, operated e.g. by Google Inc. In fact, one strategy of malicious software is to change the DNS server from the default to a DNS server under the control of the cyber criminals. The data used for the presented analysis was captured during the second half of 2012. Based on the analysis of 3GPP-specific tunneling messages the METAWIN system provides the assignment of an irreversibly anonymized MSID to each monitored DNS packet. It is important to note that this MSID is a stable identifier that refers anonymously to one single MS, regardless of which IP address might be dynamically assigned to the MS at any point in time. Within our analysis, DNS request and response pairs are correlated online and are forwarded to the centralized DSW TicketDB, where all subsequent analysis are performed. The architecture of the TicketDB system is presented in Section 3.2 of this thesis.

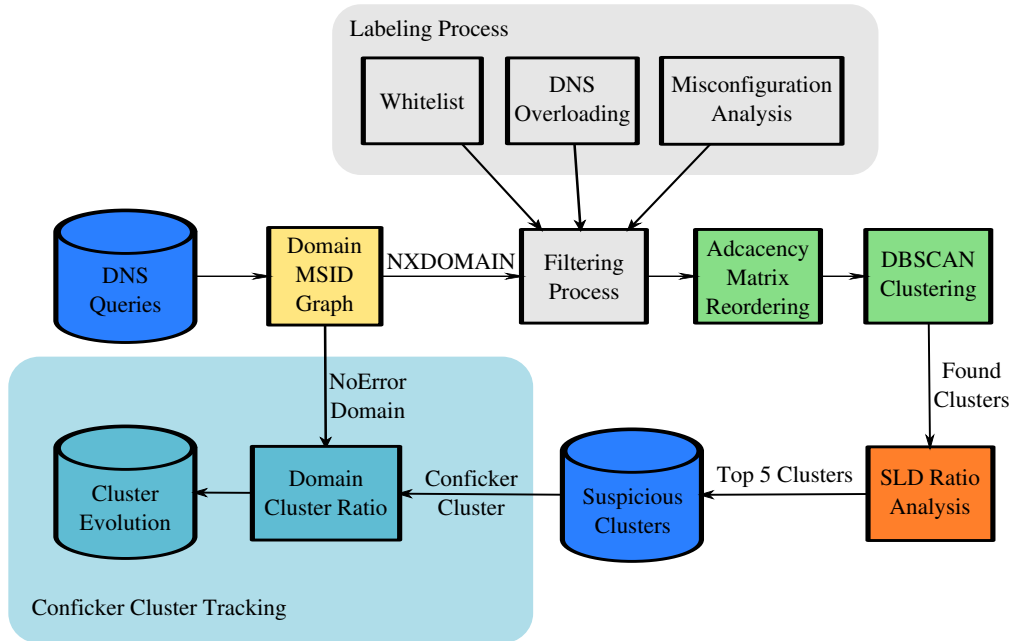


Figure 8.1.: General overview of the whole botnet detection system.

In Figure 8.1 we give a general overview of the steps performed by our DNS-based, online botnet detection approach. We start the generation of the domain name to MS

graph (called domain-MS graph in the following) by the separation of DNS traffic into unproductive (i.e. all DNS queries which did not receive an answer) and productive (i.e. all DNS which were answered) queries. Unproductive DNS traffic is defined as all DNS queries which are answered with the NXDOMAIN flag set, indicating that no IP address could be found for the queried name in the whole world-wide DNS subsystem. For the identification of groups of malicious hosts we mainly rely on unproductive DNS traffic. Later on, we learn from the productive DNS traffic which kind of valid domains are contacted by infected hosts. This allows us to track the active communication of the identified botnets.

In our analysis we found that many failed queries are caused by out-dated services, mis-spellings, wrongly configured clients and DNS overloading (i.e. applications using the DNS subsystem for services other than resolving host names to IP addresses). Those NXDOMAIN queries do not correspond to botnet activity and are therefore labeled accordingly. The full details of the labeling system, which assigns labels not only to domain names, but also to MSs and DNS servers, are presented in Section 8.3.2. As shown in Figure 8.1 the established labels are forwarded to the filtering process which discards all NXDOMAIN queries which are highly likely to not correspond to malicious activities. Within the filtering process the domain-MS graph of the remaining unproductive queries is analyzed in detail. Note that this graph is bipartite, i.e. all edges in the graph are connected from the set M of MSs to the set D of domain names. From this bipartite graph, we calculate the corresponding adjacency matrix. This matrix has one row for each MS and one column for each domain name. We then apply the *Adjacency Matrix Reordering* algorithm based on the Hamming distance (described in detail in Section 8.3.3) to this adjacency matrix. The algorithm works by exchanging rows with each other in such a way that more similar rows end up closer to each. It is not only applied to rows, but also to the columns of the matrix. The resulting adjacency matrix visually reveals groups of MSs contacting certain overlapping groups of NXDOMAIN names. The next challenging task is to extract those groups from the reordered matrix. For this purpose, we apply the density-based DBSCAN clustering algorithm introduced in [53]. The full details of this clustering approach are presented in Section 8.3.3.

Not all clusters extracted from the reordered matrix are associated with malicious botnet activities. Therefore, we introduce a novel sorting mechanism based on the number of unique Second Level Domains (SLDs) accessed by the MS of a cluster. We call this sorting mechanism, detailed in Section 8.3.3, *SLD Ratio Analysis*. We validate our approach by manually investigating the top five clusters, presented in Section 8.4. This investigation revealed that two of the clusters can be associated with the previously

8. Botnet Detection

known botnets called Conficker and Torpig, another group might be involved in email spamming and yet another group show the behavior typical for a browser hijacker. After a detailed study of the last group of hosts we discovered that it is part a novel, previously unknown botnet. This botnet is used for Bitcoin mining and was also reported by other researchers in [74] nearly two years after our original study was conducted.

For further investigation of the identified Conficker cluster — which forms the largest group of malicious hosts — we finally extract the participating MSs and implemented a novel tracking approach, based on the Domain Cluster Ratio (DCR) (see also Figure 8.1). We present the results obtained from tracking the Conficker cluster over a period of two weeks and another cluster over a period of three month in Section 8.4.

8.3.1. Initial Data Processing

The first step of our botnet detection approach is the generation of the domain-MS graph. Therefore, per day we create a table from the DNS request response pairs generated by the monitoring system. This table has the following schema: anonymized MSID, DNS RCODE¹ (identifying if the query was answered or not), IP address of the used DNS server and the queried domain name. From this daily table, we create three derived tables. The first contains unique domain names and RCODEs. The second table contains each distinct DNS server. The last table connects the first two and contains for each distinct MSID two sets of pointers, one referencing the accessed domain names and the other referencing the accessed domain names.

8.3.2. Labeling Unproductive DNS Traffic

In our approach, we assign labels to MS, domain names and DNS servers to separate suspicious from legitimate, non-suspicious unproductive DNS traffic. The amount of unproductive DNS traffic is a considerable share of all DNS traffic. In the 3G network under study, 13% of all distinct domain names are only used in unproductive DNS queries.

We label domain names as *whitelisted* by comparing them with a manually created list of non-suspicious domains. In total, this whitelist consists of 90 entries, 50 of which are derived from the top 50 Alexa domain names of Austria². Additional whitelist entries are create by a manual inspection of unproductive domain names. We sort the domain names by the amount of unproductive DNS traffic they produce and create whitelist

¹Please refer to the RCODE definition in the DNS RFC [108] for the exact details.

²See www.alexacom/topsites/countries/AT for details.

entries from the domain names known to be not suspicious. Most of the manually created whitelist entries are regular expressions matching domain names with a common suffix (e.g. facebook.com). Other entries contain fully-qualified domain names or IP reverse lookup domains. In total, whitelisting plus queries for non-Top Level Domain (TLD) domains (meaning the last part of the domain is not among the valid TLDs like e.g. .at, .com, .org, etc.) correspond to 41% of all unproductive DNS traffic.

Next, we also exclude *DNS misconfigurations*, causing approx. 5% of all distinct unproductive domain names in the first day of our analyzed dataset. A typical misconfiguration example looks like this: *www.example.com.example.com*. In our analysis, we consider all unproductive domains, which have a TLD as a third level domain to be misconfigurations. Although our misconfiguration analysis may produce false positives we manually verified that the amount of false positives is negligible.

Another source of failed queries is *DNS overloading*, where the DNS infrastructure is used for other services, like e.g. blacklist lookups. DNS overloading is responsible for approx. 4% of unproductive distinct domain names. Therefore, failed DNS queries, produced by known services are labeled as DNS overloading. In order to increase the reproducibility of the obtained results, we provide here a list of all DNS overloading domains we use: spamhaus.org, rfc-ignorant.org, surbl.org, uribl.com and spamcop.net.

Last but not least, we label DNS servers belonging to the private IP space, like e.g. 192.168.*.* as *private DNS servers*. We exclude unproductive DNS traffic sent exclusively to servers inside the private IP space as defined by RFC1918 [112]. This type of legitimate unproductive DNS traffic corresponds to approx. 2% of all unproductive distinct domain names.

8.3.3. Identifying Groups of Suspicious Clients

In this section we present the algorithm we apply to the bipartite domain-MS graph to identify suspicious clusters. The idea of reordering the adjacency matrix of this graph was introduced in [79], but no specific details of the used reordering strategy are given by the authors. In this chapter, we provide the full details of our reordering approach able to detect several DGA-based malicious botnets as shown in Section 8.4.

First we exclude all legitimate unproductive domain names labeled as whitelisted, misconfiguration or DNS overloading from the list of domains. We also exclude domain names sent to DNS servers in the reserved IP address space. For each remaining unproductive domain name we count the number of distinct MSs requesting that domain name. Since we focus on the discovery of malicious group activity we exclude domain names requested only by one single MS. Next, we extract the domain-MSIDs subgraph

8. Botnet Detection

only for the remaining domain names. Subsequently, we export the adjacency matrix of this subgraph and apply our simple and yet effective Hamming distance based reordering algorithm presented in Algorithm 6.

Algorithm 6 Hamming distance reordering algorithm.

```
function HAMMING_REORDER(vectors)
    output = dict()
    next = find_longest_vector(vectors)
    output[0] = next
    for i = 1; i < len(vectors); i++ do
        min = MAXINT
        for vec in vectors do
            dist = hamming_dist(output[i], vec)
            if dist < min then
                min = dist
                next = vec
            end if
        end for
        output[i] = next
        remove(vectors, next)
    end for
    return output
end function
```

From the exported adjacency matrix, we first generate a hash table that maps each MS to the set of domain names it contacted. Now, we reorder this hash table by means of the `hamming_reorder` function. Next, we take the output and transform it into a domain name hash table, mapping each domain name to the set of MS requesting it. This step is equivalent to creating the transposed reordered adjacency matrix. Again, we apply the `hamming_reorder` algorithm to this hash table.

In Figure 8.2 the effect of our Hamming distance reordering algorithm is shown. The fully reordered matrix is shown in Part 8.2b. In this part, several groups of MS contacting sets of overlapping domain names can be visually identified. In order to automatically extract those dense groups, we apply the DBSCAN algorithm introduced in [53], using the reordered adjacency matrix as a two dimensional space. After evaluating several other clustering algorithms, we choose DBSCAN for to the following reasons:

- It is a density-based algorithm, and the groups exposed by the Hamming distance reordering are dense.
- It is very stable w.r.t. noise.

