Doctoral Thesis

Indexing and search on complex data warehouses and rapidly-changing data

Author(s):
Blunschi, Lukas

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007579172

Rights / License:
In Copyright - Non-Commercial Use Permitted
INDEXING AND SEARCH ON COMPLEX DATA WAREHOUSES AND RAPIDLY-CHANGING DATA

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

LUKAS BLUNSCHI

Master of Science ETH in Computer Science, ETH Zurich

April 21, 1981

citizen of

Oberrohrdorf, Aargau

accepted on the recommendation of

Prof. Donald Kossmann, examiner
Prof. Gustavo Alonso, co-examiner
Prof. Karl Aberer, co-examiner
Dr. Kurt Stockinger, co-examiner

2012
Abstract

Data analysis is a large field which has multiple facets and encompasses diverse techniques in a variety of domains. This thesis looks at two problems from the data analysis field: Keyword search on data warehouses and indexing of moving objects.

Problem I: Keyword search on data warehouses has recently become more and more important due to the availability of large amounts of data stored in data warehouses. Today, big financial institutes like Credit Suisse replicate most of their data in a data warehouse. The data warehouse is used for analysis purposes, e.g., to find customers of interest to the company, to discover new trends, or to make any other kind of business decision.

The original task of a data warehouse was to run large SQL queries on relatively old data to produce reports which help business people to make long term business decisions. Nowadays, a data warehouse is used for an increasing number of applications. Each application has new data requirements: An application might require data which is not yet stored in the data warehouse, it might ask for data in a different format, or with shorter time constraints. To handle these new requirements, the data warehouse is reloaded with new data more frequently and the schema of the data warehouse becomes increasingly complex.

Furthermore, to circumvent the delay of producing a new report and going through the IT department, business people now also want to query the data warehouse directly. Keyword search on data warehouses is intended to help these business users to write their queries and to find the data items they are looking for.

This thesis investigates a new approach to keyword search on data warehouses that uses semantics in addition to base data stored in a data warehouse. Semantic information in a data warehouse is stored in metadata, such as the relational schema, domain ontologies, or synonyms. Our approach models this metadata as a graph and uses metadata graph patterns to find relevant information.

We implemented a prototype system which takes a simplified natural language query and automatically generates correct SQL. Such a system enables business analysts to run ad-hoc queries on complex data warehouses.
Problem II: Moving objects indexing is needed in domains such as car and mobile phone tracking, airplane surveillance, emergency services, and gaming engines. Updates in these applications are usually messages containing the new position of an object. Queries may be range queries, nearest-neighbor queries, or queries asking for predicted positions of an object in the future. The main problem is how to support efficient query processing under high update rates.

Considerable work has been done in the area of moving objects indexing. Many of the existing approaches are centered around extending external memory structures. All of these methods assume that data would not fit into main memory. The most relevant work for us is the B\textsuperscript{x}-tree as, conceptually, it has some similarities to the indexing strategy presented in this thesis. Besides mapping a two-dimensional space to a single dimension by using a space filling curve, the B\textsuperscript{x}-tree also partitions data into phases corresponding to future time intervals. For each phase, it uses a separate subtree to index moving objects and predicate positions. Besides methods that extend external memory structures, techniques that use main memory for monitoring queries are also relevant to us. These approaches usually work with fix-sized grids where the grid-size is chosen with respect to the average query window size. Each grid cell maintains pointers to the query results.

Traditionally, index creation has been considered an extremely costly process. For that reason, research on moving object indexes has been centered around creating sophisticated index structures. These indexes are created once, kept, and then modified according to incoming updates. This has led to a plethora of complex index structure proposals in the past. However, with the rise of large main memories and fast multi-core CPUs, this “natural law” of maintaining a moving object index can be questioned.

This thesis presents a novel, main-memory based method that never updates an index, but instead creates as many new indexes as possible. As long as the index build rate is high, an illusion of a continuously up-to-date index will be created. We will show that—surprisingly—index creation can be a matter of subseconds even for large datasets.

