Doctoral Thesis

Confidentiality and Performance for Cloud Databases

Author[s]:
Braun-Löhrer, Lucas Victor

Publication Date:
2017

Permanent Link:
https://doi.org/10.3929/ethz-a-010866596

Rights / License:
In Copyright - Non-Commercial Use Permitted
Confidentiality and Performance for Cloud Databases

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

LUCAS VICTOR BRAUN-LÖHRER

Master of Science ETH in Computer Science, ETH Zurich

born on June 18, 1987

citizen of Aadorf, Thurgau

accepted on the recommendation of

Prof. Dr. Donald A. Kossmann, examiner
Prof. Dr. Timothy Roscoe, co-examiner
Prof. Dr. Renée J. Miller, co-examiner
Prof. Dr. Thomas Neumann, co-examiner

2017
Typeset with \LaTeX

© 2017 by Lucas Victor Braun-Löhrer
Permission to print for personal and academic use, as well as permission for electronic reproduction and dissemination in unaltered and complete form are granted.
All other rights reserved.
Abstract

The twenty-first century is often called the century of the information society. The amount of collected data world-wide is growing exponentially and has easily reached the order of several million terabytes a day. As a result, everybody is talking about “big data” nowadays, not only in the research communities, but also very prominently in enterprises, politics and the press. Thanks to popular services, like Apple iCloud, Dropbox, Amazon Cloud Drive, Google Apps or Microsoft OneDrive, cloud storage has become a (nearly) ubiquitous and widely-used facility in which a huge portion of this big data is stored and processed. This makes cloud computing one of the most important, but also most exciting technologies in this present age.

In the course of the cloud’s adoption by industry, several cloud service models have developed. Starting with Infrastructure-as-a-Service (IaaS), the most basic model, the cloud service stack was extended to Platform-as-a-Service (PaaS) and finally Software-as-a-Service (SaaS). Most recently, a new class of PaaS has evolved: Database-as-a-Service (Daas), often also simply called cloud databases. Cloud databases provide developers with the ease of use of a well-known SQL (or NoSQL) interface which they can program their applications against. Moreover, they make the tempting promise that developers do not need to worry about the availability, performance or confidentiality of the database, let alone the operating system or hardware it runs on, because this is all ensured by the cloud provider.

Despite all these advantages, a lot of companies are still reluctant to move to the cloud. According to CloudTweaks, a famous cloud computing blog, the two major arguments for not adopting the cloud model are price and security concerns. Both of these arguments stem from the fact that today’s cloud databases do not fully keep the aforementioned promises. First, they do not optimize the performance of database queries and updates as much as they could, which means they use more resources than needed and hence
charge the customers too much. Second, the typical measures employed to address security are the use of SQL access control, secure connections and protocols like HTTPS and SSH. While this measures do satisfactorily protect the data against adversaries from outside, there is nothing that prevents the cloud provider herself to sneak into it. Even worse, in some countries, cloud providers can be legally forced by a court to disclose the data of their customers.

Hence, this dissertation studies the performance and confidentiality of cloud databases and proposes solutions that allow cloud providers to get closer to keeping their promises and to make potential customers more confident that moving to the cloud is the right thing to do. The first part of the dissertation argues why existing cloud databases are not properly optimized for modern workloads which typically comprise a demanding mix of transactions and real-time analytics. It studies the design space for scalable, elastic database systems capable to handle such workloads and argues why certain combinations of design decisions are more favorable than others. This part also presents two concrete system implementations and shows selected results that not only demonstrate their superiority over state-of-the-art competitors, but also their potential to enhance current cloud databases with regard to performance and ultimately costs.

The second part of this dissertation studies the usefulness of different encryption schemes for cloud databases. As existing schemes are either secure, but low-performance or the other way round, we propose a novel technique that can trade-off between confidentiality and performance and therefore achieves an excellent compromise between the two. In addition, we also show how to increase the value of a cloud database by processing data across tenants confidentially, which helps cloud providers to compensate for some of their costs and therefore offer better prices to their customers.
Zusammenfassung


Daher untersucht diese Dissertation die Leistung und die Datensicherheit von Cloud-Datenbanken und schlägt einige Lösungen vor, die es Cloud-Anbietern ermöglichen sollten, näher an das Einhalten ihrer Versprechen zu gelangen und so ihre Kunden zu überzeugen, dass die Cloud auch für sie die richtige Plattform ist. Der erste Teil der Abhandlung erklärt, wo es bei bestehenden Cloud-Anbietern noch Optimierungspotential gibt, insbesondere was das Verarbeiten von grossen, laufend hereinkommenden Datenmengen und das gleichzeitige Analysieren dieser Daten in Echtzeit betrifft. Es werden verschiedene Lösungsansätze für skalierbare, elastische Datenbanksysteme für solche Anforderungen diskutiert und es wird argumentiert, welche Kombinationen von Lösungsansätzen am sinnvollsten sind. In diesem Teil werden auch zwei konkrete Systeme vorgestellt und ausgewählte Resultate gezeigt, die nicht nur die Überlegenheit dieser Systeme gegenüber anderen Produkten auf dem neuesten Stand der Technik belegen, sondern auch aufzeigen, wie künftige Cloud-Datenbanken leistungsfähiger und somit kostengünstiger gemacht werden können.

Der zweite Teil der Abhandlung beschäftigt sich mit der Nützlichkeit verschiedener Verschlüsselungs-Algorithmen, welche für Cloud-Datenbanken in Frage kommen. Weil bestehende Algorithmen meist entweder sicher und leistungsschwach oder leistungsstark und unsicher sind, schlagen wir eine neuartige Technik vor, welche Leistung und Sicherheit gegeneinander abwägen kann und somit einen exzellenten Kompromiss zwi-
schen diesen beiden Eigenschaften erreicht. Ausserdem wird auch noch aufgezeigt, wie der Wert von Cloud-Datenbanken gesteigert werden kann, indem Daten von mehreren Kunden zusammen (in einer vertraulichen Art und Weise) analysiert werden. Dieser ge-
steigerte Wert hilft wiederum den Cloud-Anbietern, einen Teil ihrer Kosten zu amortisieren und damit ihren Kunden günstigere Preise anzubieten.
Acknowledgments

First of all, I would like to thank my advisor, Prof. Donald A. Kossmann. Although being physically absent for most of the time of my doctoral studies, he has always been 100% mentally present when I needed his advice, no matter at which time of the day. He is a real motivator and a great source of inspiration, even if some of his ideas seem totally crazy at first sight.

Furthermore, I would like to thank all the co-examiners of my examination committee, Prof. Timothy Roscoe, Prof. Renée J. Miller, and Prof. Thomas Neumann for reading my doctoral thesis and improving it by providing very helpful and valuable feedback. I would also like to thank Prof. Thomas Hofmann for chairing my doctoral examination.

Working at the Computer Science Department of ETH Zurich in general and in particular being a member of Systems Group does not only mean doing research in a very friendly and fruitful environment, but also getting many opportunities to get in touch with industry. I especially want to thank the people at Huawei ERC, Eliezer Levy, Anthony Iliopoulos, Ning Liang, and Aharon Avitzur. Working together with them on a real industrial use case and installing the software as a beta version on their infrastructure was a truly valuable and motivating experience. I also would like to thank Ramarathnam Venkatesan and Dennis Majoe from Microsoft Research for their support and input to our database encryption project. A big thanks goes to Ravinshankan Ramamurthy who supervised my internship at Microsoft Research Redmond and who, through many fruitful discussions, critical questions and a lot of optimism, motivated me to intensify the work on cross-tenant query processing.

I also had the opportunity to work together with people at the Chair of Database Systems at the Technical University of Munich in order to benchmark the AIM workload on a couple of different open-source and research systems. I would like to thank Andreas Kipf, Varun Pandey, Jan Böttcher, Prof. Alfons Kemper and especially Prof. Thomas Neumann who
also contributed to the original AIM paper by creating new HyPer binaries with additional SQL features even in the middle of the night.

I would like to thank my former and present colleges at Systems Group who encouraged and supported me throughout the course of my doctoral studies. Especially, I would like to thank Tahmineh Sanamrad with whom I had the pleasure to work on database encryption, Georgios Gasparis, Thomas Etter, Martin Kaufmann and Daniel Widmer who contributed greatly to the Analytics in Motion project, and Kevin Bocksrocker with whom I had the pleasure to work on the Tell project. It would also like to thank Renato Marroquín with whom I worked in several projects and who was so kind to read and give feedback on the early version of this thesis. Thank you, Kai-En Tsay, for writing the first version of the MT-H benchmark. A special and very grateful thank goes to Markus Pilman. Not only did I have the pleasure to work with him on the Tell project, but also to learn from him a great deal about programming, languages, tools and the history of computer science in general. Markus seems to be a living CS library, which may relate to the fact that he had actually worked in the computer science library of our department for some time. He is, apart from Donald, the person who most influenced and inspired my life and work at Systems Group.

Last, but not least, I would like to thank my family. My parents-in-law, Verena and Bruno Löhrer, and my parents, Bernadette and Vernerio Braun, supported me with their advice, patience and concrete support in taking care of the children whenever their father was busy with conferences or deadlines. My grand parents, Margrit and Kastor Locher, sponsored the final printing of this thesis. I would also like to thank my children, Alessandro and Philippa, for taking me as I am, even if they do not yet understand why one has to work from time to time and cannot play with them all day long. Finally, I want to thank from all my heart my wife, Mariélène, for her great love, but also for believing in me even in moments where I did not. From the first moment when she suggested considering doctoral studies until writing the final page of this dissertation, it was her continuous support and encouragement that let me fight it out.
# Contents

1 Introduction .......................................................... 1  
   1.1 Background and Motivation ................................... 1  
   1.2 Problem Statement ............................................. 2  
      1.2.1 Context and Design Goals ............................... 3  
      1.2.2 Optimization Metrics .................................. 4  
      1.2.3 Research Questions .................................... 4  
   1.3 Contributions .................................................. 5  
   1.4 Structure of this Thesis ..................................... 7  

I Performance .......................................................... 9  

2 Emerging Streaming Workloads and the Challenges they impose 11  
   2.1 Stream Processing and Analytics ............................ 12  
      2.1.1 Stateless Stream Processing ........................... 13  
      2.1.2 Stateful Stream Processing ............................ 13  
      2.1.3 Analytics on Stateful Streams ....................... 16  
      2.1.4 Transactions on Stateful Streams .................... 17  
   2.2 Stream Processing on Database Management Systems ....... 17  
      2.2.1 OLTP and OLAP workloads ............................. 18
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2 Data Warehousing</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3 Main-Memory OLTP systems</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4 HTAP systems</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Stream Processing on Key-Value Stores</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Key-Value Store API</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Availability, Elasticity and Transactional Support</td>
<td>21</td>
</tr>
<tr>
<td>2.3.3 Key-value stores on RDMA</td>
<td>22</td>
</tr>
<tr>
<td>2.3.4 Support for Streaming Workloads</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Concluding Remarks</td>
<td>24</td>
</tr>
<tr>
<td>3 Benchmarks for Analytics and Transactions on Stateful Streams</td>
<td>25</td>
</tr>
<tr>
<td>3.1 The Huawei-ALM Benchmark</td>
<td>26</td>
</tr>
<tr>
<td>3.1.1 The Huawei Use Case</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2 Data Model and Data Population</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 Transactional Workload: Event Stream Processing</td>
<td>28</td>
</tr>
<tr>
<td>3.1.3.1 Event Format and Event Generation</td>
<td>28</td>
</tr>
<tr>
<td>3.1.3.2 Analytics Matrix Update</td>
<td>30</td>
</tr>
<tr>
<td>3.1.3.3 Business Rule Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>3.1.4 Analytical Workload</td>
<td>32</td>
</tr>
<tr>
<td>3.1.4.1 Average Duration of Local Calls Query (Q1)</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4.2 Most Expensive Call Query (Q2)</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4.3 Cost Ratio for Frequent Callers (Q3)</td>
<td>34</td>
</tr>
<tr>
<td>3.1.4.4 Call-intensive Cities Query (Q4)</td>
<td>34</td>
</tr>
<tr>
<td>3.1.4.5 Highest Revenue Query (Q5)</td>
<td>35</td>
</tr>
<tr>
<td>3.1.4.6 Longest Call Query (Q6)</td>
<td>35</td>
</tr>
<tr>
<td>3.1.4.7 Flat Rate Subscribers Query (Q7)</td>
<td>37</td>
</tr>
<tr>
<td>3.1.5 Benchmark Parameters and Figures of Merit</td>
<td>37</td>
</tr>
</tbody>
</table>
## Contents

4.3.4 Business Rule Evaluation ................................. 61
4.3.5 ColumnMap .................................................. 61
4.3.6 Keeping Track of Updates ................................. 63
4.3.7 Query Processing ............................................ 65
   4.3.7.1 Single Instruction Multiple Data (SIMD) .......... 66
4.3.8 Distributed Execution ..................................... 67
4.4 Experiments ..................................................... 69
   4.4.1 Setup ....................................................... 70
   4.4.2 Optimal Number of Data Partitions and Bucket Size .... 71
   4.4.3 Peak Performance of AIM compared to other Systems ... 72
   4.4.4 Strong Scale-Out ......................................... 73
   4.4.5 Weak Scale-Out ........................................... 74
   4.4.6 Total Cost of Ownership ................................. 74
4.5 Concluding Remarks .......................................... 75

5 Fast Scans on Key-Value Stores ................................. 77
   5.1 Introduction .................................................. 78
   5.2 Requirements for a Scannable Key-Value Store ............ 80
      5.2.1 SQL-over-NoSQL Architecture ......................... 80
      5.2.2 Why is it difficult? ................................... 82
   5.3 Design Space ................................................ 83
      5.3.1 Where to put Updates? ................................. 83
      5.3.2 How to allocate Records? ............................. 84
      5.3.3 How to arrange Records? ............................... 84
      5.3.4 How to handle Versions? .............................. 85
      5.3.5 When to do Garbage Collection? ....................... 85
      5.3.6 Conclusions ............................................. 86
II Confidentiality 109

6 Randomly-Partitioned Encryption 111

6.1 Introduction 112

6.1.1 Background and State of the Art 113

6.1.2 Contributions and Main Results 114

6.1.3 Overview 116

6.2 Client-Server Architecture 116

6.3 Adversary Models 117

6.3.1 Domain Attack 118

6.3.1.1 Rank One-Wayness 118

6.3.2 Frequency Attack 119

6.3.2.1 Frequency One-Wayness 119

6.4 Deterministic RPE 120

6.4.1 Construction 122

6.4.2 Confidentiality under Domain Attack 123

6.5 Probabilistic RPE 125

6.5.1 Construction 126

6.5.2 Confidentiality under Frequency Attack 126

6.6 Modular RPE 128

6.6.1 Construction 128

6.6.2 Confidentiality under Known-Plaintext Attack 129

6.7 Implementation 130

6.7.1 Encryption 130

6.7.1.1 Mapping SQL types to integers 131

6.7.1.2 Efficient Sampling 132

6.7.1.3 Multiple Ciphers within the same Run 132
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7.2 Database Functionality</td>
<td>132</td>
</tr>
<tr>
<td>6.7.2.1 Range and Rank Queries</td>
<td>133</td>
</tr>
<tr>
<td>6.7.2.2 SQL Operator Overview</td>
<td>135</td>
</tr>
<tr>
<td>6.7.2.3 Updates</td>
<td>135</td>
</tr>
<tr>
<td>6.8 Performance Analysis and Experimental Results</td>
<td>136</td>
</tr>
<tr>
<td>6.8.1 Setup and Methodology</td>
<td>137</td>
</tr>
<tr>
<td>6.8.2 General Encryption Overhead</td>
<td>137</td>
</tr>
<tr>
<td>6.8.3 Overhead for varying Number of Runs</td>
<td>138</td>
</tr>
<tr>
<td>6.9 Related Work</td>
<td>138</td>
</tr>
<tr>
<td>6.10 Concluding Remarks</td>
<td>140</td>
</tr>
<tr>
<td>7 MTBase: Optimizing Cross-Tenant Database Queries</td>
<td>141</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>142</td>
</tr>
<tr>
<td>7.1.1 The Spectrum of Multi-Tenant Databases</td>
<td>142</td>
</tr>
<tr>
<td>7.1.2 Cross-Tenant Query Processing</td>
<td>143</td>
</tr>
<tr>
<td>7.1.3 State of the Art in Cross-Tenant Query Processing</td>
<td>144</td>
</tr>
<tr>
<td>7.1.4 SQL Limitations</td>
<td>145</td>
</tr>
<tr>
<td>7.1.5 Overview</td>
<td>146</td>
</tr>
<tr>
<td>7.2 MTSQl</td>
<td>147</td>
</tr>
<tr>
<td>7.2.1 MTSQl API</td>
<td>148</td>
</tr>
<tr>
<td>7.2.2 Data Definition Language</td>
<td>150</td>
</tr>
<tr>
<td>7.2.2.1 CREATE TABLE Statement</td>
<td>151</td>
</tr>
<tr>
<td>7.2.2.2 Conversion Functions</td>
<td>151</td>
</tr>
<tr>
<td>7.2.2.3 Further Conversion Function Properties</td>
<td>153</td>
</tr>
<tr>
<td>7.2.2.4 Integrity Constraints</td>
<td>155</td>
</tr>
<tr>
<td>7.2.2.5 Other DDL Statements</td>
<td>156</td>
</tr>
<tr>
<td>7.2.3 Data Control Language</td>
<td>156</td>
</tr>
</tbody>
</table>
7.2.4 Query Language .................................................. 157
  7.2.4.1 Client Presentation ...................................... 157
  7.2.4.2 Comparisons ............................................. 157
7.2.5 Data Manipulation Language ................................. 158
7.3 MTBase .............................................................. 158
  7.3.1 Client-Server Protocol .................................. 159
  7.3.2 Canonical Query Rewrite Algorithm ...................... 160
  7.3.3 Algorithm Correctness ................................... 164
  7.3.4 Rewriting DDL statements ............................... 164
  7.3.5 Rewriting DML statements ............................... 165
7.4 Optimizations ..................................................... 166
  7.4.1 Trivial Semantic Optimizations ......................... 166
  7.4.2 Other Semantic Optimizations .......................... 167
    7.4.2.1 Client Presentation and Conversion Push-Up ....... 167
    7.4.2.2 Aggregation Distribution .......................... 169
    7.4.2.3 Function Inlining .................................. 172
  7.4.3 Cost-based optimizations ............................... 173
7.5 MT-H Benchmark .................................................. 174
7.6 Experiments and Results ...................................... 175
  7.6.1 Setup ....................................................... 176
  7.6.2 Workload and Methodology .............................. 176
  7.6.3 Benefit of Optimizations ............................... 177
  7.6.4 Cross-Tenant Query Processing at Large ............... 179
  7.6.5 Experiments on System C ............................... 181
    7.6.5.1 Optimization Benefits ............................. 181
    7.6.5.2 Tenant Scaling .................................... 181
Contents

7.7 Related Work .................................................. 182
7.8 Concluding Remarks ............................................ 186

8 Summary and Conclusions ........................................ 187
  8.1 Performance ..................................................... 188
  8.2 Confidentiality and Value .................................... 189
  8.3 Combining Confidentiality, Performance and Value .......... 190
  8.4 Ongoing and Future Work ..................................... 191
Indisputably, cloud computing has been one of the fastest growing businesses related to the field of computer science in the last decade. However, moving databases into the cloud brings about a couple of questions. This thesis looks at some of these questions related to the confidentiality and performance of cloud databases.

1.1 Background and Motivation

To achieve good performance for mixed transactional and analytical workloads on cloud databases is a difficult problem because building an architecture that is both, scalable and elastic and can serve these two kinds of workloads at the same time and with good performance often means to optimize for conflicting goals. As we will see, some of these goals are truly conflicting and the best we can do is to offer “tuning knobs” that allow a user to prioritize one design goal over another. Sometimes, however, design goals are not as much conflicting as they seem to be in the first place and we are able to engineer a system which does not require such tuning knobs because it achieves all design goals with only small compromises.

Moving databases, or more generally data, to the cloud almost immediately raises confidentiality concerns. Especially enterprises that deal with highly-sensitive data, like financial institutions, health care providers and government agencies, are often reluctant
to move to the cloud because they fear security breaches. An effective measure to protect data from illegal access is encryption. However, using a strong encryption scheme implies a dramatic reduction of the cloud processing capabilities: The cloud database is degraded to a “dumb data repository” to store and serve data while the heavy burden of query processing (which involves data encryption and decryption and hence access to the secret keys) is put on the client-side application. In this thesis, we propose two measures to address this confidentiality/performance trade-off: First, in order to protect cloud database tenants from an honest-but-curious cloud provider, we design a new encryption scheme that achieves a good compromise between confidentiality and performance. Second, in order to protect the different cloud database tenants from each other, but assuming a trustworthy cloud provider, we revisit and extend SQL access control. As we will see, having this extended access control functionality also opens new opportunities: If tenants agree to join their individual data sets, new insights can be gained and their data as a whole becomes more valuable.

Deploying databases in the cloud involves many different challenges. While this thesis focuses on scalability, elasticity, confidentiality and value, there are many other important aspects that we do not address, but nevertheless want to acknowledge: For instance, as cloud databases use the cloud as a platform (PaaS), hardware is virtualized, which means that system performance as a whole is hard to predict. Another problematic issue is the difficulty of moving data from one cloud provider to another. In fact, researching how to make data migration easier is interesting for cloud customers as this would allow them to go with the lowest price. However, and this is another challenge with cloud databases, price models are very complex and hence it is often not even clear what that lowest price is.

1.2 Problem Statement

Before going into the details of how we addressed the aforementioned problems, let us describe them a little more explicitly by stating the “rules of the game” in terms of context, design goals and optimization metrics.
1.2. Problem Statement

1.2.1 Context and Design Goals

The context common to all data processing systems and techniques presented in this thesis is the cloud. The cloud, besides relieving enterprises from the burden of maintaining their own expensive on-promise infrastructure, promises elasticity and a corresponding pay-as-you-go pricing model. Elasticity with regard to a cloud infrastructure means that machines, in order to adapt to changing requirements or workloads, can be added or removed dynamically or alternatively, applications can be moved from one machine to another (with more or less resources), all with minimal configuration effort. Consequently, elastic distributed applications are characterized by the fact that they can adapt to changes in the provisioned cloud resources without the need to be restarted or re-deployed.

In order to make best use of cloud elasticity and hence minimize costs, applications need to scale in both dimensions: First, they need to scale up in order to be able to use a bigger machine if the workload becomes bigger. Second, if there are no bigger machines available and the workload still grows, applications need to scale out, which means that they need to adapt to a bigger number of machines. For both dimensions of scalability, an application scales well if for a workload that becomes $x$ times bigger, the costs to meet the same performance also grow by a factor of at most $x$. Besides this notion of weak scaling, there is also the notion of strong scaling. In strong scaling, an application scales well if investing $x$ times more (in terms of resources and ultimately dollars) into the same workload, results in an $x$-fold performance boost.

The fact that the cloud is the common context of this thesis goes hand in hand with the assumption of thin clients: if an application is deployed in the cloud to reduce the costs of on-promise infrastructure, using these cloud applications must be cheap.

We hence summarize the common design goals for all the systems and techniques that we are going to present in this thesis:

- elasticity
- weak and strong scalability in terms of scaling up and scaling out
- low processing costs for clients
- low total cost of ownership (TCO): this includes all costs for using and running the cloud applications, but also for acquiring and operating the client-side devices.
Chapter 1. Introduction

1.2.2 Optimization Metrics

The first part of this thesis focuses on the performance of mixed-workload cloud databases. This means that the concrete optimization metrics are response time and throughput of database queries and updates where we expect frequent (streaming-fashioned) updates and near-real-time ad-hoc analytical queries. The term near real time in that context means that updates to the data must be seen by the analytics (almost) immediately, e.g., within a second. The acceptable delay between updates and analytics is another optimization metric to which we will refer as data freshness.

The second part of the thesis describes techniques that can be used to increase the confidentiality and value of the data in any existing database, e.g., the high-performance cloud databases described in the first part. That means that we are interested in minimizing the performance overhead introduced by those techniques, more concretely the response time they add to updates and queries.

1.2.3 Research Questions

The question addressed in the first part of this thesis is how to build systems that provide good trade-offs between the different performance dimensions and data freshness for mixed workloads, while the key questions to be answered in the second part are: (i) how to encrypt data in a way that a good compromise between performance and confidentiality is achieved and (ii) what are the right semantics to correctly analyze data from multiple tenants in such a way that confidentiality is preserved? The short answers to these questions are visualized in Figure 1.1 and will be discussed next, when talking about the contributions.
1.3 Contributions

This thesis presents novel techniques to improve the performance, confidentiality and value of cloud databases. While the first part addresses performance in terms of scalability, elasticity and costs, the second part focuses on confidentiality and value, but in a way that keeps performance degradation at a minimal level. More concretely, we make the following contributions:

**Capturing emerging industry needs in a novel Benchmarking Suite:** Given the fact that combined streaming, transactional and analytical workloads have gained importance in different parts of the industry, we present a novel benchmark suite that captures these emerging needs. In fact, the topic of hybrid transactional-analytical processing (HTAP) systems is so timely that numerous research prototypes have been developed over the past two years. Our benchmark suite was created in close collaboration with customers from the industry and has already been adopted by other HTAP system researchers, which illustrates its potential to become the new standard benchmark for the comparison of such systems.
Chapter 1. Introduction

Data Structures and Algorithms for Mixed Workloads: We present a novel, PAX-inspired storage layout called ColumnMap that exhibits an excellent compromise between single-record reads and full-table scans and can be configured to make best use of the underlying hardware (SIMD registers and instructions, caches and memory). Together with a two-step scan/merge algorithm for delta update processing and a data and thread allocation scheme specifically designed to work well in conjunction with this data structure, ColumnMap enables building high-performance in-memory HTAP systems. Two such systems, Analytics in Motion (AIM) and TellStore, are presented in order to support this claim. As visualized in Figure 1.1a, AIM allows tweaking data freshness against the performance of analytics, while TellStore can trade-off between update and analytics performance by allocating threads accordingly.

Confidential and High-Performance Database Encryption: Randomly-partitioned encryption (RPE), as presented in this thesis, is a novel encryption scheme that allows trading off confidentiality against performance (cf. Figure 1.1b) in the presence of an honest-but-curious cloud provider. In other words, RPE optimizes performance given the required level of confidentiality as a constraint.

Language and Algorithm for Cross-Tenant Query Processing: We establish MTSQL, a new language that extends SQL with additional semantics to specifically address cross-tenant query processing in a multi-tenant (cloud) database. MTSQL thereby allows increasing the value of the data while preserving its confidentiality. We do not only present the language, but also an MTSQL-to-SQL rewrite algorithm that allows executing MTSQL on top of any given database system. This algorithm is embodied in a system called MTBase that not only serves as a proof of concept, but also allows studying different optimization strategies for the rewrite algorithm. As illustrated in Figure 1.1b, MTSQL gives database tenants the ability to lower their confidentiality requirements and share their data with other tenants or the cloud provider. The more tenants share data, the more valuable insights can be gained from that data, which, if the cloud provider returns some of this added value to the tenants as a compensation, makes the cloud financially more attractive.
1.4 Structure of this Thesis

The first part of this thesis investigates performance in cloud databases.

Chapter 2 summarizes the different streaming workloads relevant to the emerging use cases in the industry and presents a survey on related work in the areas of streaming systems, database management systems and key-value stores.

Chapter 3 formalizes the two benchmarks that capture the more complex among these workloads. The Huawei-AIM benchmark captures the needs of integrated event-processing and real-time analytics, while the YCSB# benchmark integrates real-time analytics with transactional workloads.

Chapter 4 presents AIM, a system for processing high-volume event streams and analytics on the same data. We first analyze the design space for building a system that integrates stream processing with analytics and argue why our chosen design within that space is promising. After describing implementation details, we show its superior performance compared to alternative systems using the Huawei-AIM benchmark and conclude with a number of propositions how to extend AIM to make it suitable for more general database workloads, namely processing of arbitrary transactions.

Chapter 5 describes the design, implementation and evaluation of TellStore, a versioned key-value store with fast scans, which evolved from the lessons learned from AIM. Together with its asynchronous processing model, integration with low-latency networks and efficient use of RDMA, TellStore is an enabler for systems that need to run short- and long-running transactions as well as real-time analytics on one unified storage back-end. An example for such a system is Tell [Loe15, Pil17], which will be shortly summarized in order to illustrate how TellStore can be used to execute arbitrary transactions and analytics on the same integrated data. We finish that chapter with the evaluation of TellStore which includes both benchmarks of Chapter 3 and shows excellent performance compared to state-of-the-art competitors.

The second part of this thesis describes techniques to add confidentiality and value to cloud databases. In contrast to the first part where related work is summarized in a survey, this part summarizes relevant related work at the end of each chapter separately.
Chapter 1. Introduction

Chapter 6 describes and evaluates the performance overhead of RPE and analyzes its security under two different attacks. The chapter aims at giving an overview on what RPE can achieve and how it was implemented. A more thorough analysis of RPE has already been presented by Sanamrad [San14].

Chapter 7 presents MTSQL, a new language paradigm that preserves the confidentiality of the data while increasing its value through cross-tenant query processing. We first revisit existing approaches to cross-tenant query processing, before defining the syntax and semantics of MTSQL. After that, we describe the design of MTBase, a system that implements MTSQL, and evaluate its performance overhead with a benchmark derived from TPC-H [Tra16c].

Chapter 8 concludes this thesis by summarizing what we presented, drawing conclusions from the experiments and looking ahead on how the presented systems could be further integrated into a unified processing stack.
Part I

Performance
The growing popularity of the internet of things [MF10], including mobile phones, connected vehicles, health monitoring devices and a lot of other connected sensors and devices gave rise to a variety of new business use cases and applications that serve them. These applications are typically built around streaming systems that allow ingesting and aggregating enormous amounts of events from different data sources. Given the spike of interest in building such applications, it is rather unsurprising that dedicated stream processing systems are getting significant attention, not only in the database, but also in the data science and in the open-source community.

However, there are also other emerging use cases that require near-real-time analytics on this streaming data. This chapter illustrates why handling this new scenario is too much of a challenge for streaming systems, but also why other lines of related work fall short. This essentially motivates two major contributions of this thesis: the design of new benchmarks that capture this new class of workloads and the design and implementation of novel systems that can deal with it.¹

¹Parts of this chapter have been presented at SIGMOD [BEG⁺15] and will be presented at EDBT [KPB⁺17].
Chapter 2. Emerging Streaming Workloads and the Challenges they impose

2.1 Stream Processing and Analytics

There are different kinds of mixed streaming and analytical workloads that recently emerged from a broad variety of business domains like the telecommunication industry, financial institutions or data center operators. In order to illustrate these different workloads, we will use an actual business use case from the telecommunication industry, more concretely from Huawei [hua12]. In that use case, events are represented as variations of call detail records or charging data records (CDRs) that are generated by probes at nodes of the communication network, e.g., base stations or routers. These events must be processed in real-time in order to maintain an accurate picture of the state of the network, which is the basis for managing the network effectively. This state is, in turn, represented by a large number (hundreds) of indicators. Examples of such indicators are billing and CRM-related information such as the total call duration per day per subscriber and indicators related to the network quality of service such as the number of dropped calls for each cell.

We identify for different types of workloads that a (streaming) system must be able to handle in order to support that use case. These workloads are summarized in Table 2.1 where entity is a general term that may refer to subscribers, cells or other objects of interest, like base stations or routers.

<table>
<thead>
<tr>
<th>Workload</th>
<th>System State</th>
<th>Event Stream Processing</th>
<th>Analytics</th>
</tr>
</thead>
<tbody>
<tr>
<td>stateless stream processing</td>
<td>none</td>
<td>non-transactional</td>
<td>none</td>
</tr>
<tr>
<td>stateful stream processing</td>
<td>indicators per entity</td>
<td>single-row transactions</td>
<td>none</td>
</tr>
<tr>
<td>analytics on stateful streams</td>
<td>indicators per entity +</td>
<td>single-row transactions</td>
<td>read-only queries</td>
</tr>
<tr>
<td></td>
<td>dimensional data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transactions on stateful streams</td>
<td>indicators per entity +</td>
<td>multi-row transactions</td>
<td>read-mostly queries</td>
</tr>
<tr>
<td></td>
<td>dimensional data + tagging</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>state</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Overview on different Streaming Workloads
2.1. Stream Processing and Analytics

2.1.1 Stateless Stream Processing

First, a streaming system could react to single events, as for instance giving a discount to a subscriber who just had a phone call with very low quality or informing a subscriber about saving options if the duration of her call exceeded a certain threshold. In that case, the system directly reacts to an event and does not need to maintain any state, which is why we would call this a *stateless streaming* workload. Stateless streaming is illustrated in Figure 2.1.

2.1.2 Stateful Stream Processing

Second, as shown in Figure 2.2, a streaming system might trigger alerts or discounts based on multiple, correlated events. For instance, it might give a discount to a subscriber who did already ten international phone calls this week or alert an infrastructure maintenance team if the percentage of dropped phone calls in a cell exceeds a certain threshold. Being able to process such triggers requires to maintain a number of indicators. As we will see in this thesis, the number of these indicators can become very large, especially if we maintain indicators at fine granularity. This is why such *stateful streaming* workloads often already exceed the capacity of dedicated streaming systems like Apache Storm [Apa16c], Spark Streaming [spa16] or Apache Flink [fli16].

In some literature, stateful streaming is also referred to as *complex event processing (CEP)* [GWC+06]. This highlights the fact that actions are triggered not only on the basis of a single (simple) event as in stateless streaming, but on a combination of correlated events (complex event). Despite the fact that such complex events involve more than one simple event, they are still bound to a very small subset of the entire system state. More concretely, checking whether a specific subscriber has reached ten international phone calls this week only involves events of that particular subscriber. If the system maintains an indicator that counts international calls of this subscriber this week, it suffices to check that indicator. Otherwise, the system must rely on an archive of raw events (*event archive*) to determine how many international calls this particular subscriber has already made during the current week.
Chapter 2. Emerging Streaming Workloads and the Challenges they impose

Figure 2.1: Stateless Streaming: An event enters the system (1) and is evaluated against the set of stateless trigger rules (boolean predicates on event attributes). The system reports the set matching triggers (2).

Figure 2.2: Stateful Streaming: An event enters the system (1) and updates the set of related indicators (2). The updated indicators and the event itself are checked against the set of stateful trigger rules which are boolean predicates on event attributes and indicators (3). The system reports the set of matching triggers, resp. raises a set of complex events (4).
2.1. Stream Processing and Analytics

Figure 2.3: Analytics on Stateful Streams: The entire system state maintained by stateful streaming is made available to ad-hoc real-time analytical queries (i) that potentially join the indicators with additional dimensional data (ii). Queries might happen in response to a complex event (4), e.g., find the base station that caused a specific alert.

Figure 2.4: Transactions on Stateful Streams: Events can update several sets of related indicators (2). Transactions may involve iterative queries (i) that repeatedly query the indicators and update the tagging state (ii). They can also be part of interactive sessions (i) – (iii).
Chapter 2. Emerging Streaming Workloads and the Challenges they impose

2.1.3 Analytics on Stateful Streams

The third class of streaming workloads, which we call *analytics on stateful streams* (or *streaming analytics* for short), extends stateful streaming with additional analytical queries that investigate the whole system state, respectively the entire set of indicators maintained (cf. Figure 2.3). In addition, these queries might join indicators with static *dimensional data*, for example the geographical locations of cells or the customer billing addresses of a subscribers. Such queries can be used for example by the sales department of the telecommunication company to determine geographical regions with potential high-value customers or by the infrastructure maintenance team to identify the cause of low-quality phone calls in real-time. Other queries could be used to determine the best locations for new routers or base stations based on prior usage.

Such workloads are clearly beyond what traditional streaming systems can handle, respectively what they were designed for. Streaming systems must ingest high volumes of events which is why they partition state to allow updates to be executed in parallel on these partitions. Therefore, processing analytical queries across different sets of indicators means to perform computations across partitions in a consistent manner. Consistency in that context means that queries see a snapshot of the indicators where a well-defined subset of the event stream has been fully integrated, whereas the remaining events are not reflected at all. In fact, none of the aforementioned streaming systems (Apache Storm [Apa16c], Spark Streaming [spa16] and Apache Flink [fli16]) offers this kind consistency.

In order to address fault-tolerance, these systems periodically flush their state to durable storage, *e.g.*, HDFS. This means that a system that can query this durable storage, *e.g.*, Spark [AXL+15] on HDFS, could be used for analytics. However, if the analytical queries need to get access to the most up-to-date state, this solution no longer works as the delay between the streaming engine and HDFS is too big as we will further explain when looking at data warehousing in Subsection 2.2.2.

---

2 In this particular context, the term *static* highlights the fact that this data is not changed as a result of event processing, but rather by explicit write operations of an administrator. Dimensional data is rather static because it does not change often.
2.2. Stream Processing on Database Management Systems

2.1.4 Transactions on Stateful Streams

The fourth class of workloads relevant to our use case are transactions on stateful streams (or simply streaming transactions) as displayed in Figure 2.4. Whereas in all prior workloads, we assumed that an event changes only one set of indicators (for example all indicators related to a specific subscriber or a specific cell), we now also consider events that update several sets of indicators at once as well as transactions that interleave analytical queries with updates to the system state. For instance, an event resulting from subscriber $X$ calling subscriber $Y$ on his mobile phone, might not only change the set of (cost-related) indicators of $X$, but also the set of (quality-related) indicators of $X$ and $Y$. Transactions are read-mostly, i.e., they run analytical queries, but might also make some small changes to the system state.

In order to illustrate read-mostly transactions, let us consider the example where the maintenance team has identified a couple of overloaded routers and wants to find out how to reroute traffic to ameliorate the situation. The way to do this is to run what-if simulations [HMST11]. First, such a simulation is iterative and therefore queries the systems state several times before returning the result. Second, the algorithm can also be interactive in the sense that it might ask the group whether they are happy with the result and, if not, propose alternative solutions. In order to support such interactive algorithms, a transaction needs to store the (intermediary) results of a computation, which means to update the system’s tagging state. Another example of a read-mostly transaction would be a marketing analyst that categorizes customers into high-value, medium-value and low-value and stores the result in the corresponding dimension column.

2.2 Stream Processing on Database Management Systems

The reason why we make a clear distinction between analytics and transactions on stateful streams becomes clear when we consider the fact that these two kinds of workloads are difficult if not impossible to serve for streaming systems. In turn, they are rather in the scope of what database management systems (DBMS) were designed for: If we assume that the entire systems state is kept in a (potentially very wide) table as shown in Figures 2.2 to 2.4, event processing for the first three classes of workloads only
requires single-row (also called single-key) transactions, while transactions on stateful streams require more general multi-row (multi-key) transactions.

2.2.1 OLTP and OLAP workloads

The database community only recently started to look into streaming-like workloads. In fact, the traditional workloads considered in the design of a DMBS are on-line transactional processing (OLTP) and on-line analytical processing (OLAP). These two workloads, as their names suggest, became popular in the mid 1990ies with the rise of the world wide web. According to Chaudhuri and Dayal, OLTP applications “typically automate clerical data processing tasks such as order entry and banking transactions that are the bread-and-butter day-to-day operations of an organization”, while OLAP applications, also called data warehouses, target decision support, which means that “historical and consolidated data is more important than detailed, individual records” [CD97].

2.2.2 Data Warehousing

The popular approach at the time to address both types of workloads was to use two separated dedicated sub-systems. This paradigm, colloquially known as data warehousing, was described by Kimbal [KR11], Chaudhuri and Dayal [CD97] and many others.

In data warehousing, the first sub-system is a write-optimized transactional DBMS that handles (short-lived) OLTP transactions. Every now and then, typically at the end of a business day, relevant information is extracted from the OLTP database, consolidated (resp. transformed) and loaded into a read-optimized database, called the OLAP database or simply the data warehouse. The purpose of that data warehouse is to serve long-running, analytical, read-only decision-support queries. For such queries, consistency is more important than freshness. In other words, it is tolerable that the data is slightly out-dated (which is the case if only updated once a day) as long as there is a guarantee that all records belonging to a certain business day, week or year are present. The clear separation of OLTP and OLAP sub-system has the advantage that it separates the two different concerns: while the OLTP engine is typically designed around a row-oriented storage that allows fast updates, the data warehouse is usually read-only and optimized for read access. As a matter of fact, the data warehousing paradigm gave rise to a whole line
of research on column stores [BMK99, BZN05, SAB+05, RAB+13], which make use of vectorized single-instruction-multiple-data (SIMD) execution [ZR02, WPB+09], dictionary compression [PP03, Fea12, RAB+13] or even bit-level binary compression [FLKX15] to save space and also speed up query execution in the OLAP sub-system.

While the data warehousing paradigm is a very effective approach whenever data staleness is tolerable, there is no way how it could be used to address streaming workloads like those described above. For instance, if an event triggers a maintenance alert, the infrastructure maintenance group has to be able to query the system state in real-time in order to find out about defect or overloaded base stations. If the data is out-dated by a couple of hours and hence consists of an old system state where everything was still in order, the group has no chance to identify the source of the alert and take action. Consequently, one it gets the necessary information a couple of hours later, it might be already too late and bad things (like service outage) might already have happened. Data warehouses cannot simply reduce their staleness to a level close to real-time because of the extract-transform-load (ETL) pipelines used to transfer data from the OLTP to the OLAP sub-system. These are complex, long-running operations, which is underlined by the fact that they were traditionally run overnight.

2.2.3 Main-Memory OLTP systems

The fact that we could use an OLTP database to address stateful streaming has been explored in the VoltDB streaming pipeline [vol16], which is a commercial main-memory database with additional streaming support. However, this pipeline only solves half of the problem because its core component, VoltDB [SW13], is optimized for transactions and not for analytics. The preferred way to do analytics in this pipeline is to export the data into an OLAP database, queryable storage (e.g., HDFS) or files (e.g., CSV). This means that this system inherits the problems from data warehouses, reps. streaming systems as explained in the subsections before: exporting data to another systems adds a data freshness delay to analytical queries, which is not tolerable if they need to be real-time. Moreover, even though VoltDB is a distributed scale-out database, it suffers from severe performance problems if transactions spawn multiple partitions as shown by Loesing et al. [LPEK15].
Chapter 2. Emerging Streaming Workloads and the Challenges they impose

2.2.4 HTAP systems

Approximately a decade ago, when more and more applications moved into the cloud and many of them exhibited the need of analyzing data in real-time, the database, but also the open-source communities settled off to explore alternatives to data warehousing. The result were systems that called themselves combined OLTP/OLAP [KN11], mixed workload [LPEK15] or, most recently, hybrid transaction-analytical processing (HTAP) systems [APM16]. The aim of HTAP systems is to integrate OLTP and OLAP workloads more tightly into a unified system that allows analytical queries to process the data in (near) real-time. Most HTAP systems are in-memory and only persist a (redo or undo) log, which facilitates the integration of OLTP and OLAP. One approach, introduced in HyPer [KN11], is to periodically fork the OLTP process and let the OLAP process(es) use the virtual memory snapshots to execute analytics. Another idea, used in SAP HANA [Fea12], is the differential updates approach where updates resulting from OLTP are kept in a separate row-oriented (delta) buffer and are periodically merged into a column-oriented, dictionary-compressed (main) data structure used for analytics. A very recent system is Peloton [APM16]. Peloton maintains a flexible hybrid row/column storage model, but with a unified query processing model based on an abstraction called logical tile algebra.

In essence, HTAP systems could be used for analytics or even transactions on stateful streams as the results of the short-running transactions used for event processing would be visible shortly after for analytical queries. However, two challenges remain: First of all, none of these systems supports specific support for streaming triggers, i.e., a mechanism that, after each transaction, evaluates a number of rules in order to see whether immediate action (like sending an alert) is required. However, adding such a mechanism is mainly an engineering effort and not a fundamental problem. The second issue, however is more severe. If a database should be deployed in the cloud, which as motivated in the Chapter 1 is the trend today, it has to be elastic in order to adapt to variations in the system load. All the above mentioned HTAP systems are single-node systems that can only scale-up, but not scale out, which clearly defeats elasticity.
2.3 Stream Processing on Key-Value Stores

Another approach to address streaming workloads are key-value stores (KV stores). Before showing how KV stores can be used to address the different streaming workloads, we would like to give a short overview on the evolution of KV stores and what they offer today. KV stores are a major outcome of the NoSQL (not only SQL) movement, a group of people, mainly from the tech industry, that started looking into simpler data models and better scalability for the workloads of their applications. These workloads were characterized by massive single-record read and read-modify-write operations with stringent latency requirements, not unlike streaming workloads. Thus, the solutions they came up with, different implementations of KV stores, are also attractive for stream processing.

2.3.1 Key-Value Store API

A key-value store, as its name suggests, is a system that can store and deliver key-value pairs where the value is simply a byte sequence of arbitrary length. The interface of such stores is very simple: put (also called write) operations are used to store a new key-value pair, respectively replace an existing pair with the same key. Conversely, a get (also called read) operation is used to retrieve a certain key-value pair given its key. Finally, delete operations are used to delete an existing key-value pair.