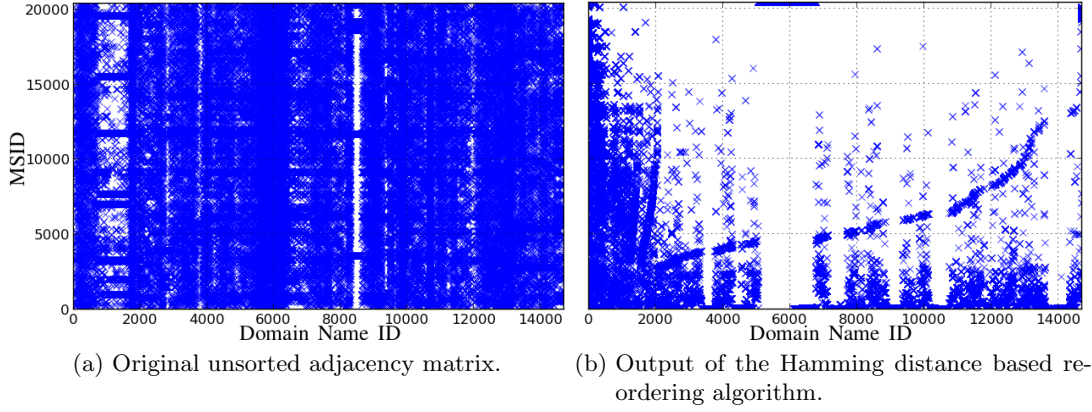


Figure 8.2.: Visual representation of the effect of the Hamming distance based reordering algorithm.

- It does not require the number of clusters, which is typically unknown, as a parameter like e.g. the k-means algorithm.

We use the DBSCAN implementation provided by the tool Environment for Developing KDD-Applications Supported by Index-Structures (ELKI) [4], along with the parameters $eps = 20$ and $minpts = 50$. This parameter setting was found empirically by visually evaluating several different sets of parameters and the resulting cluster quality. The output of the DBSCAN algorithm is a cluster ID per domain name and MS pair, which we import into TicketDB for further analysis. In total we obtain 63 clusters of which not all are caused by malicious activities.

Recall that we focus on the detection of DGA-based botnets which produce pseudo random SLDs in order to make SLD-based traffic filtering more cumbersome for security experts. Accordingly, we propose a cluster sorting approach referred to as SLD ratio, which is based on the number of unique unproductive SLDs. More specifically, the SLD ratio (SR_K) of cluster K is the number of unique SLDs ($SLD(D(K))$) divided by the number of unique domain names ($D(K)$) contacted by cluster K , given by

$$SR_K = \frac{SLD(D(K))}{D(K)}. \quad (8.1)$$

The SLD ratio analysis may exclude malicious clusters querying the same SLD over and over again. While our focus lies on the detection of botnets deploying a diverse set of SLDs, we are aware that the above introduced filtering strategy might result in botnets being undetected if they are designed to rely only on a few different SLDs. However,

our approach focuses on the detection of malicious botnets using DGAs, which explicitly rely on using many different SLDs. Finally, we order all found clusters by their SLD ratio.

8.4. Tracking Clusters of Malicious Hosts

In this section we introduce the Domain Cluster Ratio (DCR), measuring to which extent a certain productive domain name has been accessed by the hosts of a malicious cluster. In Section 8.4.1, we present the details of the top five malicious clusters according to the SLD-ratio ordering. Finally, in Section 8.4.2 we present the results of tracking two botnets over an extended time period.

Our DCR approach is based on the following idea: If a domain name is almost exclusively accessed by a cluster of malicious hosts we can identify additional infected hosts, if they access the same active domain name. This approach enables us to identify also infected hosts not generating a lot of unproductive DNS traffic.

The DCR analysis works as follows: For a malicious cluster K we define $S(K)$ to be the set of MSIDs belonging to cluster K . We compute the DCR for all active domain names accessed by this cluster, defined as:

$$DCR(j, K) = \frac{N_K(j)}{M(j)}. \quad (8.2)$$

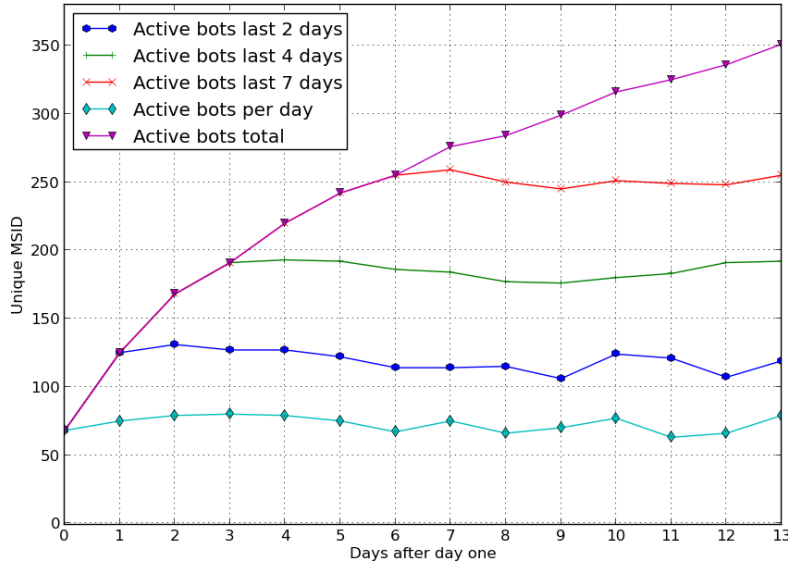
Where $M(j)$ is the set of all MSIDs requesting the domain name j , and $N_K(j)$ is the number of MSIDs of cluster K requesting j .

In addition, we define the Cluster Internal Ratio (CIR) as:

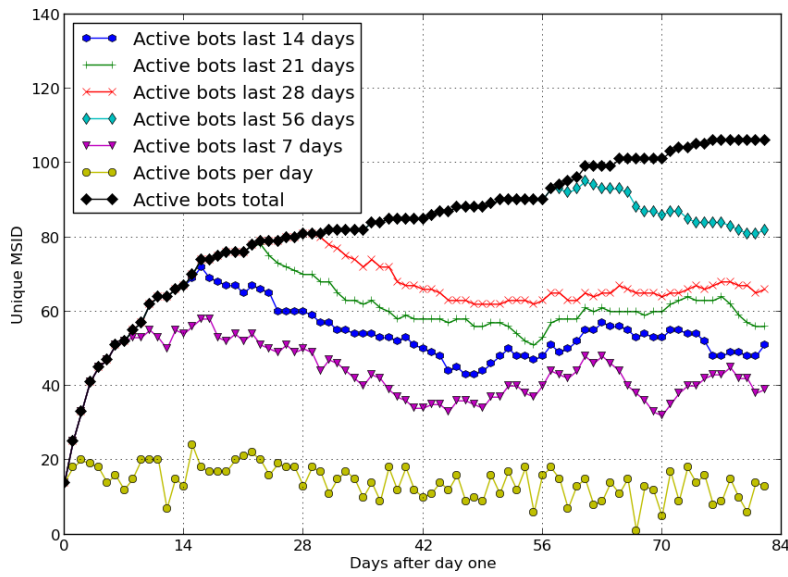
$$CIR(j, K) = \frac{N_K(j)}{S(K)}. \quad (8.3)$$

In order to separate the malicious from non-malicious domain names, we first compute the DCR and CIR for each active domain name of cluster K . In a second step we filter non-malicious domain names of cluster K if the $CIR(j, K) < a$ and $DCR(j, K) < b$. For each cluster the parameters a and b have to be found empirically, by manually investigating the distribution of the DCR and CIR values. While we are aware that the manual tuning of this threshold parameters is a critical step, the presented filtering algorithm is on-going work and subject to future refinements. In this respect, the results presented in the next section are to be considered preliminary.

8.4. Tracking Clusters of Malicious Hosts



(a) Cluster A (Conficker) tracked over 14 days.



(b) Cluster B (tang0-hotel.com) tracked over 83 days.

Figure 8.3.: Number of MSIDs vs. time for clusters A and B.

8.4.1. Dissecting Identified Clusters

In this section, we present our findings about the top five clusters of malicious hosts sorted according to their SLD ratio. Table 8.1 shows all five clusters along with their size and SLD ratio.

ID	SLDs	SLD Ratio	Unique MS	Activity
A	91	1	62	Conficker botnet
B	13	1	20	New botnet (Bitcoin mining)
C	44	1	7	Ad/Spyware
D	81	1	6	Email spam
E	58	1	5	Torpig botnet

Table 8.1.: Top five malicious clusters identified on day one.

Cluster A - The active domain names of this cluster include: e.g. lhnwrocb.org (registered by Conficker Cabal, Microsoft³), iogwldyctr.info (registered by Conficker Holding Account, Afiliast) and uulxrhfnxs.cn (unknown). In total four unique IP addresses are returned for more than 100 different randomized domain names. While three IP addresses are hosted by a known Conficker honeypot, the fourth IP address is listed in the email blacklist of spamhaus.org and it resolves to an access network of an ISP providing Internet access to home customers. This strongly suggests that this IP address is in fact the operational C&C server a Conficker botnet.

Cluster B - Nearly all MSs of this cluster contact a domain name called **tang0-hotel.com**. While it is hard to get reliable information about this domain name, we discovered that two of the four associated IP addresses are blacklisted by www.apews.org. In addition, a Google search for "tang0-hotel.com" returns around 80 results, many of them relating this domain to botnet activity. The first occurrence is from February 2012, where it is related to the "XP Internet Security 2012" malware. Another occurrence relates it to the Dropper malware, which is then used to install a BitMiner malware.

Now, nearly two years after the original botnet detection was conducted it is known that those hosts participated in a malicious botnet. The authors of [74] report that the domain name **tang0-hotel.com** was used as a proxy server for the Bitcoin mining botnet ZeroAccess.

Cluster C - All seven unique MSs of this cluster ask for domain names related with **incredimail.com**. In total, they request 165 distinct domain names,

³In this chapter details about the requested domain names (e.g. registrant, blacklist membership etc.) have been retrieved by relying on www.robtex.com.

e.g. www.incredidating.com or www.smileymessage.com, which all resolve first to www6.incredimail.com and then to the IP address 149.126.77.220. We believe those hosts are infected with *incredibar*, a malicious browser hijacker, responsible for changing DNS settings (Please refer to www.2-spyware.com/remove-incredibar.html for more details).

Cluster D - This cluster is, to the best of our knowledge, involved in email spamming. The top MS produces 120k and the bottom MS still 14k DNS MX queries per day. DNS MX queries are used by email servers to find the mail server for a given domain. The exceptionally high amount of DNS MX queries sent by the hosts of this cluster strongly indicates that this cluster corresponds to a botnet used for email spamming.

Cluster E - Active domain names of this cluster often start with four random characters and end in "ierihon.com". The IP addresses of those domain names are associated with the Torpig bot (e.g. ssahierihon.com and skohierihon.com are registered by Torpig Cabal, Holland).

8.4.2. Long-term Botnet Tracking

We now present our findings obtained by the long-term tracking of the cluster A and B mentioned above. Cluster A was tracked by relying on the CIR and DCR defined earlier. We manually evaluated the best values for the *CIR* and *DCR* and set them to 0.1 and 0.3 respectively, providing the best results for cluster A.

Figure 8.3a shows the evolution of the number of distinct MSs over time for cluster A evaluated over a total period of 14 days. We observe a steady increase over time in the number of distinct, participating MSs. Although the growth rate of cluster A decreases over the measurement period, even after 14 days the total number of participating MS seems not to be reached yet. Accordingly, qualifying the total amount of hosts infected by this botnet requires an extended measurement period, longer than the 14 days analyzed here. It is interesting to note that the number of unique active bots per day is around 75, while it is around 120 when measured for two consecutive days. As shown in Figure 8.3a, we observe a similar increase for measurement periods over larger time intervals.

This increase in the number of active MS may be explained by different phenomena. For instance, an infected MS might be inactive on some days, since e.g. the infected laptop was switched off on a particular day. The size of the botnet might change dynamically, increasing by newly infected MS and, at the same time, decreasing when malware is removed from previously infected hosts. Moreover, the botnet might also use other communication channels which we have not discovered yet. Ultimately, at this point we may even speculate that different bots are activated only on some days

8. Botnet Detection

in order to hide the total size of the botnet infection from the network operators and security experts. The latter phenomenon is difficult to investigate when only relying on dynamically assigned IP addresses for host identification as it is the case in most botnet identification approaches. The website www.shadowserver.org tracks about 2 million unique IP addresses of Conficker-infected hosts per day. Our results indicate that the number of infected hosts might be one order of magnitude larger than the one reported there, when measured over longer periods and by means of a stable host identifier.