In our prototype querying is fast, because indexes can be fully read-optimized. Updating is also fast, because updates can simply be collected before creating an index. The index rebuild process can run separately without influencing querying and updating.
Zusammenfassung


Problem I: Schlagwortsuche in Data-Warehouse-Systemen wurde in letzter Zeit immer wichtiger, weil immer größere Datenmengen in Data-Warehouse-Systemen abgespeichert werden. Heute replizieren große Finanzinstitute wie die Credit Suisse nahezu alle ihre Daten in ein Data-Warehouse. Data-Warehouse-Systeme werden für Analysezwecke verwendet, z.B. um für die Firma interessante Kunden aufzuspüren, um neue Trends abzufangen, oder um die Entscheidungsfindung zu erleichtern.

Die ursprüngliche Aufgabe eines Data-Warehouses war es grosse SQL Abfragen auf relativ alten Daten laufen zu lassen, um Berichte zu generieren, die Entscheidungsgrundlage für längerfristige Geschäftsentscheidungen waren. Heutzutage wird ein Data-Warehouse für eine steigende Anzahl von Anwendungen verwendet und jede dieser Anwendungen bringt neue Anforderungen an das Data-Warehouse mit sich: Entweder braucht eine Anwendung Daten, welche noch nicht im Data-Warehouse gespeichert sind, Daten in einem anderen Format, oder die Daten müssen aktueller sein, als sie momentan im Data-Warehouse existieren. Um diesen neuen Anforderungen zu genügen, wird das Data-Warehouse immer öfter mit neuen Daten geladen und das Datenbankschema wird dabei zunehmend komplexer.

Zusätzlich kommt hinzu, dass Geschäftsleute, um Zeit zu sparen, Abfragen direkt auf dem Data-Warehouse absetzen möchten. Dadurch kann das IT Departement umgangen werden und die Erstellung komplizierter Berichte entfällt. Schlagwortsuche in Data-Warehouse-Systemen soll Geschäftsleuten helfen ihre Abfragen speditiv zu schreiben und so die gesuchten Informationen rasch zu finden.

In dieser Doktorarbeit untersuchen wir einen neuen Ansatz zur Schlagwortsuche in Data-Warehouse-Systemen welche die Semantik (Bedeutung) der Daten berücksichtigt. Semantische Informationen werden in einem Data-Warehouse in
Metadaten zusätzlich zu den eigentlichen Basisdaten abgelegt. Beispiele von solchen Metadaten sind das Datenbankschema, Domain-Ontologien oder Synonyme. Unser Ansatz modelliert diese Metadaten als Graph und verwendet Graphmuster, um relevante Daten zu finden.

Wir haben einen Prototypen programmiert, welcher eine vereinfachte, natürlichsprachige Abfrage entgegen nimmt und daraus automatisch korrektes SQL generiert, welches dann auf dem Data-Warehouse ausgeführt werden kann. Ein solches System ermöglicht es Geschäftsleuten jederzeit beliebige Abfragen auf komplizierten Data-Warehouse-Systemen auszuführen.


Diese Doktorarbeit präsentiert eine neue, auf Hauptspeicher basierende Metho-
de, welche einen fertigen Index nicht mehr anpasst, sondern so viele neue Indexte erstellt wie nur möglich. Solange genug schnell genug viele neue Indexte erstellt werden, entsteht eine Illusion eines immer aktuellen Indexes. Wir zeigen, dass - überraschenderweise - sogar für große Datenmengen das Erstellen eines Indexes unterhalb einer Sekunde bleibt.

In unserem Prototypen sind Abfragen schnell, weil die Indexte speziell für den Lesezugriff optimiert werden können. Anpassungen sind ebenfalls sehr schnell, weil diese einfach gesammelt werden können, bevor ein neuer Index aufgebaut wird, der die neuen Anpassungen dann enthalten wird. Einen neuen Index aufzubauen kann ablaufen, ohne die Abfragen oder die Anpassungen zu behindern.
Acknowledgments

First of all, I would like to thank my advisor, Prof. Donald Kossmann. He not only has crazy ideas and works at the strangest times of day, but he is also a very nice person who understands that there is more to life than only work.