2.3.2 Availability, Elasticity and Transactional Support

Most KV stores are in-memory, highly-available and elastic (storage nodes can be added and removed accordingly according to the current load). They can either be used as caches in front of a database back-end as in Memcached [Fit04] or as primary storage as in Redis [red16], Dynamo [DHJ+07], Bigtable [CDG+08], Cassandra [LM10], RamCloud [OAE+11], Kudu [Apa16a], LevelDB [Goo16a] and RocksDB [Fac16b] which optimizes LevelDB for server workloads. When being used as primary storage, KV stores also need to be fault-tolerant, so they need to implement persistence (Redis, Bigtable, Cassandra, Dynamo, Kudu, LevelDB/RocksDB) and/or replication (Redis, Bigtable,
Cassandra, Dynamo, Kudu, RamCloud). Redis, Dynamo and Bigtable achieve their high write throughput by allowing multiple versions of each key-value pair.

In Redis and Dynamo, versioning also helps to implement multi-row transactions with optimistic multi-version concurrency control (MVCC). Cassandra and LevelDB/RocksDB on the other hand employ (pessimistic) locking to implement transactions. Finally, Bigtable and RamCloud and Kudu do not support multi-row, but only single-row transactions. Bigtable achieves this by fully exposing timestamps (version numbers) to the client who can read from and write to a particular version. In Kudu, all row updates that happen within the context of a client session are atomic. Ramcloud, which stores only one version of each pair at the time, delivers that version number as part of the result of a read operation. The client can then use this version number in conjunction with a special operation called conditional write, which only writes the new key-value pair if the provided timestamp matches the current timestamp of the pair, which means that the pair has not changed since last read by the client.

### 2.3.3 Key-value stores on RDMA

As applications that use KV stores (like Facebook, Twitter and many more) are often latency-sensitive, a lot of recent work also investigated the ability to reduce read and write latency to KV stores with remote direct memory access (RDMA). RDMA is a networking paradigm in which processes of one machine can read from and write directly into the memory of another (remote) machine by by-passing the operating system and thereby preventing CPU, cache and context switching overheads [Rec16]. Typically, RDMA operations are executed on a low-latency network like Infiniband [Inf16], but there also exists RDMA implementations over converged ethernet (RoCE) as explained by Dragojević et al. [DNCH14]. The ability of using RDMA to speed-up Memcached was explored by Jose et al. [JSL11], while Pilaf [MGL13] and HERD [KKA14] are KV stores that were redesigned from scratch with a strong focus on RDMA. Another interesting project in that regard is FaRM, which is a system that exposes the memory of a cluster of machines as shared address space over RDMA [DNCH14]. As such, FaRM can be used to build a high-performance low-latency KV store atop (FaRM-KV). Moreover, the FaRM follow-up paper [DNN15] describes in detail the implementation of distributed transactions in FaRM-KV.
2.3. Stream Processing on Key-Value Stores

2.3.4 Support for Streaming Workloads

While Memcached, Redis, Dynamo, RamCloud, LevelDB/RocksDB and the RDMA KV implementations (Pilaf, HERD and FaRM-KV) can be regarded as pure KV stores in the sense that their values are completely unstructured, Bigtable and Cassandra are also called structured data stores because their values have a structured schema. In this schema, columns are grouped into so-called column families and reads and writes can be issued on the granularity of a specific column or column family. Kudu offers both APIs: clients can either just read and write binary values (as in pure KV stores) or they can use a strongly-typed schema which is automatically mapped to binary byte strings within the Kudu runtime such that clients do not have to worry about the mapping. Kudu supports updates on the granularity of a key-value pair, but also on specific (strongly-typed) columns.

KV stores are very-well suited for stateless and stateful streaming because their design decision to optimize the performance of gets, puts and single-row transactions exactly matches the requirements of those streaming workloads. On the other hand, KV stores are ill-suited for analytics on stateful streams. Analytical queries typically end up reading most of the records of a given table. Thus, the basis for fast analytics is the existence of a fast scan, which we define as the capability to filter values based on a given set of predicates and apply a given set of aggregation function to these values. While pure KV stores do not even offer predicate-based filtering, structured data stores support filtering, but not aggregation. Instead, the preferred way to do analytics is similar to analytics in main-memory OLTP systems (cf. Section 2.2): exporting or exposing state to a dedicated analytics engine (like Spark, HBase or Impala). As in OLTP systems, this approach brings with it the drawback of increased data staleness. In addition, as we are going to see in this thesis, (partially irrelevant) data is moved forth and back much more often, which dramatically increases the overall latency of analytical queries.

The KV stores that offer multi-row transactions, i.e., Redis, Dynamo, LevelDB/RocksDB and FaRM-KV, are able to process the sub-class of streaming transactions where no analytics are needed (which corresponds to steps 1 to 4 in Figure 2.4). However, as general streaming transactions also include streaming analytics and none of the KV

While LevelDB/RocksDB offer range scans on the key and Redis and RamCloud have a multi-get operation which can also be used to retrieve short ranges, none of these pure KV stores can execute predicates on the value. This is not surprising because the value is assumed to be unstructured.
stores mentioned in this section suit them well, we conclude that none of these systems can execute transactions on stateful streams with satisfactory performance.

2.4 Concluding Remarks

We conclude from the previous sections that building a single system that delivers good performance across all different streaming workloads and is at the same time elastic enough to be deployed in the cloud is difficult. Streaming systems can scale out and therefore support nearly arbitrary high update loads, but they are ill-suited for analytics and transactions on stateful streams because they lack consistency across partitions. On the other hand, traditional DBMS and even modern HTAP systems are limited in the number of updates (events) they can process per second. Moreover, there ability to efficiently process (streaming) triggers is often limited. Key-Value stores are elastic and offer a high get/put performance, but lack a fast scan and therefore support for efficient streaming analytic and transactions.
As motivated in the previous chapter, analytics and transactions on stateful streams are emerging workloads that become more and more important in many business applications and are therefore especially relevant to cloud databases. There are many existing benchmarks that cover different aspects of these workloads, but none of them combines all the required elements. In order to close that gap, we developed two new benchmarks. The first benchmark, called \textit{Huawei-AIM}, covers analytics on stateful streams and was derived from a use case from the telecommunication industry. The second benchmark, \textit{YCSB#}, abstracts streaming transactions, but without trigger processing. Finally, the two benchmarks are composable which allows us to cover the entire workload of transactions on stateful streams.\footnote{Parts of this chapter have been presented at SIGMOD [BEG+15] and are under submission to a systems conference [PBB+17].}
3.1 The Huawei-AIM Benchmark

The Analytics in Motion (AIM) benchmark was developed in collaboration with Huawei Technologies Co. Ltd, which is why we simply refer to it as Huawei-AIM. The benchmark models a use-case from the telecommunication industry that we call the Huawei use case [hua12]. This use case, which comes from a prominent Huawei customer, is a real-live example for analytics on stateful streams as defined in Subsection 2.1.3. Consequently, we define the system under test (SUT) as the box in Figure 2.3 that needs to process different kinds of request while the requests itself are outside the SUT. In the following, we will shortly motivate the use case and define the benchmark by presenting the data model and the three concurrent workloads to be handled (stream processing, trigger evaluation and analytical queries). This is followed by the description of benchmark parameters and figures of merit. We conclude this section with enumerating the different variations of the benchmark that were used in practice and a short paragraph that contrasts the Huawei-AIM to other existing benchmarks.

3.1.1 The Huawei Use Case

Traditionally, billing data is stored off-line in the data warehouse of a mobile phone operator and is used to implement marketing campaigns, such as offering discounts or new calling plans to loyal customers. The goal is to make this analysis more flexible so that customers can benefit immediately from such marketing campaigns.

Typical marketing campaigns and analytical queries do not depend on single events (caused by phone calls, messages or network requests), but on calculating indicators per entity (i.e. per subscriber or per cell). All the indicators of an entity (also referred to as maintained aggregates) are held in an entity record which is part of a huge materialized view that we call the analytics matrix.

3.1.2 Data Model and Data Population

The general data model of the benchmark is depicted in Figure 3.1. As we can see, the analytics matrix plays a central role as it is by far the largest (many records) and widest (many attributes) table in the schema. Besides a couple of attributes that describe a
subscriber and link into dimension tables as foreign keys, it keeps a large number of indicators for every subscriber (aggregate 1, aggregate 2, ...).

There is basically such an aggregate for each combination of aggregation function (min, max, sum), aggregation window (this day, this week, ...) and several event attributes that will be explained later. Table 3.1 shows a snippet of the conceptual schema of these aggregates that includes e.g., an aggregate for the shortest duration of a long-distance phone call today (attribute min in Table 3.1). The number of such aggregates (which hence also determines the number of columns of the analytics matrix) is a workload parameter with default value 546.\(^2\) The structure of aggregates can be generated using our open-source AIM schema generator [Sys16a].

Once the database schema, including the schema of the analytics matrix which is determined from a meta database, is loaded, the population of the database works straight-forward: the dimension tables, (RegionInfo, SubscriptionType, SubscriberCategory and SubscriberValue), are initialized with a small set of static records and will

\(^2\)The number of unique aggregates is the number of different aggregation functions on duration and cost plus the count \((3 \cdot 2 + 1)\) multiplied by the number of different windows (this day/this week \(\rightarrow 2\)) and the number of different locality values (local/long-distance/all \(\rightarrow 3\)), which makes \((3 \cdot 2 + 1) \cdot 2 \cdot 3 = 42\) in total. As Huawei could only tell that the real number of aggregates of their customer is something “clearly above 500”, but could not disclose what exactly these aggregates were, we decided to just replicate the attributes a number of times, concretely 13 times, and got 546. Thus the number of aggregates always have to be a multiple of 42.
never be changed again. Their exemplary values are shown in Figure 3.2. The analytics matrix is initialized with $sf \cdot 10^7$ records where parameter $sf$ is the scaling factor and the values are chosen as follows: subscriber_id is an integer that starts at 1 and is atomically incremented with each record. The foreign keys (zip, subscription_type, category, value) are drawn uniformly at random from the set of available values in the corresponding dimension table. last_updated is a time stamp that stores the information when the record was last updated by an event. Its initialization value as well as the initialization value of all count and sum aggregates is 0 while min aggregates are initialized to the maximal value of the corresponding data type and max aggregates to the min value.

### 3.1.3 Transactional Workload: Event Stream Processing

This section describes the transactional, more stream-fashioned workload of the benchmark, event stream processing (ESP), which itself contains two phases: First, for each event that enters the system, the corresponding entity record in the analytics matrix has to be updated. Second, the updated entity record and the event itself are checked against a set of triggers as described in Subsection 2.1.3. As in the Huawei use case these triggers are used to raise alert and also give campaign-based rewards to loyal customers, we will also refer to them as campaigns. Before explaining ESP in more detail, let us shortly explain how events look like and how they are generated.

#### 3.1.3.1 Event Format and Event Generation

As we can see on the top of Figure 3.3, an event in the Huawei-AIM benchmark consists of the subscriber id of caller and callee (from_id, to_id), the time stamp of the beginning of the call, a boolean flag that indicates whether this call is local or long-distance (is_local), the call's duration in seconds and its cost in dollars. Events are generated by
3.1. The Huawei-AIM Benchmark

<table>
<thead>
<tr>
<th>id</th>
<th>zip</th>
<th>city</th>
<th>state</th>
<th>country</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CH-1000</td>
<td>Lausanne</td>
<td>Vaud</td>
<td>Switzerland</td>
<td>EUROPE</td>
</tr>
<tr>
<td>2</td>
<td>CH-8000</td>
<td>Zurich</td>
<td>Zurich</td>
<td>Switzerland</td>
<td>EUROPE</td>
</tr>
<tr>
<td>3</td>
<td>DE-80801</td>
<td>Munich</td>
<td>Bavaria</td>
<td>Germany</td>
<td>EUROPE</td>
</tr>
<tr>
<td>4</td>
<td>ARG-B6500</td>
<td>Buenos Aires</td>
<td>Buenos Aires</td>
<td>Argentina</td>
<td>SOUTH AMERICA</td>
</tr>
<tr>
<td>5</td>
<td>CHI-100000</td>
<td>Beijing</td>
<td>Beijing</td>
<td>China</td>
<td>ASIA</td>
</tr>
<tr>
<td>6</td>
<td>CHI-101500</td>
<td>Beijing</td>
<td>Beijing</td>
<td>China</td>
<td>ASIA</td>
</tr>
</tbody>
</table>

(a) RegionInfo dimension records

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>cost [$]</th>
<th>free_call_minutes</th>
<th>free_data_mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prepaid</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Contract</td>
<td>10</td>
<td>120</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Contract</td>
<td>20</td>
<td>720</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>Contract</td>
<td>50</td>
<td>max_int</td>
<td>max_int</td>
</tr>
</tbody>
</table>

(b) SubscriptionType dimension records

<table>
<thead>
<tr>
<th>id</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Business</td>
</tr>
<tr>
<td>2</td>
<td>Private</td>
</tr>
<tr>
<td>3</td>
<td>company-internal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>threshold [$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Silver</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Gold</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>Platinum</td>
<td>150</td>
</tr>
</tbody>
</table>

(c) SubscriberCategory dimension records

(d) SubscriberValue dimension records

Figure 3.2: Exemplary Dimension Records used in AIM Benchmark

so-called ESP clients, which are themselves not part of the system under test and whose responsibility is to send events to the SUT at a certain event target rate and measure the effective event throughput of the system by counting how many events have been acknowledged. ESP clients do not wait for acknowledgment before sending the next event, but instead wait for a short fixed amount of time which is the inverse of the target event rate. The events are generated in such a way that from_id and to_id are chosen uniformly at random from the set of available entity ids (1 to \(sf \cdot 10^7\)). While the time stamp is set as the current system time and we flip a fair coin for is_local, duration is chosen uniformly at random from \([1, 10000]\) and cost from \([0.10, 100.00]\).
### Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

#### 3.1.3.2 Analytics Matrix Update

Algorithm 3.1 shows the pseudo code for updating the analytics matrix. We denote the function that updates a certain attribute group as update\_attr\_group. Attribute groups are usually small and contain interdependent attributes, as for example count, sum and average of the same metric. In order for the indicators to be consistent, lines 3 to 6 in Algorithm 3.1 must happen atomically. This means that between the get (step 3) and the put (step 6) operation of a record, no other process is allowed to modify this record in the analytics matrix.

**Algorithm 3.1: Updating Indicators in the Analytics Matrix**

```plaintext
1: function update_matrix(Event e)
2:   UUID id ← get_id(e)
3:   EntityRecord r ← get(id)
4:   for all AttributeGroup attr\_group ∈ r do
5:     attr\_group ← update\_attr\_group(e, attr\_group)
6:   put(id, r)
7:   return r
```

---

**Figure 3.3: Updating an Entity Record**

<table>
<thead>
<tr>
<th>from_id</th>
<th>to_id</th>
<th>timestamp</th>
<th>is_local</th>
<th>duration</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>134525</td>
<td>461345</td>
<td>13589390</td>
<td>false</td>
<td>583 secs</td>
<td>$0.50</td>
</tr>
</tbody>
</table>

(a) get entity record:

<table>
<thead>
<tr>
<th>subscriber_id</th>
<th>last_updated</th>
<th>count_all</th>
<th>sum_local_duration_today</th>
<th>sum_long_distance_cost_today</th>
</tr>
</thead>
<tbody>
<tr>
<td>134525</td>
<td>13573283</td>
<td>3</td>
<td>1921 secs</td>
<td>$7.50</td>
</tr>
</tbody>
</table>

(b) update entity record according to event:

<table>
<thead>
<tr>
<th>subscriber_id</th>
<th>last_updated</th>
<th>count_all</th>
<th>sum_local_duration_today</th>
<th>sum_long_distance_cost_today</th>
</tr>
</thead>
<tbody>
<tr>
<td>134525</td>
<td>13589390</td>
<td>4</td>
<td>1921 secs</td>
<td>$8.00</td>
</tr>
</tbody>
</table>

(c) put modified entity record
3.1. The Huawei-AIM Benchmark

An example execution of the algorithm is illustrated in Figure 3.3: on the arrival of an event, the entity record of the caller (from_id) is retrieved (a). This record is then updated by applying the corresponding update function on each group of attributes (b). First, the last_updated time stamp is set to the event’s time stamp. If the difference between the current and the old time stamp indicates that some time windows (e.g., the current day or week) have expired, the corresponding aggregates are reset. Next, the aggregates are updated. In the current example, the event is a long-distance call and hence all metrics that refer long-distance calls (i.e., long_distance_cost_today) as well as the metrics that refer to all calls (i.e., count_all) are updated while the metrics that include local calls (i.e., sum_local_duration_today) remain unchanged. In the end, the new version of the entity record is stored (c).

3.1.3.3 Business Rule Evaluation

The second important functionality of ESP is business rule evaluation. This evaluation must happen in real-time, which means that each rule has to be evaluated against each new event and the updated entity record. Business rules in a telecommunications billing system are mainly used for marketing campaigns (rule 1), but could also trigger alerts, e.g., for potential phone misuse (rule 2) as shown in Table 3.2. Actions not only include messages being sent back to customers or to maintenance, but could also trigger new events being fed back into the system or modify entity records. In order to prevent the system from being flooded with rules that continuously trigger once their condition is met, a firing policy is defined for each rule. This policy defines how many times a rule can trigger within a certain (tumbling or sliding) time window (cf. Gasparis [Gas13] for details). The creation of business rules is also part of the AIM schema generator [Sys16a] where the number of rules is a parameter defaulting to 300. Each rule consists of 1 to 10 conjuncts and each of these conjuncts has 1 to 10 predicates where both, the number of conjuncts and the number of predicates, are drawn uniformly at random.

A straight-forward high-level method for business rule evaluation is shown in Algorithm 3.2. The method takes as input an up-to-date entity record (as produced by Algorithm 3.1) well as the event itself and checks them against all rules. Algorithm 3.2 assumes that rules are in disjunctive normal form (DNF) and are therefore encoded as a list of conjuncts, each of which contains a list of predicates. The algorithm features early abort and early success: (a) whenever a predicate evaluates to false, the whole conjunct evaluates to
### Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

<table>
<thead>
<tr>
<th>number</th>
<th>rule</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number-of-calls-today &gt; 20 AND total-cost-today &gt; $100 AND event.duration &gt; 300 secs</td>
<td>inform subscriber that the next 10 phone minutes today will be for free</td>
</tr>
<tr>
<td>2</td>
<td>number-of-calls-today &gt; 30 AND (total-duration-today / number-of-calls-today) &lt; 10 secs</td>
<td>advise subscriber to activate the screen lock as it appears that his smart phone is making calls on its own</td>
</tr>
</tbody>
</table>

Table 3.2: Example Business Rules

```plaintext
1: function evaluate_rules(Event e, EntityRecord r)
2:     Rule-Set result ← ∅
3:     for all Rule rule : rules do
4:         for all Conjunct c: rule.conjuncts() do
5:             boolean matching ← true
6:         for all Predicate p: c.predicates() do
7:             if p.evaluate(e, r) = false then
8:                 matching ← false
9:             break
10:         if matching = true then
11:             result ← result ∪ rule
12:         break
13:     return result
```

Algorithm 3.2: Business Rule Evaluation

false and hence we can continue with the next conjunct (lines 7 to 9) and (b) whenever a conjunct evaluates to true, the whole rule evaluates to true and hence we can continue evaluating the next rule in the rule set (lines 10 to 12).

#### 3.1.4 Analytical Workload

At the same time as events are processed (and business rules evaluated), there are real-time ad-hoc analytical queries (RTA queries) to be processed. These queries simulate interactive sessions of people in the marketing department (that are for instance brain-
3.1. The Huawei-AIM Benchmark

storming for new campaigns), but also by maintenance groups that need to get access to the system state in (near) real time in order to fix an issue. It is actually hard to simulate ad-hoc queries because they can basically ask anything and we cannot make any assumptions about regularities in their access patterns. Thus, we propose to use a set of seven parametrized queries, but disallow the SUT to create or use any kind of index structures except for primary key indexes.

The rest of this subsection describes these seven queries and gives a short explanation about their purpose. There are two categories of RTA queries. While queries 1 to 3 query the analytics matrix only, queries 4 to 7 involve joins with one or several dimension tables. The parameter generation is random and summarized in the end, in Table 3.3. Each RTA client selects an initial query uniformly at random and then proceeds with query numbers in ascending order (and restarting with query 1 after query 7). It executes in a closed loop, meaning that it waits for the response of a query before executing the next one.

3.1.4.1 Average Duration of Local Calls Query (Q1)

This query inspects how long (in minutes) people talk on average on a call. We are simply interested in those people that have called a local number already at least a specific number of times this week.

```
SELECT AVG(sum_duration_of_all_calls_this_week)
FROM AnalyticsMatrix
WHERE number_of_local_calls_this_week > [ALPHA];
```

Listing 3.1: Huawei-AIM Benchmark, Query 1

3.1.4.2 Most Expensive Call Query (Q2)

This query simply wants to find what was the cost of the most expensive call this week. Again it only considers the most expensive call among the frequent callers where the number of calls this week so far exceeds a certain threshold.
Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

```sql
SELECT MAX(max_cost_of_all_calls_this_week)
FROM AnalyticsMatrix
WHERE number_of_all_calls_this_week > [BETA];
```

Listing 3.2: Huawei-AIM Benchmark, Query 2

### 3.1.4.3 Cost Ratio for Frequent Callers (Q3)

This query computes the cost ratio (in terms of dollars per minute) for each group of people with a certain number of calls this week. In essence, this is for the marketing people to verify whether the frequent callers pay the cheapest rate, which should ideally be the case.

```sql
SELECT (SUM(sum_cost_of_all_calls_this_week)) / (SUM(sum_duration_of_all_calls_this_week)) AS cost_ratio
FROM AnalyticsMatrix
GROUP BY number_of_all_calls_this_week
ORDER BY number_of_all_calls_this_week DESC
LIMIT 100;
```

Listing 3.3: Huawei-AIM Benchmark, Query 3

### 3.1.4.4 Call-intensive Cities Query (Q4)

The motivation for this query is to find the cities in which people talk much with each other, *i.e.*, where average number of local calls and total duration of local calls is high. More concretely, this means to compute for each city the average number of local calls and the total duration of local calls in the current week for the most valuable subscribers, *i.e.* subscribers whose average number of local calls and the total duration of local calls in the current week exceed certain thresholds. This is a query that could be run by the marketing department in order to see on which cities they should focus with marketing campaigns that involve local calls. Another use case for this query would be maintenance groups that try to find out in which cities new antennas would pay off most.
3.1. The Huawei-AIM Benchmark

```sql
SELECT city, AVG(number_of_local_calls_this_week), SUM(sum_duration_of_local_calls_this_week)
FROM AnalyticsMatrix, RegionInfo
WHERE number_of_local_calls_this_week > [GAMMA] AND sum_duration_of_local_calls_this_week > [DELTA] AND AnalyticsMatrix.region = RegionInfo.id
GROUP BY city;
```

Listing 3.4: Huawei-AIM Benchmark, Query 4

3.1.4.5 Highest Revenue Query (Q5)

The motivation for this query is that the marketing department might want to know in which region the company earns how much money with local and long distance calls in order to customize their campaigns. It might be interested in these values for subscribers of a certain subscriber category and value.

```sql
SELECT region, SUM(sum_cost_of_local_calls_this_week) AS local, SUM(sum_cost_of_long_distance_calls_this_week) AS long_distance
FROM AnalyticsMatrix a, SubscriptionType t, Category c, RegionInfo r
WHERE t.type = [T] AND c.category = [CAT] AND a.subscription_type = t.id AND a.category = c.id AND a.region = r.id
GROUP BY region;
```

Listing 3.5: Huawei-AIM Benchmark, Query 5

3.1.4.6 Longest Call Query (Q6)

The motivation for this query is to find the subscriber with the longest call in this day and this week for local and long distance calls. We are interested in subscribers residing in a specific country.

```sql
{  
SELECT subscriber_id, "local_day" AS property,
       max_duration_of_local_calls_this_day AS max_value
FROM AnalyticsMatrix
```
WHERE max_duration_of_local_calls_this_day = (  
    SELECT MAX(max_duration_of_local_calls_this_day) FROM AnalyticsMatrix  
    , RegionInfo WHERE country=[CTY] AND AnalyticsMatrix.region =  
    RegionInfo.id)  
LIMIT 1
)  
UNION

SELECT subscriber_id, "local_week" AS property,  
    max_duration_of_local_calls_this_week AS max_value  
FROM AnalyticsMatrix  
WHERE max_duration_of_local_calls_this_week = (  
    SELECT MAX(max_duration_of_local_calls_this_week) FROM  
    AnalyticsMatrix, RegionInfo WHERE country=[CTY] AND AnalyticsMatrix.  
    region = RegionInfo.id)  
LIMIT 1
)  
UNION

SELECT subscriber_id, "long_distance_day" AS property,  
    max_duration_of_long_distance_calls_this_day AS max_value  
FROM AnalyticsMatrix  
WHERE max_duration_of_long_distance_calls_this_day = (  
    SELECT MAX(max_duration_of_long_distance_calls_this_day) FROM  
    AnalyticsMatrix, RegionInfo WHERE country=[CTY] AND AnalyticsMatrix.  
    region = RegionInfo.id)  
LIMIT 1
)  
UNION

SELECT subscriber_id, "long_distance_week" AS property,  
    max_duration_of_long_distance_calls_this_week AS max_value  
FROM AnalyticsMatrix  
WHERE max_duration_of_long_distance_calls_this_week = (  
    SELECT MAX(max_duration_of_long_distance_calls_this_week) FROM  
    AnalyticsMatrix, RegionInfo WHERE country=[CTY] AND AnalyticsMatrix.  
    region = RegionInfo.id)  
LIMIT 1
);
3.1. The Huawei-AIM Benchmark

3.1.4.7 Flat Rate Subscribers Query (Q7)

The motivation for this query is to find out which subscriber pays the least money for her calls. In order to determine this, we compute the fraction of total cost and the total duration of all calls. In the end, we are interested in the minimum of these fractions. We will only look at one subscriber value type at the time and either analyze the current week or day.

```sql
SELECT subscriber_id, (sum_cost_of_all_calls_this_week / sum_duration_of_all_calls_this_week) AS flat_rate
FROM AnalyticsMatrix, SubscriberValue v
WHERE v.value=[V] AND (sum_cost_of_all_calls_this_week / sum_duration_of_all_calls_this_week) = (
    SELECT MIN(sum_cost_of_all_calls_this_week / sum_duration_of_all_calls_this_week)
    FROM AnalyticsMatrix, SubscriberValue v1
    WHERE v1.value=[V] AND AnalyticsMatrix.value = v1.id
);
```

Listing 3.7: Huawei-AIM Benchmark, Query 7

3.1.5 Benchmark Parameters and Figures of Merit

The benchmark has a couple of different parameters which can be mainly characterized into execution parameters, workload parameters, system parameters and service level agreements (SLAs). While the execution parameters are the ones that are randomly determined at run-time, in our case the query parameters as depicted in Table 3.3, the other parameters have to be stated beforehand.

3.1.5.1 Workload Parameters

Scaling Factor \(sf\) The scaling factor determines two things. First, it determines the number of subscribers and hence the number of entity records in the analytics matrix which is \(sf \cdot 10^7\). Second, as we expect the number of events to be proportional to the number of subscribers, \(sf\) also influences the target event rate which we define as
Table 3.3: Random Parameter Generation for the Queries of the Huawei-AIM Benchmark

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>uniform from [0,2]</td>
</tr>
<tr>
<td>BETA</td>
<td>uniform from [2,5]</td>
</tr>
<tr>
<td>GAMMA</td>
<td>uniform from [2,10]</td>
</tr>
<tr>
<td>DELTA</td>
<td>uniform from [20,150]</td>
</tr>
<tr>
<td>T</td>
<td>uniform from {“Prepaid”,“Contract” }</td>
</tr>
<tr>
<td>CAT</td>
<td>uniform from {“Business”,“Private”, “company”}</td>
</tr>
<tr>
<td>CTY</td>
<td>uniform from {“Switzerland”,“Germany”, “Argentina”, “China”}</td>
</tr>
<tr>
<td>V</td>
<td>uniform from {“None”,“Silver”, “Gold”, “Platinum”}</td>
</tr>
</tbody>
</table>

$f_{ESP} = s f \cdot 10^5$ events per second. For the default value of $s f = 1$, this means that there are 10 million subscribers that, all together, produce 10,000 events per second.

**Number of Indicators / Maintained Aggregates ($N$)** The number of indicators, as mentioned before, essentially defines the width of the analytics matrix. As every indicator is updated on every event, we expect event processing to take longer if $N$ is bigger. $N$ has to be a multiple of 42 (cf. footnote 2, page 27 for an explanation) and its default configuration with $N = 546$ results in entity records of size 3 KB each. Thus, the total size of the analytics matrix in default configuration is $10^7 \cdot 3 \text{ KB} = 30 \text{ GB}$.

**Number of Campaigns / Business Rules ($BR$)** As stated above, the number of campaigns, $BR$, is another workload parameter that is set in the beginning. The more campaigns there are, the more time the system needs for event processing. The default number of campaigns is 300.

**Execution Time ($t$)** The execution time states how long the experiment runs. Population of the database is not part of the experiment and hence does not contribute to $t$. $t$ defaults to 5 minutes.
3.1. The Huawei-AIM Benchmark

3.1.5.2 Service-Level Agreements (SLAs)

Often, service providers provide performance guarantees to their customers in terms of service-level agreements (SLAs). These SLAs are just another class of parameters of the benchmark.

**Maximum Event Processing Time** \( (t_{ESP}) \): an upper bound on how much time the SUT is allowed to take to process an event and evaluate the entire set of campaigns. The SUT is required to measure the event processing time and report any violation on this SLA. The default value of \( t_{ESP} \) is 10 milliseconds.

**Minimum Event Processing Rate** \( (f_{ESP}) \): a lower bound on how many events the SUT must process per second. This SLA is no independent benchmark parameter as it is determined by \( s_f \) as already mentioned. The effective event processing rate can either be measured within the SUT or by the ESP clients, but in either case, if it is below \( f_{ESP} \), the violation of the SLA has to be reported. The default value of \( f_{ESP} \) is 10,000 events per second.

**Maximum RTA Query Response Time** \( (t_{RTA}) \): an upper bound on how much time the SUT is allowed to take to answer a RTA query. As response times have to be measured end-to-end, they are measured within the RTA client who needs to report any violations of this SLA. The default value of \( t_{RTA} \) is 100 milliseconds.

**Minimum RTA Query Rate** \( (f_{RTA}) \): lower bound on how many RTA queries the SUT must answer per second. Query throughput is measured locally at each RTA client and summed up in the end of the experiment. It is only then that we know whether \( f_{RTA} \) was violated. The default value of \( f_{RTA} \) is 100 queries per second.

**Freshness** \( (t_{fresh}) \): upper bound on the time that the SUT is allowed to take between the moment an event arrives and the time when the affected entity record is visible to RTA queries. Any violations on \( t_{fresh} \) have to be detected in the SUT by using appropriate (implementation-specific) mechanisms. The default value of \( t_{fresh} \) is 1 second.
Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

3.1.5.3 System Parameters

The system parameters include all the relevant specification parameters about hardware that runs the SUT, e.g., the type and number of CPU cores, cache sizes, memory sizes and alike. They also include the specification details of the network used between the clients and the SUT as well as any SUT-specific implementation parameters. In addition, the following two parameters must be chosen in a way that optimally supports the SUT to satisfy the SLAs.

**Number of RTA clients (c)** As RTA clients execute in closed loops, $c$ is a crucial parameter because it is essentially an upper bound on the number of pending RTA queries within the SUT.

**Number of ESP clients (e)** As ESP clients do simply send events at a fixed rate, $s$ is of minor importance. It is however important to mention that the ESP clients have to be configured in such a way that their aggregated event rate meets the target event rate. For instance, if $c = 8$ and $s f = 1$, this means that each of the clients sends at a rate of 1,250 events per second which sums up to $8 \cdot 1,250 = 10,000$ events per second.

3.1.5.4 Methodology and Figures of Merit

The primary figure of merit for the Huawei-AIM benchmark, given a specific configuration of workload parameters and SLAs, is the total cost of ownership (TCO) for resources used to execute the benchmark while satisfying all SLAs. We do not dictate the exact way how to measure TCO, but acceptable metrics include the monetary cost of running the workload in the cloud (e.g., on a couple of Amazon EC2 instances) or the number of CPUs or machines used (which makes most sense for a homogeneous set of machines).

To differentiate between implementations with the same TCO, secondary figures of merit can be applied. These are average event processing time, average response time and throughput of RTA queries. Event processing throughput does not make sense as a figure of merit because the way ESP clients send events, the throughput is either equal to $f_{ESP}$ (if the SLA can be met) or below (otherwise).
The usual *methodology* recommended (but not mandatory) to run the benchmark is to first choose the system parameters in such a way that the ESP SLAs \((t_{ESP} \text{ and } f_{ESP})\) are met and then increasingly add resources to meet the RTA SLAs \((t_{RTA}, f_{RTA} \text{ and } t_{freshness})\).

### 3.1.6 Benchmark Variations

In addition to the standard benchmark just described, there exists several variations that have also been used in practice, especially in systems for which implementing the entire benchmark was difficult.

**Huawei-AIM-Standalone**  Response times and throughput in the original benchmark are intentionally measured end-to-end with a network in between. However, if one wants to be sure that the network (respectively the cost of event (de-) serialization) is not the bottleneck, the RTA and ESP clients can also be placed on the same machine as the SUT and communicate through local memory. This variation of the benchmark is called *Huawei-AIM-Standalone*.

**Huawei-AIM-Simple**  Although campaign processing is an essential part of the original benchmark (and use case), it is not a necessary ingredient to test a system for its ability to process mixed transactional/real-time analytics workloads. Hence, the Huawei-AIM-Simple benchmark consists of the original benchmark without campaign processing. Huawei-AIM-Simple and Huawei-AIM-Standalone can of course also be combined, resulting in something called the Huawei-AIM-Simple-Standalone benchmark. This benchmark was actually used in a survey on different research prototypes and open-source data processing systems [KPB+17].

**Huawei-AIM+**  The Huawei-AIM benchmark can be combined with the YCSB# benchmark in order to get full coverage on *transactions on stateful streams*. The resulting benchmark, called *Huawei-AIM+*, will be further explained in Section 3.3.
3.1.7 Related Work

To the best of our knowledge, the Huawei-AIM benchmark was the first and so far only benchmark that included all the necessary elements of analytics on stateful streams as defined in Subsection 2.1.3: event stream processing, real-time ad-hoc analytical queries and triggers. However, there exist a couple of benchmarks that share some similarities with Huawei-AIM.

The Telecommunication Application Transaction Processing (TATP) Benchmark was developed within IBM and also targets a use case from the telco industry. While its transactional workload is somewhat similar to the one in the Huawei use case (read and update subscriber-related information), it does not include any kind of analytics.

Another breed of interesting benchmarks are the DEBS Grand Challenges, which also focus on (near-) real-time analytics on stateful streams and include a variety of different data sets coming from smart plug sensors [JZ14], taxi trips [JZ15] or, most recently, Facebook [GJVZ16]. Compared to Huawei-AIM, the state of the Analytics Matrix in these challenges is much smaller as it either consists of much fewer columns, less rows or both.

The data model depicted in Figure 3.1 has a star-shaped schema which naturally brings up the question why we could not simply use the star schema benchmark (SSB) [OOCR09]. There are a couple of significant differences between the two benchmarks. First, the facts table in SSB has only 17 attributes, which is much less than the analytics matrix in a typical installation. Second, SSB is mainly an analytics benchmark. If there updates at all (which does not become 100% clear from the benchmark description), these are executed in batches of inserts and deletes (as in TPC-H [Tra16c]). This compares in no way to the access pattern of the analytics matrix which is dominated by highly frequent updates on most of the attributes of an entity record. However, Huawei-AIM inherits the ability of SSB to de-normalize the schema and therefore supersede joins.

3.2 The YCSB# Benchmark

As illustrated in Subsection 2.1.4, streaming transaction workloads include multi-key transactions which are not part of the standard Huawei-AIM benchmark. This is why we developed a second benchmark to combine multi-key transactions with analytics.
3.2. The YCSB# Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) [CST+10] is a popular benchmark for KV stores and cloud service providers. It includes multi-key transactions and a number of queries which are mostly point lookups and range scans on a schema that consists of a single table. YCSB, however, does not include projection or aggregation queries which are at the core of analytical processing. In addition, like the analytics matrix, it only consists of fixed-sized values. However, in order to simulate dimensional data and make the database more realistic for analytics, variable-sized values are also essential.\(^3\) This is why we extended the benchmark in order to test these additional features. This new benchmark, which we will call YCSB#, makes the following modifications: First, it changes the schema and the transactional workload to include variable-sized columns and second, it introduces three new analytical queries that inspect big portions of the data. The rest of this section describes the YCSB# benchmark in more detail, following the same structure as in the description of Huawei-AIM before.

### 3.2.1 Data Model

Like in the analytics matrix of Huawei-AIM, the number of records in the main table of YCSB# is defined with respect to a scaling factor \(sf\), namely \(sf \cdot 10^6\). The default of \(sf = 50\) consequently means that the main table hosts 50 million records. Likewise, the number of columns is a workload parameter. This parameter, \(N\), defines the number of non-key columns of the main table. Together with its single key column, this makes the total number of columns of the \(N + 1\). \(N\) has to be a multiple of 10 (or \(N = 1\)) and its default value is 10.

\[
\begin{array}{cccccccccc}
\text{P} & \text{A1} & \text{B1} & \text{C1} & \text{D1} & \text{E1} & \text{F1} & \text{G1} & \text{H1} & \text{I1} & \text{J1} \\
\text{BIGINT} & \text{DOUBLE} & \text{INT} & \text{INT} & \text{SMALLINT} & \text{SMALLINT} & \text{BIGINT} & \text{BIGINT} & \text{DOUBLE} & \text{STRING} & \text{STRING} \\
\text{key, not null, unique} & \text{not null, uniform from [0,1]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [0,10 000]} & \text{not null, uniform from [0, 255]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [minint, maxint]} & \text{not null, uniform from [minint, maxint]} \\
\end{array}
\]

Figure 3.4: YCSB# Data Model for \(N = 10\)

As shown in Figure 3.4, the schema in its default configuration consists of an 8-byte integer key, named \(P\), and tuples that consist of eight fixed-sized values, named \(A1\) to \(J1\).

---

\(^3\)Variable-sized attributes are a special challenge for analytical queries because they make it harder to predict the size of a query result and hence to allocate buffers of the right size.
Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

We used the following data types: 2-byte, 4-byte and 8-byte integers as well as 8-byte double-precision float. Each of these types appears twice and the values are chosen from the corresponding distributions shown in Figure 3.4. The two variable-sized fields, \( I_1 \) and \( J_1 \), are the two-syllable, respectively three-syllable values for \( p\_type \) as specified by the TPC-H benchmark [Tra16c]. They are short strings of a variable length of at most 25 characters. The schema for \( N = 1 \) simply consists of \( P \) and \( A1 \) which represents a simple mapping from an 8-byte integers to 8-byte double precision floats. For \( N > 10 \), the columns \( A1 \) to \( J1 \) are simply replicated as many times as needed as shown in Figure 3.5.

\[
\begin{array}{|c|c|}
\hline
N = 1 & \hline
P & A1 \big| BIGINT \quad DOUBLE \\
\hline
N = 10 & \hline
P & A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 \\
& BIGINT DOUBLE INT INT SMALLINT SMALLINT BIGINT DOUBLE DOUBLE STRING STRING \\
\hline
N = 100 & \hline
P & A1 B1 C1 D1 E1 F1 G1 H1 I1 J1 A10 B10 C10 D10 E10 F10 G10 H10 I10 J10 \\
& \hline
\end{array}
\]

Figure 3.5: YCSB# Data Model for different Values of \( N \)

3.2.2 Transactional Workload

We want the transactional workload to include all relevant CRUD operations: Create, Read, Update, Delete. However, we also would like to make the transactions as realistic as possible which means that read, update and delete operations should just be executed on records that actually exist and insert operations should just happen for keys that do not yet (or not anymore) exist. Moreover, in order to make sure that the response times of the analytical queries stay in a comparable range, we would prefer the size of the main table to stay constant, which implies that inserts and deletes need to be balanced.

Every transactional client in YCSB# works in a closed loop and on its own set of keys. This key set is defined with respect to the total number of transactional clients, \( e \), as well as the three variables \( client\_id \), \( base\_insert\_key \) and \( base\_delete\_key \) which are local
3.2. The YCSB# Benchmark

to each client. The latter two variables have the invariant that they point to the newest, respectively oldest record in the main table that is owned by that client. Ownership in that context is modulo-based, which means that a client owns all records with key \( k \) for which \( k \mod e = \text{client}_\text{id} \) holds. As during population, records with keys in the range \([0, sf \cdot 10^6 - 1]\) are created, the initial values for these latter two variables are \( \text{base}\_\text{insert}\_\text{key} = \text{client}_\text{id} \) and \( \text{base}\_\text{delete}\_\text{key} = 10^6 - e + x \).

Hence, we can define the four YCSB# operations in pseudo code as shown in Algorithms 3.3 to 3.6 where the symbol \( \overset{\$}{\leftarrow} \) just means to draw from a set uniformly at random.

Clearly, these operations are all very short and not really meaningful on their own. This is why transactional clients always execute a batch of such operations within the context of one single big transaction where the fraction of inserts, updates, deletes and reads within this transaction is a benchmark parameter with the only constraint that inserts and deletes must be balanced.

```
1: function read
2:   k \overset{\$}{\leftarrow} \{\text{base}_{\text{delete}}\_\text{key} + e, \text{base}_{\text{delete}}\_\text{key} + 2e, \ldots, \text{base}\_\text{insert}\_\text{key}\}
3:   r \leftarrow \text{readRecord}(k)
```

Algorithm 3.3: YCSB# random read operation

```
1: function update
2:   k \overset{\$}{\leftarrow} \{\text{base}_{\text{delete}}\_\text{key} + e, \text{base}_{\text{delete}}\_\text{key} + 2e, \ldots, \text{base}\_\text{insert}\_\text{key}\}
3:   r \leftarrow \text{readRecord}(k)
4:   r[A1] \overset{\$}{\leftarrow} [0, 1]
5:   r[B1] \overset{\$}{\leftarrow} [\text{minint}, \text{maxint}]
6:   \ldots
7:   \text{writeRecord}(k, r)
```

Algorithm 3.4: YCSB# random update operation

3.2.3 Analytical Workload

The YCSB# also defines three analytical queries with changing characteristics ranging from simple aggregation on one column to aggregating on one column while filtering on
Chapter 3. Benchmarks for Analytics and Transactions on Stateful Streams

Algorithm 3.5: YCSB# random insert operation

1: function insert
2:   \( r[A1] \leftarrow [0, 1] \)
3:   \( r[B1] \leftarrow [\text{minint, maxint}] \)
4:   . . .
5:   insertRecord(base_insert_key, r)
6:   base_insert_key \leftarrow base_insert_key + e

Algorithm 3.6: YCSB# delete operation

1: function delete
2:   deleteRecord(base_delete_key)
3:   base_delete_key \leftarrow base_delete_key + e

different one to finally retrieving a significant portion of the main table to see how well
the SUT actually performs for large result sets:

- **Query 1**: a simple aggregation on the first floating point column to calculate the
  maximum value:
  
  ```sql
  SELECT max(A) FROM main_table;
  ```

- **Query 2**: does the same aggregation as query 1, but additionally selects only values
  in the second floating point column hosting values in the entire double-precision
  floating point range:
  
  ```sql
  SELECT max(A) FROM main_table
  WHERE H > 0 AND H < 0.5;
  ```

- **Query 3**: retrieves all records where the first 2-byte integer column is between 0
  and 26, which results in retrieving approximately 10% of the entire dataset.
  
  ```sql
  SELECT * FROM main_table
  WHERE E > 0 AND E < 26;
  ```

In that description the columns are always chosen uniformly at random from the set of
identical columns. For instance, if \( N = 100 \), column \( A \) in query 1 is chosen from the set
of columns \( \{A1, A2, \ldots, A10\} \).
### 3.2.4 Benchmark Parameters

Many of the parameters in YCSB# have the same semantics as in Huawei-AIM. These parameters are:

- **scaling factor** \((s_f)\): the number of records is \(s_f \cdot 10^6\).
- **number of non-key columns** \((N)\): the total number of columns including the key is \(N + 1\).
- **execution time** \((t)\)
- **standard system parameters**
  - **number of (closed loop) analytical clients** \((c)\)
  - **number of (closed loop) transactional clients** \((e)\)

It is worth mentioning that in YCSB#, also the transactional clients operate in closed loops and that they process batches of operations with batch size \(B\) (defaulting to 200). An additional parameter is the **transaction mix**, \(M\), which defines the fractions of the different transaction types within a transaction batch of size \(B\). By default, \(M\) is defined as: \(M[\text{read}] = 1/2, M[\text{update}] = M[\text{insert}] = M[\text{delete}] = 1/6\). Notably, the last equation, \(M[\text{insert}] = M[\text{delete}]\), makes sure that inserts and deletes are balanced as required.