In order to investigate cluster B we simply track all MSs querying **tang0-hotel1.com**. Figure 8.3b shows the evolution of cluster B over a total period of 83 days. While the qualitative shape of the time series is comparable to the one of Figure 8.3a, we observe a decrease of the number of infected hosts in the first half of the measured time period.

8.5. Summary

In this chapter we validated our approach to DNS failure graph based botnet detection originally introduced in [79] by applying it to an operational 3G network. We evaluated the performance of our approach and report in detail on five malicious clusters found in the network under study. The MS of those clusters were involved in activities such as e.g. unwillingly mining Bitcoins and browser hijacking. We also revealed communication activities with C&C servers of known botnets like e.g. Conficker and Torpig. In contrast to the work in [79] our stable MSIDs allow us to track clusters of infected hosts over extended time periods.

In the future we plan to improve our approach by integrating data from multiple separated networks. For instance, if the botnet detection approach presented in this chapter is applied only within a single network, local DNS misconfigurations or other problems affecting certain network-specific user groups may complicate the detection of actual botnets. If it is possible to demonstrate that groups of hosts showing the same suspicious behavior exist in both networks it is more likely that this behavior is caused by an external source, like e.g. a botnet. Therefore, a combination of monitoring data from different networks may increase the accuracy of our approach.

9. M2M Traffic Classification

Machine-to-Machine (M2M) network traffic is becoming highly relevant in nowadays mobile networks. The ever-increasing number of M2M devices is heavily modifying the observed traffic patterns and the interest in discovering and tracking these devices is rapidly growing among operators. In this chapter we introduce MTRAC, a complete approach for M2M TRAffic Classification, capable of discovering M2M devices from coarse-grained measurements. MTRAC uses several different Machine Learning (ML) algorithms to unveil previously undetected M2M devices in mobile networks. It relies on very simple traffic descriptors to characterize the communication patterns of each device. These descriptors are robust against traffic encryption techniques, and improve the portability of the MTRAC approach to other types of networks or usage scenarios. The MTRAC approach is implemented on top of the DBStream system presented in Section 3.4. Utilizing DBStream allows to classify M2M devices on an online basis, using different temporal and logical traffic aggregations. We study the performance of MTRAC for the online classification by applying it to more than two months of traffic observed in an operational, nationwide 3G/4G mobile network. The flexibility of DBStream enables us to compare many different ML algorithms and different traffic aggregation techniques. To the best of our knowledge, MTRAC is the first ML-based approach for automatic M2M device classification in operational mobile networks.

This chapter is based on the publication [21] accepted at IEEE International Conference on Communications 2015 (ICC) and will be presented in June of 2015. The author of this thesis is responsible for conducting the experiments, the main writing and coordination of the work. Philipp Svoboda, is responsible for analyzing and selecting the most promising features from the dataset. Pedro Casas is responsible for guiding the work in a scientific way and several writing improvements and additions as well as the name MTRAC.

9.1. Related Work

The field of automatic network traffic classification has been extensively studied during the last decade [42, 138]. The specific application of ML techniques to the traffic classification problem has also attracted large attention from the research community. A non-exhaustive list of standard supervised ML-based approaches includes the use of Bayesian classifiers, linear discriminant analysis and k -nearest-neighbors, decision trees and feature selection techniques, and support vector machines. Many unsupervised and semi-supervised learning techniques have been used for network traffic classification, including the use of k -means, DBSCAN, and AutoClass clustering. Also the GRIDCLUST algorithm presented in [123] and its later extension to the BANG-clustering system [124] are good candidates for being used in NTMA applications, due to their computational efficiency on large datasets. We point the interested reader to [102] for a detailed survey on the different ML techniques applied to network traffic classification.

More recent approaches for traffic classification focus on the specific analysis of the applications running on top of HTTP/HTTPS [54, 23], including modern the analysis of modern Internet services such as e.g. YouTube, Facebook, WhatsApp, etc.

The particular classification and analysis of M2M traffic and M2M devices has very recently emerged as a need to understand the novel traffic patterns such devices introduce. So far only a few papers are available. The most relevant work on M2M traffic characterization is provided in [125], where authors present an extensive analysis of the traffic generated by M2M devices in the AT&T US mobile network. They apply a Type Allocation Code (TAC)-based approach to separate M2M from other devices. Finally, the authors of [89, 86, 88] have recently started to investigate the problem of M2M device classification. They mainly work with offline datasets and use fine-grained traffic descriptors at the packet level to perform the analysis.

The MTRAC approach presented in this chapter aims at the classification of M2M devices in mobile network traffic. The whole system is operated in an online basis and relies only on coarse-grained traffic descriptors at the user session level. To the best of our knowledge, MTRAC is the first system able to perform online ML-based M2M classification in operational mobile networks.

9.2. Motivation

Mobile ISPs have witnessed an astonishing increase of heterogeneous devices connected to their networks in the last years. From end-user devices such as smartphones and

tablets, to Machine-to-Machine (M2M) devices such as telemeters, POS terminals, telematic sensors, etc. These novel devices introduce new traffic patterns and impose new challenges to the operation of mobile networks. The M2M scenario is of particular interesting among mobile ISPs, as the traffic generated by M2M devices might result in difficult to handle or even harmful traffic patterns. As previous studies [125] have shown, specific M2M devices may generate traffic in a synchronized fashion, which may ultimately lead to a denial of service caused by limited resources at the Radio Access Network (RAN) or even the Core Network (CN).

In general terms, M2M traffic and M2M communication refers either to the automated transfer of data among devices and central servers or the communication between such devices connected via P2P networks [143]. The main purpose of such devices is monitoring and/or controlling of remote processes, without direct human interaction. With the availability of affordable technology and the ever-increasing penetration of mobile connectivity, billions of M2M devices will be connected to mobile networks within the next decade. In fact, the Cisco VNI forecasts about 2 billion M2M devices worldwide already by the end of 2018. Especially the data plane of current mobile networks is primarily designed and optimized for smartphone usage. The increasing population of M2M devices may very soon increase the network management complexity.

For those reasons, a main goal for mobile ISPs is to track the evolution and the traffic generated by M2M devices connected to their networks. A first step is to distinguish between M2M and non M2M devices. One standard approach followed by mobile ISPs to identify M2M devices is by its hardware model [125], which can be inferred from its TAC by using the TAC databases of the GSM Association. This hardware model information is generally complemented with device templates which provide a categorization of M2M devices, based on the device type (e.g., laptop, modem, POS, router, telemetry, etc.)¹. These templates are manually produced from public information such as product brochures and specification sheets. Due to its manual construction, the TAC-based approach imposes several limitations to the classification and discovery of M2M devices. In particular, new devices which are not yet included in the templates can not be classified. In other network scenarios, like e.g. fixed-line or WiFi networks, the TAC number is not available as the used protocols do not expose TAC information. In addition, M2M devices might be connected to the 3G network via USB-sticks and multiple M2M devices might be connected over one single 3G router. Other devices might be connected via the smartphone of the user over Bluetooth. Therefore, it is also interesting for mobile

¹For example, the AT&T specialty vertical devices template at http://www.rfwel.com/support/hw-support/ATT_SpecialtyVerticalDevices.pdf.

9. M2M Traffic Classification

ISPs to identify devices which show a behavior similar to M2M devices.

To avoid the limitations of the TAC-based approach, we introduce MTRAC, a novel traffic classification approach to automatically identify M2M devices in operational 3G networks. MTRAC relies on supervised machine learning algorithms to extract behavioral models reflecting the traffic patterns generated by M2M devices. These models are then applied to classify the devices observed in the network in an online basis, relying on the DBStream system presented in Section 3.4. The models are extracted in a learning phase, using the labels provided by the TAC-based approach as ground truth. One of the salient characteristics of MTRAC is that it relies only on very simple traffic descriptors or *features* to characterize the associated devices. In particular, for each session of a device only the total uploaded and downloaded bytes and the start and end of the session are kept as features. Multiple sessions associated with the same device are then aggregated into a single feature vector and used for classification. The notion of session used in this chapter is further described in Section 9.3. Relying only on such a simple and small number of features has multiple advantages:

1. It improves the portability of the MTRAC approach to other contexts such as fixed-line and WiFi networks, as those features are available in any ISP data warehouse.
2. It avoids visibility problems induced by traffic obfuscation and encryption approaches like e.g. Hypertext Transfer Protocol Secure (HTTPS), as the selected features are robust towards those techniques.
3. It results in an improved system efficiency in terms of computational time and storage capacity, as the data volumes that have to be stored and analyzed are highly reduced.

9.3. Feature Extraction and Selection

This section describes the features extracted from the monitored network traffic, which are used as input to MTRAC to discriminate between M2M and non M2M devices.

The authors of [86] define a set of features able to identify M2M devices with high accuracy. However, the assumption in this work is the availability of fine-grained traffic descriptors at the packet level. In contrast, in this work we rely only on coarse-grained network data on a per session level. A session in this chapter is defined to be equivalent to a PDP-context of a device in a mobile network [114], but could be defined in any other standardized way, like e.g. based on traffic time-outs common in fixed-line or

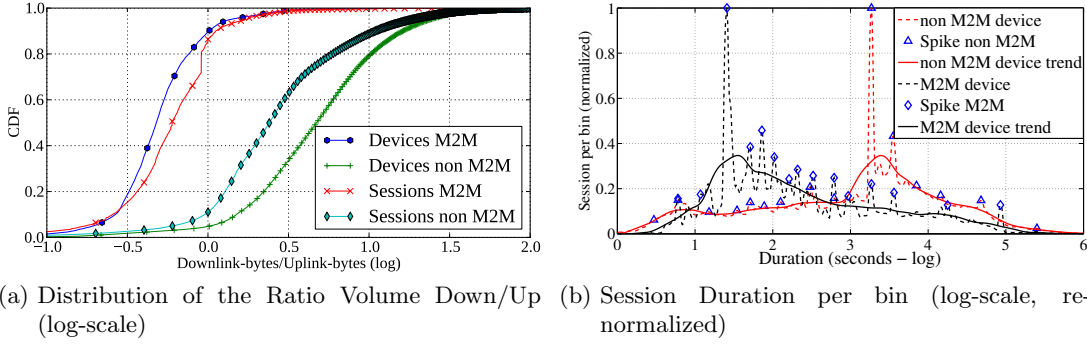


Figure 9.1.: Example features derived for M2M and non M2M devices.

WiFi networks. For each session we collect the following basic descriptors: `<uploaded, downloaded bytes, session start, session end, anonymized device ID>`. Generally speaking, these simple descriptors are available and collected in any mobile network for billing purposes, thus making our approach easily portable to other networks. In addition, as none of these descriptors require any information inside the payload of the traffic, they are privacy preserving and, at the same time, insensitive to encryption and obfuscation techniques.

In [89] it was shown that the statistical properties of the traffic aggregated at the session level differ significantly from those at the device level. Next, we review some of the features extracted from the aggregation of the traffic at both session and device levels, which are then used in the classification process. The curves in this section are labeled as M2M and non M2M based on the TAC-based technique described before. Figures in this section are re-normalized due to NDA constraints with the mobile network operator.