Furthermore, I would like to thank my examination committee, Prof. Gustavo Alonso, Prof. Karl Aberer, and Dr. Kurt Stockinger. They provided me with invaluable feedback towards the end of my thesis. I also thank Prof. Markus Gross for chairing my examination.

A big thank you goes to my former advisor Jens Dittrich and my former co-worker Marcos vaz Salles. Together we developed several highly-innovative systems and had a good time.

When I was here at ETH, I enjoyed the open atmosphere and perfect infrastructure of this university. I highly recommend it for new PhD students.

I would also like to thank the people in the Systems Group for their work-related inputs and for the fun outside work. Especially our ski trips together are a wonderful memory.

Last, I would like to thank my parents, Diana and Gallus, and my sister, Andrea, for their continuous support and motivation.
# Contents

## 1 Introduction

1.1 Motivation ................................................. 1  
   1.1.1 Keyword Search on Data Warehouses ................. 1  
   1.1.2 Moving Objects Indexing .............................. 2  
1.2 Contributions ............................................. 3  
   1.2.1 Keyword Search on Data Warehouses ................. 3  
   1.2.2 Moving Objects Indexing .............................. 4  
1.3 Structure of this Thesis ................................. 5

## I Keyword Search on Data Warehouses

## 2 Introduction

2.1 Contributions ............................................. 10

## 3 SODA

3.1 Introduction ............................................. 11  
   3.1.1 Problem Statement .................................. 11  
   3.1.2 SODA Overview .................................... 12  
3.2 Running Example .......................................... 14  
   3.2.1 Example Schema .................................... 14  
3.3 Extended Metadata Graph ................................. 16  
   3.3.1 Integrated Schema .................................. 16  
   3.3.2 Domain Ontologies .................................. 16  
   3.3.3 DBpedia ............................................ 19  
   3.3.4 Base Data .......................................... 19  
3.4 SODA in a Nutshell ....................................... 19  
   3.4.1 Step 1 - Lookup .................................... 19  
   3.4.2 Step 2 - Rank and top N ............................. 20  
   3.4.3 Step 3 - Tables .................................... 21  
   3.4.4 Step 4 - Filters .................................... 21
II Moving Objects Indexing

8 Introduction

9 MOVIES
  9.1 Preliminaries ................................................. 61
    9.1.1 Problem Statement .................................. 61
    9.1.2 Formal Argument ................................... 62
  9.2 The MOVIES Algorithm ..................................... 64
    9.2.1 Algorithmic Walkthrough ............................ 64
    9.2.2 Comparison to Differential Files .................... 65
    9.2.3 MOVIES Core Algorithm ............................... 67
  9.3 Update Buffers ............................................. 67
    9.3.1 Logging or Aggregation? ............................. 67
    9.3.2 Data Entry Layout ................................. 69
  9.4 Read-Only Indexes ......................................... 70
    9.4.1 Spatial Indexing Method ............................. 70
    9.4.2 Index Layout ....................................... 71
    9.4.3 MOVIES buildIndex algorithm ....................... 72

10 Tuning Spatial Indexing ................................ 75
  10.1 Linearized kd-tries .................................... 75
    10.1.1 Z-Codes ........................................ 76
  10.2 Partial In-Place Location Codes ....................... 76
    10.2.1 Precomputed Snippets ............................. 77
    10.2.2 Reversing Partial Z-Codes ......................... 78
  10.3 Experiments .............................................. 78
    10.3.1 Partial In-Place Z-Codes Computation ............. 78
    10.3.2 Sorting Depending on Z-Code Precision ............ 79

11 Quality-Aware Range Query Processing .................. 81
  11.1 Standard Range Query Processing ................. 81
  11.2 Quality-Aware Range Query Partitioning .......... 84
  11.3 Experiments ........................................... 85
    11.3.1 Quality-Aware Query Partitioning ............... 86
    11.3.2 Query-Method Comparison ........................ 86
    11.3.3 Query-Method Scalability ......................... 89