### 3.2.5 Figures of Merit

Compared to the Huawei-AIM benchmark, the YCSB# does not really focus on TCO, but is rather intended to be used to test the scalability of the SUT with respect to both, the transactional and the analytical workload. The figures of merit are therefore equivalent to the secondary figures of merit of the Huawei-AIM benchmark: response time and throughput of transactions as well as response time and throughput of analytical queries given a specific combination of workload and system parameters.
3.2.6 Related Work

A well-known benchmark that also tries to combine frequent updates and analytics is the CH benchmark \cite{CFG+11}. This benchmark essentially tries to combine two well-established existing benchmarks, namely TPC-C (transactional) \cite{Tra16a} and TPC-H (analytical) \cite{Tra16c}. However, the CH benchmark has the drawback that due to the nature of TPC-C, the database is constantly growing. This is a problem for analytical queries because they have to query more data on each iteration, which leads to ever-increasing response times for analytical queries. In contrast, YCSB# balances insert and delete operations, thereby rendering the database size constant.

3.3 The Huawei-AIM+ Benchmark

The last section of this chapter shortly sketches the idea how the two benchmarks described so far can be combined in order to fully cover the scenario of transactions on stateful streams as defined in Subsection 2.1.4. This benchmark has never been used in practice and is presented primarily to complete the picture. An alternative to running the combined benchmark is to show that a system performs good in both benchmarks in isolation, which implies good performance in the combined benchmark if we can argue that the SUT can be configured in such a way that the two workloads can run independently of each other. This is actually what we will do in the context of TellStore in Chapter 5.

Looking again at Figure 2.4, the idea is the following: we take the Huawei-AIM benchmark as a starting point, but extend the analytics matrix with \(NYCBS\) tags for every entity. These tag columns correspond to the non-key columns of the main table in YCSB#. Next, in addition to the Huawei-AIM transactional clients (which send events), we add \(eYCSB\) additional transactional clients which update these tags according to the YCSB# logic as described in Subsection 3.2.2. These additional clients represent people from the marketing department that do some analysis to categorize customers into different market segments. Finally, the analytical queries of the Huawei-AIM benchmark have to be extended to take these additional tags into account.
The way we propose to extend these analytics is to leave queries 1 to 7 unchanged, but to add four more queries. These queries would then be clones of the Huawei-AIM queries 4 to 7 where the filter is extended with a predicate on a random $E$ column as in query 2 of YCSB#: $E > 0 \text{ and } E < 26$.

### 3.4 Concluding Remarks

This chapter presented two newly developed benchmarks, Huawei-AIM and YCSB#, as well as their combination, Huawei-AIM+. It thereby sets the stage for reasoning about and implementing new systems able to achieve outstanding performance.
Chapters 2 and 3 motivated the newly emerging streaming workloads with a use case from the telecommunication industry and devised a set of new benchmarks to model their characteristics and needs. Moreover, Chapter 2 also revisited related work and explained the need for a new class of integrated solutions for handling such hybrid streaming analytics workloads. This chapter presents a novel architecture that integrates key-value-based event processing and SQL-based analytical processing on the same distributed store while minimizing the total cost of ownership.

Our approach combines several well-known techniques such as shared scans, delta processing, a PAX-fashioned storage layout and an interleaving of scanning and delta merging in a completely new way. Performance experiments with the Huawei-AIM benchmark show that our system scales out linearly with the number of servers. For instance, our system sustains event streams of 100,000 events per second while simultaneously processing 100 ad-hoc analytical queries per second, using a cluster of 12 commodity servers. In doing so, our system meets all response time goals of our telecommunication customers, *i.e.*, 10 milliseconds per event and 100 milliseconds for an ad-hoc analytical query. Moreover, our system beats commercial competitors by a factor of 2.5 in analytical and two orders of magnitude in update performance.¹

¹Parts of this chapter have been presented at SIGMOD [BEG⁺ 15].
Chapter 4. Analytics in Motion

4.1 Introduction

The data processing requirements for network monitoring are significant. In recent years, a number of additional applications have emerged that need to analyze CDR (call detail records or charging data records) events in real time (e.g., marketing campaigns and quality of service and experience [hua12]) and the demands are growing as these applications are becoming more sophisticated. They typically involve a large number of ad-hoc queries on the key indicators of the network or usage of classes of subscribers. These queries need to be answered in real time (within tens of milliseconds) and need to operate on fresh data. It is not acceptable to load the CDR data into a data warehouse and run the new, emerging analytics applications against the data warehouse because that would violate the data freshness requirements. Furthermore, such a data warehouse would be too expensive as it would require to store the data twice and it would significantly increase the overall total cost of ownership (TCO).

To address such analytics on stateful streams workloads that involve a high volume of events and carrying out complex analytics in real-time on these events, a number of alternative architectures have been proposed in the past. One approach is to scale out in a cluster of machines. The most prominent systems that enable such scale out are Apache Storm [Apa16c] for event processing, Apache Hadoop [SKRC10] or Apache Spark [ZCF+10] for the analytics and Druid [YTM+14] for real-time analytics and fast data ingestion. While these systems scale well with the number of machines, they are simply not fast enough in event processing if hundreds of indicators have to be maintained at a speed of 100,000s of events per second as it is usually the case in the telecommunications industry. Moreover, they copy events multiple times between the event processing system (e.g., Storm) and the analytics system (e.g., Hadoop) which is a costly operation. For instance, we tested Storm on our workload and found that it was able to handle only some dozens of indicators at a rate of several thousands events per second on a cluster of 12 commodity machines. While Druid achieves sufficiently high event processing rates, the reported numbers were also obtained from only few dozens of indicators.

In the other extreme, there has been recent work to handle mixed OLTP and OLAP workloads on a single machine in an integrated way [KN11, AIA14, GKP+10]. While the proposed systems achieve extremely good performance and avoid the cost of moving
data around, they are too limited because they cannot scale beyond a single machine. As applications get more demanding, it is critical to go beyond the processing capabilities of a single machine.

The purpose of this chapter is to present the *analytics in motion* (AIM) system which was designed to process mixed workloads with high volume of events, a high and diverse analytical query workload and strict response time guarantees for event stream processing and analytic queries. The system must scale as well as distributed systems such as Storm and Hadoop and it must be as efficient as the integrated approaches that process mixed workloads on a single machine. AIM was specifically designed for the needs of the telecommunication industry and it is a core building block in a number of Huawei products. However, we are convinced that its general ideas apply to many verticals.

Figure 4.1: AIM Architecture

Figure 4.1 shows the architecture of AIM which is in essence a simplified version of the *SQL-over-NoSQL* architecture that we will introduce in Chapter 5. The AIM architecture
Chapter 4. Analytics in Motion

is composed of three layers: (a) an event stream processing layer (ESP), (b) a storage layer implemented as an in-memory key-value store and (c) a SQL query processing layer for real-time analytics (RTA). Each of these layers scales separately as the processing requirements increase. For instance, if the event load increases (more CDRs per second), machines are added to the event stream processing layer.

While the architecture of Figure 4.1 has a number of crucial advantages, it also imposes a number of new challenges. One particular challenge is to implement the storage layer in such a way that it can sustain both, the read/update (get/put) workload of the event stream processing system and at the same time the bulk read workload of the real-time analytics. AIM addresses this challenge by combining a number of techniques that were recently designed for main-memory database systems: (a) shared scans in order to sustain the high workload [UGA+09]; (b) deterministic scheduling of read and write operations in order to control consistency [TA10, HAMS08, KKN+08, SW13]; and (c) a novel columnar storage layout that is based on PAX in order to meet the response time goals of analytic queries [ADHS01]. Furthermore, AIM natively supports the analytics matrix, which is a materialized view on the CDR events and pre-computes hundreds of indicators.

The remainder of this chapter is organized as follows: Section 4.2 draws the design space for systems implementing the AIM architecture. Following up on that, Section 4.3 explains which point in this space we decided for and why and also highlights some implementation details. Section 4.4 evaluates AIM with what it was designed for: the Huawei-AIM benchmark. Finally, we draw some conclusions from this evaluation in Section 4.5 and sketch how the lessons learned from AIM could be used in order to implement a system that also includes transactions on stateful streams.

4.2 Combining Stream Processing and Analytics

As we have seen in Subsection 2.2.2, the data warehousing approach [KR11] works well as long as the data in the warehouse can be refreshed every couple of minutes or hours. However, what we want to achieve with AIM is analytical query processing on real-time data, i.e., data not older than something in the range of one second. Moreover, our design goal is to minimize TCO and maintaining two separate stores strongly contradicts this goal. As already mentioned, our proposed architecture that employs a single shared store
faces several challenges that call for subtle design decisions. This section further details some of these challenges and illustrates different options to address them, thereby drawing the design space for implementations of the AIM architecture. We do not claim that this enumeration of options for different dimensions is complete, neither was there enough time to (experimentally) contrast pros and cons of each and every option. Nevertheless, we try to argue why in some cases we preferred one option over another for our AIM implementation.

### 4.2.1 Separating Updates from Query Processing

As we have a single store shared by ESP and RTA, we need to solve the problem how to process updates (puts) in a way that they do not interfere with longer-running analytical queries. Recent research has proposed two different solutions to this problem, both of which are shown in Figure 4.2.

**Copy-on-write**, also referred to as lazy copying, is the mechanism employed by most modern operating systems to efficiently manage the initially common memory state of parent and child process after a fork system call. Systems like HyPer [KN11] used this OS mechanism to manage different snapshots of their database. While puts are processed by the parent process on the most current version of the data, analytical query processing happens in the child process(es) on older snapshots.

**Differential updates** is a mechanism proposed by Krueger et al. [KKG+11]. Their idea is to accumulate all incoming puts in one data structure (called delta) and to process
analytical queries in a separated structure (main). Periodically, the delta records are applied to the main, which is referred to as merge. If the response time for puts is a critical factor, we can maintain two deltas, one for new puts and one for records currently being merged, and atomically switch their reference at the starting point of a merge. This approach also guarantees snapshot isolation for the analytical queries as they work on a slightly outdated, but consistent version of the data.

Our current AIM system employs a slightly modified differential updates technique\(^2\) instead of copy-on-write, the rationale for this being that the SLAs on ESP are so rigorous that the overhead caused by page faults in copy-on-write is unacceptable, at least on vanilla Debian-based Linux systems like the one we use in our evaluation.

### 4.2.2 Thread Model

As stated in the introduction, our architecture features a distributed key-value store, which means that it must support get and put functionality, \(i.e.,\) single-row lookups and updates. In addition to that, we expect our store to support a fast data scan in order to achieve reasonable throughput and response time for RTA processing. This raises the question of how to best utilize the available CPUs. We identify two options: (a) process RTA queries in a multi-threaded way, \(i.e.,\) employ a separate scan thread for each incoming query and possibly use a thread pool for recycling and (b) partition the data, thereby assigning one scan thread for each partition. Incoming queries are batched and then processed by all scan threads in parallel, each of them performing a \(shared\) \(scan.\) As the second approach gives us extremely high throughput and guaranteed latency [GAK12], this is the approach we chose for AIM.

An alternative to a fixed thread-partition assignment is to partition the data into many small chunks at the start of a scan and then continuously assign chunks to idle threads until every chunk is processed. This is a simple load-balancing mechanism that overcomes the problem that partitions could become imbalanced. This approach, also known as work stealing [BL99], comes at the additional cost of chunk management.

\(^2\)In contrast to the original proposition by et al. [KKG\(^+\)11], our delta and main data structures are indexed, which allows a more efficient merge step as no sorting is required. Our merge step is further simplified by omitting dictionary compression (which would only be beneficial if we would have strings in our analytics matrix.)
4.2. Combining Stream Processing and Analytics

4.2.3 Architecture Layering

We consider different options to physically place the three architecture components shown in Figure 4.1. Although logically separated, it is an option to place ESP and RTA functionality and a storage partition on the same physical node. This approach, to which we refer as the fully integrated approach, has the advantage of fast data access through local memory. However, we lose the advantage of the clear separation between storage and processing, which is flexibility. The fully separated approach (three separated layers) is more flexible in the sense that it allows provisioning resources in a more fine-grained manner. For instance, if we need faster storage access, we just add nodes to the storage layer, leaving the ESP and RTA processing layers unchanged. Obviously, there is a wide range of hybrid models that lie in between fully integrated and fully separated architecture layering. Our AIM implementation actually follows such a hybrid approach (further described in Subsection 4.3.2) in order to get as close as possible to the primary optimization goal which is to reduce TCO while satisfying all SLAs (cf. Subsection 3.1.5.4).

4.2.4 Data Placement and Join Processing

While the analytics matrix is distributed among the different storage nodes, the question where to store and maintain the rest of the AIM data structures remains. It makes sense to place the ESP Business Rules on the node(s) where the rule evaluation happens, which means to replicate the rule set in several places. Another question is where to place the dimension tables and it is closely related to the question where and how to do the join processing. Executing joins in the storage layer is fast as it happens closer to the data while executing joins in a separate processing layer allows for more flexibility in the overall design and is preferable if the storage nodes become overloaded. As the dimension tables are relatively small and static, we can replicate them at each storage node and inline the dimension records into the entity records. This approach, also known as data de-normalization [KR11] is the one that we used in the AIM implementation reported in this chapter.
4.3 AIM Implementation

We implemented our in-memory AIM system in such a way that it should be able to meet the default SLAs of the Huawei-AIM benchmark (cf. Section 3.1). These default values are summarized again in Table 4.1. The system is expected to scale well for a scaling factor between 1 and 10, i.e., a number of entities between 10 and 100 million.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{ESP}$</td>
<td>10 milliseconds</td>
<td>$f_{ESP}$</td>
<td>10,000 $sf$ events per second</td>
</tr>
<tr>
<td>$t_{RTA}$</td>
<td>100 milliseconds</td>
<td>$f_{RTA}$</td>
<td>100 queries per second</td>
</tr>
<tr>
<td>$t_{fresh}$</td>
<td>1 second</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Default SLAs as used in AIM Implementation

4.3.1 Observations

Let us start with some general observations about the workload of the Huawei-AIM benchmark: (a) the OLTP workload (generated by the event stream) consists of single-row transactions always referring to the primary key (entity-id), which simplifies finding the location of the corresponding record, (b) the analytics matrix uses the same primary key and can hence easily be horizontally partitioned in a transparent way, (c) RTA queries are read-only and can therefore be executed on a read-only snapshot of the analytics matrix, (d) the set of business rules and the dimension tables are small relative to the analytics matrix and are updated infrequently, which is why they can be replicated with minimal overhead.

4.3.2 System Architecture

The 3-tier architecture of our AIM implementation is depicted in Figure 4.3. It can be seen as a client-server architecture where the storage component acts as a server and the RTA and ESP component nodes are clients.

We decided to use a dedicated storage layer to store our data structures. As such, it hosts the analytics matrix and the dimension tables. Note that the analytics matrix is distributed (i.e., horizontally partitioned by entity_id) among all storage nodes, while dimension
4.3. AIM Implementation

tables are replicated at each node. Partitioning the analytics matrix was beneficial because we wanted to speed up RTA query processing by scanning the analytics matrix in parallel on different nodes. However, as we wanted to reduce communication cost between server and clients, we opted for replicating dimension data at each storage node which allows performing joins locally. This is valid because the dimension tables are assumed relatively static as mentioned above.

![Logical 3-tier Architecture of AIM Prototype]

At the bottom of Figure 4.3, we have the RTA nodes that are in fact lightweight processing nodes that take a query, redirect it to all storage nodes and, later on, merge the partial results before delivering the final result to the end user. As the bigger part of the RTA query
processing happens on the storage nodes anyway, we need far fewer RTA nodes than storage nodes. In addition, as RTA nodes are stateless, they can scale independently on demand.

Above the storage nodes, we can see the ESP processing nodes. In contrast to the lightweight RTA processing nodes, they have a much bigger CPU load as they process events and evaluate business rules, thereby using the storage nodes only for getting and putting entity records. Each ESP node has a copy of the entire set of business rules and can optionally use a rule index in order to make evaluation faster. Like RTA nodes, ESP nodes are essentially stateless and can hence scale independently on demand.

Communication between ESP and storage nodes happens synchronously (using the get/put interface), while communication between RTA and the storage is asynchronous (answers are sent whenever they are available). In order to be able to test different installations we implemented AIM for two different communication protocols: TCP/IP and Infiniband [Inf16]. As Infiniband provides very low network latency, we prefer Infiniband wherever possible.

The fact that the logical design of our implementation has 3 tiers does not imply that the physical design has 3 tiers as well. In fact, we tested two configurations for the ESP/storage nodes layout and their interaction: (a) separate physical tiers and communication using RDMA over Infiniband and (b) placement at the same physical machine (on different cores) and communication through local memory. While (a) is very beneficial in terms of flexibility of the whole system, (b) helps to tweak the system for the last bit of performance because instead of sending relatively large entity records (3 KB) over the network, we can simply send the considerably smaller events (64 B). Consequently, performance results for option (b) were slightly better and we chose this option for our main performance evaluation.

### 4.3.3 Updating the Analytics Matrix

Recalling Algorithm 3.1, we know that each attribute group of the analytics matrix has its own, customized update function. Whenever we add a new attribute group to the analytics matrix, we construct its update function by combining different (templated) building blocks from a small C++ kernel. This kernel consists of code for different event extraction functions (which allow to extract certain event attributes like time stamp, cost
or duration of a phone call), different aggregation functions (min, max, sum, count) for different data types (integer, long, float, double) and different window semantics. This essentially means that creating a new attribute group (which happens rarely, typically only during the setup phase of an experiment) is a bit expensive because we have to construct its update function with a huge nested switch statement. However, the resulting function (which will be called many times, essentially once per event) is highly efficient because (a) it is free of conditional jumps and (b) it is called by simply de-referencing a function pointer, which again prevents the CPU pipeline from stalling due to branch mis-prediction.

4.3.4 Business Rule Evaluation

As the set of business rules is relatively static, it makes sense to consider indexing them in order to make rule evaluation fast. We therefore implemented a rule index based on the ideas of Fabre et al. [FJL+01]. Interestingly, it turned out that for the 300 business rules that we have in our benchmark, this index is no faster than just processing rules without an index in a straight-forward manner as shown in Algorithm 3.2. A micro-benchmark in which we varied the number of rules showed that using a rule index started paying off for a rule set size of about 1000 and above [Gas13, p. 26]. We conclude that, as long as the rule set is small, we can avoid using an index at all, thereby eliminating the complexity of maintaining such an index.

4.3.5 ColumnMap

As mentioned in the introduction, the analytics matrix is implemented as a distributed in-memory key value store. Early experiments showed that for achieving the SLAs of ESP, RAMCloud [OAE+11] works well as a key value store [Gas13]. RAMCloud not only provides fast record lookups and writes, but also supports durability and fault tolerance as it follows a log-structured design. However, just as with any row store, we cannot get to a fast enough scan speed for RTA query processing. In order to allow for a high-speed scan, traditional analytical query processing engines use a column-oriented storage layout, which in turn is not particularly well-suited for high update rates because of poor locality of the attributes of a record.
Chapter 4. Analytics in Motion

Our solution to overcome this dilemma, is to use the partition attributes across (PAX) approach [ADHS01]. This is an attempt to find the sweet spot between purely row-oriented and purely column-oriented storage layouts. The original idea of PAX was to group records into chunks that fit into a memory page and within a page store records column-wise, i.e., values of a particular attribute are grouped together. Analytical queries that usually process a small subset of the attributes can then profit from data locality as well as from the fact that the entire records of a chunk are present in memory at the same time. We designed ColumnMap, a data structure that follows this design with the difference that it is optimized for cache size rather than the size of memory pages. This makes sense as all data structures in AIM are held in memory anyway.

The basic structure of ColumnMap is depicted in Figure 4.4. We group a fixed number of records into logical blocks called buckets. In our current implementation, the default number of records per bucket (bucket size) is 3072. Within a bucket, data

We chose 3072 because this is the highest power of two such that a bucket, consequently having size $3072 \times \text{record size (3 KB)} \simeq 9 \text{ MB}$, fits into the 10 MB L3 cache of our hardware.

62
4.3. AIM Implementation

is organized into columns. Each column holds the values for a particular attribute, *e.g.*, `all_cost_this_month`. This approach allows increasing columnar locality, which is beneficial for scan processing of individual attributes (cf. Subsection 4.3.7). In addition to these buckets, ColumnMap features a small hash map that keeps track of the mapping between the key (`entity_id`) and the actual record identifier. The reason for this level of indirection is the fact that in general keys can be arbitrary application-dependent identifiers, while the record identifiers are contiguous numbers starting from 0. Since the records are of constant size and each bucket consists of a constant number of records, we can compute the address of a specific field of a record from its record identifier. This makes lookups for single fields really fast.

It is worth mentioning that we can also configure ColumnMap to work as a pure row store (by setting the bucket size to one) or as pure column store (if bucket size = total number of entity records). In fact, ColumnMap only outperforms a column store with respect to its update performance when records are small enough to fit into a cache line. If they do not (as in our use case where we deal with huge 3 KB records), bucket size does not have such a huge impact, neither on RTA nor on ESP performance (cf. Subsection 4.4.2) and we could as well use a pure column store. There are two reasons why we prefer ColumnMap over using an off-the-shelf column store in our system: (a) it has the tunable parameter `bucket size`, which makes it a row, column and hybrid store at the same time and therefore enhances flexibility and (b) we have direct access to the raw data in the store without the need of going through a SQL interface.\(^5\)

4.3.6 Keeping Track of Updates

As stated in Subsection 4.2.1, we have to ensure that updates (puts) produced by ESP do not interfere with RTA queries because these queries should return a consistent result and therefore work on a consistent snapshot of the analytics matrix. In order to solve this, we implemented a variant of differential updates [KKG++11].

As we cannot afford to block the ESP subsystem at any time, *e.g.*, during the *merge* phase, we have to allocate the new *delta* right before merging, which means that we have two

\(^4\)However, bucket size must be large enough to ensure buckets saturate the SIMD registers, see Subsection 4.3.7.1.

\(^5\)While there are some mentionable exceptions like Google Supersonic [Goo16b], most available off-the-shelf column stores do not expose their internal data structures.
deltas during the merge phase. Get and put operations have to be adapted accordingly, as shown in Algorithms 4.1 and 4.2. Note that these algorithms test whether variable newDelta exists in order to determine if there is currently a merge being performed (newDelta exists) or not (newDelta does not exist). As the algorithms are not thread-safe, we perform gets and puts of a specific entity by one dedicated ESP thread. This decision also allows updating entity records atomically by employing conditional write, which is an important functional requirement (cf. Subsection 3.1.3.2).

```
1: function Get(UUID id)
2:     EntityRecord result ← NULL
3: if ∃ newDelta then
4:     result ← get(newDelta, id)
5: if result = NULL then result ← get(delta, id)
6: if result = NULL then result ← get(main, id)
7: return result

Algorithm 4.1: Analytics Matrix Get
```

```
1: function Put(EntityRecord r, EntityID id)
2: if ∃ newDelta then
3:     put(newDelta, r, id)
4: else
5:     put(delta, r, id)

Algorithm 4.2: Analytics Matrix Put
```

In fact, as mentioned in Subsection 4.2.1, we pre-allocate two deltas at startup and always use one of them as new and the other as old delta. This means that new delta allocation effectively becomes swapping the two delta pointers and resetting the new delta, which saves us a lot of memory management and is blazingly fast. However, this delta switching process has to block the ESP thread for a short moment in order to make sure it will deliver correct results (Algorithm 4.1), respectively write to the right delta (Algorithm 4.2). This can be implemented very efficiently with two atomic flags. We point the interested reader to the appendix of our SIGMOD paper [BEG+15] for further details.

Each get returns a record that comes along with a time stamp indicating when this record was last modified. The conditional write function takes this time stamp as an additional argument and only overrides a record if its time stamp matches this argument. If there is no match, an error is returned, which triggers the ESP node to restart the single-row transaction for the current event.
It is worth mentioning that AIM favors hot spot entities as this means that corresponding entity records might be overridden several times in the delta and therefore automatically compacted before being written to the main.

As the delta should be optimized for single-row transactions, we implemented it using Google’s dense hash map [Goo]. Additionally, the main must feature a fast scan and needs to be indexed in order for the single-row transactions to work at considerable speed. The index on the primary key (entity_id) is also a requirement for an efficient implementation of the merge where we want to be able to do a single pass through the delta, lookup the positions of the corresponding records in the main and simply replace them. We implemented the main as ColumnMap, which is in our case an optimal fit as explained in Subsection 4.3.5. There remains the question when and how often we should merge. In order to prevent the delta structure from growing too large, it would be beneficial to merge as often as possible. On the other, merge steps interrupt RTA query processing and therefore the right moment for merging has to be chosen carefully. Luckily, the merge step can be interleaved optimally with query processing as we show next.

### 4.3.7 Query Processing

Traditional database systems process one query at a time. Inspired by SharedDB [GAK12], we try to achieve a higher throughput by using a batch-oriented processing technique instead. Our storage server keeps a queue of queries that were submitted by the RTA clients. Once a new scan is started, all queries currently in the queue are processed together in one single scan pass. Such a shared scan reduces queuing time for individual queries and at the same time increases query throughput. Moreover, the batch-oriented query execution model nicely fits our delta-main storage layout because scan and merge steps can be interleaved. An RTA query processing thread therefore works in a loop with the following two steps as illustrated in Figure 4.5:

**scan step** scan the entire main (ColumnMap) as shown in Algorithm 4.3. During that phase the main is read-only and therefore concurrent accesses by the ESP thread (performing Gets) and the RTA thread are safe.

---

8 In order to satisfy the real-time SLAs on both, response time and throughput, batch size, as will be shown later, has to be chosen carefully.
merge step  the RTA thread scans the delta and applies the updates to the main in-place. At the beginning of this step, the delta becomes read-only as Puts are redirected to the newly allocated delta (see Subsection 4.3.6). The ESP thread never reads an item that the RTA thread is currently replacing, simply because if an item is currently updated in the main, this means it must also exist in the delta, which implies the ESP thread will get it from there and not from the main (cf. Algorithm 4.1).

4.3.7.1 Single Instruction Multiple Data (SIMD)

Many current processors feature explicit single-instruction-multiple-data (SIMD) machinery such as vector registers and specialized instructions to manipulate data stored in these registers. They allow for one instruction to be performed on multiple data points in parallel. Intel's streaming SIMD extensions (SSE) and advanced vector extensions (AVX) operate on registers of up to 512-bit side. The size of these registers allows concatenating up to 8 floating-point operands into a single vector and processing arithmetical or logical operations in parallel.
As the research community already pointed out, SIMD instructions allow an additional degree of parallelism and often eliminate some of the conditional branch instructions by reducing branch mis-predictions [WPB09, ZR02]. This makes SIMD instructions very useful for high-performance databases that are more often CPU-bound than memory-bound due to the increase in RAM capacities. We therefore exploited SIMD instructions to build a fast scan on ColumnMap. This scan includes filtering (selection) and aggregation (projection) as illustrated in Figure 4.6:

**Filtering:** Filtering with SIMD instructions means to first load a column into one vector register and the operand in the other register and then perform a SIMD comparison instruction (e.g. SIMD_<). This results in a bit mask that states whether to include a value in the result (0xF..F) or not (0x0..0). We combine bit masks from different filters by either SIMD_& or SIMD_| according to the WHERE clause of the query.

**Aggregation:** We intersect (SIMD_&) the data vector with the bit mask resulting from filtering and then apply an aggregation operator (SIMD_MIN, SIMD_MAX or SIMD_+).

As becomes clear, such SIMD instructions can only be used in conjunction with a columnar storage layout like ColumnMap. Therefore we have to choose *bucket size* large enough to fill the entire SIMD registers with attribute values. For 256-bit SIMD registers and attribute values of at least one byte, we would need at least \( \frac{256}{8} = 32 \) records per bucket.

### 4.3.8 Distributed Execution

As explained in Subsection 4.2.2, we not only distribute the *analytics matrix* among different nodes, but also partition it further within a node as shown in Figure 4.7. There are two parameters that determine resource provisioning: number of ESP threads \( e \) and number of RTA threads \( n \) where \( n \) also equals the number of data partitions. Each RTA thread is related to exactly one data partition while an ESP thread works on the delta of several (up to \( k \)) partitions. In our implementation we used the simple strategy to first choose \( e \) large enough to achieve the SLAs of ESP and then use the remaining cores for RTA processing and communication. As 2 threads for communication with the other two tiers are used, this means \( n = \text{number-of-cores} - e - 2 \). Note that we use the terms...
core and thread interchangeably here as we have as many threads as cores. This is to avoid the performance penalty of over-subscription (cf. Subsection 4.4.2). What is more, our implementation is NUMA-aware in the sense that RTA threads are pinned to different cores in such a way that they are collocated with their assigned data partition and can therefore access it through local memory.

Routing a get or put request to the correct data storage partition works as follows: first, use a global hash function $h$ to route the request to the node with the node identifier $h(key)$. Next, within node $i$, apply a node-specific hash function $h_i(key)$ to determine the identifier of the partition that hosts this key. Finally, route the request to the ESP thread responsible for this partition.

The distribution of data immediately raises the question of consistency. We implemented intra-node consistency by coordinating the start of the scan step for all RTA threads on a storage node. This is also beneficial because if all threads start at the same time, they
can work on the same query batch. It is not necessary to provide inter-node consistency as events do not have a global order. Distributed transactional consistency is a clearly more complex problem that we will shortly look into in the beginning of Chapter 5 where we will revisit the ideas described by Loesing et al. [LPEK15].

4.4 Experiments

This section describes the evaluation of our AIM system with the standard Huawei-AIM benchmark as detailed in Section 3.1. As stated in the beginning of Section 4.3, the AIM system should be able to cope with an event rate of 10,000 events per second per ten million entities and scale from 10 million to 100 million entities. Following the methodology proposed in Subsection 3.1.5.4, we first executed a number of experiments to determine the optimal combination of system parameters to execute the benchmark for the default workload parameters and SLAs: First, we varied the implementation-specific parameters bucket size and \( n \), the number of data partitions, respectively number of server-side RTA threads on a single storage node (Subsection 4.4.2). Next, we tested the robustness of our system by increasing the RTA load (Subsection 4.4.3), which also allowed us to conclude the optimal number of RTA client threads, \( c \). In addition, as this experiment was still single-node, we used it to compare the AIM system against its
closest counterparts from research and industry. Finally, we tested the AIM system’s capability to react to changing workload parameters. Subsection 4.4.4 examines the flexibility of AIM to react to changing SLAs, concretely to more stringent values of $f_{RTA}$ or $t_{RTA}$ while Subsection 4.4.4 demonstrates its ability to cope with an increasing scaling factor $sf$. All these experiments taken together will allow us to conclude this section with a statement about the primary figure of merit of the Huawei-AIM benchmark, the total cost of ownership (TCO).

Otherwise, we will mainly report the secondary figures of merit, average end-to-end response time and overall query throughput of RTA queries because we need them to determine whether and how SLAs can be met. As the event rate was configured to meet $f_{ESP}$, we only report measured ESP throughputs that deviated from the event target rate. $t_{ESP}$ was always met and is therefore excluded from the results. SLAs are depicted as dotted lines in the performance graphs. Where not mentioned otherwise, we used the following default parameter values: $sf = 1$, $f_{SEP} = 10,000$ events/sec, 8 ESP client threads, 8 RTA client threads ($c = 8$), 1 ESP per-server thread ($s = 1$), 5 RTA per-server threads $n = 5$ (number of data partitions), a bucket size of 3K (3072 records per bucket) and 1 AIM server (storage node).

### 4.4.1 Setup

Our experiments were conducted on servers equipped with a dual-socket 4 core Intel Xeon E5-2609 CPU, each core operating at 2.40 GHz. Each server features 32 KB L1 cache, 256 KB L2 cache and 10 MB L3 cache as well as 4x32GB Samsung DDR3-DIMM, resulting in a total of 128 GB RAM. We used a vanilla Debian Linux 4.6.3-1 running kernel 3.4.4 and GCC-4.7.2 and communicated over a 40 Gbit QDR Infiniband network.

As explained in Subsection 4.3.2, we decided to host ESP and storage on the same physical nodes (communicating through local memory) and RTA processing nodes separately. We used one dedicated machine for generating random events and measuring end-to-end throughput and response time of the ESP workload. This machine could be configured to send events at a certain rate (as specified by the benchmark). The RTA clients, responsible for the creation of random RTA queries and end-to-end measurements of throughput and response time, were be placed on a single node, using $c$ execution threads. As preliminary results revealed, one node for RTA processing was
4.4. Experiments

enough to saturate up to 10 storage server nodes and it could therefore be collocated with the RTA clients on the same physical machine.

4.4.2 Optimal Number of Data Partitions and Bucket Size

In the first experiment, we tried to find the best number of storage partitions (= RTA server threads) as well as the optimal bucket size for the used ColumnMaps. Figure 4.8 shows response time and throughput for different combinations of these two parameters on a single storage server. As hypothesized in Subsection 4.3.8, we get optimal performance when allocating exactly as many threads as there are cores. As we have one ESP thread and two communication threads, this means 5 RTA server threads on an 8-core machine. Moreover, we can see that with 4 and 5 partitions, all SLAs are met. For $n = 6$, the ESP throughput was below 10,000 at about 8,000 events/sec (similar for all different bucket sizes), which is a direct consequence of the thread thrashing at the storage nodes. The experiment further indicates that bucket size does not have a major impact on performance as long as it is large enough. Notice that ColumnMap (for different bucket sizes) slightly outperforms the pure column store (which is the line denoted by all). We conclude that the optimal setting for further experiments is 5 data partitions per storage server and a bucket size of 3K (3072).

Figure 4.8: RTA Strong Scale-Up on AIM: $sf = 1$ (10M entities, 10,000 events/sec), $c = 8$, vary $n$ from 1 to 6, different bucket sizes (1K, 3K, 16K, all = pure column store)
4.4.3 Peak Performance of AIM compared to other Systems

RTA clients work in a closed loop and submit only one query at the time which is why their number is also an upper-bound on the query batch size at the storage server(s). If we want to test the robustness of our AIM system, we can therefore simply increase the RTA load by varying $c$ from 1 to 16 as shown in Figure 4.9. We see that AIM is robust in the sense that once saturation is reached (somewhere between 8 and 16 threads), throughput stays constant but does not drop. Response time increases linearly, but not exponentially, just as we would expect. The fact that we reach peak performance with 8 threads, \textit{i.e.}, satisfy both RTA SLAs ($t_{RTA} < 100$ milliseconds and $f_{RTA} > 100$ queries/second), suggests to limit query batch size at the storage server to about 8, respectively to use $c = 8$ for further experiments.

In order to rank the secondary figures of merit, we compared AIM to two commercial database products: \textit{System M}, a main-memory-based column store optimized for real-time analytics and \textit{System D}, a disk-based row-organized database system with support for fast updates. Even though we implemented \texttt{update\_matrix(.)} as a stored procedure, both systems clearly failed in handling the required amount of events: System M could handle about 100, System D about 200 events per second. This is why we only measured their (read-only) RTA performance. To achieve the best possible query and update performance for System D, we let it use its \textit{index advisor} to create indexes on the relevant columns despite the benchmark forbidding precisely this. In addition to these commercial systems, we also executed the benchmark on \textit{HyPer} (the version described in the 2011 paper [KN11]) which seems to be the best fit for the Huawei use case among the available research prototypes because it processes analytical queries on consistent data snapshots with configurable freshness. In contrast to systems M and D, \textit{HyPer} could handle a reasonable number of events per second (5,500 in isolation and 1,940 with one concurrent RTA client). As systems M and D, as well \textit{HyPer} are database management systems with no special support for event stream processing, we only tested them with the \textit{Huawei-AIM-Simple} benchmark (\textit{i.e.}, without campaign processing).

Figure 4.9 shows that AIM clearly outperforms all three competitors (system M and D and \textit{HyPer}) by a factor of at least 2.5 for RTA response time and throughput. Given the fact that RTA performance of the competitors was measured in isolation (without concurrent ESP processing) and only with the Huawei-AIM-Simple benchmark, we think that this
improvement is quite significant. In addition, AIM achieves an event processing rate improvement ranging from 5x (over HyPer) to two orders of magnitude (over systems M and D).

As far as HyPer is concerned, it is worth mentioning that its newest version that does not use forking, but physical MVCC [NMK15], is able to beat AIM’s performance under certain circumstances. As shown in our streaming analytics survey [KPB+17], HyPer can interleave the execution of several pending RTA queries and thereby achieve a 27% RTA throughput improvement over AIM in an RTA-only execution of the Huawei-AIM-Simple-Standalone benchmark.

### 4.4.4 Strong Scale-Out

The previous experiments illustrated that one storage server is enough to accommodate 10M entities. However, as SLAs might become more stringent, e.g., $t'_{RTA}=12$ milliseconds and $f'_{RTA}=725$ queries per second, it is important to know whether provisioning more resources would solve the problem. In order to analyze this, we increased the number of storage servers from 1 to 10 as illustrated in Figure 4.10. We see a near-linear increase in throughput as well as response time. We conclude that it is possible to strongly scale out with a satisfactorily small overhead.
Figure 4.10: RTA Strong Scale-Out on AIM: $s_f = 1$ (10M entities, 10,000 events/sec), vary AIM servers from 1 to 10

4.4.5 Weak Scale-Out

The last experiment concerns weak scale-out, or in other words, how the performance measures change if we do not only increase the number of servers, but also the load (scaling factor $s_f$) accordingly. In this experiment, for each added server, we also added 10M entities and 10,000 events per second. Figure 4.11 shows a decent weak scale-out. Ideally, throughput and response time would be horizontal lines. The fact that they are not illustrates the two types of overhead that additional servers create: (a) synchronization overhead and (b) result merging. The more AIM servers, the more partial results have to be collected and merged at the RTA node. As long as the RTA node is not fully utilized, we can sacrifice some response time to improve throughput scale-out by adding more client threads at the RTA node. As we can see from the dotted lines in Figure 4.11, increasing $c$ from 8 to 12 for 60M Entities and beyond helps us to stay sharply within the SLA limits. If we want to go beyond 100M customers (and 100,000 events/sec), we can still stay within SLA limits, but may have to provision more servers machines per entity (cf. Subsection 4.4.4). If the ESP or RTA layer become the bottleneck, they can scale out individually.

4.4.6 Total Cost of Ownership

In conclusion, the AIM architecture is particularly well-suited for provisioning resources just when and where they are needed (depending on changes in the workload and/or
4.5 Concluding Remarks

This chapter introduced AIM, a new architecture for addressing systems with stringent requirements on streaming, frequent updates and execution of analytical queries in real-time as proposed in the Huawei-AIM benchmark. We discussed the design space of such an architecture and implemented the AIM system, a distributed and flexible implementation for a specific workload that features a novel combination of well-established technologies. This exactly matches the optimization goal of elasticity as defined in the introduction of this thesis. It also implies that the total cost of ownership is minimized, meeting another expectation of the introduction. In the concrete case of the AIM system, the TCO for satisfying the given SLAs for \(10 \times sf\) million entities (for \(1 \leq sf \leq 10\)), is the operation costs of \(sf + 2\) of our cluster machines (\(sf\) AIM storage nodes, 1 ESP node, 1 RTA node) plus the cost of the Infiniband network connecting them. In addition, as Subsection 4.4.4 revealed, AIM can easily be configured to satisfy more stringent SLAs, concretely \(t'_{RTA}=12\) milliseconds and \(f'_{RTA}=725\) queries per second, with only a 4-fold investment.\(^9\)

\(^9\)The TCO for meeting the default SLAs for \(sf = 1\) is roughly \(sf + 2 = 3\) machines while increasing the number of AIM servers to 10 resulted in \(1 + 1 + 10 = 12\) machines used. Dividing the latter by the former and being aware that adding network costs above and below the fraction bar only makes the fraction smaller, yields the estimate of an investment of \(12/3 = 4\).

Figure 4.11: Weak Scale-Out of RTA Queries on AIM, vary AIM servers along with scaling factor \(sf\) from 1 to 10
principles, such as materialized view, horizontal data partitioning, shared scan, the PAX paradigm, efficient distributed super-scalar query execution with SIMD and a new variant of differential updates for real-time data management.

The experimental evaluation of the AIM system with the Huawei-AIM benchmark showed that we can indeed meet the required SLAs with minimal resources (TCO) and beat commercial analytical database systems by a factor of 2.5 in analytical and two orders of magnitude in update performance. This minimal resource allocation features two processing nodes and one storage server node per 10M Entities up to 100M Entities and 100,000 events per second. Moreover, we demonstrated that ESP, RTA and storage layer scale linearly and independently from each other, which allows the system to scale far beyond 100M entities with a minimum of additional resources, hence meeting the elasticity, scalability and TCO optimization goals as stated in the introduction of this thesis. The introduction also claimed that AIM can trade-off between data freshness and analytical performance. AIM achieves this by changing the number of consequent scan steps it executes between two merge steps. The less often we merge, the better our analytics performance gets (at the cost of reduced data freshness).

The production version of AIM which was deployed as beta version within Huawei includes more features that are out of the scope of this thesis. For instance, this production version consists of a persistent event archive, a special PAX-based main-memory layout that includes support for variable-length data, incremental check-pointing and zero-copy logging, as well as policy-enforced and self-balancing queuing mechanisms for handling skews in the ESP subsystem and multi-threaded NUMA-aware main memory real-time analytics. These features are mostly used for making the system available and durable while preserving its performance.

As we will see in the next chapter of this dissertation, the work on AIM can be extended to make the system able to cover a wider range of OLTP/OLAP workloads with stringent real-time requirements that include transactions, e.g., the YCSB# benchmark. We will also see that our distributed storage server with its get/put/scan interface can serve as an important building block for a more general OLTP/OLAP engine built atop, e.g., Tell [LPEK15]. The only additional features needed for AIM’s ColumnMap to become useful as a storage back-end for Tell is the support for versioning (simultaneously keeping several versions of a record) and variable-sized fields.
Fast Scans on Key-Value Stores

After having learned some best practices about how to build a systems specifically designed to handle the streaming analytics workload of the Huawei-AIM benchmark, we want to go one step further and design a system suitable to even address streaming transactions. The strategy used to achieve this is to extend an existing high-performance OLTP database (Tell) that operates on a pure KV store (Ramcloud) with fast analytics by replacing Ramcloud with a scan-enabled, versioned KV store (TellStore) and making the necessary adjustments to Tell (resulting in a novel data management system called Tell 2.0).

The goal of this chapter is hence to explore the design space for such a “scannable” versioned KV store and evaluate different alternatives. As we will see, the data structure presented in the last chapter, ColumnMap, also exhibits very good performance if used as the primary component in such a KV store. It clearly outperforms alternative designs, like purely row- or column-based solutions or state-of-the-art high-performance KV stores.¹

¹Parts of this chapter are under submission to a systems conference [PBB+17].
5.1 Introduction

Key-value stores (KV stores or simply KVS) are becoming increasingly popular. Unlike traditional database systems, they promise elasticity, scalability and are easy to set up and maintain. Furthermore, the performance characteristics of a KVS are predictable: Each get/put request finishes in constant time. This feature helps to give latency guarantees and support SLAs for applications built atop.

Recent work [DNN+15, LPEK15, fou16] showed that KVS can be used to run OLTP workloads in an efficient and scalable way. All that work adopted a SQL-over-NoSQL approach in which the data is stored persistently and served using a KVS (i.e., NoSQL) and the application logic (with SQL support) is carried out in a separate processing layer. The big question that we would like to address in this chapter is whether such an architecture can support both, analytical workloads and OLTP workloads using the same KVS and if yes, how this can be achieved.

This question is relevant because the access patterns of OLTP and analytical workloads are different. The get/put interface of most KVS is sufficient for OLTP workloads, but it is not a viable interface for analytical workloads because these workloads involve reading a large portion, if not all, of the data: Retrieving data points one-by-one with get requests incurs prohibitively high network latencies. As a result, systems for analytical workloads provide additional access methods: they either allow data to be fetched all at once (full table scan) or to push down selection predicates and/or projections to the storage layer. Most KVS do not have such capabilities and those that do, cannot execute scans with acceptable performance.