Figure 9.1 shows the impact of aggregating single sessions by the device generating them. The general rationale behind the discrimination between M2M and non M2M devices is that the former behave less randomly than the latter. As a result, we expect that by aggregating sessions per device, the resulting patterns will diverge stronger. Figure 9.1a shows the distribution of the downlink to uplink ratio of traffic volume per session and per device, for M2M and non M2M traffic. This feature has been shown to be highly efficient in discriminating between M2M and non M2M devices as reported in [125]. We calculate the Empirical Distribution Function (eCDF) of the downlink to uplink ratio once for individual sessions and once after aggregating the volume per device, considering all the sessions of a single day. Note how the separation between M2M and non M2M devices increases for the per device aggregation. This feature allows for a first strong separation of M2M and non M2M devices. However, due to difference of one

9. M2M Traffic Classification

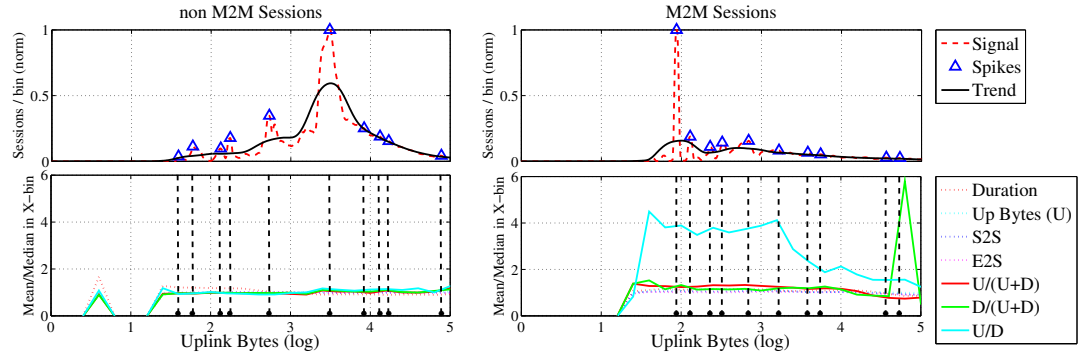


Figure 9.2.: Example of extended features extracted for M2M and non M2M sessions.

order of magnitude in population size between the two groups, the remaining FPR for the M2M class is still above 50%.

The authors of [88] have shown that M2M sessions are discrete in many dimensions, e.g., constant packet size or frequency. Therefore, we extract similar dimensions, available in our reduced dataset to be used as features in the later classification process. As an example, Figure 9.1b gives the normalized histogram of the session duration on a log-scale for both groups. As expected, the M2M group shows a high number of spikes in the histogram. Interestingly, similar spikes are also present for the non M2M group. A deeper analysis revealed that the main spikes found in the non M2M group can be accounted to timeout settings in common operating system, like e.g. Windows 7. Besides this unexpected results, there is still a very clear shift of one to two orders of magnitude between both groups, making it a good feature candidate. Similar results are observed for other dimensions such as uplink or downlink session size.

Figure 9.2 shows a similar histogram as the one presented in Figure 9.1b, but for the uplink bytes per session. The left side of the figure shows the results for non M2M group and the right side gives the M2M results. The first row depicts the estimated distribution of the bytes per session in the uplink direction. The second row shows what we call extended features, referring to additional features which are extracted from the estimated distributions. These extended features are computed from the statistical properties of different dimensions when binning the distributions around their spikes. For example, one might compute the total uplink bytes for each of the modes of the distribution (Up Bytes), or the duration of the binned spikes (Duration), or the time elapsed between the start of one session and the following (S2S), and so on. In particular, the second row of Figure 9.2 shows the mean to median ratio for some of these extended

9.3. Feature Extraction and Selection

ID	Feature
Per device and day	
1	Sum of upload volume (bytes)
2	Sum of download volume (bytes)
3	Sum of connection time (seconds)
Histogram per day	
4	Strongest spikes upload volume (bytes)
5	Strongest spikes download volume (bytes)
6	Strongest spikes connection time (seconds)
7	Strongest spikes time between starts (seconds)
Variations per day	
8	(binned) Unique values upload volume (bytes)
9	(binned) Unique values download volume (bytes)
10	(binned) Unique values connection time (seconds)
11	(binned) Unique values between starts (seconds)
Correlation time series per device per week	
12	Upload volume (bytes)
13	Download volume (bytes)
14	Connection time (seconds)
15	Time between starts (seconds)
Per device and week	
16	Frac. of upload to download volume
17	Download rate (bytes/s)
18	Upload rate (bytes/s)
Conditional threshold	
19	Frac. upload/download volume inside download spikes
20	Frac. upload/download volume inside upload spikes
21	Frac. upload/download volume inside duration spikes
22	Frac. upload/download volume inside start to start spikes
Conditional variations per day	
23	Frac. Mean to Median of uplink rate inside duration spikes
24	Frac. Mean to Median of downlink rate inside start to start spikes

Table 9.1.: Features investigated for M2M device classification.

features. Note that the mean to median indicator is used to flag strong variations in the underlying set of values.

Adding the extended features to the classification process results in an increase of the detection rate to more than 90%. Table 9.1 presents all different investigated features. Whereas features from ID 1 to 15 all contain direct values, derived from the monitored sessions, features with the IDs 16 to 24 correspond to the extended features. Based on these two groups, we selected the best feature set for detecting M2M traffic. We used the well known Weka ML toolbox [72] to identify the strongest candidates, using correlation-based feature selection as evaluation approach, and best first search as exploring algorithm. In the remainder of the chapter, we consider the best 10 features obtained through this selection approach. Unfortunately, due to NDA constraints we

are not allowed to reveal which exact features are used in the later classification process.

9.4. Online M2M Classification

In this section we show how the features described in Section 9.3 are used in MTRAC to identify M2M devices in the operational mobile network of a European ISP. The features extracted from the session data of each device are stored along with the corresponding ground truth of the device, obtained through the TAC-based approach. Part of these data are used to build the ML-based classification models, training different classifiers using Weka. The trained models are finally installed into DBStream, and used in an online basis to assign a class to each of the monitored devices. Recall that our classification problem is a dichotomic one, in which a device is either classified as M2M or non M2M. As described in detail in Section 9.5, we use multiple ML-based approaches to improve the classification performance of MTRAC.

9.4.1. DBStream Weka Integration

MTRAC runs on top of our operational DSW DBStream presented in Section 3.4, tailored for the analysis of network monitoring data. We would also like to point the reader to Section 3.5 for a full description of CEL used to implement the classification jobs underlying the MTRAC approach presented in this chapter.

We extended DBStream by adding a new module able to interface with Weka, to perform online classification based on ML algorithms. This interface, enables its user to write jobs which take a table of feature vectors as input and output a new table containing the classification results. This interface relies on previously trained classification models and can be used for any classification purpose. Weka is instrumented to classify an exported time window of data. Then the classification results are imported back into DBStream. As soon as the time window is imported, it becomes available to other DBStream jobs for further processing or visualization.

9.4.2. Aggregating Sessions per Device

The main challenge when aggregating sessions per device is correct and sufficient timing. In an offline setting, one can aggregate all sessions per device available in the whole dataset and select only those devices which meet certain criteria to perform the classification. For example, one might restrict the classification only to those devices for which a minimum amount of at least N sessions have been observed. In contrast, in an

operational online setting, the dataset does not have a defined end. On the one hand, the amount of sessions used as input to the aggregation should be as high as possible. For example, features based on the session inter-arrival time can only be generated if more than one session is available and many statistical features benefit from more input data. On the other hand, the time it takes until the classification results are available should be as short as possible, thus reducing the number of sessions available in the aggregation period. As we show in Section 9.5, the classification performance increases with the number of aggregated sessions. Therefore, the user of such a system is facing an interesting discrepancy. She can either wait longer time until more sessions are available for aggregation and gain a higher classification performance, or receive the results earlier and accept the resulting lower classification performance.

We implemented two different session aggregation approaches to visualize this trade off. The first approach is called **Simple Daily Aggregation (SDA)** and is based on time, meaning that we execute the aggregation from sessions to feature vectors e.g. after 1, 2, ..., N days. The second approach, called **Threshold Based Aggregation (TBA)**, is based on the number of sessions observed per device. The TBA approach is implemented using a rolling DBStream job (please refer to Section 3.5.3 for details on rolling/incremental DBStream jobs). The job starts from a table **A** containing all sessions from all devices as they are produced. We now want to produce a new table **B** in which all sessions are kept, until at least S sessions for a single device have been gathered. For simplicity reasons, let us assume we update **B** only once a day. A new time window of **B** is thus created by the union over all sessions of the current day stored in **A**, plus all sessions of all devices from the last time window of **B** which have not reached S sessions yet. The CEL job updating the tables for the TBA approach is shown in Algorithm 7. The result is that a session is moved from the old to the new time window of **B**, until there are S sessions available for that device. The session aggregation can now be applied on those devices of **B**, having at least S sessions and is stored in a new DBStream **CT**.

9.4.3. Establishing Ground Truth

Supervised classifiers need to be trained on a dataset containing the real class of the devices, i.e., the ground truth. Getting access to such labeled datasets is generally a very cumbersome process, especially in the case of an operational network. As mentioned before, we use the TAC-based approach to label most of the M2M devices. We then train classifiers using only those devices for which the real class is known. In the later evaluation of the different algorithms we reuse this ground truth to compute the accu-

Algorithm 7 Incremental CEL job used in the TBA approach. Sessions are gathered in table A until at least S sessions per device have been collected.

```

<job inputs="A (window 1day primary), B (window 1day delay 1day)"
  output="B (window 1day)"
  schema="serial_time int4, devID text, s_start int4, s_dur int4,
          bytes_up int8, bytes_down int8" >
  <query>
with t as (
  select devID, count(*) as cnt from B group by 1 having count(*) > S
)
/* Add all new sessions to B */
select * from A
  union all
/* Add all sessions of yesterdays B, if they had less than S sessions */
select __STARTTS, devID, s_start, s_dur, bytes_up, bytes_down, from B old
  where not exists (select 1 from t where old.devID=t.devID)
  </query>
</job>

```

racy, the True Positive (TP) and the False Positive (FP) ratios to evaluate the applied algorithms.

9.5. Classification Performance Evaluation

In this section, we provide a detailed evaluation of the classification performance of MTRAC. In total we evaluate six different ML algorithms.

Decision Stump is a decision tree algorithm generating trees of only one level, therefore only a single feature is used to decide to which class a device belongs.

J48 is a Java implementation of the well-known C4.5 decision tree learner provided by the Weka toolkit.

Random Forest is an algorithm which trains an ensemble of decision tree learners, each on a randomly selected subset of the given features using bootstrapping to generate unique sample subsets for each tree.

Hoeffding Tree is a special decision tree learning algorithm. It produces classification models quickly, which can be updated dynamically as soon as new items arrive.

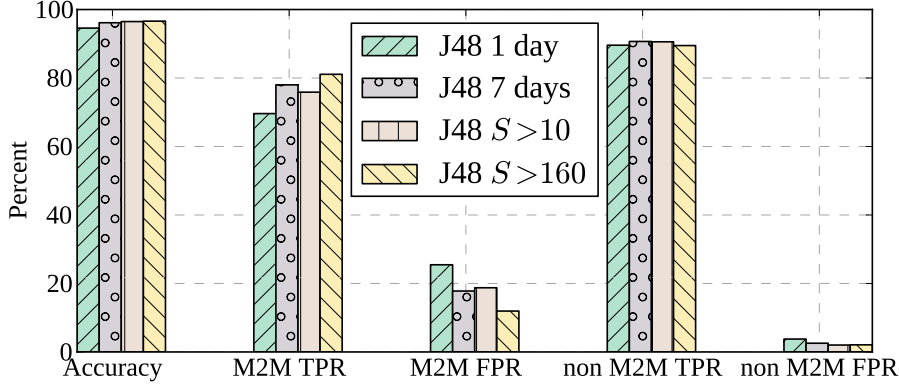


Figure 9.3.: Accuracy, TPR and FPR for J48 algorithm with different session aggregations.

Naive Bayes is a statistical classifier based on Bayes theorem with a strong (naive) assumption that each feature is independent from each other feature.

SVM is a non-probabilistic binary classifier. Support Vector Machines (SVMs) typically provide high classification performance at the cost of long training phases.

Figure 9.3 depicts the accuracy, False Positive Ratio (FPR) and True Positive Ratio (TPR) for different session aggregations. In this figure, we only show the results of the J48 algorithm, given its dominant performance among the tested ones. The reported values are averages over the whole investigation period, excluding the training phase. The overall accuracy is high, which is mainly caused by the rather small fraction of M2M devices in the network, reported in detail in Figure 9.6. Therefore, the most relevant metrics for the classification performance are the TPR and FPR of the M2M class. The FPR of the M2M class is around 25% for the one day SDA aggregation. It improves to around 17.8% for the 7 day SDA approach, which is similar to the M2M FPR of 18.1% for the 10 session TBA approach. We achieve the best M2M FPR of only 11.7% by aggregating 160 sessions, meaning that instead of every 4th, we only misclassify every 9th device.