12 Time-Parameterized Query Processing ................... 91
  12.1 Predictive MOVIES ..................................... 91
    12.1.1 Indexing Strategy ............................... 91
# CONTENTS

12.1.2 Query Strategy ........................................... 92
12.1.3 Choice of Index Time .................................... 92

12.2 Non-Predictive MOVIES ..................................... 93
12.2.1 Indexing Strategy .......................................... 93
12.2.2 Query Strategy ............................................ 93

13 MOVIES Parallelization ....................................... 95
13.1 Intra-node Parallelization .................................. 95
13.1.1 General Trade-Off ......................................... 95
13.1.2 Parallel Update and Index Merge ......................... 96
13.1.3 Parallel Sorting ........................................... 96
13.2 Inter-node Parallelization ................................... 96
13.2.1 OID-partitioning .......................................... 96
13.2.2 Spatial Partitioning ....................................... 96

14 Experiments ................................................... 99
14.1 Experimental Setup .......................................... 99
14.1.1 Hardware and Implementation Parameters ............... 100
14.1.2 Data and Queries .......................................... 100
14.2 MOVIES Experiments ......................................... 101
14.2.1 Scalability in Index Size .................................. 101
14.2.2 Scalability in Update Rate ................................. 103
14.2.3 Scalability in Update Rate and Index Size .............. 105
14.2.4 Main Memory Consumption ................................. 107
14.3 Shared-Nothing Scale-Out ................................... 109
14.3.1 Maximum Update Rate ...................................... 109
14.3.2 Shared-Nothing Scalability in Index Size ............... 111

15 Possible Extensions of MOVIES .............................. 113
15.1 Algorithmic Extensions ..................................... 113
15.1.1 Fuzzy Aggregated MOVIES ............................... 113
15.1.2 Velocity-Dependent MOVIES .............................. 114
15.1.3 Adaptive Index Rebuild .................................... 114
15.1.4 Keeping Multiple Index Frames ........................... 114
15.1.5 Supporting Insertions and Deletions ...................... 114
15.2 Indexes ..................................................... 115
15.2.1 Index Compression ......................................... 115
15.2.2 Cache-Optimized Indexes .................................. 115
15.2.3 Extension to High-Dimensional Data ..................... 115
15.3 Recovery ..................................................... 116
15.3.1 Recover by Wait ............................................ 116
15.3.2 Recover by System-based Redundancy .......................... 117
15.3.3 Log-based Redundancy ........................................ 117

16 Related Work ................................................................ 119
  16.1 With Time-Parameterized Queries .............................. 119
    16.1.1 External Memory ............................................. 119
    16.1.2 Main Memory .................................................. 120
  16.2 Without Time-Parameterized Queries ......................... 120
    16.2.1 Main Memory .................................................. 120
  16.3 Extensions for Efficient Updates .............................. 121
    16.3.1 External Memory ............................................. 121
    16.3.2 Main Memory .................................................. 121
  16.4 Experimental Studies ............................................. 121

17 Conclusions .................................................................. 123

18 Summary and Conclusions ............................................. 125
  18.1 Summary .............................................................. 125
  18.2 Ongoing and Future Work ........................................ 126
Chapter 1

Introduction

In this thesis we look at two problems from the data analysis domain: Keyword search on data warehouses and moving objects indexing.

Keyword search on data warehouses is difficult, because the returned results are not documents—as usual in the text search domain, but rather a complex combination of small information tuples. To find and connect these separate information tuples, we make use of the semantic of the data. That is, we use metadata to understand the data and then create SQL queries to generate the results. We will show how semantics can be used in a data warehouse and we will present an algorithm to process extended keyword search.

Moving objects indexing is difficult, because a high number of moving objects change their position all the time. All these positional changes need to be stored and possibly indexed to provide querying capabilities for the moving objects. In the literature, a number of sophisticated data structures have been presented. Our technique differs from these approaches by applying an old, well-known technique to process large amounts of updates: Delta Indexing. In the course of investigating the moving objects problem, we will introduce staleness of query results. Furthermore, we will show a plethora of small algorithmic improvements for indexing moving objects.