To illustrate that current state-of-the-art KVS are not well-suited for analytics, Table 5.1 shows the running times of executing YCSB# query 1 (cf. Subsection 3.2.3). For this experiment, we used four storage nodes (configured as will be explained in Subsection 5.5.1). It took Cassandra [LM10] about 19 minutes to process this simple query and RAMCloud [OAE+11] and HBase [Geo11] about half a minute. For these three KVS, we used server-side processing of the aggregate value because shipping all data and executing the aggregation at the client would have made the response times even worse. Given that the entire dataset fits into main memory, these running times are unacceptably high. The only system that had acceptable performance in this experiment was Kudu [Apa16a]. Kudu is a NoSQL, column-oriented data store specifically designed
### 5.1. Introduction

<table>
<thead>
<tr>
<th>Key-Value Store</th>
<th>Scan Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kudu [Apa16a]</td>
<td>1.8 seconds</td>
</tr>
<tr>
<td>HBase [Geo11]</td>
<td>36 seconds</td>
</tr>
<tr>
<td>RAMCloud [OAE+11]</td>
<td>46 seconds</td>
</tr>
<tr>
<td>Cassandra [LM10]</td>
<td>19 minutes</td>
</tr>
<tr>
<td>TellStore-Column</td>
<td>84 milliseconds</td>
</tr>
<tr>
<td>TellStore-Log</td>
<td>190 milliseconds</td>
</tr>
<tr>
<td>TellStore-Row</td>
<td>197 milliseconds</td>
</tr>
</tbody>
</table>

Table 5.1: Response Time of YCSB# Query 1 (max aggregation) on popular KV Stores and TellStore, for \( sf = 50 \) (50M records), running on 4 storage nodes

for analytics, but even Kudu is far away from what could be achieved. The support for this claim comes from the last three rows in Table 5.1 which show that the different implementations of TellStore, which will be presented in this chapter, outperform Kudu by another order of magnitude.

The poor performance of Kudu which was specifically designed to perform well for such queries shows that it is not easy to achieve fast scans on KVS. The problem is that there are a number of conflicting goals when supporting get/put and scan operations. Efficient scans require a high degree of spatial locality whereas get/put requires sparse indexes. As we will see, versioning and garbage collection are additional considerations whose implementation impact greatly the performance of a KVS for OLTP and analytical workloads. This chapter shows that with reasonable compromises it is possible to support both workloads as well as mixed workloads (like the ones described in Chapter 3) in the same KVS, without copying the data.

Specifically, this chapter makes the following contributions: First, we amend the SQL-over-NoSql architecture of Tell and show how it can be used to process mixed workloads (Section 5.2). Second, we present the design space for developing a KVS that supports both, point operations (get/put) and bulk operations (scans), efficiently (Section 5.3). Third, we present TellStore, a new, scalable, in-memory KVS (Section 5.4). Finally, we give the results of comprehensive performance experiments using the YCSB# and the Huawei-Simple benchmarks Section 5.5. The main result of this work is that it is indeed possible to build a KVS that has acceptable performance for get/put workloads and is highly competitive for analytical workloads.
5.2 Requirements for a Scannable Key-Value Store

This section shows how the SQL-over-NoSQL architecture of Tell can be extended to support mixed transactional/analytical workloads. This new architecture, Tell 2.0, though very promising, can only deliver its best performance if built on top of a KVS that fulfills a very specific list of requirements. As we will see, these requirements are partially contradicting, which makes the design of a suitable KVS for Tell 2.0 an intriguing problem.

5.2.1 SQL-over-NoSQL Architecture

Figure 5.1 depicts the SQL-over-NoSQL architecture to support mixed OLTP/OLAP workloads on top of a KVS as used in Tell 2.0. As in the AIM architecture (cf. Section 4.1), the data is stored in a distributed KVS which features a get/put and scan interface. Transactions and queries are processed by machines in the processing layer. The processing layer also synchronizes concurrent queries and transactions. Throughout this work, we have been using snapshot isolation (SI) [BBG+95a] for synchronization. SI can be implemented in a distributed setting using a commit manager as shown in the original Tell [LPEK15, Loe15]. This commit manager simply assigns transaction IDs and keeps track of active, committed and aborted transactions and, thus, rarely becomes the bottleneck of the system [LPEK15]. SI (or other forms of multi-version concurrency control) have emerged as the de-facto standard for synchronization; in particular, in distributed systems and for mixed workloads.

The big advantage of the SQL-over-NoSQL architecture is that it is elastic. Additional machines can be added to both layers (storage and processing) independently. For instance, additional OLAP nodes can be added at the processing layer to process a complex analytical query; these nodes can be shut down or re-purposed for other tasks once the query is completed. It is also possible to provision specialized machines for different tasks at the processing layer; e.g., machines with a great deal of main memory for analytical queries and machines with high I/O bandwidth for OLTP. This architecture also enables the efficient processing of HTAP workloads (cf. Subsection 2.2.4): Both kinds of workloads can run concurrently with dedicated resources on a single, fresh copy of the data.
5.2. Requirements for a Scannable Key-Value Store

To implement this SQL-over-NoSQL architecture that supports mixed workloads efficiently, the distributed KVS at the storage layer must meet a number of requirements:

**Scans** In addition to *get/put* requests, the KVS must support efficient scan operations. In order to reduce communication costs, the KVS should support selections, projections and simple aggregates so that only the relevant data for a query is shipped from the storage to the processing layer. Furthermore, support for shared scans is a big plus for many applications [Fer94, AIP+ 12, BGVK+ 06, UGA+ 09].

**Versioning** To support *snapshot isolation* (or other forms of multi-version concurrency control), the KVS must maintain different versions of each record and return the right version of each record depending on the time stamp of the transaction. Versioning implies the need for garbage collection to reclaim storage occupied by old versions of records that are guaranteed to be no longer needed for any active transaction or query.
Chapter 5. Fast Scans on Key-Value Stores

**Batching and Asynchronous Communication** To achieve high OLTP performance, it is critical that OLTP processing nodes batch several requests to the storage layer. This way, the costs of a round trip message from the processing to the storage layer are amortized for multiple concurrent transactions [LPEK15]. Furthermore, such batched requests must be executed in an asynchronous way so that the processing node can collect the next batch of requests while waiting for the previous batch of requests to the KVS to complete.

### 5.2.2 Why is it difficult?

The big problem of these three requirements is that they are in conflict. That is why most KVS today (with the notable exception of Kudu) have been designed to support get/put requests only (e.g., Cassandra and HBase), possibly with versioning (e.g., RAMCloud) and sometimes with asynchronous communication (cf. Section 2.3). All these features are best supported with sparse data structures: To read a specific version of a record as part of a get operation, it is not important that this record is clustered and stored compactly with other records. Scans, however, require a high degree of data locality and a compact representation of all data so that each storage access returns as many relevant records as possible. Locality is important for both, disk-based and in-memory scans. Specifically, adding scans creates the following locality conflicts:

- **scan vs. get/put:** Most analytical systems use a columnar storage layout to increase locality. KVS, in contrast, typically favor a row-oriented layout in order to process get/put requests without the need to materialize records [SAB+05].

- **scan vs. versioning:** Irrelevant versions of records slow down scans as they reduce locality. Furthermore, checking the relevance of a version of a record as part of a scan is prohibitively expensive.

- **scan vs. batching:** It is not advantageous to batch scans with get/put requests. OLTP workloads require constant and predictable response times for get/put requests. In contrast, scans can incur highly variable latencies depending on predicates selectivities and the number of columns needed to process a complex query.
Fortunately, as we will see, these conflicts are not fundamental and can be resolved with reasonable compromises. The goal of this chapter is to study the design space of KVS and to demonstrate experimentally which compromises work best.

5.3 Design Space

This section gives an overview of the most important design questions to build a KVS that supports bulk operations and scans as required by the SQL-over-NoSQL architecture of Tell 2.0.

5.3.1 Where to put Updates?

There are three possible designs to implement updates (put and insert requests): update in place, log-structured and delta-main. We shortly sketch the first two approaches and defer the reader to Subsection 4.2.1 for a description of delta-main.

update in place is the approach taken in most relational database systems, e.g., Oracle or the InnoDB storage system for MySQL. New records (inserts) are stored in free space of existing pages and updates to existing records (puts) are implemented by overwriting the existing storage for those records. This approach works great if records are fixed-sized and there is no fragmentation. However, this approach is trickier with versioning. If versions are kept in place as well (i.e., at the same location as the records), versioning can result in significant fragmentation of the storage and loss of locality. Another problem with the update-in-place approach is that it limits concurrency: To create a new version of a record, the whole page needs to be latched.\(^2\)

log-structured storage designs were first introduced by Rosenblum and Ousterhout [RO92] for file systems. This principle is also popular for KVS; e.g., RAMCloud has a log-structured design [OAE+11]. The idea is to implement all updates (puts and inserts) as appendes to a log. A variation, referred to as log-structured merge-trees (LSM) [OCGO96] is used in LevelDB [Goo16a], RocksDB [Fac16b] and Kudu [Apa16a]. This variation involves periodic reorganization of the log to improve read performance. Log-structured

\(^2\)A latch is a short-term lock that can be released once the page has been updated and need not be kept until the end of a transaction.
storages have two important advantages: (a) there is no fragmentation and (b) there are no concurrency issues as appends can be implemented in a non-blocking way [LLS+15]. A major drawback is that scans become very expensive in a pure log-structured storage system because the locality can become poor. Furthermore, it is difficult to garbage collect/truncate a log if some records are rarely updated. That is why most systems implement the LSM variant.

5.3.2 How to allocate Records?

Choosing the right memory allocation granularity is key to finding the right balance between the additional management load on get/put and scan operations. Clearly, neither storing all records in a single huge chunk of memory (which would have to be re-allocated whenever there are too many put requests), nor allocating every new record separately on the global heap (which would force the scan to iterate through the record index and perform one random memory lookup per entry) are viable solutions. A traditional compromise (described by Gray and Reuter [GR93]) is to pack data into fixed-sized blocks (usually called pages). This allows the scan to load whole blocks of data into cache and therefore improves run time drastically. Furthermore, it is much easier to implement a fast allocator/deallocator for fixed-size pages. While the original idea behind paging is to trade memory for disk accesses [GP87], we use it to trade get/put performance for scan speed: The bigger the page, the higher the scan performance, but the more page management work needs to be done during a put operation.

5.3.3 How to arrange Records?

The two most popular designs are row-major and column-major. Row-major means that a record is stored as a contiguous sequence of bytes in a page. This design is great for get/put operations. Column-major vertically partitions the data and stores a whole column of a table (or set of records) as a contiguous sequence of bytes. Such a column-major layout is great for scans as analytical queries often only involve a subset of the columns [HBND06, AMH08, ABH09, SAB+05, BMK99, BGVK+06]. Furthermore, column-major supports vector operations (SIMD) which is also great for bulk operations and scans [WPB+09, ZR02].
As already mentioned and used in Chapter 4, there is a third (hybrid) approach that proposes to store data column-wise only within a bucket/page. As the experiments with ColumnMap (in Section 4.4) showed, this approach improves get performance compared to pure column stores while preserving their scan speed.

5.3.4 How to handle Versions?

As described in Section 5.2, we want to support versioning of tuples. This means that for each key, we need to potentially store several tuple versions. There are mostly two ways to do that: store them at different locations and link them together or try to keep all versions belonging to the same key in contiguous memory. Building a linked list of versions has the advantage that writing a new version is much cheaper as we do not need to move old versions. As a drawback, all get (and potentially many put) requests need to iterate through this (random access) list in order to find the version of interest.

As described in Section 5.2, we need to support versioning of records. There are two approaches: (a) store all versions of a record at the same location and (b) store the most recent versions of all records together and store old versions of records in a separate store as a log. The second variant involves chaining the versions in a linked list in order to quickly find the right version of a record as part of a get request. These pointers consume space and pointer chasing is expensive because it involves multiple cache misses. On the positive side, this second approach simplifies garbage collection because garbage collection can be implemented as a truncation of the log of old versions. Furthermore, the second approach involves less fragmentation of data pages.

5.3.5 When to do Garbage Collection?

With versioning comes garbage collection of old versions. There are two possible strategies: (a) do periodic garbage collection in a separate dedicated thread/process and (b) piggy-back garbage collection with (shared) scans. Approach (b) increases scan time, but it also trades off garbage collection investment for garbage collection benefits: Tables which are scanned frequently and greatly benefit from garbage collection are garbage collected more frequently than other tables. Another advantage of the piggy-
back approach is that it does garbage collection while the data is processed anyway, thereby avoiding additional cache misses.

5.3.6 Conclusions

Not every possible combination of these techniques makes sense. For example, a log-structured column store makes no sense. For this study, we decided to implement three combinations which appeared to be the overall most promising designs. We call these three designs TellStore-Col, TellStore-Row and TellStore-Log.

TellStore-Log uses a log-structured storage in which the versions of all records are linked. Garbage collection is carried out by copying records to the front of the log and truncating the log. Garbage collection is only carried out as a side-effect of scans.

TellStore-Col and TellStore-Row are both based on a delta-main organization of the store. We use a log-structured design for the delta (write optimized) and a regular, page-based organization for the main (read-optimized). The difference between TellStore-Col and TellStore-Row is that TellStore-Col uses a PAX (column-major) layout in main whereas TellStore-Row has a row-major layout in main. Both use a row-major layout in delta. The next sections contain details of our implementation of these three variants.

5.4 TellStore

This section first describes some high-level implementation principles common to all three different TellStore approaches. Next, we want to highlight how each of these approaches strives to fulfill the wish list presented in Section 5.2. Most importantly, all TellStore approaches consist of carefully designed, completely lock-free, concurrent main-memory data structures. We briefly summarize TellStore-Log and TellStore-Row and then focus a little more on the implementation specifics of TellStore-Column which, as our evaluation will prove, seems to be the most promising design. We highly encourage the interested reader to also check out our open-source code project [Sys16e] to see how TellStore, respectively Tell 2.0 perform in the wild.
5.4. TellStore

5.4.1 Asynchronous Communication

In order to make best use of the available CPU time and network bandwidth, a processing node should not idle while waiting for the result of a storage request, but instead do some other useful work, e.g., process other pending transactions or process other parts of the same transaction that do not depend on the outcome of the pending request. This is why TellStore uses an asynchronous communication library, called InfinIO [Sys16b, Boc15], for its communication to the processing layer. InfinIO, which was built specifically to run on top of RDMA technology (i.e., Infiniband [Inf16]), is a library that employs user-level threads and call-back functions through an API very similar to the one provided by Boost.Asio for Ethernet communication.

Using this library, all requests to TellStore immediately return a future object, on which the calling process can wait later on. InfinIO then batches all these requests at the network layer before actually sending them to TellStore. Likewise, InfinIO batches TellStore’s results before sending them back to the processing nodes. This batching greatly improves the overall performance because it cuts down the message rate on the Infiniband link, which would otherwise become the performance bottleneck quite fast (cf. Subsection 5.5.2.3).

5.4.2 Thread Model

The general thread model, common to all three TellStore variants, is illustrated in Figure 5.2. There are two different kinds of threads: get/put threads and scan threads, both of which are pinned to CPU cores as a simple mean of performance isolation. Whenever a client opens a connection, one of the get/put threads (usually the one with the smallest load) accepts it and from now on takes the sole responsibility for processing incoming get and put request issued by that particular client. Whenever it receives a scan request, it simply places it on a queue and thereby delegates its execution to the dedicated scan threads.

One of the scan threads has the role of the scan coordinator, which means it empties the request queue and transforms its elements into one single shared scan request (as will be explained below). Next, it partitions the set of storage pages to be scanned into equal shares and notifies the other scan threads to process this request. All the scan
threads (including the coordinator) then process their shares individually. (Partial) scan results are directly written into the client’s memory using RDMA writes. Notably, using this threading model allows careful provisioning of resources for get/put requests on one side and scan requests on the other, depending on the expected workload.

### 5.4.3 Scan Interface and Execution

In order to speed up analytical queries, our scan interface allows the client to define selections, projections and simple aggregations, which requires the existence of a well-defined *table schema*. When transforming a scan batch into a single shared scan request, we use just-in-time compilation with LLVM [LLV16] to get highly-tuned machine code. In essence, we combine the shared scan techniques introduced by Giannikis *et al.* [GAK12] (where gcc is used instead of LLVM) with LLVM query code generation as described by Neumann [Neu11] and Klonatos *et al.* [KKRC14] (which does not take advantage of shared scans).

As described in more detail in Pilman’s thesis [Pil17], we carefully decided which optimizations to do in C++ (while generating the LLVM code) and which to delegate to the LLVM compiler (by choosing the right set of optimization passes). We made this distinction, rather than letting LLVM do all optimizations, because LLVM compilation is slow and
our just-in-time compilation required fast compilation times: This way, we were able to reduce compilation times from 100 milliseconds to less than 20 milliseconds for a typical batch of queries. The generated code is highly-efficient for a number of reasons: First, if some queries share a common predicate (e.g., \( a < 4 \)), this predicate is only evaluated once. Second, vector instructions are used to evaluate a predicate on several tuples at once (the degree of vectorization depends on the chosen record layout). Third, the code generation takes the table schema into account and compiles into code that performs operations on memory locations with static offsets, which facilitates pre-fetching.

### 5.4.3.1 Predicate Evaluation and Result Materialization

We require all selection predicates to be in the conjunctive normal form (CNF). As shown in Figure 5.3, scanning a page works in two steps. First, we allocate a (zero-initialized) bit matrix where columns represent unique OR-predicates and there is a row for each tuple in the memory page. We then scan the page and evaluate all these predicates. If a predicate evaluates to true for a specific tuple, the corresponding bit in the matrix is set to 1. Notably, there is no need to check its previous value as \( true \lor x = true \).

![Figure 5.3: Data Structures used for TellStore Predicate Evaluation and Result Materialization, example with two different Query Predicates in CNF](image)

Q1: \( a < 4 \land (a > 0 \lor b > 10) \)

Q2: \( a < 4 \land b > 10 \land b < 20 \)
Chapter 5. Fast Scans on Key-Value Stores

In the second step, we create a bit vector for every query of the original batch where bit \( i \) is 0 if the \( i \)-th OR-predicate appears in that query and 1 otherwise. Like this, finding matching tuples for a specific query is very efficient, we just need to compute the bit-wise expression \( r \mid \sim v \) between the query vector \( v \) and the corresponding row \( r \) in the bit matrix\(^3\) and check whether this expression has all bits set to 1. Once all matching tuples are identified, they are materialized and written into a local buffer. As soon as this buffer is full, it is copied back into the client’s memory using RDMA write.

5.4.4 TellStore-Log

The implementation of TellStore-Log was highly inspired by RAMCloud [OAE\(^1\) 11], which is also reflected in its architecture in Figure 5.4. In contrast to the log in RAMCloud, TellStore keeps additional meta data for every record, namely a validFrom field (the version number of the transaction that created this record), a validTo (the version number of the transaction that replaced this version with a new one), as well as a pointer to the previous version of that record (if such a record exists). This data is useful in two different ways: First, the scan is able to efficiently check whether a record matches the snapshot descriptor by inspecting validTo and validFrom. Second, the garbage collector can inspect these fields in order to decide whether a record can be safely discarded. The previous pointer helps with a get request in the case where the newest version in the log is too new to match the snapshot descriptor and hence older versions need to be considered.

The hash table serves as point of synchronization: two concurrent updates both append an entry to the log and try to atomically update the hash table using compare-and-swap (CAS). If both threads try to update the same key, the CAS will fail for one of them, which will case this thread to report a conflict back to the client.

The layout of the actual data of a record (referred to as data in Figure 5.4) is the same as for records in TellStore-Row as shown in the top of Figure 5.6a.

\(^3\)The logical semantics of this expression are \( v \implies r \) which makes sense because we only have to check whether the predicate is satisfied if that predicate is actually relevant for the given query.
5.4. TellStore

5.4.5 Delta-Main Approaches

The main idea behind our delta-main implementations, *TellStore-Row* and *TellStore-Column*, is to keep the *main* mostly read-only while writing changes into a *delta*. As shown in Figure 5.5, each table holds five data structures: a list of pages that hold the data of the *main*, a hash index for the main, two logs that store the delta (one for inserts and one for updates/deletes\(^4\)) and a list of hash tables which index the delta.

5.4.5.1 Record Layout

Entries in the main are clustered by version and ordered by decreasing version number, which allows a quick computation of *validFrom* and *validTo* without the need for storing these values explicitly. Each cluster keeps a pointer to the newest version of a record in the update log if such a record exists. So do entries in the insert log. Update log entries, on the other hand, keep *backward pointers* to the previous version in the update log. However, in order to prevent cycles, there are no such backward pointers back into the

\(^4\)We do never write about deletes in this chapter. The reason is that in TellStore, deletes are simply updates with a zero-payload.
main or insert log. This design facilitates the construction of the delta index because it is sufficient to index insert log entries. Update log entries are always reachable by following the newest pointers in the main- or delta index. All log entries (be it in the update or in the insert log) have a valid-bit to indicate whether an entry is valid or just garbage.

5.4.5.2 Synchronization between Get/Put/Scan and Garbage Collection

It is important to keep in mind that a key always appears either in the main- or in the delta index, but never in both. If two concurrent updates hit the same key, they are both redirected to the corresponding entry, either in a main page or in the insert log, which then serves as the point of synchronization. Get and put requests first need to check both indexes and then, depending on the provided snapshot descriptor and the record versions they encounter, follow the newest pointer and potentially its chain of backward links in order to find the record with the desired version. The reason why also put requests have to do that is that they need to check whether they are allowed to update a record or whether there exists already a newer version, in which case they are not allowed to write, but have to report a conflict.\(^5\)

\(^5\)This basically implements the semantics of conditional write as defined in Subsection 2.3.2.
The scan needs to iterate through all pages in the main as well as all pages between the head and the tail of the insert log. For each entry, it also needs to check all newer versions in the update log by following the newest pointer and the corresponding predecessor list. Garbage collection completely rewrites the page of the main (except if a page did not change at all) in order to preserve the record clustering. This is indeed a quite expensive operation, but it comes with the clear benefit of improved scan performance.

5.4.6 TellStore-Row

TellStore-Row is the delta-main implementation that stores its main pages in a row-organized record format as shown in Figure 5.6a.

The (multi-version) record stores its size, the key, the newest pointer and the number of versions, followed by an array of versions, an array of offsets and the tuples themselves. The record size is used by the scan so that it can skip the whole record if no query is interested in it. The offset array stores offsets from the beginning of the record to the beginning of the tuple (in bytes), which, together with the version array, allows us to get a specific version of a tuple without looking at the other tuples.

![Figure 5.6: Multi-Version Record in two different Formats](image)

The layout of a tuple itself is quite simple: we first store a bitmap where position \( i \) is 0 if attribute \( i \) of this tuple is NULL and 1 otherwise. Next, we store all fixed-sized fields (sorted by size so that we only waste minimal space for alignment), followed by the
variable-sized fields. Each such field consists of a 4-byte integer storing the size of the field, followed by a byte array. Note that the size of a tuple (as required, e.g., in a get request) can be calculated simply from the offset array.

5.4.7 TellStore-Col

TellStore-Col uses ColumnMap to store the main. As we showed in Subsection 4.3.5, the locations of all values that make up a tuple can be computed from the location of the first attribute, the total number of records in a page/bucket and the table schema as long as all values of that schema are fixed-size.

However, if values can have arbitrary sizes (as for example var chars), this simple computation falls apart and get/put operations become horribly slow. This is why state-of-the-art systems avoid variable-sized values, either by simply disallowing them (as we have done in AIM in the previous chapter), allocating them on a global heap and storing pointers (as in MonetDB [BGVK+06] and HyPer [Neu11]) or using a dictionary and store fixed-sized dictionary code words (as in SAP/HANA [Fea12] and DB2/BLU [RAB+13]). For our design, global allocation does not work because it makes garbage collection virtually impossible and dictionary encoding would slow down get operations dramatically. Therefore, we decided to extend ColumnMap with the necessary features as shown in Figure 5.6b. In addition to variable-sized values, our improved version of ColumnMap (called ColumnMap 2.0) also supports versioning.

5.4.7.1 ColumnMap 2.0 Page Layout

Each page starts with some meta data, namely the tuple count and an array that stores key-version-newest triples. Thus scanning this first array is sufficient to check whether a page needs garbage collection. The next array keeps the serialized length for each tuple. The meta data is followed by the fixed-sized columns and the variable-sized meta columns which, for every value on the var-sized heap, store a (4-byte) page offset to point to that value as well as its (4-byte) prefix. This has two advantages: first, by computing the difference of two consequent offsets, we can compute the size of a heap.

---

6 These numbers could be computed from other meta information, but are very handy for get and scan operations because they provide a shortcut to find out how much space tuples will need in a result buffer.
value and second, we get a speedup on prefix scan queries because we can identify a set of candidate tuples without having to look at the heap at all. Last, but not least, there is a heap to store variable-sized values. We make sure that entries belonging to the same tuple are sequentially stored in that heap, which speeds up get operations.

5.4.7.2 Materialization & Scans with LLVM

In a columnar layout, getting a single tuple may be expensive: instead of just copying the tuple out of the page, a loop needs to iterate over all columns and copy each field from each column into a new buffer. Moreover, this loop has to compute the offset of the next field incrementally (by looking at the table schema), which creates data dependencies in the code. To speed up this materialization step, we generate a materialize function whenever we create a new table or change a schema. This generated function then does not need to iterate over the schema anymore, it just contains one copy operation per column. Given only the tuple count and the offset of the tuple, this function can calculate all necessary offsets before it starts serializing. We compile this function with LLVM and add pre-fetch instructions at the beginning to make sure that all memory is loaded into the caches as early as possible. As stated before, we also use LLVM to create the shared scan execution code. This becomes even more efficient when using a columnar layout as we only need to scan columns of interest. Furthermore, LLVM helps vectorizing the code to use even more parallelism.

5.5 Experiments

This section evaluates the different TellStore approaches and some of other KV stores with respect to their ability to serve as a storage back-end for Tell 2.0. As Tell 2.0 targets mixed transactional/analytical workloads, we use the YCSB# and the Huawei-AIM-Simple (cf. Chapter 3) benchmark for this evaluation.

5.5.1 Configurations and Methodology

We ran all our experiments on a small cluster of 12 machines. Each machine is equipped with two quad core Intel Xeon E5-2609 2.4 GHz processors, 128 GB DDR3-RAM and
Chapter 5. Fast Scans on Key-Value Stores

a 256 GB Samsung Pro SSD. Each machine has two NUMA units that consists each of one NUMA processor and half of the main memory. Furthermore the machines are equipped with a 10 Gb Ethernet adapter and a Mellanox Connect X-3 Infiniband card, installed at NUMA region 0.

The KV stores we benchmarked in this section are all NUMA-unaware, which has implications on the performance. In order to get consistent numbers, we decided to run every process on only one NUMA unit. Therefore, throughout this chapter, the term node refers to a NUMA unit which can use half of a machine’s resources (4 cores and 64 GB memory). Storage nodes always run on NUMA region 0 such that they have fast access to the Infiniband card, while processing nodes can run on both regions.

The system under test (SUT) for all experiments is a KV store. Whenever we compare TellStore to other KV stores, we use TellStore-Log. If we compare different TellStore implementations to each other, we also use Kudu as a baseline. We use Kudu because it is the only KV store that can provide a reasonable scan performance (Table 5.1 in the introduction).

To make sure that the load generation does not become a bottleneck, our experiment setting always uses three times as many processing nodes as storage nodes. For all our measurements, we first populated the data and then ran the experiment for seven minutes. We ignored the numbers from the first and the last minute to factor out the warm-up and cool-down time. If a query ran for more than seven minutes, as was the case for some of the experiments in Table 5.1, we waited until it finished.

We did a considerable effort to benchmark all KV stores at their best configuration. In order to achieve a fair comparison to the in-memory systems, we put Cassandra’s and HBase’ data on a RAM disk. For Kudu, we used the SSD and configured its block cache such that it would use all the available memory. We collaborated closely with the Kudu developers in order to make sure that we had the best possible Kudu configuration. The benchmarks for HBase and Cassandra were implemented in Java using the corresponding client libraries [hba16, cas16]. For RAMCloud and Kudu, we implemented the benchmarks in C++ using the systems’ native libraries [ram16, kud16]. We used multi-put and multi-get operations in RAMCloud whenever possible and projection and selection push-down in Kudu. TellStore was benchmarked with TellDB, a shared library that incorporates the native TellStore client library and allows executing get, put and scan requests in a transactional context as well as managing secondary indexes (cf. Pilman’s thesis [Pil17].
for details). We turned off replication for all KV stores, except for HBase where this is not possible, which is why we used three storage nodes (HDFS data nodes) instead of only one for that particular case.

For TellDB and Kudu, we batched several get/put request into one single transaction,\(^7\) be it for operations in YSCS# or for event processing in Huawei-AIM. For TellStore, a batch size of \(B = 200\) proved to be useful, while a good batch size for Kudu sessions was 50.

### 5.5.2 YCSB#

We performed several experiments with the YCSB# benchmark. Where not mentioned otherwise, we used the following default parameters: 50 million records (\(sf = 50\)), 11 columns (\(N = 10\)), execution time \(t = 7\) minutes, one analytical client (\(c = 1\)) and a number of transactional clients that scaled with the number of storage nodes (\(sn\)): \(e = 30 \cdot sn − 1\). As mentioned already, we used \(B = 200\) for TellStore and \(B = 50\) for Kudu. We used three different configurations for the transaction mix \(M\):

- **read-only**: \(M[read] = 1, M[update] = M[insert] = M[delete] = 0\)
- **insert-only\(^8\)**: \(M[read] = M[update] = M[delete] = 0, M[insert] = 1\)

### 5.5.2.1 Get/put workload

In the first experiment, we ran YCSB# in a setting very similar to the original YCSB that does not execute any scan queries. We measured the operation throughput (number of gets and puts per second) of TellStore-Col against all other considered KV stores as presented in in Figure 5.7. It becomes immediately clear that TellStore is indeed very competitive for this typical KV store workload. In fact, only RAMCloud’s performance is

---

\(^7\)Kudu actually uses the notion of a **session** which has weaker transactional semantics than a ACID transaction in TellDB, but is still a useful concept.

\(^8\)This setting violates the balancing requirement on updates and deletes. However, as we are not going to run analytics concurrent with this workload, this is fine.
Chapter 5. Fast Scans on Key-Value Stores

Figure 5.7: Throughput of two different Transactional YCSB# Workloads on various KV stores, scaling from 1 to 4 Storage Nodes

on par with TellStore. This can be attributed to two factors: RAMCloud and TellStore are both in-memory and can make efficient use of the Infiniband network. In addition, RAMCloud outperforms TellStore for the get-only workload. This is mostly due to the fact that TellStore reserves three CPU cores for scans and garbage collection (and consequently only one for get/put requests), while RAMCloud can use all four cores to process operations. For the balanced workload, however, TellStore and RAMCloud perform equally well.

Figure 5.8 still shows the same experiment, but compares the three TellStore approaches against each other and shows Kudu as a baseline. While the two row-based storages (TellStore-Log and TellStore-Row) outperform TellStore-Col, this difference is not as large as one might expect. As a put operation writes into a row-oriented log in all approaches, its performance is likely to be the same. For get operations, however, TellStore-Col needs to assemble the records from different memory locations, which comes at slightly bigger cost. This is also why the gap between TellStore-Col and the other approaches is slightly bigger in case of the read-only workload. Compared to Kudu, this experiment confirms the results already shown in Figure 5.7: TellStore’s throughput is nearly two orders of magnitude higher than the one of Kudu. What is more, we can also observe a perfect scale-out for all approaches (including Kudu, just in a very different ball park).
5.5. Experiments

Figure 5.8: Throughput of two different Transactional YCSB# Workloads on TellStore variants and Kudu, scaling from 1 to 4 Storage Nodes

5.5.2.2 Insert-only

Our approaches do not only differ in the way the data is organized but also in the hash table implementation. TellStore-Log wastes memory because it allocates a hash table big enough such that it never has to be re-sized. We cannot afford, however, to do that for the delta index of TellStore-Row and TellStore-Col as we re-create this index on each garbage collection step. Instead, we use a linked list of fixed-sized hash tables. Whenever an insert operation finds the current hash table to be full, it allocates a new one and links it to its predecessor. Allocation of a new hash table (even if small) is relatively costly compared to the other costs involved with an insert. This means that the latency of inserts varies greatly between the ones that did and the ones that did not have to allocate a new hash table.

This effect is confirmed by the experiment shown in Figure 5.9 where we measured the response time of transactions with the insert-only workload in a setting with two storage nodes. As one can see, the delta-main approaches show a bigger variation in the latency of these transactions. In fact, the presented box-plot shows the 1, 5, 50, 95 and 99 percentile and not the minimum and maximum. This is because the maximum value is so far apart that the rest of the plot would not be visible. A small number of transactions needed more than half a second to complete.
5.5.2.3 Batching

As explained in Subsection 5.4.1, the processing layer of Tell 2.0 uses batching to get a significant throughput improvement at the cost of a slightly increased latency. Whenever a transaction issues a request, this request gets buffered within InfinIO and sent to TellStore whenever the buffer is full. While a transaction is waiting for a result, the InfinIO runtime system schedules other transactions that can then help to fill up the buffer.

Batching is important, as get/put messages are often very small and the message rate is limited on Infiniband. Therefore, in Figure 5.10, we show the throughput of the balanced workload for various batch sizes, using two storage nodes. It is not surprising that all storage implementations benefit equally well from batching. Furthermore, we can see that a batch size of 16 seems to be a good default value.
5.5. Experiments

![Throughput graph]

Figure 5.10: Throughput of Transactional balanced YCSB# Workload for varying InfinIO Batch Size on different TellStore variants, 2 Storage Nodes

### 5.5.2.4 Scans

In order to demonstrate the raw scan performance, we first ran the YCSB# queries in isolation (without concurrent transactional load). Figure 5.11 shows such results. We do not show the response time of query 2 as it is nearly identical to query 1.

![Response time graph]

Figure 5.11: Response Times of two different YCSB# Queries running in isolation on TellStore variants and Kudu, scaling from 1 to 4 Storage Nodes
As one would expect, TellStore-Col has the lowest response time for query 1 and query 2 (not shown). For query 3 however, TellStore-Log is slightly faster. This is due to the fact that query 3 does not do any projection and hence TellStore is required to fully materialize the records. This has the same consequence as already described in Subsection 5.5.2.1, namely that TellStore-Col has to fetch values from different locations and therefore performs slightly worse.

The surprising result, however, is that the log-based implementation performs so well at all. The reason is that it does garbage collection while it scans, which leaves it with one additional core (three instead of two) for scanning.

All three TellStore variants exhibit perfect scalability while Kudu does not seem to benefit that much from added storage nodes. For query 3, there is some improvement from 1 to 2 nodes, but only about 10% (note that the y-axis is cut at two places).

5.5.2.5 Mixed Workload

To see how well a scan performs while the system also needs to execute get and put requests, we ran the YCSB# queries with the workloads shown in Subsection 5.5.2.1 (balanced and get-only) and compared these numbers to the results obtained in Subsection 5.5.2.4. However, in order to make all systems comparable, we fixed the get/put request rate to 35,000 operations per second, 35,000 being the highest load that Kudu could sustain. In Figure 5.12, we show the response time for query 1 and four storage nodes. For the other queries and with a different number of storage nodes, the results look similar. Comparing the different scenarios, we can see no notable difference for TellStore. Kudu, on the other hand, does experience a significant increase in its query response time for concurrent puts. This clearly demonstrates TellStore’s robustness under mixed workloads.

5.5.3 Huawei-AIM Benchmark

To show that TellStore is able to perform more complex, interactive workloads, we also executed the Huawei-AIM-Simple benchmark for a scaling factor of $sf = 2$. As Figure 5.13 reveals, TellStore-Col outperforms all other implementations significantly. Furthermore, the difference between TellStore-Log and TellStore-Col is larger than in the
5.5. Experiments

![Figure 5.12: YCSB# Q1 Response Time for different concurrently-running YCSB# Transactional Workloads on TellStore variants and Kudu, 4 Storage Nodes](image)

![Figure 5.13: Response Times of different Huawei-AIM Queries for \( sf = 2 \) (20M entities, 20,000 events per second) on TellStore variants and Kudu, 4 Storage Nodes](image)

experiments shown before. There are two main reasons for that: First, the main table (analytics matrix) consists of 500 columns. Therefore, the columnar layout has a much
higher impact on query performance than before. This is also why Kudu, which also features a columnar layout, is much closer to the TellStore performance now. Second, the queries are much more complex. As mentioned already several times, TellStore supports storage-side aggregation, which is heavily used here. While all TellStore approaches offer this feature, TellStore-Col can make much better use of the vectorization features provided by the CPU (and the LLVM compiler).

We can also see that TellStore-Col is the only approach that can meet the required SLA on response time, $t_{RTA} = 100$ milliseconds. As none of the approaches can meet $f_{RTA} = 100$ queries per second (TellStore-Log reaches roughly 30 queries per second, the others are clearly below), no statement about TCO can be made.

5.5.4 Summary

We conclude from our experiments that TellStore is a very competitive KV store because it can deliver a high throughput for get and put requests. It also provides an order of magnitude higher scan performance than all other KV stores we tested against, both in terms of latency and throughput. Compared to alternative designs like Kudu, TellStore’s scan performance does not significantly deteriorate when a moderately-sized get/put workload is executed in parallel. Moreover, TellStore shows a perfect strong scale-out, the most important dimension of scalability.

It is not surprising that the TellStore variant with the columnar layout (TellStore-Col) provides the lowest scan latencies, already in the YCSB#, but even more apparent in the Huawei-AIM-Simple benchmark. At the same time, it is also able to sustain a very high get/put load, which hence makes it the preferred implementation for mixed workloads.

5.6 Excursus: Performance Isolation

Cloud databases are inherently multi-tenant because they have many different users that all access and process their data. In order to stay in business, a cloud database provider needs to isolate these tenants well from each other. Tenant isolation has two aspects: performance isolation and protection. The first aspect, performance isolation, relates to the requirement that tenants’ perceived user experience should always be the
same, independent of the fact how many other tenants are present in the database and what workloads they execute. The second aspect, protection is not discussed here, but will be elaborated in detail in the context of *cloud confidentiality* in the second part of this thesis.

In addition to the remark in Subsection 5.4.2 which classifies the separation of update and scan threads within TellStore as a simple mean of performance isolation, the literature describes several approaches to tackle performance isolation at different levels: A foundational paper by Verghese *et al.* [VGR98] explains how to enable performance isolation between different processes in shared-memory multiprocessor systems and with respect to CPU time, memory and disk bandwidth. Follow-up work by Fedorova *et al.* [FSS07] extends this work to also isolate caches, while Urgaonkar *et al.* present a technique that overbooks resources in a cluster of machines by priorly profiling the resource demands of applications using it. The advent of hardware virtualization gave rise to an entire line of work on performance isolation in hypervisors, *e.g.*, Xen [GCGV06]. Xen tries to achieve performance isolation between several virtual machines on the same physical machine.

There is literature that specifically studies performance isolation in cloud databases: SQLVM [NDS+13] studies how to allocate resources to tenants in a database given their specific CPU, I/O and memory requirements. SQLVM uses the abstraction of a virtual machine (VM) and hence metrics similar to those of Verghese *et al.* [VGR98], but performs the allocation within the DBMS and not in the operating system. This is more light-weight because it saves the costs of context switching. The follow-up work on SQLVM [DLNK16] raises the abstraction even more such that clients do not have to specify their hardware resources, but simply their monetary budget and response time constraints. This is in fact nothing else but specifying SLAs. A similar approach is followed by Liu *et al.* [LHM+13] who model tenant placement as an optimization problem where the overall SLA revenue for the cloud provider must be maximized. DBSeer [MCM13] complements these works by providing accurate estimation models for resource consumption in OLTP workloads and illustrating some roads for future research directions (some of which have already been explored by the aforementioned work [LHM+13, DLNK16]). A useful summary of what has been achieved and what should be researched in the future with regard to tenant isolation, was presented by Elmore *et al.* [ECAEA13].

Another interesting work that implements tenant isolation directly in the application is by Shue *et al.* [SFS12] who show how to address performance isolation and fairness
Chapter 5. Fast Scans on Key-Value Stores

between different tenants in a key-value store.

This thesis does not specifically address performance isolation between tenants. Instead, as stated in Chapter 1, the presented techniques follow the approach to optimize for best overall performance. For example, we prefer an overall higher query throughput for all tenants over lower response times for some queries for a particular sub-set of tenants. However, many of the above mentioned approaches to performance isolation could be integrated into Tell 2.0, respectively AIM. Especially the work by Das et al. [DLNK16] and Shue et al. [SFS12] would be interesting to further investigate as they share many common aspects with our systems.

5.7 Concluding Remarks

Our experiments show that it is possible to build a KV store with a high scan performance that does not sacrifice on get/put performance and scalability. The presented system, TellStore, is able to run different kinds of workloads with competitive performance without over-specializing in a certain area. From the three presented implementations, TellStore-Column shows the best overall performance across the different experiments, which makes it a preferred choice to system to run mixed OLTP/OLAP workloads, respectively transactions on stateful streams.

TellStore is elastic by design: processing nodes are stateless and can therefore be added or removed as required by the workload. Also the storage layer is elastic: First, while offering good performance for both, updates and queries, the performance ratio of these two parts of the workload can be further tuned by changing number of update versus scan threads dynamically. Second, storage nodes can also be added or removed fairly easily with a minimal number of keys that need to be transferred [GBPK16]. Furthermore, the experiments also proved TellStore to be highly-scalable and as performance scales well with the investment (on CPUs and memory) we arguably also minimized TCO, thereby fulfilling all the promises that we made in the introduction.

TellStore’s current design does not include durability or replication to secondary storage nodes. In order to address this, we recently started building a replication feature into TellStore [GBPK16]. Our first experiences make us confident that this should not be too hard as a enough related work exists in that area [Geo11, LM10, RKO14].
Running analytical workloads efficiently is a hard problem and having a fast scan in the storage only solves it half-way. In this thesis, we focused on queries that can efficiently be processed on a single processing node, taking full advantage of TellStore’s tremendous scan capacity. However, there exist other analytical queries that can only be efficiently executed in a distributed way. Examples of such distributed query processing systems include Spark [ZCF+10] and Presto [Fac16a]. We implemented TellStore adapters for both of these systems [Sys16e] and ran the TPC-H benchmark queries [Tra16c]. Unluckily, we found query response times with TellStore to be on par with other storage layer back-ends (Parquet and ORC). Extensive profiling on these platforms revealed that this is because neither Spark nor Presto can make efficient use of TellStore’s scan feature, which is why we started implementing a distributed query processing engine for Tell 2.0. This system will need an optimizer that takes into account TellStore’s asynchronous processing model as well as its capabilities to process selections, projections and aggregations directly on the data.
Part II

Confidentiality
Randomly-Partitioned Encryption

The previous chapters focused on the performance of cloud databases and briefly touched on performance isolation. Besides performance isolation, protection was mentioned as another important aspect of tenant isolation within the cloud. Data protection is a very broad term that subsumes many different aspects, like different kinds of data sensitivity (confidential vs. secret) as well as different attacker models (active vs. passive adversaries, untrusted/malicious tenants and/or cloud providers). Depending on this context, protection would be referred to as security, privacy or confidentiality. The second part of this thesis is about confidentiality because the use cases it addresses do not involve strictly private data (e.g., blueprints of a secret government program), but data that, if leaked, incurs a significant economic damage. Examples of such use cases include the management of an enterprise’s customer data or analytics on the history of financial transactions of a bank.

This chapter presents randomly-partitioned encryption (RPE) as a novel encryption scheme to preserve the confidentiality of a tenant’s data against other tenants as well as an honest-but-curious cloud provider. RPE bridges the gap between strong-security/bad-performance encryption (e.g., AES [KL07]) and good-performance/weak-security encryption (e.g., OPE [AKSX04, BCLO09]). RPE achieves the best of both worlds because it can trade-off confidentiality against performance and thereby allows tenants to tune their database for performance given their required level of confidentiality.\(^1\)

\(^1\)Parts of this chapter have been presented at DBSec [SBKV14].
Remark: This thesis mainly targets the database and systems community. However, the following chapter also uses some notations and terms commonly used in the information security, respectively crypto community. This is why this chapter uses footnotes extensively to describe the relevant crypto terms. Readers familiar with these terms can easily skip those footnotes.

6.1 Introduction

As mentioned already in Chapter 1, one of the biggest issues able to stop cloud computing and in particular cloud databases, are concerns about security, respectively confidentiality [CGJ+09, clo16]. The events that motivated this work were privacy violations in a private cloud by honest-but-curious adversaries\(^2\) with insider knowledge. Encryption is a possible way to protect data against such attackers and the spectrum of available encryption schemes ranges from strong semantically-secure (but typically inefficient) encryption schemes to weak (but efficient) encryption schemes. The problem that randomly-partitioned encryption (RPE) targets is to combine the advantages of low-confidentiality/high-performance schemes like order-preserving encryption (OPE) [AKSX04] with high-confidentiality/low-performance schemes like probabilistic AES (AES in CBC mode [KL07]), in order to achieve good confidentiality and reasonable performance. Throughout this chapter, performance refers to the average response time of queries in the TPC-H benchmark [Tra16c], while confidentiality is the ability to resist the following attacks that all model different kinds of insider knowledge (and will be thoroughly defined in Section 6.3):

- **domain attack**: The adversary has knowledge of the plaintext domain.\(^3\)

- **frequency attack**: The adversary has knowledge of the plaintext domain and its frequency distribution.