In Figure 9.4 we compare the FPR achieved by different classification algorithms. Here, we aggregate device sessions through the 10 session TBA approach. The days two to eight are used as training set, therefore this period shows a decreased FPR, most prominent for the random forest algorithm. In this classification problem, complex tree algorithms like the Hoeffding, J48 and random forest achieve the lowest FPRs. The very best performance is achieved by the random forest algorithm, but its very long training phase of several hours is a major drawback. The J48 algorithm provides the

9. M2M Traffic Classification

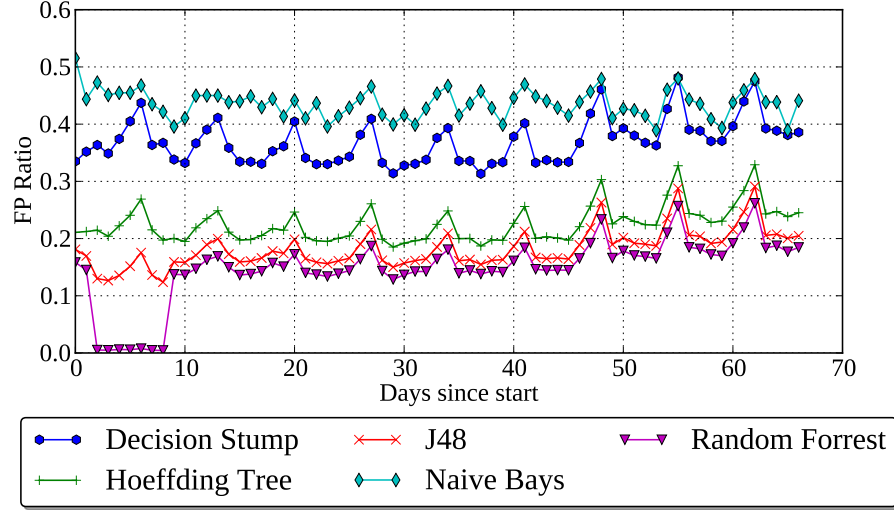


Figure 9.4.: FPR per day for selected classifiers.

best balance between training time and classification performance. Therefore, we use the J48 algorithm for the remainder of this chapter. We also trained a SVM model, which surprisingly resulted in a classifier of very low performance, where nearly all the devices were classified as non M2M. We think this is caused by a strong overfitting of the SVM model to the training data, resulting in a low classification performance in the test dataset.

As shown in Figure 9.6, the fraction of M2M devices is small but represents definitely an important share of the devices in the network. Over the weekends, the relative share of M2M devices decreases. In our opinion this is the main cause for the decrease in the classification performance during weekends, which results in the spikes of the FPR shown in Figure 9.4. Identifying a device correctly as a M2M becomes harder since the overall fraction of M2M devices is lower. We tried to overcome this limitation by training different models for week and weekend days, which did lead to a very small but not substantial classification performance improvement.

In Figure 9.5 we compare the SDA to the TBA session aggregation approach, using J48 models in both cases. In the upper Part 9.5a we show the FPR for the SDA approach, aggregating sessions based on an increasing number of days. For each aggregation we export the first part, i.e. the first day, the first two days, etc., as training set for the J48 classifier. The FPR decreases with longer aggregation intervals, although aggregation intervals longer than 7 days do not seem to decrease the FPR any further. In the lower Part 9.5b we show the classification performance for the TBA approach. In general,

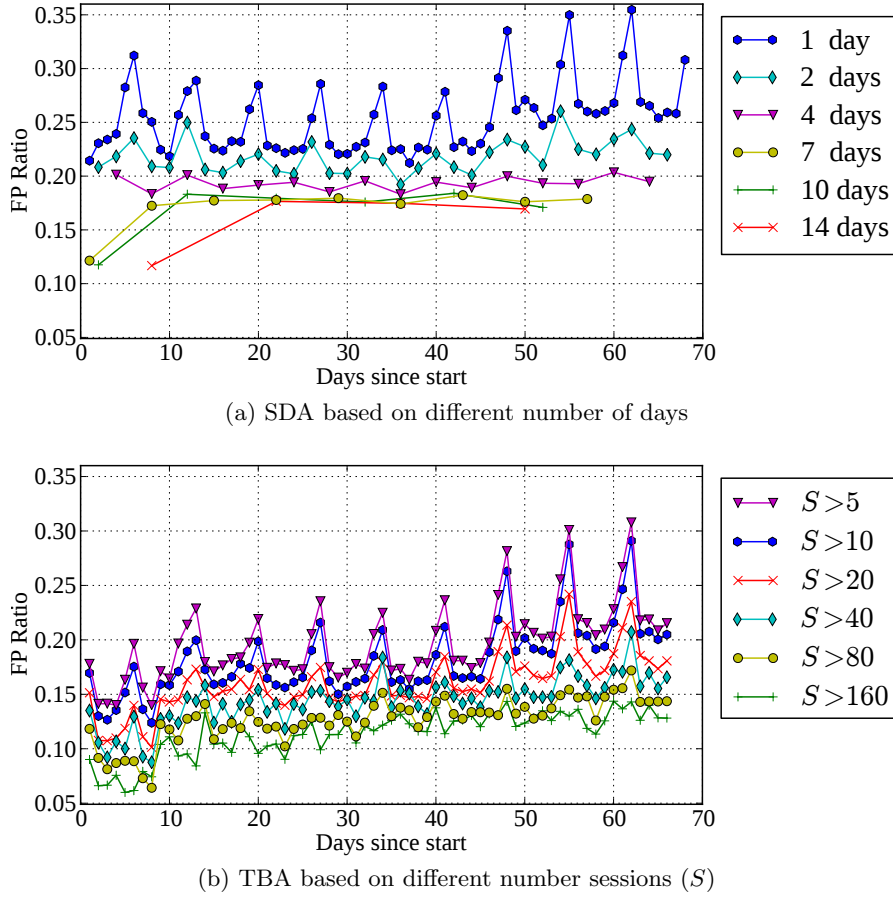


Figure 9.5.: Comparison of our different session aggregation approaches SDA and TBA using a J48 classifier.

the FPR is lower than for the SDA. Also here, longer aggregation intervals result in a decreased FPR, which can get as low as 11.6% in average for $S > 160$. In total, the TBA approach performs considerably better for longer aggregation intervals as compared to the SDA approach.

Especially for the TBA approach, it is interesting to analyze how long it takes until a device is classified. To provide this insight, Figure 9.7 shows the normalized cumulative amount of devices reaching at least S sessions. It is shown that the number of devices with more than S sessions grows slower for larger thresholds. For example, for the $S > 160$ TBA approach, even after an investigation period of more than two months, only 33.8% of devices pass this threshold.

9. M2M Traffic Classification

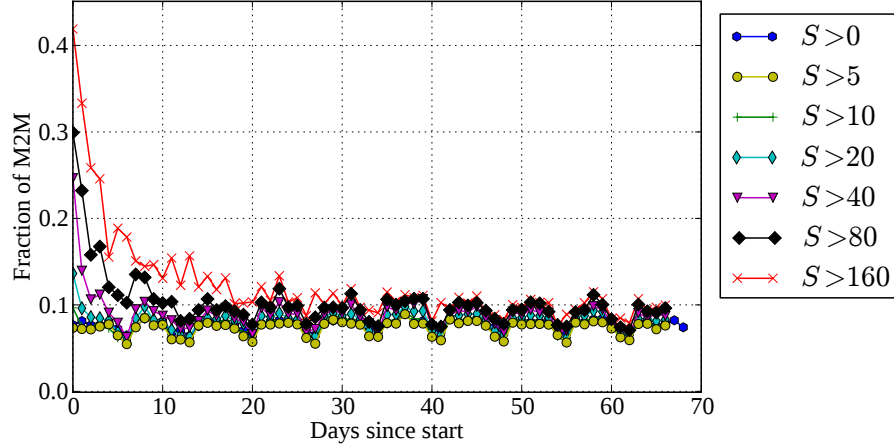


Figure 9.6.: Fraction of M2M devices

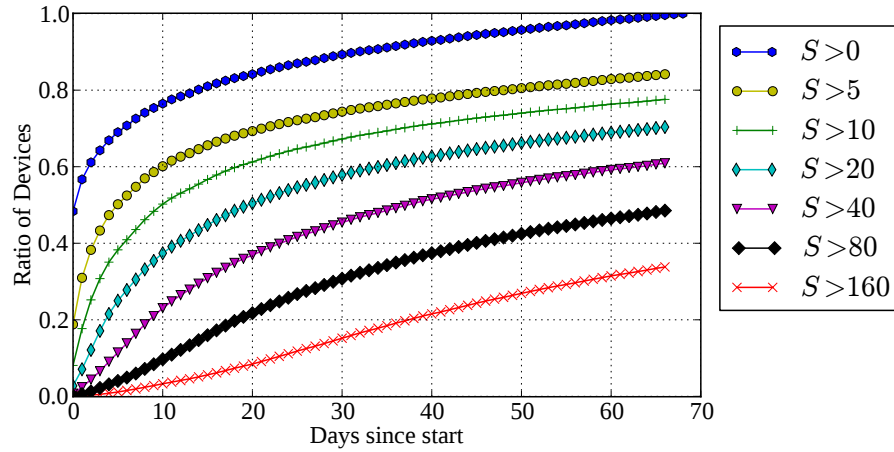


Figure 9.7.: Cumulative ratio of classified devices

9.6. Summary

In this chapter we introduced MTRAC, a novel approach for online M2M classification based on coarse-grained measurements. We started by explaining which informative features can be extracted from high-level session data. Next, we applied multiple different machine learning algorithms to train classification models and compared their performance. We presented two different approaches to aggregate sessions per device. Whereas the SDA approach can be applied early for nearly all devices, the classification performance of the TBA approach is considerably better. However, the TBA approach might take a long time until enough sessions for a device are available. We leave it as a choice to the user of MTRAC to decide whether to favor a low FPR or the ability to

classify a device as fast as possible.

In the future, we plan to extend the MTRAC approach to work with longer periods of data. As it already appears, in the last three weeks of Fig. 9.5b the FPR gradually increases. Therefore, we plan to apply advanced online learning methods to cope with this concept drift. In addition, we want to study the increased FPR over weekends in more detail. During weekends, the FPR is up to approx. 10% worse than on weekdays. Last but not least, we are planning to extend our approach from using session based to flow based data. Since the number of flows is a lot higher than the number of sessions, more information is available per device which might improve the performance of the overall approach.

10. Conclusion and Future Research Directions

In this thesis, we presented the evolution of DBStream, a Data Stream Warehouse (DSW) tailored for, but not limited to, Network Traffic Monitoring and Analysis (NTMA) applications. We have shown, that if instrumented correctly, a PostgreSQL database engine can process big amounts of data in a fast and efficient way. Furthermore, we demonstrated the practicability of the early TicketDB system by a botnet detection approach for mobile networks. This approach lead to the detection of a previously unknown botnet, which was afterwards confirmed by other scientists [74]. From this application, as well as others, we inferred useful insights on how to improve such a system to offer increased performance and higher flexibility at the same time. In a performance study, we demonstrated that the DBStream system can, installed only on a single node, outperform a cluster of 10 Spark nodes by a factor of 2.6, running the same query workload on the same dataset. The flexibility of DBStream was presented in another application, where it was instrumented to run multiple complex machine learning tasks. The resulting MTRAC approach, based only on the analysis of coarse-grained network descriptors, shows a very high accuracy for the continuous classification of M2M devices in a 3G mobile network. Although those results indicate that the DBStream is already a very good system for network monitoring, many technical challenges and interesting research questions are still waiting to be solved.

In the future, we plan to apply DBStream to other application domains with similar properties, such as intelligent transportation systems, smart grid and smart city data. Those application domains have similar data properties, such as big data streams of append only data, and similar processing and analysis requirements. Therefore, DBStream can be used to store and analyze data from those domains as successfully as data from mobile networks.