1.1 Motivation

1.1.1 Keyword Search on Data Warehouses

Querying large data warehouses is very hard for non tech-savvy business users. Deep technical knowledge of both SQL as well as the schema of the database is required in order to build correct queries and to come up with new business insights. Instead of directly writing SQL queries, a typical business analyst requires
a way of exploring the data warehouse and creating the queries while doing so. Extended keyword search is one option of such an explorative tool.

Implementing extended keyword search in a data warehouse is, unfortunately, not that easy. Whereas regular keyword search deals with large corpora of text content, the data in a relational data warehouse is structured in tables, rows, and columns. Interestingly, this additional structure makes keyword search harder, because we not only want to return a single row where a keyword appears, but we want to capture the meaning of a keyword query and translate it into a SQL query that returns all the data related to it. To approach this problem, we will use different types of metadata, such as domain ontologies and relational schemas at different granularities.

A typical keyword query and its SQL counterpart is shown in Query 1.

<table>
<thead>
<tr>
<th>Keyword query:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rich customers zürich</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM customers, assets, addresses WHERE assets.customer = customers.id AND customers.livingAt = addresses.id AND assets.balance &gt; 50,000 AND addresses.city = 'Zürich'</td>
</tr>
</tbody>
</table>

**Query 1**: Keyword query example: Keywords vs. SQL.

### 1.1.2 Moving Objects Indexing

Indexing data that changes continuously is difficult, because one either gets high query performance or high update performance. As shown in Figure 1.1, to get high query performance, we have to create indexes. On the other hand, to get high update performance, we have to remove all possible indexes and make persistency as cheap as possible, for example, by using a log to store the changes on persistent storage. To avoid this dilemma, we introduce a third dimension: staleness.

The staleness of a query result is the time since when incoming updates are no longer considered in the result. Zero-staleness is achieved if all updates known to the system are considered in the result. On the other hand, staleness increases, the longer updates are not considered in the result.
1.2. CONTRIBUTIONS

To implement staleness, we make use of a well known technique called Differential Files [70]—more recently called Delta Indexing. Staleness allows us to build an index for querying the data without having to consider the latest incoming updates. Because of the indexes, we get good query performance and because updates do not have to be indexed immediately, we still support high update rates.

1.2 Contributions

1.2.1 Keyword Search on Data Warehouses

Our main contribution on keyword search on data warehouses is the design and implementation of SODA (Search Over DAta Warehouse), a system that takes a business analyst query and a metadata graph as input and generates a SQL query as output. SODA resembles a Web search engine and actively tries to engage the user into the process of finding the right query and result.

We define a simple domain specific language (DSL) to query a data warehouse. This DSL is based on keywords to express values from the base data and dedicated operators such as range predicates, sum, group-by, and top N to specify filters and aggregates.
CHAPTER 1. INTRODUCTION

We present a graph pattern matching algorithm which uses the input query and the metadata of the data warehouse to find answers to business questions. Both, the input query and the metadata of the data warehouse, are modelled as graphs. Computing the result then becomes a graph matching problem.

We show graph patterns which are used to query the Credit Suisse data warehouse. Our graph patterns are flexible enough such that SODA can evolve over time—adapting to the needs of the company. Furthermore, the patterns are general enough such that SODA is not only limited to the financial industry domain, but is applicable in all kinds of domains.

We report on experiments and experiences with a real-life, hundreds of terabyte data warehouse with extremely complex schemas. These experiments assess the sensitivity of the system with regard to different ways to ask the same query and different ways to model the metadata.

Finally, we also implemented a visual user interface that explains the lineage of query results and allows a user to select the most appropriate result.

1.2.2 Moving Objects Indexing

Our main contribution on this problem is a novel approach to effectively index moving objects. This approach provides at all times a read-optimized index which does not suffer from update handling. We call our approach MOVIES (MOVing objects Indexing using frEquent Snapshots), because the key idea of our approach is to use short-lived index snapshots.

We show how to scale MOVIES to multiple cores and multiple nodes and we run a thorough experimental study of MOVIES using standard hardware and realistic data sets unmatched by previous work.