\(^2\)adversaries that do not actively manipulate data, but try to infer information [KL07]

\(^3\)In cryptography, the term plaintext or cleartext refers to information that is not encrypted, while encrypted information is called ciphertext.
6.1.1 Background and State of the Art

The main idea of RPE is to randomly partition the plaintext domain and apply an order-preserving encryption scheme to each partition. This makes RPE a partially-order-preserving encryption scheme as each partition is ordered, but the total order is hidden. In the following, we try to summarize the state of the art in the context of existing OPE schemes and argue why they cannot withstand domain and frequency attacks. A more detailed overview of related work will be provided towards the end of that chapter in Section 6.9.

Order-Preserving Encryption (OPE) An order-preserving symmetric encryption (OPE) scheme is a deterministic symmetric encryption scheme whose encryption algorithm produces ciphertexts that preserve numerical ordering of the plaintexts. This property makes OPE very attractive for database applications since it allows efficient range and rank query processing on encrypted data. However, the order relationship between plaintext and ciphertext remains intact after encryption, making it an easy target for a domain attack. Moreover, being deterministic makes OPE particularly vulnerable against frequency attacks. OPE was first proposed in the database community by Agrawal et al. [AKSX04] and treated cryptographically for the first time by Boldyreva et al. [BCLO09], followed by many other researchers searching for an ideal object [PLZ13, MTY13, WRG+13]. However, the problem of dealing with domain and frequency attacks remains.

Probabilistic Order-Preserving Encryption (Prob-OPE) A probabilistic order-preserving encryption scheme is a probabilistic symmetric encryption scheme whose algorithm not only produces ciphertexts that preserve numerical ordering of the plaintexts, but also generates different ciphertexts for the same plaintext. This property flattens out the original frequency distribution of the plaintext values, therefore resisting statistical attacks where an adversary knows the frequency of one or several plaintexts [WL06, EWSG04, YZW06, KAK10a]. However, probabilistic schemes still leak

4An ideal object is a ciphertext that was not produced by the encryption function, but by a random permutation function. Ideal objects are typically used in so-called indistinguishability proofs where an encryption scheme is assumed secure if an adversary cannot distinguish ciphertexts from ideal objects.
total order and are hence exposed to domain attacks (and consequently also frequency attacks\(^5\)).

**Modular Order-Preserving Encryption (MOPE)** One way to efficiently secure OPE schemes against domain attacks is to prepend the OPE construction with *random shift cipher*, i.e., to add a big (secret) number (called the *modular offset*) to each plaintext value before encryption [BCO11]. As this addition is modular (modulo the size of the plaintext domain), nothing can be leaked about the rank of an underlying plaintext. While being resistant against domain attacks, frequency attacks remain a problem.

**Partially-Order-Preserving Encryption (POPE)** A partially-order-preserving encryption scheme is a symmetric encryption scheme whose algorithm partitions the domain. Within each partition the order is preserved but across the partitions the order is distorted. There are several ways how to partition the domain: Kadhem *et al.* [KAK10b] as well as Lee *et al.* [STJD\(^+\)09] partition the domain into several bins. These solutions are secure against domain attacks, but vulnerable against frequency attacks because the queries leak the bin boundaries. RPE on the other hand, partitions the data in a fine-grained manner and proposes a confidentiality-tunable method to rewrite queries.

**6.1.2 Contributions and Main Results**

To the best of our knowledge, this work is the first that formally defines domain and frequency attack. Moreover, we introduce a couple of novel encryption techniques and analyze their security under these attacks. We start with a scheme that protects against the domain attack called *deterministic randomly-partitioned encryption* (Det-RPE) and then make this scheme probabilistic (Prob-RPE) in order to address the frequency attack. Det-RPE and Prob-RPE can also be prepended with a *random shift cipher*, just like in MOPE. The resulting schemes, Det-MRPE and Prob-MRPE have a number of practical advantages and are even more resilient against domain attacks.

What is more, we assess the confidentiality and performance of these encryption techniques and compare them to relevant related work. Table 6.1 shows a summary of this

---

\(^5\)As the adversary in the frequency attack also gets all information needed to launch a domain attack, vulnerability to domain attacks always implies vulnerability to frequency attacks.
6.1. Introduction

assessment. The middle two columns depict confidentiality and have a tick if a method is secure against a certain attack (a tick in brackets means that the confidentiality depends on a tuning parameter). The last column shows a very rough binary performance measure that states whether or not range queries can be answered within 30 minutes in the 10-GB TPC-H dataset. As we can see, the MRPE variants are Pareto-optimal as they have unique confidentiality/performance characteristics that differentiate them from existing solutions. This is why these are the preferred variants to be used in practice. We use Det-RPE and Prob-RPE mainly for confidentiality analysis and as fall-back solutions in case of known-plaintext attacks.6

<table>
<thead>
<tr>
<th></th>
<th>Domain Attack</th>
<th>Frequency Attack</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPE [AKSX04, BCLO09, BCO11, PLZ13, MTY13, XYH12, WRG13]</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Prob-OPE [YZW06, EWSG04, KAK10a]</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>MOPE [BCO11]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>POPE [STJD09, KAK10b]</td>
<td>[✓]</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>AES-CBC [FGK03]</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Det-RPE</td>
<td>[✓]</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Det-MRPE</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Prob-RPE</td>
<td>[✓]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prob-MRPE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Qualitative Confidentiality & Performance Overview of selected Database Encryption Schemes where [✓] means that the confidentiality depends on one or several tunable parameters

One important feature of our methods is their composability. RPE schemes can be composed with any order-preserving encryption scheme approved by the security com-

6A known-plaintext attack (KPA) [KL07] is launched by an adversary who knows one or more cleartext-ciphertext pairs and tries to use this information to infer additional cleartext-ciphertext pairs
Chapter 6. Randomly-Partitioned Encryption

munity, thereby inheriting their latest breakthroughs. Additionally, by adding a layer of randomness on top of a chosen underlying OPE scheme, RPE schemes amend the weaknesses of OPE schemes. A further advantage of probabilistic RPE schemes that differentiates them from ordinary probabilistic schemes that they support dynamic databases (that include updates).

6.1.3 Overview

The intention of this chapter is to serve as an overview on the implementation of RPE and what it can achieve with respect to confidentiality and performance. It thereby rather highlights the most important outcomes than explaining each and every detail. The reason for this is that RPE has already been presented with an in-depth security analysis and an extensive performance evaluation [San14].

The remainder of this chapter is organized as follows: Section 6.2 presents our assumed client-server architecture. Section 6.3 formally defines the newly introduced adversary models. Section 6.4 starts with describing the basic idea of randomly-partitioned encryption, then formally defines Det-RPE and analyses the confidentiality of Det-RPE and other OPE schemes under domain attack. Section 6.5 formally defines probabilistic RPE and analyses its confidentiality under frequency attack. In Section 6.6, we present the modular variants of RPE, Det-MRPE and Prob-MRPE and show how they further improve confidentiality against those attacks as well as what happens in presence of a known plaintext attack. Sections 6.7 and 6.8 describe implementation details and how they influence the performance of these schemes with respect to the TPC-H benchmark. Section 6.9 discusses related work in more detail and Section 6.10 briefly concludes.

6.2 Client-Server Architecture

Figure 6.1a shows the traditional client-server architecture of running applications on top of a database system. The application or end user issues SQL statements to the database server. The database server executes these SQL statements and returns the results.

Figure 6.1b shows the extended architecture used in this work which is the de-facto standard used in all related work on client-side database security [Sio07, DDJ+03,
6.3 Adversary Models

As explained in the introduction of this chapter, there were use cases from the industry that triggered new adversary models for cloud databases. However, these models have not been cryptographically treated before this work. This section closes that gap and formally introduces the new adversary models as well as the methods needed to assess the confidentiality of different encryption schemes within the context of these models.

PRZB11, ABE+13]. In this architecture, the application remains unchanged and issues the same (unencrypted) SQL statements as in the traditional system of Figure 6.1a. The encryption functionality is implemented as part of an encryption layer sitting between client and server and serving the following purpose: First, it rewrites the statements that are to be submitted to the encrypted database. Second, it decrypts and post-processes the results returned from the encrypted database.

As stated in Chapter 1, we assume thin clients across all works presented in this thesis. With regard to the encryption layer which is assumed to be located on the client’s device, this means that it not only needs to be thin, but also trusted. As it is constrained by energy, the heavy weight-lifting of executing joins, aggregates and alike is expected to be carried out in the (cloud) database.

Figure 6.1: Client-Server Database Architecture with and without Encryption
Chapter 6. Randomly-Partitioned Encryption

Notation  Let $\mathcal{X}$ be the (multi-)set of plaintext values in a domain and $\mathcal{Y}$ the (multi-)set of ciphertexts.\(^7\) The size of $\mathcal{X}$ is denoted as $|\mathcal{X}|$ and similarly, the size of $\mathcal{Y}$ is $|\mathcal{Y}|$. Plaintext elements are denoted as $x$ and ciphertext elements as $y$. Additionally, we define the key space to be $\mathcal{K}_{eys}$ where $K$ is an element from that key space. $K$ is a function that randomly selects an element from $\mathcal{K}_{eys}$, denoted as $K \leftarrow \mathcal{K}_{eys}$.\(^8\) Let $\mathcal{E}_{nc}$ be the encryption function for encryption method $ES$ with key $K$ and plaintext value $x$ as its input parameters. Thus, we have: $y = \mathcal{E}_{nc}ES(K, x)$. Symmetrically, $\mathcal{D}_{ec}$ is the decryption function of $ES$, taking $y$ and $K$ as input, yielding: $x = \mathcal{D}_{ec}ES(K, y)$. Let $\text{Rank}_x$ be the rank of $x$ within $\mathcal{X}$ and $\text{Rank}_y$ the rank of $y$ in $\mathcal{Y}$. Let $\text{rank}$ be the function that returns the rank of an element in its corresponding space. The frequency distribution of $\mathcal{X}$ and $\mathcal{Y}$ is denoted as $\mathcal{F}_X$ and $\mathcal{F}_Y$ respectively. Finally, let $\text{freq}$ be the function that returns the frequency of an element in its corresponding space.

6.3.1 Domain Attack

A domain attack is launched by an adversary $\mathcal{A}^D$ that has a-priori knowledge of the plaintext domain. In our motivating use cases from the industry, this corresponds to a database administrator who is already in the possession of a (cleartext) list of all customers of that enterprise and now wants to retrieve sensitive (encrypted) information for some of these customers (for instance, the ones regarded high-value).

6.3.1.1 Rank One-Wayness

In order to model the success probability of $\mathcal{A}^D$ in breaking an encryption scheme $ES$, we introduce a new notion called rank one-wayness (ROW). $\mathcal{A}^D$ is given the plaintext domain $\mathcal{X}$ and the rank of a ciphertext $y$ (i.e., $\text{rank}_y$) and is asked to return the underlying plaintext $x$, i.e., an $x$ for which $y = \mathcal{E}_{nc}(K, x)$ holds. The ROW advantage of $\mathcal{A}^D$ against $ES$ is defined as the probability of Experiment 6.1 to return 1:

$$Adv_{ES}^{\text{ROW}}(\mathcal{A}^D) = \Pr[Exp_{ES}^{\text{ROW}}(\mathcal{A}^D) = 1] \quad (6.1)$$

\(^7\)Typically, $\mathcal{X}$ and $\mathcal{Y}$ are pure sets. However, when we talk about the frequency distribution (number of occurrences of each value) within a domain, we will sometimes use multi-sets to express which value appears how often.

\(^8\)The $\leftarrow$ sign on top of the $\leftarrow$ indicates randomness.
6.3. Adversary Models

Experiment 6.1  \( \text{Exp}^{\text{ROW}}(\mathcal{A}^D): \mathcal{A}^D \) guesses from plaintext domain

1: \( K \xleftarrow{\$} \text{Keys} \)
2: \( x \xleftarrow{\$} \mathcal{X} \)
3: \( y \leftarrow \text{Enc}_{\mathcal{E}}(K, x) \)
4: \( \text{Rank}_y \leftarrow \text{rank}(y) \)
5: \( x' \xleftarrow{\$} \mathcal{A}^D(\mathcal{X}, \text{Rank}_y) \)
6: if \( x = x' \) then return 1
7: else return 0

To further clarify Experiment 6.1, let us consider the following example: Let \( \mathcal{X} = \{\text{'Beatles'}, \text{'Metallica'}, \text{'U2'}\} \) and \( \mathcal{E} = \text{OPE} \). Assume the ciphertext space to be \( \mathcal{Y} = \{143, 465, 706\} \). We start the experiment by choosing a random element from \( \mathcal{X} \), for instance \text{'Metallica'} and encrypt it. We provide \( \mathcal{A}^D \) with \( \mathcal{X} \) and \( \text{Rank}_{465} = 2 \). \( \mathcal{A}^D \) has to return an element from \( \mathcal{X} \) which she thinks corresponds to the second element of \( \mathcal{Y} \). The probability that \( \mathcal{A}^D \) guesses \text{'Metallica'} correctly is called the \textit{ROW advantage}.

6.3.2 Frequency Attack

A \textit{frequency attack} is an attack that is launched by an adversary \( \mathcal{A}^{DF} \) that has a-priori knowledge of the plaintext values as well as their frequency distribution (how often each plaintext value appears in the database).

6.3.2.1 Frequency One-Wayness

In order to model the success probability of \( \mathcal{A}^{DF} \) in breaking an encryption scheme, \( \mathcal{E} \mathcal{S} \), we introduce another notion called \textit{frequency one-wayness} (FOW). Similarly to ROW, the FOW advantage is defined to be the probability of Experiment 6.2 returning 1:

\[
\text{Adv}_{\mathcal{E}\mathcal{S}}(\mathcal{A}^{DF}) = Pr[\text{Exp}_{\mathcal{E}\mathcal{S}}^{\text{FOW}}(\mathcal{A}^{DF}) = 1]
\] (6.2)

\(^9\)Providing the rank of a ciphertext to the adversary makes sense because the rank can be computed by simply looking at all ciphertexts present in a given database.
Chapter 6. Randomly-Partitioned Encryption

Experiment 6.2 \( \text{Exp}^{\text{FOW}}_{\text{EES}}(A^{\text{DF}}) \): \( A^{\text{DF}} \) guesses from plaintext domain values and frequencies

1. \( K \leftarrow \text{Keys} \)
2. \( y \leftarrow \mathcal{Y} \)
3. \( x \leftarrow \text{Dec}_{\text{EES}}(K, y) \)
4. \( \text{Freq}_y \leftarrow \text{freq}(y) \)
5. \( x' \leftarrow \text{A}^{\text{DF}}(\mathcal{X}, \mathcal{F}_x, \text{Freq}_y) \)
6. \( \text{if } x = x' \text{ then return 1} \)
7. \( \text{else return 0} \)

Experiment 6.2 chooses a random element \( y \) from the ciphertext space \( \mathcal{Y} \) and decrypts \( y \) to get its underlying plaintext \( x \). The experiment then computes the frequency of \( y \) in the ciphertext space \( \mathcal{Y} \), using \( \text{freq}(\cdot) \). The adversary receives the plaintext domain \( \mathcal{X} \), the frequency distribution of the plaintext values in \( \mathcal{X} \), \( \mathcal{F}_x \) and the frequency of the ciphertext \( \text{Freq}_y \) and needs to guess the corresponding plaintext \( x \) in \( \mathcal{X} \).

Again we illustrate this with an example. This time, let \( \mathcal{X} = \{\text{'Beatles', 'Beatles', 'U2'}\} \) and \( \mathcal{Y} = \{143, 143, 706\} \) while \( \mathcal{E} = \text{OPE} \) remains. We start the experiment by choosing a random element from \( \mathcal{Y} \), for instance 143, and decrypt it. We provide \( A^{\text{DF}} \) with \( \mathcal{X} \), \( \mathcal{F}_x = \{2, 1\} \) and \( \text{freq}_{143} = 2 \). \( A^{\text{DF}} \) has to return an element from \( \mathcal{X} \) which she thinks corresponds to 143. The probability that \( A^{\text{DF}} \) guesses correctly is called the FOW advantage.

6.4 Deterministic RPE

This section presents the deterministic randomly-partitioned encryption scheme, Det-RPE. As illustrated in Figure 6.2, the key idea of RPE is to take an existing weak encryption method such as OPE \([\text{AKSX}04, \text{BCLO}09, \text{BCO}11, \text{XYH}12, \text{MTY}13]\) as a building block and to enhance its security by applying it separately on different random partitions of the data that we call runs. In contrast to other partially-order-preserving encryption schemes \([\text{STJD}^{+}09, \text{KAK}10b]\) where the plaintext domain is partitioned into random-length (or fixed-sized) range partitions (cf. Figures 6.2a and 6.2b), RPE creates partially-ordered partitions by randomly assigning each domain value to a run and thereby creating more uncertainty (cf. Figure 6.2c).
6.4. Deterministic RPE

Figure 6.2: Different Approaches for Partially-Order-Preserving Encryption

(a) Equally-sized Range Partitions [KAK10b]
(b) Randomly-sized Range Partitions [STJD+09]
(c) Random Partitions (RPE)

Figure 6.3: RPE Principle

(a) Standard OPE
(b) RPE-enhanced Encryption

Figure 6.3a shows the workings of a traditional OPE encryption function. It receives a plaintext $x$ and produces a ciphertext $y$ where $K$ is the set of parameters of $Enc$, e.g., a secret key. Figure 6.3b shows how RPE composes this traditional scheme to become more secure and have a number of additional operational advantages, e.g., support for updates. Instead of a single $Enc$ function per domain, RPE makes use
Chapter 6. Randomly-Partitioned Encryption

of $U$ encryption functions per domain, $Enc_1$, $Enc_2$, ..., $Enc_U$, where $U$ is the number of runs. These $U$ functions typically all have the same structure and just differ in the secret key they use. Given a plaintext $x$, $ChooseRun$ generates a number between 1 and $U$ and encrypts $x$ using the corresponding $Enc$ function, i.e., $y = Enc_{ChooseRun(x)}(x)$. $ChooseRun$ is a pseudo-random function with parameters $K_{map}$. In the case of Det-RPE, $ChooseRun$ and $Enc_1$, $Enc_2$, ..., $Enc_U$ all have to be deterministic. That means that Det-RPE can be composed with any OPE scheme referenced in this chapter so far [AKSX04, BCLO09, BCO11, MTY13, XYH12, WRG+13].

6.4.1 Construction

After having described the idea behind Det-RPE, let us formally define it by construction:

Construction 1. Let $OPE = (K, Enc, Dec)$ be a deterministic symmetric order-preserving encryption scheme and $ChooseRun(k, u, x)$ a deterministic, uniform, pseudo-random function that given as input a secret key $k$, an integer $u$ and an arbitrary value $x$, computes a number between 1 and $u$ with equal probability. We define a deterministic symmetric partially-order-preserving scheme, $Det$-$R$-$PE(K_{Det$-$R$-$PE}$, $Enc_{Det$-$R$-$PE}$, $Dec_{Det$-$R$-$PE}$, $U$), as follows:

- $K_{Det$-$R$-$PE}$ runs $K$ independently for each run and returns $U$ keys, namely $(K_1, ..., K_U)$. Also, it runs $K$ independently to generate $K_{map}$ for the $ChooseRun$ function.
- $Enc_{Det$-$R$-$PE}$ takes $K_u$ and $x$, as input where $u = ChooseRun(K_{map}, U, x)$. It returns $(u, y)$ where $y = Enc(K_u, x)$.
- $Dec_{Det$-$R$-$PE}$ takes $(u, y)$ as input and returns $x = Dec(K_u, y)$.

Table 6.2 gives an example of a set of customer names encrypted with Det-RPE with $U = 2$ using an OPE scheme. The performance and confidentiality characteristics of Det-RPE do not only depend on $ChooseRun$ and $Enc$, but also on the number of runs $U$: In the extreme case of $U = 1$, RPE is the same as $Enc$ which is typically a weak, yet high-performance encryption scheme. In the other extreme, $U = \infty$, RPE cannot be distinguished from a random mapping and hence has similar properties like a strong encryption scheme, e.g., AES-ECB [KL07], but with possibly much better performance as only relevant data has to be shipped to the (thin!) client.
6.4. Deterministic RPE

<table>
<thead>
<tr>
<th>Cleartext</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Code (⟨run, y⟩)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beatles</td>
<td>1</td>
<td>1</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>Beatles</td>
<td>1</td>
<td>1</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>Beatles</td>
<td>1</td>
<td>1</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>Elton John</td>
<td>1</td>
<td></td>
<td>(1, 1)</td>
</tr>
<tr>
<td>Madonna</td>
<td>2</td>
<td></td>
<td>(2, 2)</td>
</tr>
<tr>
<td>Madonna</td>
<td>2</td>
<td></td>
<td>(2, 2)</td>
</tr>
<tr>
<td>Metallica</td>
<td>3</td>
<td></td>
<td>(2, 3)</td>
</tr>
<tr>
<td>Nelly Furtado</td>
<td>2</td>
<td></td>
<td>(1, 2)</td>
</tr>
<tr>
<td>Tina Turner</td>
<td>3</td>
<td></td>
<td>(1, 3)</td>
</tr>
</tbody>
</table>

Table 6.2: Example of Det-RPE for \( U = 2 \)

6.4.2 Confidentiality under Domain Attack

This subsection analyses the confidentiality of different encryption schemes under domain attacks, including Det-RPE. The confidentiality is expressed as ROW advantage (cf. Subsection 6.3.1) against the different schemes and is proven directly or indirectly with a couple of lemmas. These ROW advantages can then be used to rank Det-RPE within related work.

An ordinary OPE scheme [AKSX04, BCLO09, BCO11, PLZ13, MTY13, XYH12, WRG+13] allows the domain adversary, \( \mathcal{A}^D \), to efficiently break the encryption solely by sorting the ciphertext values and map them to the sorted domain of plaintexts. In other words:

**Remark 1.** The ROW advantage of \( \mathcal{A}^D \) on OPE is:

\[
\text{Adv}^{\text{ROW}}_{\text{OPE}}(\mathcal{A}^D) = Pr[\text{Exp}^{\text{ROW}}_{\text{OPE}}(\mathcal{A}) = 1] = 1
\]  

(6.3)

On the other hand, a modular OPE scheme, as proposed by Boldyreva et al. [BCO11], is completely resilient against domain attacks.

**Lemma 1.** The ROW advantage of \( \mathcal{A}^D \) on MOPE is:

\[
\text{Adv}^{\text{ROW}}_{\text{MOPE}}(\mathcal{A}^D) = Pr[\text{Exp}^{\text{ROW}}_{\text{MOPE}}(\mathcal{A}) = 1] = \frac{1}{\mathcal{X}}
\]  

(6.4)

**Proof.** In order to win Experiment 6.1 on MOPE, the adversary needs to know the modular offset. Since the offset is chosen randomly from \( \mathcal{X} \), the adversary can guess it correctly with a probability of \( \frac{1}{\mathcal{X}} \).  

\[\square\]

123
Nevertheless, as we will see in Section 6.6, MOPE is vulnerable against known plaintext attacks.

The probabilistic OPE schemes [YZW06, EWSG04, KAK10a] have the following ROW advantage against a domain attack:

**Lemma 2.** The ROW advantage of $\mathcal{A}^D$ on Prob-OPE is:

$$Adv_{\text{Prob-OPE}}^{\text{ROW}}(\mathcal{A}^D) = Pr[Exp_{\text{Prob-OPE}}^{\text{ROW}}(\mathcal{A}^D) = 1] = \left(\frac{\text{Rank}_y - 1}{\text{Rank}_x - 1}\right) \left(\frac{Y - \text{Rank}_y}{X - \text{Rank}_x}\right) \left(\frac{Y - 1}{X - 1}\right) \tag{6.5}$$

Det-RPE amends all variants of OPE schemes to a great extent by randomly partitioning the domain into runs, thereby breaking the total order into $U$ partial orders. This is beneficial because the problem of finding the right total order out of $U$ partial orders is known to be inapproximable [GMR08, BS90].

**Lemma 3.** The ROW advantage of $\mathcal{A}^D$ on Det-RPE is defined as:

$$Adv_{\text{Det-RPE}}^{\text{ROW}}(\mathcal{A}^D) = Pr[Exp_{\text{Det-RPE}}^{\text{ROW}}(\mathcal{A}^D) = 1] = \left(\frac{\text{Rank}_x - 1}{\text{Rank}_{(y,r)} - 1}\right) \left(\frac{X - \text{Rank}_x}{Y - \text{Rank}_{(y,r)}}\right) \left(\frac{X}{Y}\right) \tag{6.6}$$

The probability distribution conforms to a negative hypergeometric probability distribution which applies to sampling without replacement from a finite population, in our case the domain $\mathcal{X}$. As random selections are made from the population, each subsequent draw decreases the population causing the probability of success to change with each draw. The detailed proofs of Equation (6.5) and Equation (6.6) are quite involved and can be found in our technical report [SBM+13] and Sanamrad’s thesis [San14].

Figure 6.4 compares the ROW advantage of the different analyzed encryption schemes in this section. As a baseline, we plot the advantage of an adversary that outputs a random $x$ regardless of the domain. This is considered to be ideal and is exactly what MOPE [BCO11] achieves. We see that Det-RPE amends OPE schemes (in this plot concretely ROPE [BCLO09]) with high ROW advantage to get closer to the ideal threshold with increasing the number of runs.

The equations presented in this section can be tuned to meet a tenant’s negligibility requirement. For instance, in Equation (6.6), depending on the domain size, the number of runs can be tuned to meet a given confidentiality/performance requirement. The higher the number of runs, the closer one gets to the ideal threshold, but also the bigger is the performance overhead.
6.5 Probabilistic RPE

In this section we present the probabilistic variant of randomly-partitioned encryption (Prob-RPE). RPE is made probabilistic in two ways: (a) by using a probabilistic order-preserving encryption scheme within a run, such as proposed by Yang et al. \cite{YZW06} and (b) by assigning the same plaintext value to different runs to guarantee database dynamism and improve confidentiality.

In Prob-RPE, ChooseRun is defined as a truly random function, \emph{i.e.}, for each plaintext element \(x\), ChooseRun randomly selects a run, \(u \overset{\$}{\leftarrow} \mathcal{U}\). Afterwards, \(x\) is to be encrypted in \(u\), using a probabilistic encryption scheme. Table 6.3 gives an example of such a scheme where the same value can even have multiple codes within a single run. For instance, \textit{Beatles} has two codes in \textit{Run 2} and one code in \textit{Run 1}, in other words \(\text{codes(Beatles)} = \{\langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 1, 1 \rangle\}\).
6.5.1 Construction

To formally construct Prob-RPE, we compose the probabilistic ChooseRun with a probabilistic order-preserving-encryption scheme (like the ones presented by Yang et al. [YZW06]) as follows:

**Construction 2.** Let $\mathcal{Prob-OLE}(K_p, \mathcal{Enc}_p, \mathcal{Dec}_p)$ be a probabilistic order-preserving encryption scheme as defined by Yang et al. [YZW06] and ChooseRun($u$) a random function that given as input an integer $u$, returns a number between 1 and $u$ with equal probability. We define a probabilistic RPE encryption scheme, $\mathcal{Prob-RPE}(K_{\mathcal{Prob-RPE}}, \mathcal{Enc}_{\mathcal{Prob-RPE}}, \mathcal{Dec}_{\mathcal{Prob-RPE}}, U)$, as follows:

- $K_{\mathcal{Prob-RPE}}$ runs $K_p$ independently for each run and returns $U$ keys, namely $(K_1, \ldots, K_U)$.
- $\mathcal{Enc}_{\mathcal{Prob-RPE}}$ takes $K_u$ and $x$, as input where $u = \text{ChooseRun}(U)$. It returns $(u, y)$ where $y = \mathcal{Enc}_p(K_u, x)$.
- $\mathcal{Dec}_{\mathcal{Prob-RPE}}$ takes $(u, y)$ as input and returns $x = \mathcal{Dec}_p(K_u, y)$.

### Table 6.3: Example of Prob-RPE for $U = 2$

<table>
<thead>
<tr>
<th>Cleartext</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Code $(\text{run}, y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beatles</td>
<td>1</td>
<td></td>
<td>$(2, 1)$</td>
</tr>
<tr>
<td>Beatles</td>
<td>2</td>
<td></td>
<td>$(2, 2)$</td>
</tr>
<tr>
<td>Beatles</td>
<td>1</td>
<td></td>
<td>$(1, 1)$</td>
</tr>
<tr>
<td>Elton John</td>
<td>3</td>
<td></td>
<td>$(2, 3)$</td>
</tr>
<tr>
<td>Madonna</td>
<td>2</td>
<td></td>
<td>$(1, 2)$</td>
</tr>
<tr>
<td>Madonna</td>
<td>4</td>
<td></td>
<td>$(2, 4)$</td>
</tr>
<tr>
<td>Metallica</td>
<td>3</td>
<td></td>
<td>$(1, 3)$</td>
</tr>
<tr>
<td>Nelly Furtado</td>
<td>4</td>
<td></td>
<td>$(1, 4)$</td>
</tr>
<tr>
<td>Tina Turner</td>
<td>5</td>
<td></td>
<td>$(2, 5)$</td>
</tr>
</tbody>
</table>

6.5.2 Confidentiality under Frequency Attack

As before, we assess the confidentiality of Prob-RPE by computing bounds on the adversary’s advantage and compare this to other encryption schemes. As Prob-RPE
was specifically designed to resist frequency attacks, we use the FOW advantage (cf. Subsection 6.3.2.1) as figure of merit this time.

A known weakness of deterministic encryption schemes is their inability to hide the original frequency distribution of the plaintext domain, which is especially bad if the plaintext domain has a skewed frequency distribution. Since OPE, MOPE and Det-RPE are all deterministic encryption schemes, the FOW advantage introduced in Subsection 6.3.2 of $\mathcal{A}^{DF}$ depends solely on the plaintext frequency distribution $F_X$. In other words, if the original frequency distribution is uniform, the adversary’s advantage is also uniform, which means optimal confidentiality. However, if the original frequency distribution is skewed, then the adversary’s advantage is larger and depends on the number of elements having the same frequency.

**Lemma 4.** Let $y = \text{Enc}(K, x)$ and $G = \{w | w \in \mathcal{X} \land \text{freq}_X(w) = \text{freq}_Y(y)\}$ be the set of distinct plaintext values having the same frequency as $y$. Then, the FOW advantage of the $\mathcal{A}^{DF}$ adversary on a deterministic encryption scheme $\text{Det}$ is defined as:

$$\text{Adv}^{FOW}_{\mathcal{A}^{DF}}(\mathcal{A}^{DF}) = \Pr[\text{Exp}^{FOW}_{\mathcal{A}^{DF}}(A) = 1] = \frac{1}{|G|}$$

(6.7)

**Proof.** In order to win Experiment 6.2 on $\text{Det}$, the adversary needs to choose a plaintext value having the same frequency as the given ciphertext in the experiment. The cardinality of set $G$ determines the number of possibilities of $\mathcal{A}^{DF}$ on $\text{Det}$. Hence, the game is won with a probability of $\frac{1}{|G|}$. \qed

**Remark 2.** A deterministic encryption scheme is optimally safe against a frequency attacks if and only if $F_X$ is uniform. A uniform $F_X$ implies a maximum $|G|$ where $|G| = X$, i.e., $|G|$ equals the domain size.

**Remark 3.** Prob-RPE as described in this section, takes care of a skewed probability distribution by creating a new ciphertext for each plaintext value. This way the skewed frequency distribution of the plaintext domain is mapped to a uniform frequency distribution in the ciphertext space, i.e., the FOW advantage under frequency attack is optimal which is $\frac{1}{X}$.

As depicted in Figure 6.5, deterministic OPE variants (including MOPE and Det-RPE) are exposed to frequency attacks if the data is skewed. Especially, values with a unique frequency (e.g., 8 in Figure 6.5) are leaked. On the other hand, an adversary’s chance to break Prob-RPE is as good as a random guess and hence ideal.
6.6 Modular RPE

An important observation to make from Equations (6.5) and (6.6) is that for RPE schemes, the ROW advantage not only depends on the domain size, but also on the rank of the plaintext within the domain. Consequently, extreme values of the domain (e.g., 1 and 64 in Figure 6.4) break the uniformity of the probability distribution. In order to fix this problem, RPE can be combined with a random shift cipher (as introduced by Boldyreva et al. [BCO11]) to form a modular ring structure and hide these extreme values. Compared to that work, we do not prepend the cipher to the plaintext, but append it to the ciphertext. This means that the underlying RPE scheme can be used without modifications and defining the modulo operation on the ciphertext is mathematically more sound because, compared to the plaintext domain (which can have any data type), the ciphertext space in RPE is always defined as a finite, contiguous range of integer numbers (cf. Section 6.7).

6.6.1 Construction

We formally construct modular RPE (Det-MRPE or Prob-MRPE depending on the underlying RPE scheme) as follows.
Construction 3. Let $\mathcal{RPE} = (K, \mathcal{E}_{nc}, \mathcal{D}_{ec}, U)$ be a symmetric (deterministic or probabilistic) RPE scheme with $U$ runs as defined in Construction 1, respectively Construction 2 and $\mathcal{R}an(d(w)$ a random function that given as input an integer $w$, returns a number between $0$ and $w − 1$ with equal probability. We define a modular symmetric RPE scheme, $\mathcal{MRPE}(K_{MRPE}, \mathcal{E}_{ncMRPE}, \mathcal{D}_{ecMRPE}, U)$ as follows:

- $K_{MRPE}$ runs $K$ to determine all keys required by $\mathcal{RPE}$. Additionally, $\mathcal{R}and(Y)$ is used to $U$ times to generate random offsets $o_1, o_2, \ldots o_U$ within the ciphertext space.
- $\mathcal{E}_{ncMRPE}(x)$ first computes $(u, y) = \mathcal{E}_{nc}(x)$ and then outputs the shifted ciphertext: $(u, y')$ where $y' = y + o_u \mod Y$.
- $\mathcal{D}_{ecMRPE}(u, y)$ first shifts back the ciphertext and then outputs the decrypted value $x = \mathcal{D}_{ec}(y')$ where $y' = y - o_u \mod Y$.

Table 6.4 shows an example of $Prob-MRPE$. Let us assume $Y = 6$ and random offsets $o_1 = 3, o_2 = 1$ for the sake of this small example. In that case, Metallica gets $\langle 1, (3 + 3) \mod 6 \rangle = \langle 1, 0 \rangle$ and Elton John becomes $\langle 2, (3 + 1) \mod 6 \rangle = \langle 2, 4 \rangle$.

<table>
<thead>
<tr>
<th>Ciphertext</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Code ($\langle$ run, y $\rangle$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beatles</td>
<td>2</td>
<td></td>
<td>$\langle 2, 2 \rangle$</td>
</tr>
<tr>
<td>Beatles</td>
<td>3</td>
<td></td>
<td>$\langle 2, 3 \rangle$</td>
</tr>
<tr>
<td>Beatles</td>
<td>4</td>
<td></td>
<td>$\langle 1, 4 \rangle$</td>
</tr>
<tr>
<td>Elton John</td>
<td>4</td>
<td></td>
<td>$\langle 2, 4 \rangle$</td>
</tr>
<tr>
<td>Madonna</td>
<td>5</td>
<td></td>
<td>$\langle 1, 5 \rangle$</td>
</tr>
<tr>
<td>Madonna</td>
<td>5</td>
<td></td>
<td>$\langle 2, 5 \rangle$</td>
</tr>
<tr>
<td>Metallica</td>
<td>0</td>
<td></td>
<td>$\langle 1, 0 \rangle$</td>
</tr>
<tr>
<td>Nelly Furtado</td>
<td>1</td>
<td></td>
<td>$\langle 1, 1 \rangle$</td>
</tr>
<tr>
<td>Tina Turner</td>
<td>0</td>
<td></td>
<td>$\langle 2, 1 \rangle$</td>
</tr>
</tbody>
</table>

Table 6.4: Example of Prob-MRPE for $U = 2$

6.6.2 Confidentiality under Known-Plaintext Attack

As shown in Lemma 1 and Figure 6.4, MOPE [BCO11] is secure against domain attacks because the random modular shift hides the extreme values in the beginning and the
end of the plaintext domain. For the same reason, \textit{Det-MRPE} and \textit{Prob-MRPE} improve the confidentiality of \textit{Det-RPE} and \textit{Prob-RPE} in the presence of such attacks such that the ROW advantage does no longer depend on the number of runs, but equals the best confidentiality that can be achieved (random guess).

However, if the adversary launches a known-plaintext attack (KPA), \textit{i.e.}, one plaintext-ciphertext pair is leaked to the adversary, she can “unwrap” the ring by sorting plaintexts and ciphertexts according to their distance do this “reference pair” [San14, p. 66]. In MOPE one such pair is enough to make the scheme degrade to ordinary OPE. In the case of \textit{modular RPE}, a known plaintext-ciphertext pair is needed for each \textit{run}. The resulting \textit{confidentiality downgrade} is illustrated in Table 6.5 and it supports a claim that we have made much earlier, in Section 6.1: despite \textit{MOPE} and \textit{Det-MRPE} having the same characteristics in Table 6.1, \textit{Det-MRPE} is clearly preferable because in the presence of KPA it degrades to \textit{Det-RPE} which has provably better confidentiality than OPE (which is what MOPE degrades to).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Encryption Scheme} & \textbf{Downgrade under KPA (Fall-back Scheme)} \\
\hline
MOPE & OPE \\
\textit{Det-MRPE} & \textit{Det-RPE} \\
\textit{Prob-MRPE} & \textit{Prob-RPE} \\
\hline
\end{tabular}
\caption{Confidentiality Downgrade under Known-Plaintext Attack (KPA)}
\end{table}

### 6.7 Implementation

So far, we have introduced the \textit{RPE} principles and have assessed the confidentiality of the different RPE variants. In order to measure their performance, we implemented them in Java. This section highlights some implementation details of the encryption schemes and shows to which extent these schemes preserve database functionality.

#### 6.7.1 Encryption

We constructed \textit{Det-MRPE} on top of \textit{ROPE} [BCLO09] where a pseudo-random lazy sampling algorithm has been used to map range gaps (in the ciphertext space) to domain
6.7. Implementation

gaps (in the plaintext space) in a recursive, binary search manner to determine the image of an input. We used a Java library from Cern to sample from the hypergeometric distribution [Zec16]. \( \text{chooseRun}(K_{\text{map}}, x) \) was implemented using Java’s standard random number generator seeded with \( K_{\text{map}} \cdot x \).

Two questions which were not fully answered by Boldyreva et al. [BCLO09] remain: (a) how to map arbitrary SQL data types to integer numbers and (b) how to efficiently sample from hypergeometric distribution if the population size becomes too big.\(^{10}\) Another question to be addressed is how to generate multiple cipher codes for the same plaintext within the same run in Prob-MRPE.

6.7.1.1 Mapping SQL types to integers

For each relevant SQL type, we have to define a mapping to (arbitrary-length) integers such that order is preserved. For most types, we can just take the binary representation, flip a couple of bits and then interpret the bit string as unsigned (arbitrary-length) integer. For signed integers (32 or 64 bits) for example, it is sufficient to flip the first bit. For IEEE floating point types, the first bit has to be flipped and if the number was originally negative, the remaining bits have to be flipped as well. Dates and time stamps are internally backed by integers, which solves the problem.

Remember that OPE schemes are intended to encrypt attributes for which we expect range and rank (top-N) queries. Therefore, encrypting arbitrary-length attributes, like texts or blobs, does not make sense. If these attributes are sensitive, one would rather use a strong encryption scheme and otherwise let them un-encrypted. Encrypting chars and var chars with an order-preserving encryption scheme can indeed make sense if we expect prefix queries like \( \text{WHERE } x \text{ LIKE 'A%' which essentially corresponds to } \text{WHERE } x \geq 'A\mu..\mu' \text{ AND } x \leq 'A\mu..M' \) where \( \mu \) corresponds to the smallest and \( M \) to the largest character in the used character encoding. For char\((n)\) strings, the character string is already order-preserving with respect to the character encoding, which is sufficient to process prefix queries. For varchar\((n)\), to preserve the same order, strings having less than \( n \) characters must be appended with the necessary number of \( \mu \) characters.

\(^{10}\)The Java library [Zec16] only implements sampling from a population with a size at most \( 2^{31} \) and if that maximal value is used, takes already quite long.
6.7.1.2 Efficient Sampling

As mentioned above, sampling from the hypergeometric distribution is expensive, especially for big population sizes. This means that for a data type with byte-size $n$, sampling from a population of $2^{8n}$ gets expensive if $n$ is bigger than one and clearly infeasible (and in the case of our used library impossible) if $n$ becomes bigger than four. Another question is also how to choose the size of the ciphertext which has to be at least twice as big as the domain space [BCLO09].

Our solution to this problem is to partition a given bit string into chunks of seven bits and encrypt each 7-bit plaintext word as 8-bit ciphertext word. For each chunk, a separate encryption box is used. This allows sampling from a population of $2^8$, which is still reasonably-fast. The danger of this approach is that an attacker can now attack the single 7-bit plaintexts separately to achieve at least partial decryption. This becomes especially bad if these 7-bit words correspond to useful information, e.g., to 7-bit ASCII characters. Luckily, modular RPE prevents this attacks, which is illustrated in Figure 6.6: As the random offset is chosen from the entire ciphertext space, which in the case of an $n$-byte data type has size $2^{8\lceil\frac{8n}{7}\rceil}$, addition of this offset very likely causes carry bits to “spill over” from one box to the next, thereby gluing the chunks together and making it impossible for an attacker to attack specific parts of the resulting ciphertext.

6.7.1.3 Multiple Ciphers within the same Run

In order to implement $\text{Prob-MRPE}$, we also need a way to encrypt the same values several times within the same run. In order to achieve this, we simply append each cleartext word with a random $b$-bit string before encryption as shown in Figure 6.7 where $z$ is the appended random bit string. As a result, we get up to $2^b$ codes for each value within the same run. $b$ is a parameter that should be configured with regard to the expected frequency of the most frequent value.

6.7.2 Database Functionality

A good way to understand to which extent $\text{Det-MRPE}$ and $\text{Prob-MRPE}$ can possibly hurt performance, it helps to understand what fraction of database functionality is preserved. The more functionality is preserved, the more query processing can happen in the cloud.
Figure 6.6: Example for the encryption of the string “ABBA” for datatype varchar(5) and a simple 7-bit ASCII character encoding: the string is first extended to 5 characters, then encrypted character-by-character using 7-to-8-bit encryption boxes and finally added to the random offset (modulo $2^{40}$). Carry bits between 8-bit chunks (bytes) are depicted in bold.

and the less work is left for the (thin) client, which is what is desirable for good overall performance (and possibly costs).

### 6.7.2.1 Range and Rank Queries

With regard to SQL queries, preserving functionality means to find a way how they can be executed directly on the encrypted data without the need for decryption. RPE was developed specifically with range and rank queries in mind and indeed such queries can be executed entirely on the encrypted data.

Range queries are defined with respect to a lower bound $l$ and upper bound $u$, both of which can be including or excluding. Such a query is rewritten by encrypting $u$ and $l$ for each run separately, using the specific $Enc$ function of each run, finally yielding
Chapter 6. Randomly-Partitioned Encryption

\[ x \rightarrow \text{ChooseRun} \rightarrow \text{RandConcat} \rightarrow b \]

\[ K_1, K_2, K_3, K_4 \rightarrow \text{Enc}_1, \text{Enc}_2, \text{Enc}_3, \text{Enc}_4 \]

\[ y \]

Figure 6.7: Generating several codes per Run

For \text{Prob-MRPE}, we do not choose \( z \) arbitrarily at random, but set it to \textit{all ones} or \textit{all zeros}, depending whether it is an upper or a lower bound and whether the bound is including or excluding. The final rewritten query asks for values that are either in the first run and between \( l_1 \) and \( u_1 \) or in the second run between \( l_2 \) and \( u_2 \) and so on. As the encryption includes a modular addition, it can happen that for some runs \( u_j < l_j \). In that case, the range condition changes from a \textit{between} to an \textit{or} predicate.

For instance, taking again Table 6.4 as an example, a typical range query (to retrieve all people that start with “M”) would be:

\[
\text{SELECT name from Person WHERE name LIKE 'M%'};
\]

Its rewritten form, using \text{Prob-MRPE} encryption, is:

\[
\text{SELECT name from Person WHERE}
\]
\[
\text{(name.run = 1 AND (name.value >= 5 OR name.value <1))}
\]
\[
\text{OR (name.run = 2 AND (name.value >= 5 AND name.value <=5))};
\]

We observe how for the first run the \textit{between} predicate changes into an \textit{or} and how the predicates for the different runs are connected together with additional \textit{or} predicates. We would like to highlight that with increasing \( U \), the rewritten range predicates can become

134
6.7. Implementation

quite large and at some point hit the limits of the SQL execution engine. However, even in that extreme case, RPE has still arguably performs better than semantically-secure schemes because only the relevant data has to be shipped to the (thin!) client.