In addition, we want to investigate if it is possible to extend the DBStream approach by replacing the database engine PostgreSQL with either the parallel database system Greenplum [110], or a MapReduce based large-scale data processing framework like,

10. Conclusion and Future Research Directions

e.g. Spark [146]. This is a logical extension of the current single machine DBStream architecture to a cluster system, thus enabling scale-out properties found in modern big data processing frameworks.

Currently the most promising future research direction is the extension of the cache-oblivious scheduling approach to the parallel execution of recurring workloads. This can be achieved, by integrating the precedence DAGs of multiple subsequent time batches into a single precedence DAG. Then, in order to extend our approach to parallel execution, we may use the WTMB approach, but allow multiple jobs to be mapped to the same position in the schedule. All jobs in the same position would be executed in parallel. By the introduction of execution borders we make sure jobs accessing the same input data do not drift too much apart. This is possible without a negative affect on the makespan of the overall schedule. We are planning to start working on this very soon and also to evaluate if cache-oblivious scheduling is applicable to modern MapReduce cluster processing systems.

10.1. Statements of the Thesis

1. *"Any big amount of data can be processed as a stream."*
2. *"In the beginning there are only few tasks, therefore you will try to invest as much resources as possible into a single task. After some time the question is how to spread the limited resources over multiple tasks."*
3. *"Stream processing does not have to be done in real-time."*
4. *"A problem can be called a big data problem, if in addition to functional also non-functional issues have to be solved."*
5. *"Do we want to live in a world where the government knows everything about us?"*
6. *"It is in our hands to use big data for the good or the bad of mankind."*
7. *"Sometimes research is not about doing something which works, but doing something which looks good on paper."*
8. *"It always seems impossible until it's done."*
Nelson Mandela
9. *"If we knew what it was we were doing, it would not be called research, would it?"*
Albert Einstein
10. *"Peace is war, love is hate and freedom is slavery."*
George Orwell

10.2. Vision

I personally would like to end this thesis with some of my thoughts about the future of computer science.

In the past 20 years, from about 1980 to 2000 the major challenge in computer science was to acquire data. Whenever any program should analyze data, those data typically had to first be manually entered into computers by humans. Therefore, the reach of computer system was rather limited. With the wide adoption of networked computer systems and the Internet also more and more data became available.

In the current period, which goes roughly from the year 2000 to the year 2020, the availability of data became and is still becoming the default case. To mention just a few examples, Wikipedia gave structure to human knowledge and social media involved everyone into the generation of new computerized data. The main problem nowadays is to make the most sense out of the available data. Some applications are already quite evolved, like e.g. the IBM Watson¹ or the self driving cars made by Google². Another very interesting approach is called DeepMind³ and is able to learn how to successfully play Atari games only from a visual input and the current score of the game. Still the major problem is that although those applications are astonishing, many parts are relying on scripted human made procedures. Another big problem is the integration of different types of data. Many solutions focus only on one specific problem and can solve that quite well, but they are not able to integrate with other solutions or other types of data.

In my opinion, the next period, from about 2020 to 2040 will be all about data integration, creating an virtual environment which is sophisticated enough to house the first real artificially intelligent beings. As of today, there is already working speech recognition, Watson can play Jeopardy better than humans, cars can drive automatically, humanoid robots can walk, jump, run and play soccer, Wikipedia stores a large part of human knowledge, news (of course also wrong ones) travel through Twitter faster than they are even shown on TV, automatic stock market trading and countless other amazing systems exist. Therefore, I think that this period will be all about systems which not only predict the future anymore, but actively change the future, our all future.

It is the task of all of us to ensure this change is for the best of all humankind.

¹<http://www.ibm.com/smarterplanet/us/en/ibmwatson/>

²<https://plus.google.com/+GoogleSelfDrivingCars/>

³<http://deepmind.com/>

Bibliography

- [1] Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., Hwang, J.-h., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. *IN CIDR*, pages 277—289.
- [2] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2003). Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139.
- [3] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D. J., Rasin, A., and Silberschatz, A. (2009). Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933.
- [4] Achtert, E., Goldhofer, S., Kriegel, H.-P., Schubert, E., and Zimek, A. (2012). Evaluation of clusterings – metrics and visual support. *Data Engineering, International Conference on*, 0:1285–1288.
- [5] Agrawal, K. and Fineman, J. T. (2014). Brief announcement: cache-oblivious scheduling of streaming pipelines. In *SPAA*, pages 79–81.
- [6] Agrawal, P., Kifer, D., and Olston, C. (2008). Scheduling shared scans of large data files. In *PVLDB 1(1)*, pages 958–969.
- [7] Ahmad, M., Aboulmaga, A., Babu, S., and Munagala, K. (2011). Interaction-aware scheduling of report-generation workloads. *VLDB J.*, 20(4):589–615.
- [8] Ahmed, R. and Dharaskar, R. V. (2012). Study of mobile botnets: An analysis from the perspective of efficient generalized forensics framework for mobile devices. *IJCA Proceedings on National Conference on Innovative Paradigms in Engineering and Technology (NCIPET 2012)*, ncipet(15).
- [9] Allahverdi, A., Ng, C., Cheng, T., and Kovalyov, M. (2008). A survey of scheduling problems with setup times or costs. In *European Journal of Operational Research* 187(3), pages 985–1032.

Bibliography

- [10] Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142.
- [11] Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear road: A stream data management benchmark. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 480–491.
- [12] Arumugam, S., Dobra, A., Jermaine, C., Pansare, N., and Perez, L. (2010). The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530.
- [13] Balazinska, M., Kwon, Y., Kuchta, N., and Lee, D. (2007). Moirae: History-enhanced monitoring. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 375–386.
- [14] Bannister, J., Mather, P., and Coope, S. (2004). *Convergence technologies for 3G networks: IP, UMTS, EGPRS and ATM*. John Wiley & Sons.
- [15] Bär, A., Barbuzzi, A., Michiardi, P., and Ricciato, F. (2011). Two parallel approaches to network data analysis. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS) 2011*, Seattle, USA.
- [16] Bär, A., Casas, P., Golab, L., and Finamore, A. (2014a). DBStream: An online aggregation, filtering and processing system for network traffic monitoring. In *International Wireless Communications and Mobile Computing Conference, IWCMC 2014, Nicosia, Cyprus, August 4-8, 2014*, pages 611–616.
- [17] Bär, A., Finamore, A., Bermudez, I., Golab, L., Mellia, M., and Casas, P. (2013). Technical Report: Continuous Analytics for Traffic Monitoring and Applications to CDN. Technical Report FTW-TECHREPORT-129, FTW.
- [18] Bär, A., Finamore, A., Casas, P., Golab, L., and Mellia, M. (2014b). Large-scale network traffic monitoring with DBStream, a system for rolling big data analysis. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, pages 165–170.
- [19] Bär, A., Golab, L., Ruehrup, S., Schiavone, M., and Casas, P. (2015). Cache-oblivious scheduling of shared workloads. In *IEEE 31th International Conference on Data Engineering ICDE 2015, April 13-17 2015, Seoul, South Korea, to appear*.

- [20] Bär, A., Paciello, A., and Romirer-Maierhofer, P. (2013). Trapping botnets by DNS failure graphs: Validation, extension and application to a 3g network. In *2013 Proceedings IEEE INFOCOM Workshops, Turin, Italy, April 14-19, 2013*, pages 393–398.
- [21] Bär, A., Svoboda, P., and Casas, P. (2015). MTRAC - Discovering M2M Devices in Cellular Networks from Coarse-grained Measurements. In *Proceedings of the IEEE International Conference on Communications 2015 (ICC) , London, UK*.
- [22] Berkeley AMPLab (2014). Big data benchmark. <https://amplab.cs.berkeley.edu/benchmark/>.
- [23] Bermudez, I., Mellia, M., Munafò, M. M., Keralapura, R., and Nucci, A. (2012). DNS to the rescue: discerning content and services in a tangled web. In *Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement, IMC '12, Boston, MA, USA, November 14-16, 2012*, pages 413–426.
- [24] Beyer, K. S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M. Y., Kanne, C., Özcan, F., and Shekita, E. J. (2011). Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283.
- [25] Borders, K., Springer, J., and Burnside, M. (2012). Chimera: A declarative language for streaming network traffic analysis. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 365–379.
- [26] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. S. (2011). Apache hadoop goes realtime at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1071–1080.
- [27] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2002). Monitoring streams - A new class of data management applications. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 215–226.
- [28] Casas, P., D’Alconzo, A., Fiadino, P., Bär, A., and Finamore, A. (2014a). On the analysis of qoe-based performance degradation in youtube traffic. In *10th International*

Bibliography

- Conference on Network and Service Management, CNSM 2014 and Workshop, Rio de Janeiro, Brazil, November 17-21, 2014*, pages 1–9.
- [29] Casas, P., D’Alconzo, A., Fiadino, P., Bär, A., Finamore, A., and Zseby, T. (2014b). When youtube does not work - analysis of qoe-relevant degradation in google CDN traffic. *IEEE Transactions on Network and Service Management*, 11(4):441–457.
- [30] Casas, P., Fiadino, P., and Bär, A. (2013). IP mining: Extracting knowledge from the dynamics of the internet addressing space. In *25th International Teletraffic Congress, ITC 2013, Shanghai, China, September 10-12, 2013*, pages 1–9.
- [31] Casas, P., Fiadino, P., and Bär, A. (2014c). Understanding HTTP traffic and CDN behavior from the eyes of a mobile ISP. In *Passive and Active Measurement - 15th International Conference, PAM 2014, Los Angeles, CA, USA, March 10-11, 2014, Proceedings*, pages 268–271.
- [32] Chaudhuri, S. and Dayal, U. (1997). An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74. ACM ID: 248616.
- [33] Chekuri, C. and Motwani, R. (1999). Precedence constrained scheduling to minimize weighted completion time on a single machine. In *Discrete Applied Mathematics 98(1-2)*, pages 29–38.
- [34] Chen, Q. and Hsu, M. (2010). Experience in extending query engine for continuous analytics. In *Data Warehousing and Knowledge Discovery, 12th International Conference, DAWAK 2010, Bilbao, Spain, August/September 2010. Proceedings*, pages 190–202.
- [35] Cho, K., Mitsuya, K., and Kato, A. (2000). Traffic data repository at the WIDE project. In *Proceedings of the Freenix Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 263–270.
- [36] Choi, H. and Lee, H. (2012). Identifying botnets by capturing group activities in DNS traffic. *Computer Networks*, 56(1):20–33.
- [37] Claise, B., Sadasivan, G., Valluri, V., and Djernaes, M. (2004). RFC 3954: Cisco systems NetFlow services export version 9. <http://www.ietf.org/rfc/rfc3954.txt>.
- [38] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

- [39] Coluccia, A., D’Alconzo, A., and Ricciato, F. (2013). Distribution-based anomaly detection via generalized likelihood ratio test: A general maximum entropy approach. *Computer Networks*, 57(17):3446–3462.
- [40] Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003). Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 647–651.
- [41] Crescenzi, P. and Kann, V. (1998). A compendium of np optimization problems. <ftp://ftp.nada.kth.se/Theory/Viggo-Kann/compendium.pdf>.
- [42] Dainotti, A., Pescapè, A., and Claffy, K. C. (2012). Issues and future directions in traffic classification. *IEEE Network*, 26(1):35–40.
- [43] D’Alconzo, A., Coluccia, A., and Romirer-Maierhofer, P. (2010). Distribution-based anomaly detection in 3g mobile networks: from theory to practice. *Int. Journal of Network Management*, 20(5):245–269.
- [44] Dean, J. and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI*.
- [45] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.
- [46] Dindar, N., Güç, B., Lau, P., Özal, A., Soner, M., and Tatbul, N. (2009). Dejavu: declarative pattern matching over live and archived streams of events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1023–1026.
- [47] Dindar, N., Tatbul, N., Miller, R. J., Haas, L. M., and Botan, I. (2013). Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.*, 22(4):421–446.
- [48] Douglas, C. and Tang, H. (2010). Gridmix3 – emulating production workload for apache hadoop. <https://developer.yahoo.com/blogs/hadoop/gridmix3-emulating-production-workload-apache-hadoop-450.html>.
- [49] Emulex (2015). Endace dag cards - 100% packet capture guaranteed - high speed packet capture, any network interface.