Furthermore, we show how to implement a main-memory moving object index to support time-parameterized queries. Previous main-memory methods did not support this type of query.

We also introduce techniques that allow us to trade index rebuild time for query response time in a very fine-granular manner. As a consequence, we may easily adapt our method to an application’s QoS-requirements, for example, its desired query rate and query result staleness. A key technique for this is partial in-place z-codes.

Finally, we present techniques to make update handling efficient. As we collect incoming updates in a buffer, the cost for collecting updates is very small. We will show that different buffer organizations have different impact on the overall performance of MOVIES.
1.3 Structure of this Thesis

In the first part of this thesis, we look at keyword search on data warehouses. Chapter 2 gives an introduction into keyword search on data warehouses and highlights our main contributions in Section 2.1. Chapter 3 describes the problem and gives an overview of the idea of SODA in Section 3.1. Section 3.2 describes the running example used to describe SODA and Section 3.3 introduces the data structures. An overview of the SODA approach is given in Section 3.4. Section 3.5 shows the user interface. Chapter 4 defines the algorithms and patterns in more detail and presents several example queries. Section 4.1 explains why and where we use patterns in SODA. Metadata graph patterns and input patterns are introduced in Section 4.2 and Section 4.3 respectively. Our query language is formalized in Section 4.4. Section 4.5 gives several examples. Chapter 5 discusses the results of comprehensive experiments that evaluate the quality of the results produced by SODA using both precision and recall as metrics. Chapter 6 evaluates related work and compares SODA against several prominent systems. Chapter 7 contains conclusions of the first part of this thesis and outlines possible avenues for future work.

In the second part of this thesis, we explore moving objects indexing. In Chapter 8 we introduce moving objects indexing. Chapter 9 starts by defining the problem and is followed by a formal argument supporting the idea of staleness in Section 9.1. The MOVIES algorithm is shown in Section 9.2. Section 9.3 and Section 9.4 introduce our update buffers and read-only indexes. In Chapter 10 we introduce a method to trade query performance for indexing time called partial in-place z-codes. Quality-aware range query processing is explained in Chapter 11. Time-parameterized query processing is explained in Chapter 12. Chapter 13 presents avenues for parallelization within a single node and also between different machines. Our extended experimental analysis is covered in Chapter 14. We finish the second part with possible extensions of MOVIES in Chapter 15, related work in Chapter 16 and a conclusion in Chapter 17.
Part I

Keyword Search on Data Warehouses
Chapter 2

Introduction

The purpose of data warehouses is to enable business analysts to make better decisions. Over the years the technology has matured and data warehouses have become extremely successful. As a consequence, more and more data has been added to the data warehouses and their schemas have become increasingly complex. These systems still work great in order to generate pre-canned reports. However, with their current complexity, they tend to be a poor match for non tech-savvy business analysts who need answers to ad-hoc queries that were not anticipated.

The traditional way of querying a data warehouse is to write SQL statements—possibly with the help of tools that offer available table names and/or column names. The current complexity of data warehouses makes this impossible for non-IT personal. First, because of the missing technical knowledge of SQL and second, because of the complexity of the data warehouse schemas.

Keyword search offers a much simpler approach to query a data warehouse. In a Google-like fashion, it would be possible to look for interesting facts in the data warehouse. Once these facts have been found, additional support to connect and aggregate these facts could be provided. Usually, keyword search is applied to large, text-mostly corpora. In such a scenario, the known information retrieval techniques [53] work perfectly well. Unfortunately, the fine granular structure of a data warehouse makes keyword search much more difficult. We not only want a single cell of a relational table as a result, but also all the data connected to that piece of information. E.g. the data values in the same row or rows which join with this row from other tables.

The first systems to support keyword search in relational databases were DB-Explorer [1], DISCOVER [32], and BANKS [6]. The key idea of these systems was to build an inverted index on the base data and to consider key/foreign key relationships when building query results. The inverted index is used to find all occurrences of the keywords of a query in tuples of the database. The key/foreign key relationships are used to compute join paths to construct business objects from
the tuples that match different keywords of the query. Based on this promising, initial work, a number of more sophisticated systems have been developed in the recent past. We present them in Chapter 6.