*Top-N queries* (rank queries) are answered by asking the query separately for each *run*, possibly in parallel, and merging the $N \cdot U$ results in a $U$-way merge sort during post-processing. Clearly, this puts some additional burden on the client application, however, as $U$ is constant, the additional overhead stays within close bounds.

6.7.2.2 SQL Operator Overview

An overview of different SQL Operators and whether or not they can be implemented directly in the corresponding encryption scheme is shown in Table 6.6.\(^\text{11}\) We can see that the RPE variants support most of the operators, which allows them to be used in practice. Meanwhile, semantically-secure encryption schemes (*e.g.*, AES in CBC mode [KL07]) become inefficient as soon as *range*, *in*, *order-by* or *group-by* queries need to be processed.

6.7.2.3 Updates

For deterministic OPE schemes like *Det-MRPE* updates are no specific challenge because for each possible plaintext, there exists exactly one ciphertext. On the other hand, using a probabilistic scheme, like *Prob-MRPE*, updates might create a situation where we want to encrypt $x$ and all available codes for $x$ have already been used at least once. Most other probabilistic schemes assume a read-only database where all data is a-priori known and no updates can occur. In such a case, the encryption function can be configured to pre-allocate enough ciphertexts such that even the most frequent values have enough codes. Should updates occur anyway, one of the codes has be used twice, thereby decreasing confidentiality.

Prob-MRPE offers an additional way out: simply create a new run dynamically as needed. This implies that $U$ is no longer constant. Compared to all prior work, a tenant has the

\(^{11}\)While we sketched the ideas for range and rank queries, the details of the rewrite and post-processing algorithms of the remaining operators are described in Sanamrads-s thesis [San14, ch. 8f] as well as our technical report [SBM+ 13].
flexibility to choose what to do: either, let the algorithm decide when to increase $U$ and get predictable confidentiality at a marginal decrease in performance or fix $U$ and let the algorithm use the same ciphertext more than once (like in related work).

### 6.8 Performance Analysis and Experimental Results

In order to measure to what extent encryption and reduce database functionality influence query performance, we implemented the TPC-H benchmark [Tra16c] on the architecture shown in Figure 6.1. We have executed a number of experiments with different varying parameters, e.g., the number of runs or the number of codes per value in a single run for Prob-MRPE (cf. Subsection 6.7.1.3). We measured total response time, post-processing time, network cost and query compilation time. In the following, we focus on the most interesting results.
6.8. Performance Analysis and Experimental Results

6.8.1 Setup and Methodology

All experiments were conducted on two separate machines for client and server. The client was written in Java, ran on a machine with 24 GB of memory and communicated to the database server using JDBC. The server machine had 128 GB of memory available and hosted a MySQL 5.6 database. Both machines had 8 cores and ran a Debian-based Linux distribution. We measured end-to-end response time for all queries in separation, using a scaling factor of 10 (which means that the size of the plaintext data set is 10 GB).

Metrics used in aggregate functions (e.g., volume of orders) and surrogates (e.g., order numbers) were left un-encrypted while all other (sensitive) attributes, such as names, dates, etc. were encrypted. Whenever SQL operators on encrypted data were not supported, the entire data was shipped and then aggregated and filtered during the post-processing step in the encryption layer. Wherever the TPC-H benchmark defines an index on an attribute \( a \), we created the same index on the encrypted value of \( a \) as well as a composite index on \((a_{-run}, a)\).

6.8.2 General Encryption Overhead

Figure 6.8 shows the response time for different encryption functions compared to plain which is the response time for query processing on un-encrypted data. MOPE [BCO11] is a deterministic modular OPE scheme and was used as a gold standard to compare against the two different modular RPE variants presented in this chapter. We can see that for a majority of queries and RPE variants, response time is at most 2.5 times higher than plain and twice as high as MOPE. Sometimes, it is even smaller than the baselines because the partitioning (together with the composite indexes) enables parallelized query processing, which was employed in the presence of Top-N queries.

As expected, Prob-MRPE can add significant overhead. However, we believe this overhead to be reasonable as the security gain is substantial as shown in the previous sections of this chapter. We also measured the performance of deterministic AES-ECB [KL07], but did not include the results in the graph because most queries did not finish within 30 minutes and therefore had an overhead ranging from about 20x to 200x. AES-CBC [KL07], a representative of semantically-secure encryption schemes, would perform even worse.
6.8.3 Overhead for varying Number of Runs

The number of runs, \( U \), is an important parameter of all RPE schemes. Figure 6.9 shows how the response time of Det-MRPE increases with the number of runs for the first six TPC-H queries, again relative to plain. We chose these queries because they cover a wide spectrum of different operators (range predicates, aggregates, Top-N, sub-selects, etc.). For Q6, there is a significant (but still linear) increase in the response time with a growing number of runs. Q2 and Q3 are Top-N queries and are therefore executed with a degree of parallelism equal to \( U \), which result in a decreasing response time. Q1, Q4 and Q5 have a similar behavior like most TPCH-queries: their curves stay fairly flat. This shows that partitioning does generally not hurt RPE.

6.9 Related Work

To fill the gap between no-confidentiality/high-performance and high-confidentiality/low-performance, there have been a number of proposals for encryption techniques that support query processing without decrypting the data [HILM02, DDJ+03, AS00]. How-
ever, these approaches return a superset of the desired result and lose a lot of time into decrypting and filtering out the false positives.

On the other hand, the goal of fully homomorphic encryption (FHE) [Gen09, SV10, VDGHV10, EIG85, Pai99] is to strongly encrypt the data and process it directly without decryption. Wang et al. [WAEA12] show, at a conceptual level, how FHE can be used in databases. However, when taken to practice, databases explode in size because of the huge keys required by FHE. Therefore, the practicality of FHE in databases is an open question and no performance evaluation has been published so far.

Another important class of encryption techniques for databases are variants of order-preserving encryption (OPE), first introduced by Agrawal et al. [AKSX04].\textsuperscript{12} Examples include random (modular) OPE [BCLO09, BCO11], mutable OPE [PLZ13], indistinguishability-based OPE [MTY13], generalized OPE [XYH12, WRG\textsuperscript{+}13], structure-preserving database encryption [EWSG04], probabilistic OPE [YZW06], OPE with splitting and scaling [WL06], multi-valued (partial) OPE [KAK10a, KAK10b] and chaotic OPE [STJD\textsuperscript{+}09]. As detailed in Section 6.7, RPE is based on random (modular) OPE, but very different from all the other aforementioned OPE derivatives in the sense that it either solves a different problem (e.g., addresses different attacks) or uses different techniques. Systems

\textsuperscript{12}This paper got the SIGMOD Test of Time award in 2014, which is no surprise looking at the follow-up work.
that exploit OPE are CryptDB [PRZB11] and Monomi [TKMZ13]. RPE can be integrated into these systems to make them more secure.

Secure hardware has recently been exploited in the TrustedDB [BS11] and Cipherbase [ABE+13] projects. They are an interesting approach, but how to reach good performance, especially for analytical workloads, is still open. Again, RPE can be plugged-in into these systems in order to enable faster, but still confidential query processing.

Most commercial database products support strong encryption using AES [KL07], e.g., Oracle [Nan05] and Microsoft SQL Server [Hsu08]. Unfortunately, they only support encryption for data at rest at the disk level, which means that these approaches do not address attacks issued by a curious database administrator or any other party that can access the data in memory.

6.10 Concluding Remarks

This chapter presented different variants of randomly partitioned encryption, a set of novel methods for encrypting cloud databases, thereby addressing real world attacker scenarios like domain and frequency attacks. Moreover, we showed that the additional performance cost introduced by these new encryption schemes is reasonably small compared to the potential confidentiality gain.

Referring back to the optimization goals and research questions of Chapter 1, we observe that RPE can indeed compromise well between confidentiality and performance, using the number of runs as a tuning knob. Moreover, RPE inherits the elasticity and scalability properties from the underlying cloud database. If scalability is good, that means data can be partitioned and still efficiently processed, which in turn means that RPE can use considerably many runs without even sacrificing on performance.

What is more, using RPE allows processing range and rank queries directly on the encrypted data in the cloud. This perfectly complies with the assumption of thin clients who can thereby focus rather on decrypting the desired results than filtering out tons of false positives. As only the relevant data is shipped back to the client, we also save communication cost in the cloud, on the network and on the client side, which helps to reduce the overall TCO.
MTBase: Optimizing Cross-Tenant Database Queries

So far, this thesis focused on performance and confidentiality in cloud databases, or in other words, multi-tenant data management systems. The underlying assumption, as in most, if not all, of today’s cloud databases, was that each tenant queries her own data. All techniques proposed so far and most of the existing work in the research community focuses on single-tenant query processing. However, being the provider of a cloud database with thousands of tenants, who might even have their data structured similarly, opens a whole set of new business opportunities if we think about processing queries across all these different datasets.

In this chapter, we would like to take a step back and study how queries can be processed across multiple tenants at minimal costs and in a confidential manner. As we will see, standard SQL semantics are insufficient to process cross-tenant queries in an unambiguous way, which is why we will first propose a set of extensions to standard SQL, called MTSQL, to fix the ambiguity problem. Next, we will develop an architecture, MTBase, that processes MTSQL on top of an existing database by rewriting the queries. Finally, we will show how MTBase optimizes cross-tenant query processing such that the performance overhead compared to single-tenant execution becomes marginal.\(^1\)

---

\(^1\)Parts of this chapter are under submission to a database conference [BMTK17].
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

7.1 Introduction

As mentioned already several times in this thesis, cloud providers promise good elasticity, high availability and a fair pay-as-you-go pricing model to their tenants. All the big players, like Google [KG15], Amazon [Ama16], Microsoft [Mic16] and recently Oracle [Ora16b], have launched their own Database-as-a-Service (DaaS) cloud products, all of which are massively multi-tenant.

7.1.1 The Spectrum of Multi-Tenant Databases

As pointed out by Chong et al. [CCW06], the term *multi-tenant database* is ambiguous and can refer to a variety of DaaS schemes with different degrees of logical data sharing between tenants. On the other hand, as argued by Aulbach et al. [AGJ+08], multi-tenant databases not only differ in the way they logically share information between tenants, but also in the physical separation of information. We conclude that the *multi-tenancy spectrum* consists of four different layouts as shown in Figure 7.1.

![Figure 7.1: The Spectrum of Multi-Tenant Databases and their Applicability to Data Integration](image-url)
7.1. Introduction

First, there are DaaS products that offer each tenant her proper database while relying on physically-shared resources (SR), like CPU, network and storage. Examples include SAP HANA [SAP16], SqlVM [NDS\textsuperscript{+}13], RelationalCloud [MCM13] and Snowflake [DCZ\textsuperscript{+}16]. Next, there are systems that share databases (SD), but each tenant gets her own set of tables within such a database, as for example Azure SQL DB [DLNK16]. Finally, there are the two schemes where tenants not only share a database, but also the table layout (schema). Either, as for example in Apache Phoenix [Apa16b], tenants still have their private tables, but these tables have the same schema (SS) or the data of different tenants is consolidated into shared tables (ST) which is hence the layout with the highest degree of physical and logical sharing. SS and ST layouts are not only used in DaaS, but also in Software-as-a-Service (SaaS) platforms, as for example in Salesforce [WB09] and FlexScheme [AGJ\textsuperscript{+}08, ASJK11]. The main reason why these systems prefer ST over SS is cost [AGJ\textsuperscript{+}08]. Moreover, if the number of tenants exceeds the number of tables a database can hold (which is typically a number in the range of 10,000), SS becomes prohibitive. Conversely, ST databases can easily accommodate 100,000s to even millions of tenants.

7.1.2 Cross-Tenant Query Processing

An important use case for multi-tenancy databases, presumably not getting the deserved attention so far, is cross-tenant query processing. In Switzerland, for instance, three big institutions, namely Swisscom (a telco provider), Ringier (a big media provider) and SRF (Swiss national TV) have recently joined into a strategic marketing alliance [swi16]. This means nothing else than cross-tenant query processing to get better insights into how to drive their marketing campaigns. Another compelling use case is health care where many providers (and insurances) use the same integrated SaaS application. If the providers would agree to query their joint datasets of (properly anonymized) patient data with scientific institutions, this could enable medical research to advance much faster because the data can be queried as soon as it gets in.

This work looks into cross-tenant query processing within the scope of SS and ST databases, thereby optimizing a very specific sub-problem of data integration (DI). DI, in a broad sense, is about finding schema and data mappings between the original schemas of different data sources and a target schema specified by the client application [HMN\textsuperscript{+}99, FHH\textsuperscript{+}09, RH01]. As such, DI techniques are applicable to the entire
spectrum of multi-tenant databases of Figure 7.1 because even if tenants use different schemas or databases, these techniques can identify correlations and hence extract useful information. Our work embraces and builds on top of the latest DI work. More concretely, we optimize conversion functions similar to those used in DI by thoroughly analyzing and exploiting their algebraic properties. In addition, instead of translating data into a specific client format (and update periodically), we convert it to any required client format efficiently and just-in-time.

### 7.1.3 State of the Art in Cross-Tenant Query Processing

There are several existing approaches to cross-tenant query processing which are summarized in Figure 7.2: The first approach is data warehousing [KR11] where data is extracted from several data sources (tenant databases/tables), transformed into one common format and finally loaded into a new database where it can be queried by the client. This approach has high integration transparency in the sense that once the data is loaded, it is in the right format as required by the client and she can ask any query she wants. Moreover, as all data is in a single place, queries can be optimized. On the down-side of this approach, as argued in Subsection 2.2.2 and also observed in related work [NMK15, APM16], are costs in terms of both, developing and maintaining such ETL pipelines and maintaining a separate copy of the data. Another disadvantage is data staleness in the presence of frequent updates.

Federated Databases [Lev98, HLR02] reduce some of these costs by integrating data on demand (no copying). However, maintenance costs are still significant as for every new data source a new integrator/wrapper has to be developed. As data resides in different places (and different formats), queries can only be optimized to a very small extent (if at all), which is why the degree of integration transparency is considered sub-optimal. Finally, systems like SAP HANA [SAP16] and Salesforce [WB09], which are mainly tailored towards single-tenant queries, offer some degree of cross-tenant query processing, but only through their application logic, which means that the set of queries that can be asked is limited. Furthermore, as argued above, all of these related systems present results of cross-tenant queries in one specific static client format.
7.1.4 SQL Limitations

The reason why none of these approaches tries to use SQL is that it is ambiguous in the context of cross-tenant query processing. Consider, for instance the ST database in Figure 7.3, which we are going to use as a running example throughout the chapter. As soon as we want to query the joint dataset of tenants 0 and 1 and for instance join Employees with Roles, joining on role_id alone is not enough as this would also join Patrick with researcher and Ed with professor, which is clear nonsense. The obvious solution is to add the tenant identifier ttid to the join predicate. On the other hand, joining the Employees table with itself on E1.age = E2.age does not require ttid to be present in the join predicate because it actually makes sense to include results like (Alice, Ed) as they are indeed the same age. As ttid is an attribute
invisible to the end client, there is no way to distinguish the two cases (the one where ttid has to be included in the join and the one where it does not) in standard SQL. Another challenge arises from the fact that different tenants might store their employee’s salaries in different currencies. More concretely, two tenants might both have a salary column of type decimal(10,2), but one of them stores the values in USD and the other one in EUR. If this is the case, computing the average salary across all tenants clearly involves some value conversions that should, ideally, happen without the end client noticing or even worrying about.

7.1.5 Overview

This chapter presents MTSQL as a solution to these ambiguity problems. MTSQL extends SQL with additional semantics specifically-suited for cross-tenant query processing. It enables high integration transparency because any client (with any desired data format) can ask any query at any time. Moreover, as data resides in a single database (SS or ST), queries can be aggressively optimized with respect to both, standard SQL semantics and additional MTSQL semantics. As MTSQL adopts the single-database layout, it is also very cost-effective, especially if used on top of ST. Moreover, data conversion only happens as needed, which perfectly fits the cloud’s pay-as-you-go cost model.

Specifically, the chapter makes the following contributions:

- It defines the syntax and semantics of MTSQL, a query language that extends SQL with additional semantics suitable for cross-tenant query processing.
- It presents the design and implementation of MTBase, a database middleware that executes MTSQL on top of any ST database.
- It studies MTSQL-specific optimizations for query execution in MTBase.
- It extends the well-known TPC-H benchmark to run and evaluate MTSQL workloads.
- It evaluates the performance and the implementation correctness of MTBase with this benchmark.

The rest of this chapter is organized as follows: Section 7.2 defines MTSQL while Section 7.3 gives an overview on MTBase. Section 7.4 discusses the MTSQL-specific
optimizations, which are validated in Section 7.6 using the benchmark presented in Section 7.5. Section 7.7 shortly summarizes lines of related work, while conclusions are drawn in Section 7.8.

7.2 MTSQl

In order to model the specific aspects of cross-tenant query processing in multi-tenant databases, we developed MTSQl, which will be described in this section. MTSQl extends SQL in two ways: First, it extends the SQL interface with two additional parameters, C and D. C is the tenant identifier (or tid for short) of the client who submits a statement and hence determines the format in which the result must be presented. The data set, D, is a set of ttds that refer to the tenants whose data the client wants to query. Secondly, MTSQl extends the syntax and semantics of the SQL Query Language, Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL, consists of GRANT and REVOKE statements).

As mentioned in the introduction of this chapter, there are several ways for a multi-tenant database to be laid out: Figure 7.3 shows an example of the ST scheme, also referred to as basic layout in related work [AGJ+08] where tenants’ data is consolidated using the same tables. Meanwhile, Figure 7.4 illustrated the SS scheme, also referred to as private table layout, where every tenant has her own set of tables. In that scheme, data ownership is defined as part of the table name while in ST, records are explicitly annotated with the tid of their data owner, using an extra column in the table which is invisible to the end client.

As these two approaches are semantically equivalent, the MTSQl semantics that we are about to define apply to both. In the case of the SS, applying a statement s with respect to D simply means to apply s to the logical union of all private tables owned by a tenant in D. In SS, s is applied to tables filtered according to D. In order to keep the presentation simple, the rest of this chapter assumes an ST scheme, but sometimes defines semantics with respect to SS if that makes the presentation easier to understand.
MTBase: Optimizing Cross-Tenant Database Queries

<table>
<thead>
<tr>
<th>E_ttid</th>
<th>E_emp_id</th>
<th>E_name</th>
<th>E_role_id</th>
<th>E_reg_id</th>
<th>E_salary</th>
<th>E_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Patrick</td>
<td>1</td>
<td>3</td>
<td>50K</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>John</td>
<td>0</td>
<td>3</td>
<td>70K</td>
<td>28</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Alice</td>
<td>2</td>
<td>3</td>
<td>150K</td>
<td>46</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Allan</td>
<td>1</td>
<td>2</td>
<td>80K</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Nancy</td>
<td>2</td>
<td>4</td>
<td>200K</td>
<td>72</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Ed</td>
<td>0</td>
<td>4</td>
<td>1M</td>
<td>46</td>
</tr>
</tbody>
</table>

Employees (tenant-specific), $E_{\text{salary}}$ of tenant 0 in USD, $E_{\text{salary}}$ of tenant 1 in EUR

<table>
<thead>
<tr>
<th>R_ttid</th>
<th>R_role_id</th>
<th>R_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>phD stud.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>postdoc</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>professor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>intern</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>researcher</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>executive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Re_reg_id</th>
<th>Re_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AFRICA</td>
</tr>
<tr>
<td>1</td>
<td>ASIA</td>
</tr>
<tr>
<td>2</td>
<td>AUSTRALIA</td>
</tr>
<tr>
<td>3</td>
<td>EUROPE</td>
</tr>
<tr>
<td>4</td>
<td>N-AMERICA</td>
</tr>
<tr>
<td>5</td>
<td>S-AMERICA</td>
</tr>
</tbody>
</table>

Roles (tenant-specific) Regions (global)

Figure 7.3: Database in Basic Layout (ST), $X_{\text{ttid}}, R_{\text{ttid}}$ not visible to clients

### 7.2.1 MTSQl API

MTSQL needs a way to incorporate the additional parameters $C$ and $D$. As $C$ is the $ttid$ of the tenant that issues a statement, we assume it is implicitly given by the SQL connection string. $ttids$ are not only used for identification and access control, but also for data ownership (cf. Figure 7.4). While this chapter uses integers for simplicity reasons, $ttids$ can have any data type, in particular they can also be database user names.

$D$ is defined using the MTSQl-specific $\text{SCOPE}$ runtime parameter on the SQL connection. This parameter can be set in two different ways: Either, as shown in Listing 7.1, as simple scope with an $\text{IN}$ list stating the set of $ttids$ that should be queried or, as in Listing 7.2, as a sub-query with a $\text{FROM}$ and a $\text{WHERE}$ clause (complex scope). The semantics of the latter is that every tenant that owns at least one record in one of the tables mentioned in the $\text{FROM}$ clause that satisfies the $\text{WHERE}$ clause is part of $D$. The $\text{SCOPE}$ variable defaults to $\{C\}$, which means that by default a client processes only
7.2. MTSQl

<table>
<thead>
<tr>
<th>E_emp_id</th>
<th>E_name</th>
<th>E_role_id</th>
<th>E_reg_id</th>
<th>E_salary</th>
<th>E_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Patrick</td>
<td>1</td>
<td>3</td>
<td>50K</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>John</td>
<td>0</td>
<td>3</td>
<td>70K</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Alice</td>
<td>2</td>
<td>3</td>
<td>150K</td>
<td>46</td>
</tr>
</tbody>
</table>

**Employees_0 (private), E_salary in USD**

<table>
<thead>
<tr>
<th>E_emp_id</th>
<th>E_name</th>
<th>E_role_id</th>
<th>E_reg_id</th>
<th>E_salary</th>
<th>E_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Allan</td>
<td>1</td>
<td>2</td>
<td>80K</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>Nancy</td>
<td>2</td>
<td>4</td>
<td>200K</td>
<td>72</td>
</tr>
<tr>
<td>2</td>
<td>Ed</td>
<td>0</td>
<td>4</td>
<td>1M</td>
<td>46</td>
</tr>
</tbody>
</table>

**Employees_1 (private), E_salary in EUR**

<table>
<thead>
<tr>
<th>R_role_id</th>
<th>R_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>phD stud.</td>
</tr>
<tr>
<td>1</td>
<td>postdoc</td>
</tr>
<tr>
<td>2</td>
<td>professor</td>
</tr>
</tbody>
</table>

**Roles_0 (private)**

<table>
<thead>
<tr>
<th>Re_reg_id</th>
<th>Re_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AFRICA</td>
</tr>
<tr>
<td>1</td>
<td>ASIA</td>
</tr>
<tr>
<td>2</td>
<td>AUSTRALIA</td>
</tr>
<tr>
<td>3</td>
<td>EUROPE</td>
</tr>
<tr>
<td>4</td>
<td>N-AMERICA</td>
</tr>
<tr>
<td>5</td>
<td>S-AMERICA</td>
</tr>
</tbody>
</table>

**Regions (global)**

Figure 7.4: Database in Private Table Layout (SS)

her own data. Defining a simple scope with an empty IN list, on the other hand, makes D include all the tenants present in the database.

Making C and D part of the connection allowed a clear separation between the end users of MTSQl (for which ttids do not make much sense and hence remain invisible) and administrators/programmers that manage connections (and are aware of ttids).

```sql
SET SCOPE = "IN (1,3,42)";
```

Listing 7.1: Simple SCOPE expression using IN
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

```
SET SCOPE = "FROM Employees WHERE E_salary > 180K";
```

Listing 7.2: Complex SCOPE expression with sub-query

### 7.2.2 Data Definition Language

DDL statements are issued by a special role called the *data modeler*. In a multi-tenant application, this would be the SaaS provider (*e.g.*, a Salesforce administrator) or the provider of a specific application. However, the data modeler can delegate this privilege to any tenant she trusts using a *GRANT* statement, as will be described in Subsection 7.2.3.

There are two types of tables in MTSQL: tables that contain common knowledge shared by everybody (like *Regions*) and those that contain data of a specific tenant (like *Employees* and *Roles*). More formally, we define the *table generality* of *Regions* as *global* and the one of all other tables as *tenant-specific*. In order to process queries across tenants, MTSQL needs a way to distinguish whether an attribute is *comparable* (can be directly compared against attribute values of other tenants), *convertible* (can be compared against attribute values of other tenants after applying a well-defined *conversion function*) or *tenant-specific* (it does semantically not make sense to compare against attribute values of other tenants). An overview of these types of *attribute comparability*, together with examples from Figure 7.3, is shown in Table 7.1.

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparable</td>
<td>can be directly compared to and aggregated with other values</td>
<td>E_region_id, E_age, Re_name, R_region_id, R_name</td>
</tr>
<tr>
<td>convertible</td>
<td>other values need to be converted to the format of the current tenant before comparison or aggregation</td>
<td>E_salary</td>
</tr>
<tr>
<td>tenant-specific</td>
<td>values of different tenants cannot be compared with each other</td>
<td>E_role_id, R.role_id</td>
</tr>
</tbody>
</table>

Table 7.1: Overview on Attribute Comparability in MTSQL


7.2.2.1 CREATE TABLE Statement

The MTSQL-specific keywords for creating (or altering) tables are `GLOBAL`, `SPECIFIC`, `COMPARABLE` and `CONVERTIBLE`. An example of how they can be used is shown in Listing 7.3. Note that `SPECIFIC` can be used for tables and attributes. Moreover, using these keywords is optional as we define that tables are global by default, attributes of tenant-specific tables default to `tenant-specific` and those of global tables to `comparable`.²

```
1 CREATE TABLE Employees SPECIFIC (
2   E_emp_id INTEGER NOT NULL SPECIFIC,
3   E_name VARCHAR(25) NOT NULL COMPARABLE,
4   E_role_id INTEGER NOT NULL SPECIFIC,
5   E_reg_id INTEGER NOT NULL COMPARABLE,
6   E_salary VARCHAR(17) NOT NULL CONVERTIBLE @currencyToUniversal
      @currencyFromUniversal,
7   E_age INTEGER NOT NULL COMPARABLE,
8   CONSTRAINT pk_emp PRIMARY KEY (E_emp_id),
9   CONSTRAINT fk_emp FOREIGN KEY (E_role_id) REFERENCES Roles (R_role_id)
10 );
```

Listing 7.3: exemplary MTSQL CREATE TABLE statement, MT-specific keywords marked in red

7.2.2.2 Conversion Functions

Cross-tenant query processing requires the ability to execute comparison predicates on `comparable` and `convertible` attributes. While comparable attributes can be directly compared to each other, convertible attributes, as their name indicates, have to be converted first, using conversion functions. Each tenant has a pair of conversion functions for each attribute to translate from and to a well-defined universal format. More formally, a conversion function pair is defined as follows:

²In fact, global tables, as they are shared between all tenants, can only have comparable attributes anyway.
Definition 1. \((\text{toUniversal} : X \times T \to X, \text{fromUniversal} : X \times T \to X)\) is a valid MTSQ\textsc{L} conversion function pair for attribute \(A\), where \(T\) is the set of tenants in the database and \(X\) is the domain of \(A\), if and only if:

(i) There exists a \textit{universal format} for attribute \(A\):

\[
\text{image}(\text{toUniversal}(\cdot, t_1)) = \text{image}(\text{toUniversal}(\cdot, t_2)) = \ldots = \text{image}(\text{toUniversal}(\cdot, t_{|T|}))
\]

(ii) For every tenant \(t \in T\), the partial functions \(\text{toUniversal}(x, t)\) and \(\text{fromUniversal}(x, t)\) are well-defined, bijective functions.

(iii) \(\text{fromUniversal}\) is the inverse of \(\text{toUniversal}\):

\[
\forall t \in T, x \in X : \text{fromUniversal}(\text{toUniversal}(x, t), t) = x
\]

These three properties imply the following two corollaries that we are going to need later in this chapter:

Corollary 1. \(\text{toUniversal}\) and \(\text{fromUniversal}\) are \textit{equality-preserving}:

\[
\forall t \in T : \text{toUniversal}(x, t) = \text{toUniversal}(y, t) \iff x = y \iff \text{fromUniversal}(x, t) = \text{fromUniversal}(y, t)
\]

Corollary 2. Values from any tenant \(t_i\) can be converted into the representation of any other tenant \(t_j\) by first applying \(\text{toUniversal}(\cdot, t_i)\), followed by \(\text{fromUniversal}(\cdot, t_j)\) while equality is preserved:

\[
\forall t_i, t_j \in T : x = y \iff \text{fromUniversal}(\text{toUniversal}(x, t_i), t_j) = \text{fromUniversal}(\text{toUniversal}(y, t_i), t_j)
\]

The reason why we opted for a two-step conversion through universal format is that it allows each tenant \(t_i\) to define her share of the conversion function pair \((\text{toUniversal}(\cdot, t_i), \text{fromUniversal}(\cdot, t_i))\) individually without the need of a central authority. Moreover, this design greatly reduces the overall number of partial conversion functions as we need at most \(2 \cdot |T|\) partial function definitions, compared to \(|T|^2\) functions in the case where we would define a direct conversion for every pair of tenants.
7.2.2.3 Further Conversion Function Properties

Listings 7.4 and 7.5 show an example of such a conversion function pair. These functions are used to convert phone numbers with different prefixes (like “+”, “00” or any other specific county exit code\(^3\)) and the universal format is a phone number without prefix. In this example, converting phone numbers simply means to lookup the tenant’s prefix and then either prepend or remove it, depending whether we convert from or to the universal format. Note that the exemplary code also contains the keyword IMMUTABLE to state that for a specific input, the function always returns the same output, which is an important hint for the (PostgreSQL) query optimizer. While this keyword is PostgreSQL-specific, some other vendors, but by far not all, offer a similar syntax.

```sql
1 CREATE FUNCTION phoneToUniversal (VARCHAR(17), INTEGER) RETURNS VARCHAR(17)
2   AS 'SELECT SUBSTRING($1, CHAR_LENGTH(PT_prefix)+1) FROM Tenant,
3      PhoneTransform WHERE T_tenant_key = $2 AND T_phone_prefix_key =
4      PT_phone_prefix_key;'  
5 LANGUAGE SQL IMMUTABLE;
```

Listing 7.4: Converting a phone number to universal form (without prefix), PostgreSQL syntax

```sql
1 CREATE FUNCTION phoneFromUniversal (VARCHAR(17), INTEGER) RETURNS VARCHAR(17)
2   AS 'SELECT CONCAT(PT_prefix, $1) FROM Tenant, PhoneTransform WHERE
3      T_tenant_key = $2 AND T_phone_prefix_key = PT_phone_prefix_key;'  
4 LANGUAGE SQL IMMUTABLE;
```

Listing 7.5: Converting to a specific phone number format, PostgreSQL syntax

It is important to mention that Corollary 1 is a minimal requirement for conversion functions to make sense in terms of producing coherent query results among different clients. There are, however conversion functions that exhibit additional properties, as for instance being:

\(^3\)The country exit code is a sequence of digits that one must to dial in order to inform the telco system that one intends to call a number abroad. A full list of country exit codes can be found on [http://www.howtocallabroad.com/codes.html](http://www.howtocallabroad.com/codes.html).
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

- order-preserving with respect to tenant $t$:
  \[ x < y \iff \text{toUniversal}(x, t) < \text{toUniversal}(y, t) \]

- homomorphic with respect to tenant $t$ and function $h$:
  \[ \text{toUniversal}(h(x_1, x_2, \ldots, t) = h(\text{toUniversal}(x_1, t), \text{toUniversal}(x_2, t), \ldots) \]

We will call a conversion function pair **fully-order-preserving** if $\text{toUniversal}$ and $\text{fromUniversal}$ are order-preserving with respect to all tenants. Consequently, a conversion function pair can also be **fully-$h$-preserving**.

Listings 7.6 and 7.7 show an exemplary conversion function pair used to convert currencies (with USD as universal format). These functions are not only equality-preserving, but also fully-SUM-preserving: as the currency conversion is nothing but a multiplication with a constant factor\(^4\) from $\text{CurrencyTransform}$, it does not matter in which format we sum up individual values (as long as they all have that same format). As we will see, such special properties of conversion functions are a crucial ingredient for query optimization.

```sql
1 CREATE FUNCTION currencyToUniversal (DECIMAL(15,2), INTEGER) RETURNS DECIMAL(15,2)
2   AS 'SELECT CT_to_universal*$1 FROM Tenant, CurrencyTransform WHERE T_tenant_key = $2 AND T_currency_key = CT_currency_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 7.6: Converting a currency to universal form (USD), PostgreSQL syntax

```sql
1 CREATE FUNCTION currencyFromUniversal (DECIMAL(15,2), INTEGER) RETURNS DECIMAL(15,2)
2   AS 'SELECT CT_from_universal*$1 FROM Tenant, CurrencyTransform WHERE T_tenant_key = $2 AND T_currency_key = CT_currency_key;'
3 LANGUAGE SQL IMMUTABLE;
```

Listing 7.7: Converting from USD to a specific currency, PostgreSQL syntax

\(^4\)We are aware of the fact that currency conversion is not at all constant, but depends on rapidly-changing exchange rates. In this chapter, we want to keep the examples as simple as possible in order to facilitate the illustration of our concepts. However, the presented ideas also apply to temporal databases, which means that dynamic exchange rates could be taken into account.
The conversion function examples shown in Listings 7.4 to 7.7 assume the existence of tables `PhoneTransform` and `CurrencyTransform`, holding additional conversion information, as well as a `Tenants` table with references into these tables. The way how a tenant can define her portion of the conversion functions is then simply to choose a specific currency and phone format as part of an initial setup procedure. However, this is only one possible implementation. MTSQL does not make any assumptions (or restrictions) on the implementation of conversion function pairs themselves, as long as they satisfy the properties given in Definition 1.

MTSQL is not the first work that talks about conversion functions. In fact, as mentioned in Subsection 7.1.2, there is an entire line of work on schema mapping techniques [HMN+99, FHH+09, AGJ+08]. These works mention and take into account conversion functions, like for example a multiplication or a division by a constant. More complex conversion functions, including regular-expression-based substitutions and other arithmetic operations, can be found in *Potter’s Wheel* [RH01] where conversion is referred to as value translation. All these different conversion functions can potentially also be used in MTSQL which is, to the best of our knowledge, the first work that formally defines and categorizes conversion functions according to their properties.

### 7.2.2.4 Integrity Constraints

MTSQL allows global integrity constraints that every tenant has to adhere to (with respect to the entirety of her data) as well as tenant-specific integrity constraints (that tenants can additionally impose on their own data). An example of a global referential integrity constraint is shown in the end of Listing 7.3. This constraint means that for every tenant, for each entry of `E_role_id`, a corresponding entry `R_role_id` has to exist in `Roles` and must be owned by that same tenant. Consider for example employee *John* with `R_role_id 0`. The constraint implies that their must be a `role 0` owned by tenant 0, which in that case is *PhD student*. If the constraint were only tenant-specific for tenant 1, John would not link to roles and `E_role_id 0` would just be an arbitrary numerical value. In order to differentiate global from tenant-specific constraints, the scope is used.\(^5\)

\(^5\)Remembering that an empty IN list refers all tenants, this is exactly what is used to indicate a global constraint. Additionally, all constraints created as part of a `CREATE TABLE` statement are global as well.
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

7.2.2.5 Other DDL Statements

CREATE VIEW statements look the same as in plain SQL. As for the other DDL statements, anyone with the necessary privilege can define global views on global and tenant-specific tables. Tenants are allowed creating their own, tenant-specific views (using the default scope). The selected data has to be presented in universal format if it is a global view and in the tenant-specific format otherwise. DROP VIEW, DROP TABLE and ALTER TABLE work the same way as in plain SQL.

7.2.3 Data Control Language

Let us have a look at the MTSQL GRANT statement:

```sql
GRANT <privileges> ON <database|table> TO <ttid>;
```

Listing 7.8: MTSQL GRANT syntax

As in plain SQL, this grants a specific set of access privileges (READ, INSERT, UPDATE, DELETE) to the tenant identified by `ttid`. In the context of MTSQL, however, this means that the privileges are granted with respect to `C`. Consider the following statement:

```sql
GRANT READ ON Employees TO 42;
```

Listing 7.9: Example of an MTSQL GRANT statement

In the private table layout, if `C` is 0, then this would grant tenant 42 read access to Employees_0, but if `C` is 1, tenant 42 would get read access to Employees_1 instead. If a grant statement grants to ALL, then the grant semantics also depend on `D`, more concretely if `D = {7, 11, 15}` the privileges would be granted to tenants 7, 11 and 15.

By default, a new tenant that joins an MTSQL system is granted the following privileges: READ access to global tables and READ, INSERT, UPDATE, DELETE, GRANT, REVOKE on his own instances of tenant-specific tables. In our example, this means that a new tenant 111 can read and modify data in Employees_111 and Roles_111. Next, a tenant can start asking around to get privileges on other tenants' tables or also on global
tables. The `REVOKE` statement, as in plain SQL, simply revokes privileges that were granted with `GRANT`.

### 7.2.4 Query Language

Just as in FlexScheme [AGJ+08, ASJK11], queries themselves are written in plain SQL and have to be filtered according to $D$. Whereas in FlexScheme $D$ always equals $\{C\}$ (a tenant can only query her own data), MTSQ allows cross-tenant query processing, which means that the data set can include other tenants than $C$ and can in particular be bigger than one. As mentioned in the introduction, this creates some new challenges that have to be handled with special care.

#### 7.2.4.1 Client Presentation

As soon as tenants can query other tenants’ data, the MTSQ engine has to be make sure to deliver results in the proper format. For instance, looking again at Figure 7.3, if tenant 0 queries the average salary of all employees of tenant 1, then this should be presented in USD because tenant 0 stores her own data in USD and expects other data to be in USD as well. Consequently, if tenant 1 would ask that same query, the result would be returned as is, namely in EUR.

#### 7.2.4.2 Comparisons

Consider a join of `Roles` and `Employees` on `reg_id`. As long as the dataset size is only one, such a join query has the same semantics as in plain SQL (or FlexScheme). However, as soon as tenant 1, for instance, asks this query with $D = \{0, 1\}$, the join has to take the `ttids` into account. The reason for this is that `reg_id` is a tenant-specific attribute and should hence only be joined within the same tenant in order to prevent semantically wrong results like John being an intern (although tenant 0 does not have such a role) or Nancy being a professor (despite the fact that tenant 1 only has roles `intern`, `researcher` and `executive`).

Comparison or join predicates containing `comparable` and `convertible` attributes, on the other hand, just have to make sure that all data is brought into universal format before
being compared. For instance, if tenant 0 wants to get the list of all employees (of both
tenants) that earn more than 100K USD, all employee salaries have to be converted to
USD before executing the comparison.

Finally, MTSQL does not allow comparing tenant-specific with other attributes. For
instance, we see no way how it could make sense to compare $E_{\text{role_id}}$ to something
like $E_{\text{age}}$ or $E_{\text{salary}}$.

### 7.2.5 Data Manipulation Language

MTSQL DML works the same way as in FlexScheme [AGJ+08, ASJK11] if $D = \{C\}$. Otherwise, if $D \neq \{C\}$, the semantics of a DML statement are defined such that it is
applied to each tenant in $D$ separately. Constants, \texttt{WHERE} clauses and sub-queries are
interpreted with respect to $C$, exactly the same way as for queries (cf. Subsection 7.2.4).
This implies that executing \texttt{UPDATE} or \texttt{INSERT} statements might involve value conversion
to the proper tenant format(s).

### 7.3 MTBase

Based on the language specification described in the previous section, we implemented
MTBase, an open-source MTSQL engine [Sys16c]. As shown in Figure 7.5, the basic
building block of MTBase is an MTSQL-to-SQL translation middleware sitting between
a traditional database management system (DBMS) and the clients. In fact, as it com-
municates to the DBMS by the means of pure SQL, MTBase works in conjunction with
any off-the-shelf DBMS. For performance reasons, the middleware maintains a cache
of MT-specific meta data, which is persisted in the DBMS along with the actual user
data. Conversion functions are implemented as \textit{user-defined functions} (UDFs) that might
involve additional meta tables, both of which are also persisted in the DBMS. MTBase
implements the basic data layout, which means that data ownership is implemented
as an additional $ttid$ (meta) column in each tenant-specific table as illustrated in Fig-
ure 7.3. There are some dedicated meta tables: Tenant stores each tenant’s privileges
and conversion information and Schema stores information about table and attribute
comparability. Additional meta tables can (but do not have to) be used to implement
conversion function pairs, \textit{e.g.}, CurrencyTransform and PhoneTransform shown in Listings 7.4 to 7.7.

![MTBase Architecture](image)

Figure 7.5: MTBase Architecture

While the rewrite module was implemented in Haskell and compiled with GHC [GHC16], the connection handling and the meta data cache maintenance was written in Python and run with the Python2 interpreter [Pyt16]. Haskell is handy because we can make full use of pattern matching and additive data types to implement the rewrite algorithm in a quick and easy-to-verify way, but any other functional language, \textit{e.g.}, Scala [Eco16], would also do the job. Likewise, there is nothing fundamental in using Python, any other framework that has a good-enough abstraction of SQL connections, \textit{e.g.}, JDBC [Ora16a], could be used.

### 7.3.1 Client-Server Protocol

Upon opening a connection at the middleware, the client’s \textit{ttid}, \textit{C}, is derived from the connection string and used throughout the entire lifetime of that connection. Whenever a client sends a MTSQL statement \(s\), first if the current scope is complex, a SQL query \(q_s\) is derived from this scope and evaluated at the DBMS in order to determine the relevant dataset \(D\). After that, \(D\) is compared against privileges of \(C\) in the Tenant table and \textit{ttids} in \(D\) without the corresponding privilege are pruned, resulting in \(D'\). Next, \(C\), \(D'\) and \(s\) are input into the rewrite algorithm which produces a rewritten SQL statement \(s'\) which is then sent to the DBMS before relaying the result back to the client. Note that in order to guarantee correctness in the presence of updates, \(q_s\) and \(s'\) have to be executed within the same transaction and with a consistency level at least \textit{repeatable-read} [BBG\textsuperscript{+}95b],...
even if the client does not impose any transactional guarantees. If $s$ is a DDL statement, the middleware also updates the MT-Specific meta information in the DBMS and the cache.

The rest of this section explains the MTSQ-L-to-SQL rewrite algorithm in its canonical form and proves its correctness with respect to Subsection 7.2.4, while Section 7.4 shows how to optimize the rewritten queries such that they can be run on the DBMS with reasonable performance.

### 7.3.2 Canonical Query Rewrite Algorithm

Our proposed canonical MTSQ-L-to-SQL rewrite algorithm works top-down, starting with the outer-most SQL query and recursively rewriting sub-queries as they come along. For each sub-query, the SQL clauses are rewritten one-by-one. The algorithm makes sure that for each sub-query the following invariant holds: the result of the sub-query is filtered according to $D'$ and presented in the format required by $C$.

The pseudo code of the general rewrite algorithm for rewriting a (sub-)query is shown in Algorithm 7.1. Note that **FROM**, **GROUP BY**, **ORDER BY** and **HAVING** clause can be rewritten without any additional context while **SELECT** and **WHERE** need the whole query as an input because they might need to check the **FROM** for additional information, for instance they must know to which original tables certain attributes belong.

```
1: Input: $C$: ttid, $D$: set of ttids, $Q$: MTSQ-L query
2: Output: SQL query
3: function RewriteQuery($C$, $D$, $Q$)
4:     new-select ← rewriteSelect($C$, $D$, $Q$)
5:     new-from ← rewriteFrom($C$, $D$, $Q$.from())
6:     new-where ← rewriteWhere($C$, $D$, $Q$)
7:     new-group-by ← rewriteGroupBy($C$, $D$, $Q$.groupBy())
8:     new-order-by ← rewriteOrderBy($C$, $D$, $Q$.orderBy())
9:     new-having ← rewriteHaving($C$, $D$, $Q$.having())
10: return new Query (new-select, new-from, new-where, new-group-by, new-order-by, new-having)
```

**Algorithm 7.1: Canonical Query Rewrite Algorithm**
In the following, we will look at the rewrite functions for the different SQL clauses. As explaining this in detail would take the space of an entire thesis on its own, we only provide the high-level ideas and illustrate them with suitable minimal examples. However, we strongly encourage the interested reader to check-out the Haskell code [Sys16d], which in fact almost reads like a mathematical definition of the rewrite algorithm.

**SELECT**  The rewritten SELECT clause has to present every attribute \( a \) in \( C \)'s format, which, if \( a \) is convertible, is achieved by two calls to the conversion function pair of \( a \) as can be seen in the examples of Listing 7.10. If \( a \) is part of compound expression (as in line 6), it has to be converted before the functions (in that case \( \text{AVG} \)) are applied. Note that in order to make a potential super-query work correctly, we also rename the result of the conversion, either by the new name that it got anyway (as in line 6) or by the name that it had before (as in line 3). Rewriting a star expression (line 9) in the uppermost query also needs special attention, in order not to provide the client with confusing information, like \( \text{ttid} \) which should stay invisible.