Bibliography

- [50] Erman, J., Gerber, A., Hajiaghayi, M. T., Pei, D., and Spatscheck, O. (2009). Network-aware forward caching. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 291–300.
- [51] Erman, J., Gerber, A., and Sen, S. (2011). HTTP in the home: it is not just about pcs. *Computer Communication Review*, 41(1):90–95.
- [52] EsperTech Inc. (2015). Esper: Event processing for java. <http://www.espertech.com/products/esper.php>.
- [53] Ester, M., Kriegel, H., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231.
- [54] Fiadino, P., Bär, A., and Casas, P. (2013). HTTPTag: A flexible on-line http classification system for operational 3g networks. In *Proceedings of IEEE Infocom 2013*, Turin, Italy.
- [55] Fiadino, P., D’Alconzo, A., Bär, A., Finamore, A., and Casas, P. (2014). On the detection of network traffic anomalies in content delivery network services. In *2014 26th International Teletraffic Congress (ITC), Karlskrona, Sweden, September 9-11, 2014*, pages 1–9.
- [56] Fiadino, P., Schiavone, M., and Casas, P. (2015). Vivisecting WhatsApp in Cellular Networks: Servers, Flows, and Quality of Experience. In Steiner, M., Barlet-Ros, P., and Bonaventure, O., editors, *Traffic Monitoring and Analysis (TMA), Barcelona, Spain*. LNCS.
- [57] Finamore, A., Mellia, M., Meo, M., Munafò, M. M., Torino, P. D., and Rossi, D. (2011). Experiences of internet traffic monitoring with Tstat. *IEEE Network*, 25(3):8–14.
- [58] Folkert, N., Gupta, A., Witkowski, A., Subramanian, S., Bellamkonda, S., Shankar, S., Bozkaya, T., and Sheng, L. (2005). Optimizing refresh of a set of materialized views. In *VLDB*, pages 1043–1054.
- [59] Frigo, M., Leiserson, C., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *FOCS*, pages 285–297.

- [60] Fusco, F. and Deri, L. (2010). High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement 2010, Melbourne, Australia - November 1-3, 2010*, pages 218–224.
- [61] Garcia-Molina, H., Ullman, J. D., and Widom, J. (2009). *Database systems - the complete book*. Prentice Hall, 2 edition.
- [62] Garey, M. and Johnson, D. (1979). Computers and intractability: A guide to the theory of np-completeness.
- [63] Garey, M., Johnson, D., and Stockmeyer, L. (1976). Some simplified np-complete graph problems. In *Theoretical Computer Science 1(3)*, pages 237–267.
- [64] Giannikis, G., Alonso, D. M. G., and Kossmann, D. (2014). Shared workload optimization. In *PVLDB 7(6)*, pages 429–440.
- [65] Giannikis, G., Alonso, G., and Kossmann, D. (2012). Shareddb: Killing one thousand queries with one stone. In *PVLDB 5(6)*, pages 526–537.
- [66] Golab, L., Johnson, T., Seidel, J. S., and Shkapenyuk, V. (2009). Stream warehousing with datadepot. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, SIGMOD '09*, pages 847–854, New York, NY, USA. ACM.
- [67] Golab, L., Johnson, T., Sen, S., and Yates, J. (2012a). A sequence-oriented stream warehouse paradigm for network monitoring applications. In *Passive and Active Measurement - 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012. Proceedings*, pages 53–63.
- [68] Golab, L., Johnson, T., and Shkapenyuk, V. (2012b). Scalable scheduling of updates in streaming data warehouses. In *TKDE 24(6)*, pages 1092–1105.
- [69] Golab, L. and zsu, M. T. (2003). Issues in data stream management. *SIGMOD Rec.*, 32(2):514.
- [70] Grimaudo, L., Mellia, M., Baralis, E., and Keralapura, R. (2014). Select: Self-learning classifier for internet traffic. *IEEE Transactions on Network and Service Management*, 11(2):144–157.
- [71] Gupta, A., Sudarshan, S., and Viswanathan, S. (2001). Query scheduling in multi query optimization. In *IDEAS*, pages 11–19.

Bibliography

- [72] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.
- [73] Härder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.
- [74] Huang, D. Y., Dharmdasani, H., Meiklejohn, S., Dave, V., Grier, C., McCoy, D., Savage, S., Weaver, N., Snoeren, A. C., and Levchenko, K. (2014). Bitcoin: Monetizing stolen cycles. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*.
- [75] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010). The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 41–51.
- [76] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010). Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, pages 159–168.
- [77] Ji, Y. (2013). Database support for processing complex aggregate queries over data streams. In *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 31–37.
- [78] Jia, Y. and Shao, Z. (2009). A benchmark for hive, pig and hadoop. <https://issues.apache.org/jira/browse/hive-396>.
- [79] Jiang, N., Cao, J., Jin, Y., Li, L., and Zhang, Z.-L. (2010). Identifying suspicious activities through DNS failure graph analysis. In *Network Protocols (ICNP), 2010 18th IEEE International Conference on*, pages 144 –153.
- [80] Jin, Y., Sharafuddin, E., and Zhang, Z.-L. (2009). Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *SIGMETRICS/Performance*, pages 49–60.
- [81] Keys, K., Moore, D., Koga, R., Lagache, E., Tesch, M., and claffy, k. (2001). The architecture of CoralReef: an Internet traffic monitoring software suite. In *Passive*

and *Active Network Measurement Workshop (PAM)*, Amsterdam, Netherlands. RIPE NCC.

- [82] Krishnamurthy, S., Franklin, M. J., Davis, J., Farina, D., Golovko, P., Li, A., and Thombre, N. (2010a). Continuous analytics over discontinuous streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1081–1092.
- [83] Krishnamurthy, S., Franklin, M. J., Davis, J., Farina, D., Golovko, P., Li, A., and Thombre, N. (2010b). Continuous analytics over discontinuous streams. In *SIGMOD Conference*, pages 1081–1092.
- [84] Labio, W., Yerneni, R., and Garcia-Molina, H. (1999). Shrinking the warehouse update window. In *SIGMOD*, pages 383–394.
- [85] Lam, W., Liu, L., Prasad, S., Rajaraman, A., Vacheri, Z., and Doan, A. (2012). Muppet: Mapreduce-style processing of fast data. *PVLDB*, 5(12):1814–1825.
- [86] Laner, M., Svoboda, P., Nikaein, N., and Rupp, M. (2013). Traffic models for machine type communications. In *ISWCS 2013, The Tenth International Symposium on Wireless Communication Systems, Ilmenau, TU Ilmenau, Germany, August 27-30, 2013*, pages 1–5.
- [87] Laner, M., Svoboda, P., Romirer-Maierhofer, P., Nikaein, N., Ricciato, F., and Rupp, M. (2012a). A comparison between one-way delays in operating HSPA and LTE networks. In *10th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), Paderborn, Germany, May 14-18, 2012*, pages 286–292.
- [88] Laner, M., Svoboda, P., and Rupp, M. (2014). Detecting M2M traffic in mobile cellular networks. In *Systems, Signals and Image Processing (IWSSIP), 2014 International Conference on*, pages 159–162.
- [89] Laner, M., Svoboda, P., Schwarz, S., and Rupp, M. (2012b). Users in cells: A data traffic analysis. In *2012 IEEE Wireless Communications and Networking Conference, WCNC 2012, Paris, France, April 1-4, 2012*, pages 3063–3068.
- [90] Lehner, W., Cochrane, R., Pirahesh, H., and Zaharioudakis, M. (2001). Fast Refresh using Mass Query Optimization. In *ICDE*, pages 391–398.

Bibliography

- [91] Li, B., Mazur, E., Diao, Y., McGregor, A., and Shenoy, P. J. (2012). SCALLA: A platform for scalable one-pass analytics using mapreduce. *ACM Trans. Database Syst.*, 37(4):27.
- [92] Liarou, E., Idreos, S., Manegold, S., and Kersten, M. L. (2012). Monetdb/datacell: Online analytics in a streaming column-store. *PVLDB*, 5(12):1910–1913.
- [93] Livingtree, L., Barrish, S., Raab, F., Askins, B., Johnson, D., Stephens, J., Majdalany, M., Hotea, A., and Rabl, T. (2015). Transaction processing performance council. <http://www.tpc.org/>.
- [94] Luo, G., Naughton, J. F., Ellmann, C. J., and Watzke, M. (2010). Transaction reordering. In *DKE 69(1)*, pages 29–49.
- [95] Mahimkar, A., Ge, Z., Wang, J., Yates, J., Zhang, Y., Emmons, J., Huntley, B., and Stockert, M. (2011). Rapid detection of maintenance induced changes in service performance. In *Proceedings of the Seventh Conference on emerging Networking EXperiments and Technologies*, CoNEXT '11, page 13:113:12, New York, NY, USA. ACM.
- [96] Maier, G., Feldmann, A., Paxson, V., and Allman, M. (2009). On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement 2009, Chicago, Illinois, USA, November 4-6, 2009*, pages 90–102.
- [97] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon (2012). From throw-away traffic to bots: Detecting the rise of DGA-Based malware. In *USENIX Security Symposium*.
- [98] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. (2010). Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339.
- [99] Michael, M. M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. (2007). Scale-up x scale-out: A case study using nutch/lucene. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8.
- [100] Mistry, H., Roy, P., Sudarshan, S., and Ramamritham, K. (2001). Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318.

- [101] mPlane Consortium (project FP7-ICT-318627) (2015). mPlane - an Intelligent Measurement Plane for Future Network and Application Management. <http://www.ict-mplane.eu/>.
- [102] Nguyen, T. T. T. and Armitage, G. J. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76.
- [103] Nykiel, T., Potamias, M., Mishra, C., Kollios, G., and Koudas, N. (2014). Sharing across multiple mapreduce jobs. *ACM Trans. Database Syst.*, 39(2):12.
- [104] Olston, C., Chiou, G., Chitnis, L., Liu, F., Han, Y., Larsson, M., Neumann, A., Rao, V. B. N., Sankarasubramanian, V., Seth, S., Tian, C., ZiCornell, T., and Wang, X. (2011). Nova: continuous pig/hadoop workflows. In *SIGMOD Conference*, pages 1081–1090.
- [105] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110.
- [106] Özsu, M. T. and Valduriez, P. (1996). Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, 28(1):125–128.
- [107] Özsu, M. T. and Valduriez, P. (2011). *Principles of distributed database systems*. Springer Science & Business Media.
- [108] P. Mockapetris (1987). DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. <http://www.ietf.org/rfc/rfc1035.txt>.
- [109] Papadimitriou, C. (1976). The np-completeness of the bandwidth minimization problem. In *Computing*, 16(3), pages 263–270.
- [110] Pivotal Software, Inc. (2015). Pivotal greenplum database - enable analytic innovation.
- [111] Psaroudakis, I., Athanassoulis, M., and Ailamaki, A. (2013). Sharing data and work across concurrent analytical queries. In *PVLDB 6(9)*, pages 637–648.
- [112] Rekhter, Y., Groot, G. J. d., Karrenberg, D., and Lear, E. (1996). RFC1918: address allocation for private internets. <http://tools.ietf.org/html/rfc1918>.