2.1 Contributions

The main contributions of our work are as follows:

a) We show how SODA can be used to generate SQL queries from a high-level query language (keywords and operators), metadata, and patterns. The key innovation is the use of patterns that help to interpret and exploit a large variety of different kinds of metadata such as homonyms and synonyms (e.g., using DBPedia), domain ontologies (Credit Suisse has its own domain ontology and imports standards from the financial industry), modeling conventions (e.g., inheritance), performance tricks (e.g., partitioning, redundancy), and last but not least base data.

b) We present the results of experiments using a real-life data warehouse with hundreds of tables and thousands of attributes from a major player in the financial industry. The experiments show that indeed complex SQL queries can be generated automatically from high-level specifications and that the gap between the world of business users and the reality of IT can be bridged. The experiments, however, also demonstrate some of the limitations of the approach and that a search tool like SODA needs to evolve based on user feedback and experience.
Chapter 3

SODA

This chapter describes the design of the SODA system (Search over DAta Warehouse) [9]. SODA bridges the gap between the business needs of analysts and the technical complexity of current data warehouses. SODA enables a Google-like search experience for data warehouses by taking keyword queries of business users and automatically generating executable SQL. The key idea is to use a graph pattern matching algorithm that uses the metadata model of the data warehouse. Our results with real data from a global player in the financial services industry show that SODA produces queries with high precision and recall, and makes it much easier for business users to interactively explore highly-complex data warehouses.

3.1 Introduction

3.1.1 Problem Statement

Modern data warehouses have grown dramatically in complexity over the last decades. In particular, the schemas of data warehouses have become increasingly complex with hundreds of tables and ten thousands of attributes for many organizations. In part, this growth in complexity has been the result of the large success of data warehousing in many organizations. Data warehouses are used for an increasing number of applications and these applications have evolved over time. Each new application and most evolutionary steps involve extending the schema in order to fiddle in the new information requirements of, say, the new application.

A second observation that can be made in modern data warehouses is that there is a growing gap between the high-level (conceptual) view of business users and the low-level (physical) perspective of database administrators. Business users still think of the data in star schemas with fact tables in the center and dimension tables as satellites [40]. Database administrators need to integrate many such star
schemas of different kinds of business users with varying information needs into a single physical schema. Their job is to optimize the data warehouse, thereby minimizing cost (i.e., \$) and meeting all performance goals (i.e., response time and throughput). At the same time, they must manage the data and the schema.

Given these differences in goals, it is not surprising that the conceptual world of business users and the physical world of database administrators is very different. For instance, database administrators may implement a simple business concept such as Customer using many different tables, thereby partitioning the data horizontally and vertically. Furthermore, database administrators may store information from different business entities in a single table if that helps improve performance or manageability. Database administrators may also implement inheritance and generalization in different ways, depending on the query workload that they anticipate. As an extreme example, database administrators may use cryptic naming schemes for table and column names, thereby helping them with certain administration tasks. What makes matters worse is that the schemas of data warehouses have already evolved for several decades and different conventions and optimizations have been applied in each generation.

In regular, every-day operations, this gap does not become apparent. The information needs of business users are typically fulfilled with the help of pre-defined reports using pre-canned queries. These pre-canned queries specify exactly how to reconstruct the business concepts (e.g., revenue of a customer) from the physical database schema. While these reports work well for periodically recurring information needs of business users, the gap between the business and IT world becomes problematic if business users want to ask ad-hoc queries or if new reports need to be generated for an optimized business process or to launch a new product. In such an event, business users and database administrators must work together and it often takes days or weeks before both groups of users have found a way to implement such a new report—given that all the information is already in the data warehouse, otherwise it could even take longer.