```
1  -- Rewriting a simple select expression:
2  SELECT E_salary FROM Employees; -->
3  SELECT currencyFromUniversal(currencyToUniversal(E_salary, ttid), C) as
   salary FROM Employees;
4  -- Rewriting an aggregated select expression
5  SELECT AVG(E_salary) as avg_sal FROM Employees; -->
6  SELECT AVG(currencyFromUniversal(currencyToUniversal(E_salary, ttid), C))
      as avg_sal FROM Employees;
7  -- Rewriting star expression, hiding irrelevant info
8  SELECT * FROM Employees; -->
9  SELECT E_name, E_reg_id, E_salary, E_age FROM Employees;
```

**Listing 7.10: Examples for Rewriting SELECT clause**

**WHERE**  There are essentially three steps that the algorithm has to perform in order to create a correctly rewritten WHERE clause (cf. Listing 7.11). First, conversion functions have to be added to each convertible attribute in each predicate in order make sure that comparisons are executed in the correct (client) format (lines 2 to 6). This happens the same way as for a SELECT clause. Notably, all constants are always in \( C \)'s format because it is \( C \) who asks the query. Second, for every predicate involving two or more
tenant-specific attributes, additional predicates on \( \text{ttid} \) have to be added (line 9), except if the attributes are part of the same table, which means they are owned by the same tenant anyway. Predicates that contain tenant-specific together with other attributes cause the entire query to be rejected as was required in Subsection 7.2.4.2. Last, but not least, for every base table in the FROM clause, a so-called \( D \)-filter has to be added to the WHERE clause (line 12). This filter makes sure that only the relevant data (data that is owned by a tenant in \( D' \)) gets processed.

\begin{lstlisting}[language=sql]
1 -- Comparison with a constant:
2 ... FROM Employees WHERE E_salary > 50K -->
3 ... WHERE currencyFromUniversal(currencyToUniversal(E_salary,ttid),C) > 50K) ...  
4 -- General comparison:
5 ... FROM Employees E1, Employees E2 WHERE E1.E_salary > E2.E_salary -->
6 ... WHERE currencyFromUniversal(currencyToUniversal(E1.E_salary,E1.ttid), C) > currencyFromUniversal(currencyToUniversal(E1.E_salary,E1.ttid),C) ...  
7 -- Extend with predicate on \( \text{ttid} \)
8 ... FROM Employees, Roles WHERE E_role_id = R_role_id -->
9 ... FROM Employees, Roles WHERE E_role_id = R_role_id AND Employees.ttid = Roles.ttid ...  
10 -- Adding \( D \)-filters for \( D' = \{3,7\} \)
11 ... FROM Employees E, Roles R ... -->
12 ... WHERE E.ttid IN (3,7) AND R.ttid IN (3,7) ...
\end{lstlisting}

Listing 7.11: Examples for Rewriting WHERE clause

**FROM** All tables referred by the FROM clause are either base tables or temporary tables derived from a sub-query. Rewriting the FROM clause simply means to call the rewrite algorithm on each referenced sub-query as shown in Algorithm 7.2. A FROM table might also contain a JOIN of two tables (sub-queries). In that case, the two sub-queries are rewritten and then the join predicate is rewritten in the exact same way like a WHERE clause.

Notably, this algorithm preserves the desired invariant for (sub)-queries: the result of each sub-query is in client format and filtered according to \( D' \) and, due to the rewrite of the SELECT and the WHERE clause of the current query, base tables are also presented.
1: **Input:** $C$: ttid, $D$: set of ttids, $FromClause$: MTSQL FROM clause
2: **Output:** SQL FROM clause
3: **function** RewriteFrom($C$, $D$, $FromClause$)
4: $res ← extractBaseTables ($FromClause$)$
5: for all $q ∈ extractSubQueries ($FromClause$)$ do
6: $res ← res ∪ \{ rewriteQuery ($C$, $D$, $q$) \}$
7: for all $(q_1, q_2, cond) ∈ extractJoins ($FromClause$)$ do
8: $q'_1 ← rewriteQuery ($C$, $D$, $q_1$)$
9: $q'_2 ← rewriteQuery ($C$, $D$, $q_2$)$
10: $cond' ← rewriteWhere ($C$, $D$, $cond$)$
11: $res ← res ∪ \{ createJoin ($q'_1$, $q'_2$, $cond'$) \}$
12: return $res$

Algorithm 7.2: Rewrite Algorithm for FROM clause

in client format and filtered by $D$. So are joins. We conclude that the result of the current query therefore also preserves the invariant.

**GROUP-BY, ORDER-BY and HAVING** HAVING and GROUP-BY clauses are basically rewritten the same way like the expressions in the SELECT clause. Some DBMSs might throw a warning stating that grouping by a comparable attribute $a$ is ambiguous because the way we rewrite $a$ in the WHERE clause and rename it back to $a$, we could actually group by the original or by the converted attribute $a$. However, the SQL standard clearly says that in such a case, the result should be grouped by the outer-more expression, which is exactly what we need. ORDER-BY clauses need not be rewritten at all.

**SET SCOPE** Simple scopes do not have to be rewritten at all. The FROM and WHERE clause of a complex scope are rewritten the same way as in a sub-query. In order to make it a valid SQL query, the rewrite algorithm adds a SELECT clause that projects on the respective ttids as shown in Listing 7.12.

```
1 SET SCOPE = "FROM Employees WHERE E_salary > 180K"; -->
2 SELECT ttid FROM Employees WHERE currencyFromUniversal(
    currencyToUniversal(E_salary,ttid),C) > 180K;
```

Listing 7.12: Rewriting a complex SCOPE expression
7.3.3 Algorithm Correctness

Proof. We prove the correctness of the canonical rewrite algorithm with respect to Subsection 7.2.4 by induction over the composable structure of SQL queries and by showing that the desired invariant, the result of each sub-query is filtered according to $D'$ and presented in the format required by $C$, holds: First, as a base, we state that adding the $D$-filters in the WHERE clause and transforming the SELECT clause to client format for every base table in each lowest-level sub-query ensures that the invariant holds. Next, as an induction step, we state that the way how we rewrite the FROM clause, as it was described earlier, preserves that property. The top-most SQL query is itself a composition of sub-queries (and base tables) for which the invariant holds. This means that the invariant holds for the entire query, which is hence guaranteed to deliver the correct result.

7.3.4 Rewriting DDL statements

Rewriting CREATE TABLE statements involves two steps: first, if the table is tenant-specific, an additional attribute ttid has to be added and the primary key needs be extended with that ttid. Second, all the MT-specific keywords have to be stripped off before the statement can be sent to the underlying DBMS. As a side effect, the MT-specific meta information, like the comparability of attributes, is stored in the meta tables.

Rewriting global constraints is straight-forward: for global constraints, the ttids have to be made part of the constraint. For instance, the foreign key constraint of Listing 7.3 just becomes:

```sql
CONSTRAINT fk_emp FOREIGN KEY (E_role_id, ttid)
    REFERENCES Roles (R_role_id, ttid)
```

Listing 7.13: Rewriting a global foreign-key constraint

Tenant-specific check constraints are rewritten just like queries, so they automatically include the ttids where needed. The tricky question is how to implement tenant-specific referential integrity constraints. The way MTBase implements this, is to rewrite these constraints as check constraints. Imagine, as an example, that there is no global foreign
key constraint on the Employees table and only tenant 0 adds this constraint privately. The way to rewrite this to a SQL check constraint is to make sure that the set of distinct keys in Employees_0 is a subset of the distinct keys in Roles_0:

```sql
CONSTRAINT fk_emp_0 CHECK (SELECT COUNT(E_role_id) FROM Employees WHERE ttid=0 AND E_role_id NOT IN (SELECT R_role_id FROM Roles WHERE ttid=0)) = 0
```

Listing 7.14: Rewriting a tenant-specific foreign-key constraint

MTBase executes CREATE VIEW statements by rewriting their WHERE clause the same way it rewrites queries, including the proper scope, \( D \). No other modifications are needed.

### 7.3.5 Rewriting DML statements

INSERT statements that consist of a sub-query have to be executed in two steps: First, the sub-query is rewritten and executed on the DBMS on behalf of \( C \). Second, for every \( d \in D \), the result (which does not include any \( ttids \)) is extended with \( ttid = d \) before being executed on the DBMS as a simple INSERT statement that contains a simple list of VALUES.

For instance, consider tenant 0 inserting data on behalf of tenant 1 (\( C = 0, D = \{1\} \)) with the following statement:

```sql
INSERT INTO Employees VALUES E_name, E_reg_id, E_salary, E_age (
SELECT E_name, E_reg_id, E_salary, E_age
FROM Employees WHERE E_age > 40
);
```

Listing 7.15: Rewriting an INSERT statement

First of all, the intension of tenant 0 here is to copy some of his recordsover to tenant 1.\(^6\) The result of the SELECT sub-query, executed on behalf of tenant 0 on the exemplary database of Figure 7.3, is (‘Alice’, 3, 150K, 46). This record is then converted

\(^6\)However, before copying data from one tenant to another, one should also consider the possibility to make this data global.
into the format of tenant 1 and extended with its \texttt{ttid}: (1, 'Alice', 3, 135K, 46), before being inserted into the \texttt{Employees} table. This examples shows some of the difficulties of executing an \texttt{INSERT} (or \texttt{UPDATE}) statement on behalf of somebody else. First, as \texttt{E_emp_id} and \texttt{E_role_id} are \texttt{NOT NULL}, they must either have a default value or the statement fails. Second, for tenant 0 to provide a useful \texttt{E_role_id} for tenant 1 is difficult because it is a \textit{tenant-specific} attribute. MTBase does not prevent a tenant from inserting \textit{tenant-specific} attributes, even on behalf of other tenants, but it throws a warning in order to notify that the value might not make sense. Luckily, these problems do not occur with \texttt{DELETE} statements.

### 7.4 Optimizations

As we have seen, there is a canonical rewrite algorithm that correctly rewrites MTSQL to SQL. However, as we will see in our experimental evaluation,\textsuperscript{7} the rewritten queries often execute very slowly on the underlying DBMS. The main reason for this is that the pure rewritten queries call two \textit{conversion functions} on every \textit{transformable attribute} of every record that is processed, which is extremely expensive. Luckily, the execution costs can be reduced dramatically when applying the \textit{optimization passes} that we describe in this section. As we assume the underlying DBMS to optimize query execution anyway, we focus on optimizations that a DBMS query optimizer cannot do (because it needs MT-specific context) or does not do (because an optimization does not matter often enough outside the context of MTBase such that the optimizer would care). We differentiate between \textit{semantic optimizations}, which are always applied because they never make a query slower and \textit{cost-based optimizations} which are only applied if the predicted costs are smaller than in the original query.

#### 7.4.1 Trivial Semantic Optimizations

There are a couple of special cases for \textit{C} and \textit{D} that allow saving \textit{conversion function} calls, join predicates and/or \textit{D-filters}. First, if \textit{D} includes all tenants, that means that we want to query all data and hence D-filters are no longer required as shown in line 3 of Listing 7.16. Second, as shown in line 6, if $|D| = 1$, we know that all data is from

\textsuperscript{7}Consider for instance the execution time of Q1 in Table 7.6 without optimizations (\textit{canonical}).
the same tenant, which means that including \texttt{ttid} in the join predicate is no longer necessary. Last, if we know that a client queries her own data, \emph{i.e.}, \(D = \{C\}\), we know that even \textit{convertible attributes} are already in the correct format and can hence remove the conversion function calls (line 9).

\begin{verbatim}
1 -- dropping D-filter if D is the default scope:
2 SELECT E_age FROM Employees WHERE E_ttid IN (1,2); -->
3 SELECT E_age FROM Employees;
4 -- dropping ttid from join predicate if |D| = 1:
5 SELECT E_age, R_name FROM Employees, Roles WHERE E_role_id = R_role_id
   AND E_ttid = R_ttid AND E_ttid IN (2) AND R_ttid IN (2); -->
6 SELECT E_age, R_name FROM Employees, Roles WHERE E_role_id = R_role_id
   AND E_ttid IN (2) AND R_ttid IN (2);
7 -- dropping conversion functions if D = \{C\}:
8 SELECT currencyFromUniversal(currencyToUniversal(E_salary, E_ttid),0) AS E_salary FROM Employees; -->
9 SELECT E_salary FROM Employees;
\end{verbatim}

Listing 7.16: Examples for trivial Semantic Optimizations

### 7.4.2 Other Semantic Optimizations

There are a couple of other semantic optimizations that can be applied to rewritten queries. While \textit{client presentation push-up} and conversion push-up try to minimize the number of conversions by delaying conversion to the latest possible moment, \textit{aggregation distribution} takes into account specific properties of conversion functions (cf. Subsections 7.2.2.2 and 7.2.2.3). If conversion functions are UDFs written in SQL it is also possible to \textit{inline} them, which typically gives queries an additional speed-up.

#### 7.4.2.1 Client Presentation and Conversion Push-Up

As conversion function pairs are \textit{equality-preserving}, it is possible in some cases to defer conversions to later, \emph{e.g.}, to the outermost query in the case of nested queries. While \textit{client presentation push-up} converts everything to universal format and defers conversion to client format to the outermost \texttt{SELECT} clause, \textit{conversion push-up} pushes this idea even more by also delaying the conversion to universal format as much as
possible. Both optimizations are beneficial if delaying the conversions allows the query execution engine to evaluate other (less expensive) predicates first. This means that, once the data has to be converted, it is already more filtered and therefore the overall number of (expensive) conversion function calls becomes smaller.\textsuperscript{8} Naturally, if we delay conversion, this also means that we have to propagate the necessary \textit{ttids} to the outer-more queries and keep track of whether data is in \textit{original}, \textit{universal} or \textit{client} format.

```sql
1 -- before optimization
2 SELECT Dom.name1, Dom.sal1 as sal, COUNT(*) as cnt FROM ( 
3  SELECT E1.name as name1, currencyFromUniversal(currencyToUniversal(E1.E_salary, E1.E_ttid), C) as sal1
4  FROM Employees E1, Employees E2 
5  WHERE currencyFromUniversal(currencyToUniversal(E1.E_salary, E1.E_ttid)
6    , C) > currencyFromUniversal(currencyToUniversal(E2.E_salary, E2.
7    E_ttid), C) 
8 ) as Dom GROUP BY Dom.name1, sal, cnt ORDER BY cnt;
9 -- after optimization
10 SELECT Dom.name1, currencyFromUniversal(Dom.sal1, C) as sal, COUNT(*) as cnt FROM ( 
11 SELECT E1.name as name1, currencyToUniversal(E1.E_salary, E1.E_ttid) as sal1
12 FROM Employees E1, Employees E2 
13 WHERE currencyToUniversal(E1.E_salary, E1.E_ttid) > currencyToUniversal
14    (E2.E_salary, E2.E_ttid) 
15 ) as Dom GROUP BY Dom.name1, sal, cnt ORDER BY cnt;
```

Listing 7.17: Example for Client Presentation Push-Up

Listing 7.17 shows a query that ranks employees according to the fact how many salaries of other employees their own salary dominates. With \textit{client presentation push-up}, salaries are compared in \textit{universal} instead of \textit{client} format, which is correct because of the \textit{order-preserving} property of \textit{currencyFromUniversal} (cf. Subsection 7.2.2.3) and saves half of the function calls in the sub-query.

\textit{Conversion push-up}, as illustrated in Listing 7.18, reduces the number of function calls dramatically: First, as salaries are only converted in the end, salaries of employees aged

\textsuperscript{8}In the worst case, the number of conversion function calls stays the same, but it can definitely note become bigger.
less than 45 do not have to be considered at all. Second, the WHERE clause converts the constant \((100K)\) instead of the attribute \((E_{\text{salary}})\). As the outcome of conversion functions is deterministic (cf. Subsection 7.2.2.2) and \(C\) is also constant, the conversion functions have to be called only once per tenant and their results can be cached by the DBMS for the rest of the query execution, which make query execution much faster as we will see in Section 7.6.\(^9\)

```
1    -- before optimization
2    SELECT AVG(X.sal) FROM {
3        SELECT currencyFromUniversal(currencyToUniversal(E_salary, E_ttid), C)
4            as sal
5        FROM Employees WHERE E_age >= 45 AND currencyFromUniversal(
6            currencyToUniversal(E_salary, E_ttid), C) > 100K) as X;
7    -- after optimization
8    SELECT AVG(currencyFromUniversal(currencyToUniversal(X.sal, X.sal_ttid),C
9      )) FROM {
10       SELECT E_salary as sal, E_ttid as sal_ttid
11       FROM Employees WHERE E_age >= 45 AND E_salary > currencyFromUniversal(
12          currencyToUniversal(100K, E_ttid), C
13      ) as X);
```

Listing 7.18: Example for Conversion Push-Up

### 7.4.2.2 Aggregation Distribution

Many analytical queries contain aggregation functions, some of which aggregate over convertible attributes. The idea of aggregation distribution is to aggregate in two steps: First, aggregate per tenant in that specific tenant’s format (which requires no conversion) and second, convert intermediary results to universal (one conversion per tenant), aggregate those and convert the final result to client format (one additional conversion).

This simple idea reduces the number of conversion function calls for \(N\) records and \(T\) different data owners of these records from \((2N)\) to \((T + 1)\). This is significant because \(T\) is typically much smaller than \(N\) (and cannot be greater).

Compared to pure conversion push-up, which works for any conversion function pair, the applicability of aggregation distribution depends on further algebraic properties of these

\(^9\)For instance, looking at the execution times of Q22 in Tables 7.5 and 7.6, we see how they decrease from \(o1\) (only trivial optimizations) to \(o2\) (including conversion push-up).
functions. Gray et al. [GCB+97] categorize numerical aggregation functions into three categories with regard to their ability to distribute: distributive functions, like COUNT, SUM, MIN and MAX distribute with functions $F$ (for partial) and $G$ (for total aggregation). For COUNT for instance, $F$ is COUNT and $G$ is SUM as the total count is the sum of all partial counts. There are also algebraic aggregation functions, e.g., AVG, where the partial results are not scalar values, but tuples. In the case of AVG, this would be the pairs of a partial sums and partial counts because the total average can be computed from the sum of all sums, divided by the sum of all counts. Finally, holistic aggregation functions cannot be distributed at all.

We would like to extend this notion of Gray et al. [GCB+97] and define the distributability of an aggregation function $a$ with respect to a conversion function pair $(from, to)$. Table 7.2 shows some examples for different aggregation and conversion functions. First of all, we want to state that, as all conversion functions have scalar values as input and output, they are always fully-COUNT-preserving, which means that COUNT can be distributed over all sorts of conversion functions. Next, we observe that all order-preserving functions preserve the minimum and the maximum of a given set of numbers, which is why MIN and MAX distribute over the first three categories of conversion functions displayed in Table 7.2.

<table>
<thead>
<tr>
<th></th>
<th>$to(x) = c \cdot x$</th>
<th>$to(x) = a \cdot x + b$</th>
<th>$to(x)$ is order-preserving</th>
<th>$to(x)$ is equality-preserving</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MIN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>MAX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>SUM</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>AVG</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Holistic</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7.2: Distributability of different Aggregation Functions over different categories of Conversion Functions

We further notice that if $to$ (and consequently also $from$) is a multiplication with a constant (first column of Table 7.2), $to$ is fully-MIN-, fully-MAX- and fully-SUM-preserving, which is why these aggregation functions distribute. As SUM and COUNT distribute, AVG, an algebraic function, distributes as well. Finally looking at the second column
of Table 7.2 where \( f_{\text{from}} \) is a linear function, it is not easy to see why such a function is still \( \text{SUM} \)-preserving and \( \text{AVG} \)-preserving, which is why we are going to prove this mathematically.

**Proof.** More formally, for the set of tenants \( T \), their bags of values \( X_1, \ldots, X_T \) and corresponding conversion functions \( f_1(x) = a_1x + b_1, \ldots, f_T(x) = a_Tx + b_T \), we can compute the total average as the weighted average of the partial averages. The following series of equations starts with the total average on converted values and ends up with weighted average of the converted partial averages:

\[
\frac{\sum_{t \in T} (\sum_{x \in X_t} f_t(x))}{\sum_{t \in T} |X_t|} = \frac{\sum_{t \in T} (\sum_{x \in X_t} a_t x + b_t)}{\sum_{t \in T} |X_t|} = \frac{\sum_{t \in T} (a_t (\sum_{x \in X_t} x) + b_t \cdot |X_t|)}{\sum_{t \in T} |X_t|} = \frac{\sum_{t \in T} (\frac{|X_t|}{|X_t|} (a_t \sum_{x \in X_t} x) + b_t \cdot |X_t|)}{\sum_{t \in T} |X_t|} = \frac{\sum_{t \in T} (|X_t| \cdot f_t(\sum_{x \in X_t} x))}{\sum_{t \in T} |X_t|}
\]

If we look more carefully at these equations, we realize that the term below the fraction bar is always the same, which means that the set of equations, if we remove this term, also shows how the total sum can be computed by multiplying the partial averages with the corresponding partial sums. 

---

```sql
-- before optimization
2 SELECT SUM(currencyFromUniversal(currencyToUniversal(E_salary, E_ttid), C)) as sum_sal FROM Employees

-- after optimization
3 SELECT currencyFromUniversal(SUM(t.E_partial_salary), C) as sum_sal FROM
   (SELECT currencyToUniversal(SUM(E_salary), E_ttid) as E_partial_salary
    FROM Employees GROUP BY E_ttid) as t;
```

Listing 7.19: Example for Conversion Function Distribution
We conclude this paragraph by observing that the conversion function pair for phone format (cf. Listings 7.4 and 7.5) is not even order-preserving and does therefore not distribute while the pair for currency format (cf. Listings 7.6 and 7.7) distributes over all standard SQL aggregation functions. An example of how this can be used is shown in Listing 7.19.

### 7.4.2.3 Function Inlining

As explained in Subsection 7.2.2.2, there are several ways how to define conversion functions. However, if they are defined as a SQL statement, potentially including lookups into meta tables), they can be directly inlined into the rewritten query in order to save calls to UDFs. Function inlining typically also enables the query optimizer of the underlying DBMS to optimize much more aggressively. In WHERE clauses, conversion functions could simply be inlined as sub-queries, which, however often results in sub-optimal performance as calling a sub-query on each conversion is not much cheaper than calling the corresponding UDF. For SELECT clauses, the SQL standard prohibits inlining them as sub-queries as this can result in attributes not being contained neither in an aggregate function nor in the GROUP BY clause.¹⁰ This is why the proper way to inline functions is by using a join as demonstrated in Listing 7.20. Our results in Section 7.6 suggest that function inlining, though producing complex-looking SQL queries, results in very good query execution performance.

```sql
-- before optimization
SELECT currencyFromUniversal(currencyToUniversal(E_salary, E_ttid), C) as E_salary FROM Employees

-- after optimization
SELECT (C1.CT_from_universal * C2.CT_to_universal * E_salary) as E_salary
FROM Employees, Tenant T1, Tenant T2, CurrencyTransform1, CurrencyTransform2
WHERE T1.T_tenant_key = C AND T1.T_currency_key = CurrencyTransform1.CT_currency_key
    AND T2.T_tenant_key = E_ttid AND T2.T_currency_key = CurrencyTransform2.CT_currency_key