- [113] Ricciato, F. (2006). Traffic monitoring and analysis for the optimization of a 3g network. *IEEE Wireless Commun.*, 13(6):42–49.
- [114] Ricciato, F., Hasenleithner, E., and Romirer-Maierhofer, P. (2008). Traffic analysis at short time-scales: an empirical case study from a 3g cellular network. *IEEE Transactions on Network and Service Management*, 5(1):11–21.
- [115] Ricciato, F., Svoboda, P., Motz, J., Fleischer, W., Sedlak, M., Karner, M., Pilz, R., Romirer-Maierhofer, P., Hasenleithner, E., and Jäger, W. (2006). Traffic monitoring and analysis in 3g networks: lessons learned from the METAWIN project. *Elektrotechnik und Informationstechnik*, 123(7-8):288–296.
- [116] Rice, D., Glick, J., McKay, C., Sandifer, C., and Cramblitt, B. (2015). Standard performance evaluation corporation. <http://spec.org/>.
- [117] Roesch, M. (2015). Snort, network intrusion detection and prevention.
- [118] Romirer-Maierhofer, P. (2012). *Online Anomaly Detection in Operational 3G Cellular Networks*. PhD thesis, Universität Wien, Vienna, Austria.
- [119] Romirer-Maierhofer, P. and Ricciato, F. (2008). Towards anomaly detection in one-way delay measurements for 3g mobile networks: A preliminary study. In *IP Operations and Management, 8th IEEE International Workshop, IPOM 2008, Samos Island, Greece, September 22-26, 2008, Proceedings*, pages 1–14.
- [120] Romirer-Maierhofer, P., Ricciato, F., and Coluccia, A. (2008). Explorative analysis of one-way delays in a mobile 3g network. In *Local and Metropolitan Area Networks, 2008. LANMAN 2008. 16th IEEE Workshop on*, pages 73–78.
- [121] Romirer-Maierhofer, P., Ricciato, F., D’Alconzo, A., Franzan, R., and Karner, W. (2009). Network-wide measurements of TCP RTT in 3g. In *Traffic Monitoring and Analysis, First International Workshop, TMA 2009, Aachen, Germany, May 11, 2009. Proceedings*, pages 17–25.
- [122] Romirer-Maierhofer, P., Schiavone, M., and D’Alconzo, A. (2015). Device-specific Traffic Characterization for Root Cause Analysis in Cellular Networks. In Steiner, M., Barlet-Ros, P., and Bonaventure, O., editors, *Traffic Monitoring and Analysis (TMA), Barcelona, Spain*, page to appear. LNCS.
- [123] Schikuta, E. (1996). Grid-clustering: an efficient hierarchical clustering method for very large data sets. In *13th International Conference on Pattern Recognition, ICPR 1996, Vienna, Austria, 25-19 August, 1996*, pages 101–105.

- [124] Schikuta, E. and Erhart, M. (1997). The bang-clustering system: Grid-based data analysis. In *Advances in Intelligent Data Analysis, Reasoning about Data, Second International Symposium, IDA-97, London, UK, August 4-6, 1997, Proceedings*, pages 513–524.
- [125] Shafiq, M. Z., Ji, L., Liu, A. X., Pang, J., and Wang, J. (2012). A first look at cellular machine-to-machine traffic: large scale measurement and characterization. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 65–76.
- [126] Sommer, J. L. E. and Strait, M. (2009). L7-Filter, application layer packet classifier for linux <http://l7-filter.sourceforge.net/>.
- [127] Stewart, R. J., Trinder, P. W., and Loidl, H. (2011). Comparing high level mapreduce query languages. In *Advanced Parallel Processing Technologies - 9th International Symposium, APPT 2011, Shanghai, China, September 26-27, 2011. Proceedings*, pages 58–72.
- [128] Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. (2009). Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security, CCS '09*, pages 635–647, New York, NY, USA. ACM.
- [129] Stonebraker, M. (2010). Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11.
- [130] StreamBase Inc. (2014). Streambase: Real-time, low latency data processing with a stream processing engine. <http://www.streambase.com>.
- [131] Systems, C. (2014). Cisco visual networking index: Global mobile data traffic forecast update, 2013-2018.
- [132] The Telecommunications Research Center Vienna (FTW) (2013). Data Analysis and Reporting for Wireless Networks (DARWIN3). http://www.ftw.at/research-innovation/projects/darwin_3?set_language=en.
- [133] The Telecommunications Research Center Vienna (FTW) (2014). Data Analysis and Reporting for Wireless Networks (DARWIN4). http://www.ftw.at/research-innovation/projects/darwin-4?set_language=en.

Bibliography

- [134] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629.
- [135] Trammell, B., Casas, P., Rossi, D., Bär, A., Ben-Houidi, Z., Leontiadis, I., Szemethy, T., and Mellia, M. (2014). mplane: an intelligent measurement plane for the internet. *IEEE Communications Magazine*, 52(5):148–156.
- [136] Transaction Processing Performance Council (TPC) (2015). TPC Benchmark DS (TPC-DS): The New Decision Support Benchmark Standard. <http://www.tpc.org/tpcds/default.asp>.
- [137] Tsaih, D., Wu, G., Chang, C., Hung, S., Wu, C., , and Lin, H. (2008). An efficient a* algorithm for the directed linear arrangement problem. In *WSEAS Transactions on Computers*, 7(12), pages 1958–1967.
- [138] Valenti, S., Rossi, D., Dainotti, A., Pescapè, A., Finamore, A., and Mellia, M. (2013). Reviewing traffic classification. In *Data Traffic Monitoring and Analysis*, pages 123–147. Springer Berlin Heidelberg.
- [139] Vigne, R., Mangler, J., Schikuta, E., and Rinderle-Ma, S. (2012). A structured marketplace for arbitrary services. *Journal on Future Generation Computer Systems*, 28(1):48–57.
- [140] White, T. (2012). *Hadoop: the definitive guide*. O’Reilly.
- [141] Williams, N., Zander, S., and Armitage, G. (2006). A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *SIGCOMM Comput. Commun. Rev.*, 36(5):516.
- [142] Wolf, J. L., Balmin, A., Rajan, D., Hildrum, K., Khandekar, R., Parekh, S., Wu, K., and Vernica, R. (2012). On the optimization of schedules for mapreduce workloads in the presence of shared scans. *VLDB J.*, 21(5):589–609.
- [143] Wu, G., Talwar, S., Johnsson, K., Himayat, N., and Johnson, N. D. (2011). M2M: from mobile to embedded internet. *IEEE Communications Magazine*, 49(4):36–43.
- [144] Xie, Y., Yu, F., Achan, K., Gillum, E., Goldszmidt, M., and Wobber, T. (2007). How dynamic are IP addresses? In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM ’07, page 301312, New York, NY, USA. ACM.

- [145] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. (2013). Shark: SQL and rich analytics at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 13–24.
- [146] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*.
- [147] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’12, Boston, MA, USA, June 12-13, 2012*.
- [148] Zukowski, M., Héman, S., Nes, N., and Boncz, P. A. (2007). Cooperative scans: Dynamic bandwidth sharing in a dbms. In *VLDB*, pages 723–734.

Curriculum Vitae

Personal

Name: Arian Bär

Address: FTW, Donau-City-Straße 1, 1220 Vienna, Austria

Contact: arian.baer [at] gmail.com

Education

Diplom Informatiker Universität (Dipl.-Inf. Univ.) Major fields of study: Artificial Intelligence, Computer Graphics, Theoretical Computer Science. Diploma thesis (German equivalent of a master thesis) title: "Similarity Search in Large-Scale Text Databases", Friedrich-Alexander Universität Erlangen-Nürnberg, 2009.

Involvement in Scientific Projects

Forschungszentrum Telekommunikation Wien (FTW)

Data Analysis and Reporting in Wireless Networks (DARWIN), Project series: DARWIN3 and DARWIN4 (2010 - to end of 2014).

mPlane – an Intelligent Measurement Plane for Future Network and Application Management (2013 - until now).

List of Publications

The following list of publications are all peer reviewed from at least 3 independent reviewers.

Relevant First Author Publications

- Bär, Arian; Svoboda, Philipp; Casas, Pedro. "MTRAC - Discovering M2M Devices in Cellular Networks from Coarse-grained Measurements." In *Proceedings of the IEEE International Conference on Communications 2015 (ICC)*, London,

UK, 2015, to appear. Acceptance rate: 38%.

- Bär, Arian; Golab, Lukasz; Ruehrup, Stefan; Schiavone, Mirko; Casas, Pedro. "Cache-Oblivious Scheduling of Shared Workloads." In *Proceedings of the 31th IEEE International Conference on Data Engineering (ICDE)*, Seoul, South Korea, 2015, to appear. Acceptance rate: not yet known (20% for ICDE 2014).
- Bär, Arian; Finamore, Alessandro; Casas, Pedro; Golab, Lukasz; Melia, Marco. "Large-Scale Network Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis." In *Proceedings of IEEE International Conference on Big Data 2014 (IEEE BigData 2014)*, Washington DC, USA 2014. Acceptance rate: 33.3%.
- Bär, Arian; Casas, Pedro; Golab, Lukasz; Finamore, Alessandro. "DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring" In *5th International Workshop on TRaffic Analysis and Characterization (TRAC) in conjunction with International Wireless Communications and Mobile Computing Conference (IWCMC 2014)* Nicosia, Cyprus 2014. Acceptance rate: 65%.
- Bär, Arian; Paciello, Antonio; Romirer-Maierhofer, Peter, "Trapping botnets by DNS failure graphs: Validation, extension and application to a 3G network." In *Proceedings of the IEEE INFOCOM Workshops 2013* Turin, Italy 2013. Acceptance rate: 29.2%.
- Bär, Arian and Golab, Lukasz. "Towards benchmarking stream data warehouses." In *Proceedings of the fifteenth international workshop on Data warehousing and OLAP (DOLAP '12)*. ACM, New York, NY, USA 2012. Acceptance rate: 54%.
- Bär, Arian, Antonio Barbuzzi, Pietro Michiardi, and Fabio Ricciato. "Two parallel approaches to network data analysis." In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS) in conjunction with VLDB 2011*,

Seattle Washington, USA 2011. Acceptance rate: unknown.

Other Journal Publications

- Pedro Casas, Alessandro D'Alconzo, Pierdomenico Fiadino, Arian Bär, Alessandro Finamore, Tanja Zseby. "When YouTube doesn't Work: Analysis of QoE-relevant Degradation in Google CDN Traffic". IN *IEEE Transactions on Network and Service Management*, 2014.
- Brian Trammell, Pedro Casas, Dario Rossi, Arian Bär, Z. Ben-Houidi, I. Leontiadis, T. Szemethy, and M. Mellia. "mPlane: an Intelligent Measurement Plane for the Internet", *IEEE Communications Magazine, Special Issue on Monitoring and Troubleshooting Multi-domain Networks using Measurement Federations*, 2014.

Other Conference Publications

- Pedro Casas, Alessandro D'Alconzo, Pierdomenico Fiadino, Arian Bär, and Alessandro Finamore. "On the Analysis of QoE-based Performance Degradation in YouTube Traffic", *10th International Conference on Network and Service Management (CNSM)*, 2014.
- Pierdomenico Fiadino, Alessandro D'Alconzo, Arian Bär, Alessandro Finamore, and Pedro Casas. "On the Detection of Network Traffic Anomalies in Content Delivery Network Services.", *26th International Teletraffic Congress (ITC)*, 2014.
- Pedro Casas, Pierdomenico Fiadino, and Arian Bär. "YouTube All Around: Characterizing YouTube from Mobile and Fixed-line Network Vantage Points.", *European Conference on Networks and Communications (EuCNC)*, 2014.
- Pedro Casas, Pierdomenico Fiadino, and Arian Bär. "Understanding HTTP Traffic and CDN Behavior from the Eyes of a Mobile ISP", *Passive and Active Measurements Conference (PAM)*, 2014.
- Pedro Casas, Pierdomenico Fiadino, Arian Bär. "IP Mining: Extracting Knowledge from the Dynamics of the Internet Addressing Space", *25th International Teletraffic Congress (ITC)*, 2013.
Best Paper Award

Curriculum Vitae

- Pierdomenico Fiadino, Arian Bär, Pedro Casas. "HTTPTag: A Flexible On-line HTTP Classification System for Operational 3G Networks", *32nd IEEE International Conference on Computer Communications (Infocom)*, 2013.
- Arian Bär, A. Berger, S. Egger, R. Schatz. "A Lightweight Mobile TV Recommender", *Changing Television Environments, 6th European Conference (EuroITV)*, 2008.
- R. Schatz, Arian Bär, A. Berger. "Matching Content to the Mobile User", *Future Internet Symposium (FIS)*, 2008