Listing 7.20: Example for Function Inlining
```

¹⁰Interestingly, most commercial DBMS reject such queries while PostgreSQL executes them anyway.
7.4.3 Cost-based optimizations

Subsection 7.4.2.1 proposed to rewrite predicates in such a way that comparisons between two attributes are always done in universal format. However, as conversion functions have to be equality-preserving (cf. Definition 1), equality predicates could be executed in any (appropriate) format. Likewise, if a conversion function is known to be order-preserving, also inequality predicates could be executed in any format. The question which format to use for comparison predicates in order to execute a query with minimal cost depends on the following cost factors: First, there are the costs of tenants’ partial conversion functions, which might not be uniform. Second, we have to know which portion of the data is in which format. For instance if 90% of the data is in a specific tenant format, that format is a good candidate format for predicate evaluation. While knowing the format usage frequency requires access to the database statistics, more specifically histograms, good cost estimations for UDFs are a research topic on its own [LCBK04, HPS+12].

Our current implementation of MTBase does not do cost-based optimizations. First, because we consider it still an open question whether such optimizations should be implemented in the MTBase optimizer or rather in the query optimizer of the underlying DMBS (by teaching it the notion of conversion function pairs). Also, as query optimizers become better and better in optimizing UDFs by analyzing their algebraic properties [HPS+12], doing cost-based optimizations in MTBase might become obsolete.

One idea that could be used if we assume a uniform cost model for conversion functions, is to minimize the total number of function calls among the entire query, thereby following the ideas for expensive predicate evaluation proposed by Hellerstein and Stonebraker [HS93]: If the total amount of tuples to be processed is \( N \), it needs \( N \) conversion function calls to bring all these tuples into universal format. If there exists a tenant-specific format \( F \) (different from the universal format) that is used in more than 50% of the tuples to be compared, using that format becomes cheaper: tuples that are in format \( F \) do not have to converted at all and the other \( M \) tuples need two conversions (first to universal and then to \( F \)). Hence the total number of conversions is smaller than \( N \), namely: \( 2 \cdot M < 2 \cdot 0.5 \cdot N = N \).
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

7.5 MT-H Benchmark

As MTSQL is a novel language, there exists no benchmark to evaluate the performance on an engine that implements it, e.g., MTBase. So far, there exists no standard benchmark for cross-tenant query processing, only for data integration [Tra16b] which does not assume the data to be in shared tables. Transactions in MTBase are not much different from standard transactions. Analytical queries, however, typically involve a lot of conversions and therefore thousands of (potentially expensive) calls to UDFs. Thus, the ability to study the usefulness of different optimizations passes on different analytical queries was a primary design goal, which is why we decided to extend the well-known TPC-H database benchmark [Tra16c]. Our new benchmark, which we call MT-H, extends TPC-H in the following way:

- Each tenant represents a different company. The number of tenants \( T \) is a parameter of the benchmark. \( t t i d s \) range from 1 to \( T \).

- We consider Nation, Region, Supplier, Part, and Partsupp common, publicly available knowledge. They are therefore global tables and need no modification.

- We consider Customer, Orders and Lineitem tenant-specific. While the latter two are quite obviously tenant-specific (each company processes their own orders and line items), customers might actually do business with several companies. However, as customer information might be sensitive and the format of this information might differ from tenant to tenant, it makes sense to have specific customers per tenant.

- All primary keys and foreign keys relating to tenant-specific tables (\( C\_custkey \), \( O\_orderkey \), \( O\_custkey \), \( L\_orderkey \)) are tenant-specific. If not mentioned otherwise, the attributes in Customer, Orders and Lineitem are comparable.

- We consider two domains for convertible attributes and corresponding conversion functions: currency and phone format. currency refers to monetary values, i.e., \( C\_acctbal \), \( O\_totalprice \) and \( L\_extendedprice \), and uses the conversion functions from Listings 7.6 and 7.7. phone format is used in \( C\_phone \) with the conversion function pair of Listings 7.4 and 7.5. We modified the data generator...
of TPC-H (dbgen) to take the specific currency and phone formats into account. Each tenant is assigned a random currency and phone format, except for tenant 1 who gets the universal format for both.

- The TPC-H scaling factor $s_f$ also applies to our benchmark and dictates the overall size of the tables. After creating all records with dbgen, each record in Customer, Orders and Lineitem is assigned to a tenant in a way that foreign-key constraints are preserved, i.e., orders of a specific tenant link to a customer of that same tenant. There are two ways how this assignment happens, either uniform (each tenant gets the same amount of records) or zipfian (tenant 1 gets the biggest share and tenant $T$ the smallest). This tenant share distribution $\rho$ is another parameter of the benchmark.

- We use the same 22 queries and query parameters as TPC-H. Additionally, for each query run, we have to define the client $C$ who runs the queries as well as the dataset/scope $D$ she wants to query.

- For query validation, we simply set $C = 1$ and $D = \{1, 2, \ldots, T\}$. That way, we make sure to process all data and that the result is presented in universal format and can therefore be compared to expected query results of the standard TPC-H. An exception are queries that contain joins on $O_{\text{custkey}} = C_{\text{custkey}}$. In MT-H, we make sure that each order links to a customer from the same tenant, thus the mapping between orders and customers is no longer the same as in TPC-H (where an order can potentially link to any customer). For such queries, we define the result from the canonical rewrite algorithm (without optimizations) to be the gold standard to validate against.

## 7.6 Experiments and Results

This section presents the evaluation of MTBase using the MT-H benchmark. We first evaluated the benefits of different optimization steps from Section 7.4 and found that the combination of all of these steps brings the biggest benefit. Second, we analyzed how MTBase scales with an increasing number of tenants. With all optimizations applied and for a dataset of 100 GB on a single machine, MTBase scales up to thousands of
tenants with very little overhead. We also validated result correctness as explained in Section 7.5 and can report only positive results.

### 7.6.1 Setup

In our experiments, we used the following two setups: The first setup is a PostgreSQL 9.6 Beta installation, running on Debian Linux 4.1.12 with gcc 4.9.2-10 on a 4x16 Core AMD Opteron 6174 processor with 256 GB of main memory. The second installation runs a commercial database, which we will call System C, on a commercial operating system and on the same processor with 512 GB of main memory. Although both machines have enough secondary storage capacity available, we decided to configure both database management systems to use in-memory backed files in order to achieve the best performance possible and compare the different approaches “at their best”. Moreover, we configured the systems to use all available threads, which enabled *intra-query parallelism*.

### 7.6.2 Workload and Methodology

As MT-H has a lot of parameters and in order to make things more concrete, we worked with the following two scenarios which were already sketched in the introduction:

**Scenario 1** handles the data of a business alliance of a couple of small to mid-sized enterprises, which means there are 10 tenants with $sf = 1$ and each of them owns more or less the same amount of data ($\rho = \text{uniform}$).

**Scenario 2** represents a big database ($sf = 100$) of medical records coming from thousands of tenants, like hospitals and private practices. Some of these institutions have vast amounts of data while others only handle a couple of patients ($\rho = \text{zipfian}$). A research institution wants to query the entire database ($D=\{1,2,...,T\}$) in order to gather new insights for the development of a new treatment. We will look at this scenario for different numbers of $T$.

In order to evaluate the overhead of *cross-tenant query processing* in MTBase compared to single-tenant query processing, we also measured the standard TPC-H queries with
different scaling factors. When \( D \) was set to all tenants, we compared to TPC-H with the same scaling factor as MT-H. For the cases where \( D \) had only one tenant out of ten, we compared with TPC-H with a scaling factor ten times smaller.

Every \textit{query run} was repeated three times in order to ensure stable results. We noticed that three runs are needed for the response times to converge (within 2\%). Thus we always report the last measured response time for each query with two significant digits.

All experiments were executed with both setups (PostgreSQL and \textit{System C}). Whereas the major findings were the same on both systems, PostgreSQL optimizes conversion functions (UDFs) much better by caching their results. \textit{System C}, on the other hand does not allow UDFs to be defined as deterministic and hence cannot cache conversion results. This eliminates the effect of \textit{conversion push-up} when applied to comparison predicates where we convert the constant instead of the attribute (cf. Listing 7.18). This being said, the next few subsections only report results on PostgreSQL while Subsection 7.6.5 confirms that the main conclusions drawn from the PostgreSQL experiments generalize.

### 7.6.3 Benefit of Optimizations

In order to test the benefit of the different combinations of optimizations applied, we tested \textit{Scenario 1} with different optimization levels enumerated in Table 7.3. From \( o1 \) to \( o4 \) we added optimizations incrementally, while the last optimization level (\textit{inl-only}) only applied \textit{trivial optimizations} and \textit{function inlining} in order to test whether the other optimizations are useful at all.

<table>
<thead>
<tr>
<th>opt level</th>
<th>optimization passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>canonical</td>
<td>none</td>
</tr>
<tr>
<td>o1</td>
<td>trivial optimizations</td>
</tr>
<tr>
<td>o2</td>
<td>( o1 + ) client presentation push-up + conversion push-up</td>
</tr>
<tr>
<td>o3</td>
<td>( o2 + ) conversion function distribution</td>
</tr>
<tr>
<td>o4</td>
<td>( o3 + ) conversion function inlining</td>
</tr>
<tr>
<td>inl-only</td>
<td>( o1 + ) conversion function inlining</td>
</tr>
</tbody>
</table>

Table 7.3: Optimization Levels used for Evaluation of MTSQL

Table 7.4 shows the MT-H queries for different optimization levels and \textit{scenario 1} \((sf = 1, T = 10)\) where client 1 queries her own data. As we can see, in that case,
applying trivial optimizations in \( o1 \) is enough because these already eliminate all conversion functions and joins and only the D-filters remain. Executing these filters seems to be very inexpensive because most response times of the optimized queries are close to the baseline, TPC-H with \( sf = 0.1 \). Queries 2, 11 and 16 however, take roughly ten times longer than the baseline. This is not surprising when taken into account that these queries only operate on \textit{shared tables} which have ten times more data than in TPC-H. The same effect can be observed in Q09 where a significant part of the joined tables are shared.

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-0.1G</td>
<td>2.6</td>
<td>0.1</td>
<td>0.27</td>
<td>0.35</td>
<td>0.15</td>
<td>0.29</td>
<td>0.18</td>
<td>0.14</td>
<td>0.59</td>
<td>0.36</td>
<td>0.081</td>
<td>0.37</td>
<td>0.26</td>
<td>0.27</td>
<td>0.77</td>
<td>0.12</td>
<td>0.081</td>
<td>0.89</td>
<td>0.12</td>
<td>0.13</td>
<td>0.57</td>
<td>0.081</td>
</tr>
<tr>
<td>canonical</td>
<td>84</td>
<td>1.0</td>
<td>0.55</td>
<td>0.65</td>
<td>0.32</td>
<td>1.0</td>
<td>0.29</td>
<td>0.36</td>
<td>4.9</td>
<td>0.91</td>
<td>0.37</td>
<td>0.55</td>
<td>0.63</td>
<td>0.98</td>
<td>3.1</td>
<td>1.2</td>
<td>0.49</td>
<td>1.7</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>2.0</td>
</tr>
<tr>
<td>o1</td>
<td>2.7</td>
<td>1.0</td>
<td>0.43</td>
<td>0.61</td>
<td>0.22</td>
<td>0.43</td>
<td>0.23</td>
<td>0.56</td>
<td>3.8</td>
<td>0.76</td>
<td>0.37</td>
<td>0.55</td>
<td>0.92</td>
<td>0.56</td>
<td>0.91</td>
<td>1.2</td>
<td>0.48</td>
<td>1.6</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>0.085</td>
</tr>
<tr>
<td>o2</td>
<td>2.7</td>
<td>1.0</td>
<td>0.42</td>
<td>0.61</td>
<td>0.22</td>
<td>0.43</td>
<td>0.22</td>
<td>0.57</td>
<td>3.9</td>
<td>0.76</td>
<td>0.38</td>
<td>0.55</td>
<td>0.89</td>
<td>0.56</td>
<td>0.96</td>
<td>1.2</td>
<td>0.5</td>
<td>1.7</td>
<td>0.3</td>
<td>2.8</td>
<td>0.67</td>
<td>0.085</td>
</tr>
<tr>
<td>o3</td>
<td>2.7</td>
<td>1.0</td>
<td>0.43</td>
<td>0.61</td>
<td>0.22</td>
<td>0.43</td>
<td>0.23</td>
<td>0.56</td>
<td>3.9</td>
<td>0.76</td>
<td>0.37</td>
<td>0.55</td>
<td>0.92</td>
<td>0.56</td>
<td>0.91</td>
<td>1.2</td>
<td>0.48</td>
<td>1.6</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>0.085</td>
</tr>
<tr>
<td>o4</td>
<td>2.7</td>
<td>1.0</td>
<td>0.43</td>
<td>0.62</td>
<td>0.22</td>
<td>0.43</td>
<td>0.23</td>
<td>0.61</td>
<td>4.1</td>
<td>0.78</td>
<td>0.39</td>
<td>0.56</td>
<td>0.9</td>
<td>0.57</td>
<td>1.0</td>
<td>1.2</td>
<td>0.51</td>
<td>1.7</td>
<td>0.31</td>
<td>3.1</td>
<td>0.67</td>
<td>0.085</td>
</tr>
<tr>
<td>inl-only</td>
<td>2.7</td>
<td>1.0</td>
<td>0.42</td>
<td>0.65</td>
<td>0.22</td>
<td>0.43</td>
<td>0.22</td>
<td>0.57</td>
<td>3.8</td>
<td>0.76</td>
<td>0.37</td>
<td>0.55</td>
<td>0.92</td>
<td>0.56</td>
<td>0.92</td>
<td>1.2</td>
<td>0.48</td>
<td>1.6</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>0.085</td>
</tr>
</tbody>
</table>

Table 7.4: Response Times [sec] of 22 MT-H Queries for MTBase-on-PostgreSQL with \( sf = 1 \), \( T = 10 \), \( \rho = \text{uniform} \), \( C = 1 \), \( D = \{1\} \), for different Optimization Levels, versus TPC-H with \( sf = 0.1 \).

Table 7.5 shows similar results, but for \( D = 2 \), which means that now conversion functions can no longer be optimized away. While most of the queries show a similar behavior than in the previous experiment, for the ones that involve a lot of conversion functions, \textit{i.e.}, Q1, Q6 and Q22, we see how the performance becomes better with each \textit{optimization pass} added. We also notice that while \textit{function inlining} is very beneficial in general, it is even more so when combined with the other optimizations.

Finally, Table 7.6 shows the results where we query all data, \textit{i.e.}, \( D = \{1, 2, \ldots, 10\} \). This experiment involves even more conversion functions from all the different \textit{tenant formats} into \textit{universal}. In particular, when looking again at queries 1, 6 and 22, we observe the great benefit of \textit{conversion function distribution} (added with \( o3 \)), which, in turn, only works as great in conjunction with \textit{client and conversion function push-up} because \textit{aggregation} typically happens in the outermost query while \textit{conversion} happens in the sub-queries. Overall, \( o4 \), which contains all optimization passes that MTBase offers, is the clear winner.
7.6. Experiments and Results

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-0.1G</td>
<td>2.6</td>
<td>0.11</td>
<td>0.27</td>
<td>0.35</td>
<td>0.15</td>
<td>0.29</td>
<td>0.14</td>
<td>0.59</td>
<td>0.36</td>
<td>0.081</td>
<td>0.37</td>
<td>0.26</td>
<td>0.27</td>
<td>0.77</td>
<td>0.12</td>
<td>0.081</td>
<td>0.89</td>
<td>0.12</td>
<td>0.13</td>
<td>0.57</td>
<td>0.081</td>
<td></td>
</tr>
<tr>
<td>canonical</td>
<td>87</td>
<td>1</td>
<td>0.5</td>
<td>0.6</td>
<td>0.28</td>
<td>1.0</td>
<td>0.26</td>
<td>0.37</td>
<td>4.9</td>
<td>0.89</td>
<td>0.37</td>
<td>0.56</td>
<td>0.65</td>
<td>1.0</td>
<td>3.2</td>
<td>1.2</td>
<td>0.49</td>
<td>1.6</td>
<td>0.31</td>
<td>2.8</td>
<td>0.66</td>
<td>2.0</td>
</tr>
<tr>
<td>o1</td>
<td>87</td>
<td>1</td>
<td>0.5</td>
<td>0.69</td>
<td>0.33</td>
<td>1.0</td>
<td>0.27</td>
<td>0.38</td>
<td>5.2</td>
<td>0.9</td>
<td>0.39</td>
<td>0.56</td>
<td>0.92</td>
<td>1.0</td>
<td>3.1</td>
<td>1.2</td>
<td>0.51</td>
<td>1.6</td>
<td>0.32</td>
<td>3.1</td>
<td>0.68</td>
<td>2.0</td>
</tr>
<tr>
<td>o2</td>
<td>87</td>
<td>1</td>
<td>0.5</td>
<td>0.61</td>
<td>0.28</td>
<td>1.0</td>
<td>0.27</td>
<td>0.38</td>
<td>5.2</td>
<td>0.9</td>
<td>0.39</td>
<td>0.57</td>
<td>0.91</td>
<td>1.0</td>
<td>3.1</td>
<td>1.2</td>
<td>0.51</td>
<td>1.6</td>
<td>0.32</td>
<td>3.1</td>
<td>0.67</td>
<td>1.3</td>
</tr>
<tr>
<td>o3</td>
<td>32</td>
<td>1</td>
<td>0.45</td>
<td>0.63</td>
<td>0.28</td>
<td>0.44</td>
<td>0.24</td>
<td>0.37</td>
<td>4.3</td>
<td>0.83</td>
<td>0.38</td>
<td>0.56</td>
<td>0.91</td>
<td>1.1</td>
<td>1.9</td>
<td>1.3</td>
<td>0.51</td>
<td>1.6</td>
<td>0.32</td>
<td>3.1</td>
<td>0.67</td>
<td>1.3</td>
</tr>
<tr>
<td>o4</td>
<td>14</td>
<td>1</td>
<td>0.49</td>
<td>0.62</td>
<td>0.22</td>
<td>0.44</td>
<td>0.22</td>
<td>0.57</td>
<td>3.9</td>
<td>0.93</td>
<td>0.38</td>
<td>0.56</td>
<td>0.89</td>
<td>0.73</td>
<td>1.3</td>
<td>1.2</td>
<td>0.49</td>
<td>1.6</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>0.27</td>
</tr>
<tr>
<td>ini-only</td>
<td>45</td>
<td>1</td>
<td>0.47</td>
<td>0.61</td>
<td>0.27</td>
<td>0.64</td>
<td>0.24</td>
<td>0.58</td>
<td>4.2</td>
<td>0.94</td>
<td>0.37</td>
<td>0.55</td>
<td>0.91</td>
<td>0.73</td>
<td>2.2</td>
<td>1.2</td>
<td>0.48</td>
<td>1.7</td>
<td>0.3</td>
<td>2.8</td>
<td>0.66</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Table 7.5: Response Times [sec] of 22 MT-H Queries for MTBase-on-PostgreSQL with $sf = 1$, $T = 10$, $\rho =$ uniform, $C = 1$, $D = \{2\}$, for different Optimization Levels, versus TPC-H with $sf = 0.1$

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-1G</td>
<td>26</td>
<td>1</td>
<td>2.4</td>
<td>4.5</td>
<td>1.4</td>
<td>1.5</td>
<td>2.9</td>
<td>3.7</td>
<td>1.3</td>
<td>9.5</td>
<td>2.2</td>
<td>0.38</td>
<td>3.9</td>
<td>8.4</td>
<td>2.7</td>
<td>5.9</td>
<td>1.2</td>
<td>0.54</td>
<td>10</td>
<td>0.3</td>
<td>2.4</td>
<td>4.8</td>
</tr>
<tr>
<td>canonical</td>
<td>870</td>
<td>1</td>
<td>6.5</td>
<td>1.5</td>
<td>3.4</td>
<td>8.7</td>
<td>3.7</td>
<td>1.7</td>
<td>19</td>
<td>11</td>
<td>0.36</td>
<td>4.1</td>
<td>4.9</td>
<td>7.3</td>
<td>28</td>
<td>1.2</td>
<td>0.57</td>
<td>12</td>
<td>0.32</td>
<td>2.6</td>
<td>5.8</td>
<td>20</td>
</tr>
<tr>
<td>o1</td>
<td>860</td>
<td>1</td>
<td>6.5</td>
<td>1.5</td>
<td>3.4</td>
<td>8.7</td>
<td>3.7</td>
<td>1.7</td>
<td>19</td>
<td>11</td>
<td>0.36</td>
<td>4.1</td>
<td>4.9</td>
<td>7.3</td>
<td>28</td>
<td>1.2</td>
<td>0.62</td>
<td>12</td>
<td>0.33</td>
<td>2.7</td>
<td>5.9</td>
<td>20</td>
</tr>
<tr>
<td>o2</td>
<td>870</td>
<td>1</td>
<td>6.5</td>
<td>1.5</td>
<td>3.4</td>
<td>8.6</td>
<td>3.7</td>
<td>1.7</td>
<td>19</td>
<td>11</td>
<td>0.35</td>
<td>4.1</td>
<td>4.9</td>
<td>7.2</td>
<td>28</td>
<td>1.2</td>
<td>0.57</td>
<td>12</td>
<td>0.32</td>
<td>2.6</td>
<td>5.9</td>
<td>13</td>
</tr>
<tr>
<td>o3</td>
<td>310</td>
<td>1</td>
<td>5.5</td>
<td>1.5</td>
<td>3.1</td>
<td>3.1</td>
<td>3.4</td>
<td>1.6</td>
<td>11</td>
<td>10</td>
<td>0.36</td>
<td>4.1</td>
<td>4.9</td>
<td>7.3</td>
<td>12</td>
<td>1.2</td>
<td>0.55</td>
<td>12</td>
<td>0.32</td>
<td>2.6</td>
<td>5.9</td>
<td>13</td>
</tr>
<tr>
<td>o4</td>
<td>130</td>
<td>1</td>
<td>3.7</td>
<td>1.5</td>
<td>1.7</td>
<td>3.1</td>
<td>3.4</td>
<td>1.4</td>
<td>11</td>
<td>8.6</td>
<td>0.38</td>
<td>4.1</td>
<td>4.9</td>
<td>4.4</td>
<td>9.1</td>
<td>1.2</td>
<td>0.59</td>
<td>12</td>
<td>0.32</td>
<td>2.6</td>
<td>5.7</td>
<td>2.2</td>
</tr>
<tr>
<td>ini-only</td>
<td>450</td>
<td>1</td>
<td>4.1</td>
<td>1.6</td>
<td>1.8</td>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>14</td>
<td>4.9</td>
<td>0.39</td>
<td>4.1</td>
<td>4.8</td>
<td>4.4</td>
<td>19</td>
<td>1.2</td>
<td>0.55</td>
<td>12</td>
<td>0.32</td>
<td>2.6</td>
<td>5.8</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Table 7.6: Response Times [sec] of 22 MT-H Queries for MTBase-on-PostgreSQL with $sf = 1$, $T = 10$, $\rho =$ uniform, $C = 1$, $D = \{1, 2, \ldots, 10\}$, for different Optimization Levels, versus TPC-H with $sf = 1$

7.6.4 Cross-Tenant Query Processing at Large

In the next experiment, we evaluated the cost of cross-tenant query processing up to thousands of tenants. More concretely, we measured the response time of conversion-intensive MT-H queries, i.e., Q1, Q6 and Q22, for scenario 2, thereby increasing the number of tenants from 1 to 100,000 on a large dataset where $sf = 100$. We measured for the best optimization level, o4, as well as for inlining-only. The obtained results were then compared to plain TPC-H with $sf = 100$, as shown in Figure 7.6. First of all, we notice that the cost overhead compared to single-tenant query-processing (TPC-H) stays below a factor of 2 and in general increases very moderately with the number of tenants. An interesting artifact can be observed for Q22 where MT-H for one tenant executes faster than plain TPC-H. The reason for this is a sub-optimal optimization decision in PostgreSQL: one of the most expensive parts of Q22, namely to find customers with a specific country code, is executed with a parallel scan in MT-H while no parallelism is used in the case of TPC-H.

179
Figure 7.6: Response Times (relative to TPC-H, $s_f = 100$) of o4 and inlining-only Optimization Levels for selected MT-H Queries, $\rho =$ zipfian, $T$ scaling from 1 to 100,000 on a log-scale, MTBase-on-PostgreSQL
7.6.5 Experiments on System C

This subsection presents the same experiments as before, but this time executed on System C and with a bigger scaling factor for the optimization experiments. As this system is a commercial database, we can only make educated guesses what happens if the behaviour is different than on PostgreSQL.

7.6.5.1 Optimization Benefits

As mentioned in Section 7.6, the performance numbers of MTBase on System C show, all in all, the similar trends for the optimization passes than to the ones on PostgreSQL. The only difference is that executing conversion functions, which are implemented as UDFs, is much more expensive in System C because results cannot be cached. As a consequence, executing queries without optimizations gets much worse and Q1 can take up to three hours to be processed on a 10 GB dataset. The results for scenario 1 are shown in Tables 7.7 to 7.9.

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-1G</td>
<td>0.6</td>
<td>0.053</td>
<td>0.1</td>
<td>0.077</td>
<td>0.18</td>
<td>0.067</td>
<td>0.13</td>
<td>0.12</td>
<td>0.28</td>
<td>0.092</td>
<td>0.078</td>
<td>0.1</td>
<td>0.089</td>
<td>0.18</td>
<td>0.18</td>
<td>0.19</td>
<td>0.071</td>
<td>0.25</td>
<td>0.072</td>
<td>5.2</td>
<td>0.21</td>
<td>0.04</td>
</tr>
<tr>
<td>canonical</td>
<td>1000</td>
<td>0.12</td>
<td>230</td>
<td>0.17</td>
<td>1.7</td>
<td>8.7</td>
<td>2.6</td>
<td>3.0</td>
<td>29</td>
<td>10</td>
<td>0.3</td>
<td>0.25</td>
<td>0.66</td>
<td>8.0</td>
<td>18</td>
<td>1.3</td>
<td>2.4</td>
<td>26</td>
<td>29</td>
<td>0.099</td>
<td>0.2</td>
<td>79</td>
</tr>
<tr>
<td>o1</td>
<td>0.78</td>
<td>0.1</td>
<td>0.23</td>
<td>0.14</td>
<td>0.087</td>
<td>0.99</td>
<td>0.95</td>
<td>0.15</td>
<td>1.1</td>
<td>0.12</td>
<td>0.29</td>
<td>0.25</td>
<td>0.65</td>
<td>0.14</td>
<td>0.14</td>
<td>1.3</td>
<td>0.13</td>
<td>8.9</td>
<td>0.91</td>
<td>0.076</td>
<td>0.19</td>
<td>0.55</td>
</tr>
<tr>
<td>o2</td>
<td>0.77</td>
<td>0.1</td>
<td>0.23</td>
<td>0.14</td>
<td>0.087</td>
<td>0.98</td>
<td>0.97</td>
<td>0.15</td>
<td>1.0</td>
<td>0.12</td>
<td>0.28</td>
<td>0.25</td>
<td>0.66</td>
<td>0.14</td>
<td>0.14</td>
<td>1.3</td>
<td>0.13</td>
<td>8.9</td>
<td>0.94</td>
<td>0.077</td>
<td>0.2</td>
<td>3.0</td>
</tr>
<tr>
<td>o3</td>
<td>0.78</td>
<td>0.1</td>
<td>0.23</td>
<td>0.14</td>
<td>0.088</td>
<td>0.97</td>
<td>0.93</td>
<td>0.15</td>
<td>1.1</td>
<td>0.12</td>
<td>0.28</td>
<td>0.25</td>
<td>0.65</td>
<td>0.14</td>
<td>0.14</td>
<td>1.3</td>
<td>0.14</td>
<td>8.8</td>
<td>0.92</td>
<td>0.079</td>
<td>0.2</td>
<td>3.1</td>
</tr>
<tr>
<td>o4</td>
<td>0.78</td>
<td>0.1</td>
<td>0.3</td>
<td>0.14</td>
<td>0.089</td>
<td>0.99</td>
<td>1.0</td>
<td>1.5</td>
<td>1.0</td>
<td>1.0</td>
<td>0.12</td>
<td>0.29</td>
<td>0.25</td>
<td>0.66</td>
<td>0.14</td>
<td>0.14</td>
<td>1.3</td>
<td>1.4</td>
<td>8.9</td>
<td>0.92</td>
<td>0.078</td>
<td>0.2</td>
</tr>
<tr>
<td>ml-only</td>
<td>0.78</td>
<td>0.1</td>
<td>0.24</td>
<td>0.14</td>
<td>0.088</td>
<td>0.97</td>
<td>0.95</td>
<td>0.15</td>
<td>1.0</td>
<td>0.12</td>
<td>0.29</td>
<td>0.25</td>
<td>0.67</td>
<td>0.14</td>
<td>0.14</td>
<td>1.3</td>
<td>1.4</td>
<td>8.9</td>
<td>0.91</td>
<td>0.076</td>
<td>0.2</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Table 7.7: Response Times [sec] of 22 MT-H Queries for MTBase-on-System-C with \( sf = 10 \), \( T = 10 \), \( \rho = \text{uniform} \), \( C = 1 \), \( D = \{1\} \), for different Optimization Levels, versus TPC-H with \( sf = 1 \)

7.6.5.2 Tenant Scaling

An interesting picture can be seen for the tenant scaling experiment in Figure 7.7. While executing the queries for a small number of tenants (\( \leq 10 \)) or a big one (\( \geq 10,000 \)) seems to be reasonably cheap, executing queries for a mid-sized number of tenants seems to increase the costs dramatically. Looking at the query plans of System C did not reveal much because the plans are (probably intentionally) very coarse-grained. Thus, we can
Table 7.8: Response Times [sec] of 22 MT-H Queries for MTBase-on-System-C with $sf = 10$, $T = 10$, $\rho = \text{uniform}$, $C = 1$, $D = \{2\}$, for different Optimization Levels, versus TPC-H with $sf = 1$

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-1G</td>
<td>0.8</td>
<td>0.053</td>
<td>0.1</td>
<td>0.077</td>
<td>0.18</td>
<td>0.067</td>
<td>0.13</td>
<td>0.12</td>
<td>0.28</td>
<td>0.092</td>
<td>0.078</td>
<td>0.1</td>
<td>0.66</td>
<td>0.095</td>
<td>0.1</td>
<td>0.19</td>
<td>0.071</td>
<td>0.25</td>
<td>0.072</td>
<td>5.2</td>
<td>0.21</td>
<td>0.04</td>
</tr>
<tr>
<td>canonical</td>
<td>1100</td>
<td>0.13</td>
<td>240</td>
<td>0.18</td>
<td>1.6</td>
<td>9.0</td>
<td>1.8</td>
<td>2.7</td>
<td>29</td>
<td>10</td>
<td>0.29</td>
<td>0.25</td>
<td>0.66</td>
<td>7.9</td>
<td>18</td>
<td>1.3</td>
<td>2.3</td>
<td>26</td>
<td>28</td>
<td>0.12</td>
<td>0.2</td>
<td>80</td>
</tr>
<tr>
<td>o1</td>
<td>1100</td>
<td>0.12</td>
<td>250</td>
<td>0.16</td>
<td>1.6</td>
<td>9.0</td>
<td>1.8</td>
<td>2.9</td>
<td>30</td>
<td>11</td>
<td>0.29</td>
<td>0.25</td>
<td>0.68</td>
<td>7.9</td>
<td>18</td>
<td>1.3</td>
<td>2.4</td>
<td>26</td>
<td>29</td>
<td>0.13</td>
<td>0.19</td>
<td>80</td>
</tr>
<tr>
<td>o2</td>
<td>1100</td>
<td>0.11</td>
<td>240</td>
<td>0.18</td>
<td>1.6</td>
<td>8.9</td>
<td>14</td>
<td>2.9</td>
<td>40</td>
<td>10</td>
<td>0.29</td>
<td>0.25</td>
<td>0.67</td>
<td>7.8</td>
<td>18</td>
<td>1.3</td>
<td>2.2</td>
<td>26</td>
<td>28</td>
<td>0.11</td>
<td>0.2</td>
<td>80</td>
</tr>
<tr>
<td>o3</td>
<td>240</td>
<td>0.12</td>
<td>4.2</td>
<td>0.18</td>
<td>1.1</td>
<td>1.0</td>
<td>3.2</td>
<td>3.0</td>
<td>17</td>
<td>3.4</td>
<td>0.3</td>
<td>0.25</td>
<td>0.66</td>
<td>8.1</td>
<td>9.2</td>
<td>1.3</td>
<td>2.4</td>
<td>26</td>
<td>28</td>
<td>0.11</td>
<td>0.2</td>
<td>79</td>
</tr>
<tr>
<td>o4</td>
<td>1.1</td>
<td>0.1</td>
<td>0.17</td>
<td>0.14</td>
<td>0.15</td>
<td>0.099</td>
<td>0.91</td>
<td>0.16</td>
<td>0.94</td>
<td>2.1</td>
<td>0.31</td>
<td>0.25</td>
<td>0.67</td>
<td>0.34</td>
<td>0.29</td>
<td>1.3</td>
<td>0.15</td>
<td>1.5</td>
<td>1.1</td>
<td>0.099</td>
<td>0.2</td>
<td>1.2</td>
</tr>
<tr>
<td>inl-only</td>
<td>1.7</td>
<td>0.13</td>
<td>0.2</td>
<td>0.14</td>
<td>0.15</td>
<td>0.1</td>
<td>0.78</td>
<td>0.17</td>
<td>1.1</td>
<td>0.19</td>
<td>0.28</td>
<td>0.25</td>
<td>0.61</td>
<td>0.33</td>
<td>0.21</td>
<td>1.3</td>
<td>0.15</td>
<td>1.5</td>
<td>12</td>
<td>0.099</td>
<td>0.2</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 7.9: Response Times [sec] of 22 MT-H Queries for MTBase-on-System-C with $sf = 10$, $T = 10$, $\rho = \text{uniform}$, $C = 1$, $D = \{1, 2, \ldots , 10\}$, for different Optimization Levels, versus TPC-H with $sf = 10$

<table>
<thead>
<tr>
<th>Level</th>
<th>Q01</th>
<th>Q02</th>
<th>Q03</th>
<th>Q04</th>
<th>Q05</th>
<th>Q06</th>
<th>Q07</th>
<th>Q08</th>
<th>Q09</th>
<th>Q10</th>
<th>Q11</th>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch-10G</td>
<td>7.9</td>
<td>0.097</td>
<td>0.94</td>
<td>0.81</td>
<td>1.6</td>
<td>0.83</td>
<td>0.92</td>
<td>0.68</td>
<td>2.5</td>
<td>0.85</td>
<td>0.27</td>
<td>1.1</td>
<td>5.5</td>
<td>0.92</td>
<td>0.9</td>
<td>1.3</td>
<td>0.7</td>
<td>2.6</td>
<td>0.76</td>
<td>0.14</td>
<td>2.0</td>
<td>0.32</td>
</tr>
<tr>
<td>canonical</td>
<td>11000</td>
<td>0.14</td>
<td>2500</td>
<td>1.7</td>
<td>28</td>
<td>90</td>
<td>20</td>
<td>38</td>
<td>200</td>
<td>1100</td>
<td>0.3</td>
<td>1.2</td>
<td>6.3</td>
<td>73</td>
<td>180</td>
<td>1.3</td>
<td>2.1</td>
<td>69</td>
<td>29</td>
<td>0.17</td>
<td>3.3</td>
<td>800</td>
</tr>
<tr>
<td>o1</td>
<td>11000</td>
<td>0.13</td>
<td>2500</td>
<td>1.6</td>
<td>28</td>
<td>90</td>
<td>21</td>
<td>37</td>
<td>190</td>
<td>1100</td>
<td>0.3</td>
<td>1.2</td>
<td>6.2</td>
<td>74</td>
<td>180</td>
<td>1.3</td>
<td>2.0</td>
<td>69</td>
<td>29</td>
<td>0.16</td>
<td>3.3</td>
<td>800</td>
</tr>
<tr>
<td>o2</td>
<td>11000</td>
<td>0.15</td>
<td>2400</td>
<td>1.7</td>
<td>29</td>
<td>90</td>
<td>24</td>
<td>39</td>
<td>310</td>
<td>1100</td>
<td>0.3</td>
<td>1.2</td>
<td>6.2</td>
<td>74</td>
<td>180</td>
<td>1.4</td>
<td>2.1</td>
<td>69</td>
<td>30</td>
<td>0.17</td>
<td>3.4</td>
<td>790</td>
</tr>
<tr>
<td>o3</td>
<td>2400</td>
<td>0.12</td>
<td>43</td>
<td>1.7</td>
<td>22</td>
<td>9.8</td>
<td>18</td>
<td>35</td>
<td>64</td>
<td>52</td>
<td>0.31</td>
<td>1.2</td>
<td>6.3</td>
<td>74</td>
<td>65</td>
<td>1.3</td>
<td>1.9</td>
<td>69</td>
<td>30</td>
<td>0.16</td>
<td>3.4</td>
<td>790</td>
</tr>
<tr>
<td>o4</td>
<td>38</td>
<td>0.13</td>
<td>1.1</td>
<td>1.6</td>
<td>0.59</td>
<td>0.97</td>
<td>1.6</td>
<td>1.2</td>
<td>5.3</td>
<td>29</td>
<td>0.31</td>
<td>1.2</td>
<td>6.2</td>
<td>1.1</td>
<td>2.4</td>
<td>1.3</td>
<td>0.83</td>
<td>11</td>
<td>1.3</td>
<td>0.14</td>
<td>3.4</td>
<td>0.75</td>
</tr>
<tr>
<td>inl-only</td>
<td>42</td>
<td>0.13</td>
<td>1.8</td>
<td>1.7</td>
<td>1.6</td>
<td>1.2</td>
<td>9.8</td>
<td>1.2</td>
<td>5.5</td>
<td>13</td>
<td>0.3</td>
<td>1.2</td>
<td>6.3</td>
<td>1.1</td>
<td>1.6</td>
<td>1.3</td>
<td>0.84</td>
<td>11</td>
<td>17</td>
<td>0.18</td>
<td>3.4</td>
<td>0.53</td>
</tr>
</tbody>
</table>

only speculate that for a mid-sized number of tenants, the optimizer does a couple of unfortunate decisions.

### 7.7 Related Work

MTBase builds heavily on and extends a lot of related work. This section gives a brief summary of the most prominent lines of work that influenced our design.

**Shared-resources (SR) systems** In related work, this approach is also often called database virtualization or database as a service (Daas) when it is used in the cloud context. Important lines of work in this domain include (but are not limited to) SqlVM/Azure SQL DB [NDS+13, DLNK16], RelationalCloud [MCM13], SAP-HANA [SAP16] and Snowflake [DCZ+16], most of which was well summarized by Elmore et al. [ECAEA13].
Figure 7.7: Response Times (relative to TPC-H, $sf = 100$) of $o4$ and inlining-only Optimization Levels for selected MT-H Queries, $\rho =$ zipfian, $T$ scaling from 1 to 100,000 on a log-scale, MTBase-on-System-C
MTBase complements these systems by providing a platform that can accommodate more, but typically smaller tenants.

**Shared-databases (SD) systems**  This approach, while appearing in the spectrum of multi-tenant databases by Chong et al. [CCW06], is rare in practice. Sql Azure DB [DLNK16] seems to be the only product that has an implementation of this approach. However, even Microsoft strongly advises against using SD and instead recommends to either use SR or ST [Mic16].

**Shared-tables (ST) systems and schema evolution**  Work in that area includes Salesforce [WB09], Apache Phoenix [Apa16b], FlexScheme [AGJ⁺08, ASJK11] and Azure SQL Database [Mic16]. Their common idea, as in MTSQL, is to use an invisible tenant identifier to identify which records belong to which tenant and rewrite SQL queries in order to include filters on this ttid. MTSQL extends these systems by providing the necessary features for cross-tenant query processing.

**Database Federation / Data Integration**  The problem that data integration (DI) systems (e.g., CLIO [HMN⁺99, FHH⁺09] and Potter’s Wheel [RH01]) try to solve is to find schema and data mappings between different schemata used by different data sources and a target schema specified by the client application (cf. Subsection 7.1.2). In order to relate DI to cross-tenant query processing, we could model each tenant as a data source. Consequently, MTBase in the case of ST solves the special sub-class of DI problems that deals with data translation. DI is often combined with database federation [Lev98, HLR02], which means that there exist small program modules (called integrators, mediators or simply wrappers) to map data from different sources (possibly not all of them SQL databases) into one common format. While data federation generalizes well across the entire spectrum of multi-tenant databases, maintaining such wrapper architectures is expensive, both in terms of code maintenance and update processing. Conversely, MTSQL enables cross-tenant query processing in a more efficient and flexible way in the context of SS and ST databases.

**Confidentiality**  How to compose MTBase with tenant data encryption as proposed by Chong et al. [CCW06] is not obvious as this opens the question how tenants can process
7.7. Related Work

data for which they have permission to process, but which is owned by another tenant (and is therefore encrypted with that other tenant’s key). Obviously, simply sharing the key of tenant $t$ with all tenants that were granted the privilege to process $t$’s data is not a viable solution, as this allows them to impersonate $t$, which defeats the whole purpose of encryption. How to address this issue also depends a lot on the given attacker scenario: Do we want to protect tenants from each other? Do we trust the cloud provider? Do we expect honest-but-curious behavior or active attacks? Also the granularity at which a tenant can share data with another tenant matters: schema- vs. table- vs. attribute- vs. row- vs. predicate-based, aggregations-only and possible combinations of these variants. For some of these granularities and attacker models, proposed solutions exist [DFJ+07, CEK+10].

Query Optimization/Compilation Using semantic optimizations to reduce conversion function costs comes at the possible drawback that the underlying DBMS might optimize for another cost metric, thereby possibly generating a sub-optimal overall plan in some cases as the two optimization steps happen independently. A possible solution towards that end would be to integrate MTSQ\(\text{L}\) directly into a query compiler/optimizer of a specific SQL-capable system instead of putting a rewrite engine on top. This would allow multi-objective query optimization (for which an efficient algorithm exists [TK16]) where tuple conversion costs is just one metric among others. The challenge that query compilers are often monolithic code monsters and therefore hard to work with, could be mitigated by using a modern layered architecture with DSLs [SKP+16].

A known performance challenge for today’s databases are dependant nested sub-queries as described by Neumann et al. [NK15] where predicates depend on the evaluation of a sub-query. The generated query plans for such queries typically incur $O(n^2)$ operations whereas an optimal unnested plan often needs only $O(n)$ for a table with $n$ records. For MTBase, such a quadratic overhead becomes even worse if the join attributes contain convertible attributes which might consequently lead to a quadratic number of conversion function calls. Hence, MTBase would potentially profit enormously from an algorithm that can unnest arbitrary queries as described in the aforementioned work [NK15].
Chapter 7. MTBase: Optimizing Cross-Tenant Database Queries

7.8 Concluding Remarks

This chapter presented MTSQL, a novel paradigm to address cross-tenant query processing in multi-tenant databases. MTSQL extends SQL with multi-tenancy-aware syntax and semantics, which allows efficient optimization and execution of cross-tenant queries in MTBase. MTBase is an open-source system that implements MTSQL. At its core, it is an MTSQL-to-SQL rewrite middleware sitting between a client and any DBMS of choice. The performance evaluation with a benchmark adapted from TPC-H showed that MTBase (on top of PostgreSQL) can scale to thousands of tenants at very low overhead and that our proposed optimizations to cross-tenant queries are highly effective.

In the future, we plan to further analyze the interplay between the MTBase- and the DBMS query optimizer in order to implement cost-based optimizations. We also want to study conversion functions that vary over time and investigate how MTSQL can be extended to temporal databases. Moreover, we would like to look more into the confidentiality issues of multi-tenant databases, in particular how to enable cross-tenant query processing if data is encrypted.

With respect to the claims in Chapter 1, we can say that MTSQL allows trading off between confidentiality and value in the sense that the more privileges a tenant gives to other tenants, the less confidential her data gets, but the bigger becomes the potential value of that data as it can be combined with other datasets. The question remains who will profit from that added value. Ideally, cloud providers would come up with new pricing models where tenants get better prices the more data they share. Notably, this should happen in a way that tenant still have full control over the disclosure of their data, which highlights again the importance of studying how confidentiality and cross-tenant query processing can work together.
Summary and Conclusions

In this thesis we looked at two important aspects of cloud databases: performance and confidentiality. Both aspects directly relate to the biggest concerns that hinder people from moving to the cloud, namely cost and privacy.

Part I of this thesis explored new directions towards the construction of a distributed integrated data storage solution that is both, highly-elastic and highly-scalable, and at the same time allows processing short-running update-intensive along long-running read-mostly analytical transactions. The final design that we propose, a versioned data store with support for fast scans built around a columnar-oriented storage layout, not only shows excellent performance with respect to real-life and synthetic benchmarks, but offers the ease of quickly reacting to changing hardware capacities as well as increasing/decreasing workloads. This makes TellStore a very attractive alternative to other NoSQL stores like Cassandra, Kudu or RocksDB, for any SQL-over-NoSQL data processing system, be it Spark, Presto or Tell.

The techniques presented in Part II, RPE and MTSQL, kept in mind that performance is key to anything that happens in the cloud, but in addition, investigated how to improve confidentiality and value of the stored data. The main result, perhaps surprising, was that, if carefully designed, confidentiality and value can be improved up to a specific level at virtually no cost. As such, these techniques are attractive to be adopted by cloud providers because they allow them to increase their customers’ satisfaction (and thus likely their own profit) with only a small, one-time investment.
Chapter 8. Summary and Conclusions

8.1 Performance

We started off in Chapter 2 by motivating the emerging class of combined streaming and analytical workloads which is of rapidly-growing importance to the industry. We assessed related work in different domains, like streaming engines, database management systems and HTAP systems, as well as key-value stores and found none of the related work suitable for processing such workloads in an elastic, scalable way.

Based on that motivation, we developed new benchmarks in Chapter 3 to capture the characteristics of analytics and transactions on stateful streams. The first benchmark, Huawei-AIM, originates from the telecommunication industry and covers the scenario of performing real-time analytics on rapidly-changing aggregated billing and marketing data. The second benchmark, YCSB#, extends the widely-used YCSB benchmark in order to model arbitrary transactions on frequently-evolving data. A system performing well in both these benchmarks proves the ability to handle not only complex and ad-hoc transactions, but also stream-fashioned updates and is therefore ready to accept the challenges of big data as many people see it evolving in the next decade.

Chapter 4 presented Analytics in Motion (AIM) as a system specifically designed to address the Huawei-AIM benchmark in the cloud context. As such, AIM can be regarded as the gold standard for this benchmark, also because it achieves outstanding performance and fulfills all the desired properties mentioned in the introduction: elasticity, scalability and low and predictable TCO. The key ingredients that allowed AIM to achieve all this, were a PAX-oriented storage design, called ColumnMap, and a two-step scan-merge algorithm for interleaved processing of updates and scans, which was not only a natural fit for delta-main updates, but also simplified concurrency. In conclusion, AIM essentially showed what can be achieved and thereby served as a starting point for further research towards a more general-purpose system.

Such a system is Tell 2.0, shortly touched upon in Chapter 5. Tell 2.0, which runs on top of a key-value store and is able to achieve outstanding performance on mixed transactional analytical workloads, i.e., Huawei-AIM and YCSB#, if that key-value store supports native versioning and fast scans. This is why we carefully researched the design space for such a store and identified asynchronous data access and communication, shared scans, page-based record allocation as well as thread pinning and a clear separation of duties between these threads as promising key design principles. Following these principles,
we implemented TellStore with three different storage layouts and experimentally proved that it is the only representative among the state-of-the-art KV stores on which Tell 2.0 can reach such an outstanding performance. Furthermore, among the three storage layouts that we tested within TellStore, ColumnMap 2.0, a data structure that evolved from the original ColumnMap in AIM, exhibited the best overall performance across all experiments that we conducted.

8.2 Confidentiality and Value

What stops business organizations, especially financial institutions, from moving to cloud databases, are confidentiality concerns. While strong encryption can in principal be used to perfectly hide the content of the data and is fine to use in data at rest, e.g., encrypted files, it is infeasible to be used in practice as soon as data needs not only be accessed, but actually processed. Consequently, using encryption in cloud databases while not killing their performance is a tough challenge that we addressed with randomly-partitioned encryption (RPE) in Chapter 6. In contrast to many related works on order-preserving encryption (OPE), we did not focus on standard cryptographical attacks, but instead analyzed the confidentiality of different schemes, including RPE, in the presence of more practical attacks derived from insider knowledge, i.e., domain and frequency attacks.

As our confidentiality analysis revealed, RPE, which is itself built on top of an existing OPE scheme, amends the confidentiality of that scheme by pseudo-randomly partitioning the data based on a secret key. Our experimental evaluation showed that analytical query processing on the TPC-H benchmark was not more than 2.5 times more expensive than processing the queries on un-encrypted data. However, what makes RPE even more appealing to be used in practice, is the fact that tuning for good confidentiality did not even hurt: For many queries, having more partitions did not decrease performance and in some cases made it even better because more partitions allowed a higher degree of processing parallelism.

Finally, we studied how we can increase the value of data in the cloud without sacrificing performance in Chapter 7. We found that the semantics of cross-tenant query processing, the technique to create more valuable insights from tenants sharing their data, cannot be thoroughly expressed in plain SQL. Consequently, we proposed MTSQL, an extension to SQL, to close this gap. What is more, we showed by construction that for a given context,
Chapter 8. Summary and Conclusions

MTSQL can be translated to plain SQL and can hence be executed on a standard database. This allowed the design of an MTSQL processing system, called MTBase, that acts as thin MTSQL-aware middleware between a client and the (cloud) database.

The experimental validation of MTBase unveiled huge costs for cross-tenant query processing caused by the conversion functions used to translate data from one tenant’s format to another. Luckily, we found a number of optimizations exploiting the mathematical properties of such conversion functions which helped to reduce the performance overhead to a minimum, even for datasets of up to 100,000 tenants.

Both of these techniques, RPE and MTSQL, have in common that they are low-hanging fruits for cloud providers to grab because they make their data more confidential, respectively more valuable, with only a small performance overhead. In addition, MTSQL is open-source and fully-compliant with the SQL standard, which facilitates its adoption even more. Both techniques minimize the processing costs on the client-side and delegate as much processing as possible to the cloud, thereby following the assumption of thin clients and minimizing TCO. Furthermore, PRE and MTSQL inherit the scalability and elasticity properties from the underlying database. This makes us confident that, as cloud databases become more and more elastic and scalable, perhaps by integrating some of the principles mentioned in Part I, the cost overhead of confidentiality and cross-tenant query processing become smaller and smaller and will eventually vanish.

8.3 Combining Confidentiality, Performance and Value

As we developed, studied and evaluated different new techniques to improve confidentiality, performance and value of cloud databases, the question arises to which extent these complementary techniques can be combined in order to get the full benefit.

As already argued in the section above, the algorithms for rewriting and optimizing MTSQL are completely independent from the underlying database and the same is true for query encryption and post-processing with RPE. That means that conceptually, MTSQL as well as RPE can individually be used in conjunction with any system that implements the presented data structures and algorithms of Part I of this thesis, for instance Tell 2.0.
Revisiting confidentiality, it is important to distinguish between threads internal and external to a cloud database. Today’s cloud providers offer a two-fold approach to external attacks: First, tenants can establish secure connections to communicate with the cloud, which ensures that the transferred data (e.g., query results) remains confidential. Second, tenants are protected from each other by the SQL API, more concretely SQL access control. MTSQL extends SQL access control and therefore fully composes with these mechanisms.

We also would like to highlight MTSQL’s composability with differential privacy. Differential privacy algorithms [DMNS06, Dwo11] protect sensitive data records by keeping track of all the queries a specific client has asked and adding random noise to a query result as soon as it would, combined with the results obtained so far, reveal confidential information. Thus, complementing MTSQL with differential privacy algorithms offers tenants more fine-grained control over their data: Instead of just allowing or disallowing another tenant to read her data, a tenant could now also allow another tenant to only compute aggregated statistics on her data.

The remaining question is how MTSQL composes with techniques that address internal attacks with searchable database encryption schemes, e.g., RPE. If every tenant uses a different secret key to encrypt her data, query processing across several tenants requires knowledge of all these keys. It is clearly not acceptable for a tenant to disclose her secret key to any other tenant just for the sake of giving this other tenant (partial) access to her data. However, it might be acceptable to use a trusted middleware to which all tenants upload their keys and which executes query encryption and result decryption on behalf of these tenants.

8.4 Ongoing and Future Work

This thesis not only looked at some questions and trade-offs regarding performance and confidentiality in the cloud, but also opened a couple of avenues for further research.

Extending the Tell Processing Stack Although, RPE and MTBase could in theory run on top of Tell 2.0, adapting the query rewrite algorithms accordingly would be a tedious, error-prone task. The reason is that Tell 2.0 currently offers two interfaces,
Chapter 8. Summary and Conclusions

TellDB and TellJava, none of which complies to the SQL standard. So clearly, building a SQL interface, which would also require to build a query compiler and probably an optimizer, is an interesting and important milestone on Tell’s road map to become a fully-fledged database suitable for the cloud. An advantage is Tell’s layered design, which allows adoption at different levels and enriches the search space for application development. For instance, graph processing applications could be built on top of TellDB or such a new SQL interface. Furthermore, it would be interesting to research whether some of the optimizations we made in MTBase would pay off even more when executed directly in TellStore and also enhancing TellStore with encryption functionality seems an interesting path to follow. Moreover, Tell 2.0 is open-source, which nourishes our hope that such new features will not only be further developed within Systems Group, but also pushed forward by the research community as a whole.

Encryption and Multi-Tenancy  Another interesting question that we would like to investigate further is how RPE (or encryption in general) composes with cross-tenant query processing. RPE allows a majority of SQL operators to be processed directly on encrypted data, but only if that data was all encrypted with the same key, respectively the same set of keys. However, if we want to query data across tenants where each tenant uses her own set of keys, the problem gets more challenging and it is a completely open question whether something towards that direction can be achieved when relying on a software-only approach and not trusting any third party. An interesting approach using trusted hardware is Cipherbase [ABE+13] where the majority of the SQL processing is done in an untrusted cloud database (SQL Server), but the key operations on encrypted data are executed on trusted (tamper-proof) hardware, in that case an FPGA. To the best of our knowledge, Cipherbase is currently also just single-tenant, but the trusted hardware offers interesting new opportunities for confidential cross-tenant query processing if the FPGA is not only used for secure computations, but also for access control. MTSQL, which allows modelling encryption and decryption as conversion functions, provides the necessary semantics to reason about such a hardware-based implementation and hence serves as an interesting starting point for further research.
List of Tables

2.1 Overview on different Streaming Workloads .............................. 12

3.1 Schema snippet of the Analytics Matrix ................................. 28

3.2 Example Business Rules .................................................. 32

3.3 Random Parameters of Huawei Benchmark ............................... 38

4.1 Default SLAs as used in AIM Implementation ........................... 58

5.1 Response Time of YCSB# Q1 on popular KV Stores ...................... 79

6.1 Confidentiality & Performance of different Database Encryption Schemes 115

6.2 Example of Det-RPE for $U = 2$ ........................................ 123

6.3 Example of Prob-RPE for $U = 2$ ........................................ 126

6.4 Example of Prob-MRPE for $U = 2$ ...................................... 129

6.5 Confidentiality Downgrade under Known-Plaintext Attack .............. 130

6.6 Supported SQL Operators in RPE ......................................... 136

7.1 Overview on Attribute Comparability in MTSQQL ......................... 150

7.2 Distributability over Conversion Functions ............................... 170

7.3 Optimization Levels used for Evaluation of MTSQQL .................... 177

7.4 MT-H Response Times for Scenario 1, $D = \{1\}$, PostgreSQL ........... 178
List of Tables

7.5  MT-H Response Times for Scenario 1, $D = \{2\}$, PostgreSQL . . . . . . . . 179
7.6  MT-H Response Times for Scenario 1, $D = \{1, 2, \ldots, 10\}$, PostgreSQL . . . 179
7.7  MT-H Response Times for Scenario 1, $D = \{1\}$, System C . . . . . . . . . . . . . . . 181
7.8  MT-H Response Times for Scenario 1, $D = \{2\}$, System C . . . . . . . . . . . . . . . 182
7.9  MT-H Response Times for Scenario 1, $D = \{1, 2, \ldots, 10\}$, System C . . . . . . . 182
List of Figures

1.1 Overview of Systems and Techniques presented in this Thesis 5
2.1 Stateless Streaming 14
2.2 Stateful Streaming 14
2.3 Streaming Analytics 15
2.4 Streaming Transactions 15
3.1 Huawei-AIM Data Model 27
3.2 Exemplary Dimension Records used in AIM Benchmark 29
3.3 Updating an Entity Record 30
3.4 YCSB# Data Model for $N = 10$ 43
3.5 YCSB# Data Model for different Values of $N$ 44
4.1 AIM Architecture 53
4.2 Separating Updates from Query Processing 55
4.3 Logical 3-tier Architecture of AIM Prototype 59
4.4 ColumnMap Structure 62
4.5 Collaboration of ESP and RTA Thread 66
4.6 SIMD Processing 68
4.7 Analytics Matrix Partitioning and Thread Model 69
## List of Figures

4.8 RTA Strong Scale-Up on AIM ............................................. 71
4.9 RTA Peak Performance on different systems ......................... 73
4.10 RTA Strong Scale-Out on AIM ........................................... 74
4.11 Weak Scale-Out of RTA Queries on AIM ............................. 75

5.1 SQL-over-NoSQL Architecture ............................................ 81
5.2 General TellStore Thread Model .......................................... 88
5.3 TellStore Page Scan ...................................................... 89
5.4 TellStore-Log: a Log-structured KVS .................................. 91
5.5 Architecture of Delta-Main Storage .................................... 92
5.6 Multi-Version Record in two different Formats ....................... 93
5.7 Throughput Scale-Out of YCSB# Transactions on various KV stores ... 98
5.8 Throughput Scale-Out of YCSB# Transactions on TellStore and Kudu .. 99
5.9 TellStore Response Time Variation for insert-only YCSB# Workload .... 100
5.10 TellStore Throughput for varying InfinIO Batch Size ................. 101
5.11 Respone Times Scale-Out of YCSB# Queries in Isolation ............ 101
5.12 Response Time of YCSB# Q1 for Concurrent Transactional Workloads .. 103
5.13 Response Times of Huawei-AIM Queries with concurrent Transactions .. 103

6.1 Client-Server Database Architecture with and without Encryption .... 117
6.2 Different Approaches for Partially-Order-Preserving Encryption ........ 121
6.3 RPE Principle ............................................................ 121
6.4 ROW Advantage under Domain Attack .................................. 125
6.5 FOW Advantage under Frequency Attack .............................. 128
6.6 RPE Encryption Example for Varchar .................................. 133
6.7 Generating several codes per Run ...................................... 134
6.8 RPE Response Times for TPC-H Queries .............................. 138
6.9  Response Times for varying number of Runs  ....................... 139

7.1  Spectrum of Multi-Tenant Databases  ............................ 142
7.2  Overview of Cross-Tenant Query Processing Systems ............ 145
7.3  Database in Basic Layout  ........................................ 148
7.4  Database in Private Table Layout  ............................... 149
7.5  MTBase Architecture  .............................................. 159
7.6  MT-H Response Time Scaling for Scenario 2, PostgreSQL ..... 180
7.7  MT-H Response Time Scaling for Scenario 2, System C ......... 183
List of Algorithms

3.1 Updating Indicators in the Analytics Matrix ........................................ 30
3.2 Business Rule Evaluation ................................................................. 32
3.3 YCSB# random read operation ......................................................... 45
3.4 YCSB# random update operation ..................................................... 45
3.5 YCSB# random insert operation ....................................................... 46
3.6 YCSB# delete operation ................................................................. 46
4.1 Analytics Matrix Get ................................................................. 64
4.2 Analytics Matrix Put ................................................................. 64
4.3 Shared Scan Query Processing ...................................................... 66
7.1 Canonical Query Rewrite Algorithm .............................................. 160
7.2 Rewrite Algorithm for FROM clause .............................................. 163
List of Listings

3.1 Huawei-AIM Benchmark, Query 1 ................................. 33
3.2 Huawei-AIM Benchmark, Query 2 ................................. 34
3.3 Huawei-AIM Benchmark, Query 3 ................................. 34
3.4 Huawei-AIM Benchmark, Query 4 ................................. 35
3.5 Huawei-AIM Benchmark, Query 5 ................................. 35
3.6 Huawei-AIM Benchmark, Query 6 ................................. 35
3.7 Huawei-AIM Benchmark, Query 7 ................................. 37
7.1 Simple SCOPE expression using IN ................................. 149
7.2 Complex SCOPE expression with sub-query ........................ 150
7.3 MSQL CREATE TABLE statement ................................. 151
7.4 Converting a phone number to universal form, postgreSQL syntax ........................ 153
7.5 Converting to a specific phone number format, postgreSQL syntax ........................ 153
7.6 Converting a currency to universal form (USD), postgreSQL syntax ........................ 154
7.7 Converting from USD to a specific currency, postgreSQL syntax ........................ 154
7.8 MSQL GRANT syntax ................................. 156
7.9 Example of an MSQL GRANT statement .............................. 156
7.10 Examples for Rewriting SELECT clause .............................. 161
7.11 Examples for Rewriting WHERE clause .............................. 162
7.12 Rewriting a complex SCOPE expression .............................. 163
7.13 Rewriting a global foreign-key constraint .............................. 164
7.14 Rewriting a tenant-specific foreign-key constraint .............................. 165
7.15 Rewriting an INSERT statement ................................. 165
7.16 Examples for trivial Semantic Optimizations .............................. 167
7.17 Example for Client Presentation Push-Up .............................. 168
7.18 Example for Conversion Push-Up ................................. 169
7.19 Example for Conversion Function Distribution .............................. 171
LIST OF LISTINGS

7.20 Example for Function Inlining ........................................... 172
Bibliography


Bibliography


<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIM</td>
<td>6, 53</td>
</tr>
<tr>
<td>AIM parameters</td>
<td>37</td>
</tr>
<tr>
<td>analytics matrix</td>
<td>26, 54, 58, 61</td>
</tr>
<tr>
<td>bucket</td>
<td>62</td>
</tr>
<tr>
<td>campaigns</td>
<td>28, 61</td>
</tr>
<tr>
<td>CAS</td>
<td>90</td>
</tr>
<tr>
<td>CDR</td>
<td>12, 52</td>
</tr>
<tr>
<td>ciphertext</td>
<td>112</td>
</tr>
<tr>
<td>column-major</td>
<td>84</td>
</tr>
<tr>
<td>ColumnMap</td>
<td>62, 94</td>
</tr>
<tr>
<td>compaction</td>
<td>65</td>
</tr>
<tr>
<td>conditional write</td>
<td>22, 64, 92</td>
</tr>
<tr>
<td>confidentiality</td>
<td>111</td>
</tr>
<tr>
<td>consistency</td>
<td>68</td>
</tr>
<tr>
<td>conversion function</td>
<td>151</td>
</tr>
<tr>
<td>copy-on-write</td>
<td>55</td>
</tr>
<tr>
<td>cost-based optimization</td>
<td>166</td>
</tr>
<tr>
<td>cross-tenant query processing</td>
<td>143</td>
</tr>
<tr>
<td>CRUD</td>
<td>44</td>
</tr>
<tr>
<td>D-filter</td>
<td>162</td>
</tr>
<tr>
<td>data integration</td>
<td>143</td>
</tr>
<tr>
<td>data warehousing</td>
<td>18, 54, 144</td>
</tr>
<tr>
<td>delta</td>
<td>55, 83</td>
</tr>
<tr>
<td>Det-RPE</td>
<td>120</td>
</tr>
<tr>
<td>deterministic RPE</td>
<td>120</td>
</tr>
<tr>
<td>differential updates</td>
<td>55</td>
</tr>
<tr>
<td>differential privacy</td>
<td>191</td>
</tr>
<tr>
<td>dimensional data</td>
<td>16, 27, 57, 58</td>
</tr>
<tr>
<td>domain attack</td>
<td>112</td>
</tr>
<tr>
<td>elasticity</td>
<td>3, 76, 80, 106</td>
</tr>
<tr>
<td>ESP</td>
<td>28</td>
</tr>
<tr>
<td>ETL</td>
<td>19</td>
</tr>
<tr>
<td>event</td>
<td>28</td>
</tr>
<tr>
<td>FOW</td>
<td>119</td>
</tr>
<tr>
<td>frequency attack</td>
<td>112</td>
</tr>
<tr>
<td>get</td>
<td>21, 76, 80, 87, 96</td>
</tr>
<tr>
<td>honest-but-curious</td>
<td>6, 112</td>
</tr>
<tr>
<td>HTAP</td>
<td>20, 80</td>
</tr>
<tr>
<td>Huawei use case</td>
<td>12, 26</td>
</tr>
<tr>
<td>ideal object</td>
<td>113</td>
</tr>
<tr>
<td>Infiniband</td>
<td>60, 70</td>
</tr>
<tr>
<td>insider knowledge</td>
<td>112</td>
</tr>
<tr>
<td>join</td>
<td>57</td>
</tr>
<tr>
<td>main</td>
<td>56, 83</td>
</tr>
<tr>
<td>merge</td>
<td>56</td>
</tr>
<tr>
<td>modular RPE</td>
<td>128</td>
</tr>
<tr>
<td>MT-H Benchmark</td>
<td>174</td>
</tr>
<tr>
<td>MTBase</td>
<td>6, 158</td>
</tr>
<tr>
<td>MTSQL</td>
<td>6, 147</td>
</tr>
</tbody>
</table>
multi-row transaction, 18, 22
multi-tenancy spectrum, 142
OLAP, 18, 52, 80
OLTP, 18, 52, 58, 80
PAX, 62
performance isolation, 87, 104
plaintext, 112
probabilistic RPE, 125
put, 21, 76, 80, 87, 96
random shift cipher, 114, 128
RDMA, 22
response time, 4
ROW, 118
row-major, 84
RPE, 6, 111, 112
RTA, 32
scalability, 3, 76, 106
scale-out, 3, 73, 74
scale-up, 3, 71
scan, 23, 76, 80, 87, 88
schema, 88, 95
semantic optimization, 166
session, 97
shared scan, 56, 65, 88
SIMD, 19, 66, 84
single-row transaction, 18, 22, 56, 58, 65
SLA, 37, 75
snapshot isolation, 80
SQL-over-NoSQL, 53, 80
stateful streaming, 13
stateless streaming, 13
streaming analytics, 16
streaming systems, 11
streaming transactions, 17
TATP Benchmark, 42
TCO, 40, 52, 74, 76, 106
Tell, 80
TellStore, 6, 79, 86
Tell 2.0, 80, 83, 86
thin client, 3
throughput, 4
versioning, 76
work stealing, 56
YCSB#, 42