3.1.2 SODA Overview

In order to support a more agile usage of a data warehouse, new search tools are required. Ideally, a business user asks a query using operators and the business concepts of her world and the search tool automatically translates these concepts into SQL queries that are executable in the current version of the data warehouse. Typical queries might be: Show me all my wealthy customers who live in Zurich. Who are my top ten customers in terms of revenue? In such queries, wealthy customers is a business concept that is defined by, say, the salary of a customer. top ten is an operator and applied to customer it asks for the customers with the highest trading volume.
3.1. INTRODUCTION

SODA addresses this need of business users by allowing them to pose queries in an intuitive, high-level language based on keywords, operators and values. SODA translates these queries into a set of alternative SQL queries, ranks those queries, and (partially) executes the Top 10 in order to generate result snippets (up to twenty tuples) for each of these queries. Just as in a Web search with Google or Bing, the user has now the choice to select one of those queries of the first result page, ask for the next set of candidate queries (i.e., the next result page), or refine the original query.

Translating keywords into SQL has been studied before in related work such as BANKS [6], DISCOVER [32], DBExplorer [1], SQAK [82] or Keymantic [5]. Like most of these systems, SODA indexes the base data and finds join paths using key/foreign key relationships of the database schema. The key innovation of SODA is its flexible way of making use of metadata that goes way beyond looking at key/foreign key relationships or lookups on column names and table names. SODA allows to define metadata patterns that specify how the database schema implements the conceptual model that the business user might have in mind. For instance, at Credit Suisse customer information is spread across several tables; different kinds of customers (e.g., organizations, wealthy customers in private equity, customers with special compliance constraints and risk profiles) are implemented in different ways. The metadata allows to bridge the gap between the low-level SQL implementation and the concepts typically used by business users and allows to generate the right SQL for a complex query.

To enable SODA (and other related tools), Credit Suisse has invested heavily in building a so-called metadata warehouse [37]. Such a metadata warehouse stores all available metadata. [37] shows, for instance, how provenance information of the metadata warehouse can be used in order to find out which applications are affected by a change in a specific data source. As will be shown, SODA exploits the definition of business terms (e.g., wealthy customer), homonyms and synonyms (e.g., information extracted from DBPedia), and data models at different levels to help business users to ask complex queries to the data warehouse. Given the growing gap between business and IT, we are aware of several other organizations that are also investing into such metadata warehouses. Obviously, these metadata warehouses will have totally different structures and model metadata in different ways. By using patterns, however, SODA is flexible and generic enough to exploit any kind of metadata. Furthermore, SODA can evolve over time thereby adapting to new patterns based on user feedback or to an evolving data warehouse.
3.2 Running Example

This section describes the simplified schema of a mini-bank with customers that buy and sell banking products (so-called financial instruments). We use this example throughout this work to demonstrate that generating SQL queries that meet a business need can be difficult for humans even for a small schema. The example, however, also illustrates that if the metadata is known and the right patterns can be extracted from the query, then generating the right SQL is quite doable for a machine.

Typical end user queries that we will analyze are as follows: (1) Find all financial instruments of customers in Zurich. (2) What is the total trading volume over the last months? (3) What is the address of Sara Guttinger?

3.2.1 Example Schema

Figure 3.1 shows the example schema. It models information about customers (referred to as Parties in our data warehouse) and the transactions these customers made; buying and selling on the stock market. Parties can be individuals for private banking or corporate customers for investment banking (i.e., organizations); both kinds of parties are modeled separately because they are supported by different sets of analysts. The technical term for products which can be bought or sold on the stock market is Financial Instrument. Financial instruments can be shares of a company (e.g., IBM shares). Financial instruments, however, can also be structured; that is, a financial instrument could relate to a fund that manages a portfolio of shares or even a hedge fund that manages a portfolio of certificates of
other funds and hedge funds. It is in part this recursive nature of financial instruments that makes it difficult for business analysts to extract the right information from a data warehouse.

As mentioned in the introduction, real-world data warehouses are far more complex. In a real data warehouse, schemas are layered with different levels of abstraction. Figure 3.1 is a dramatic simplification, which could be at the conceptual layer at which business analysts and architects meet in order to design a new report. At lower layers (i.e., logical and physical layer), the schemas become more complex as the system architects and database administrators refine the schemas in order to achieve better performance by partitioning and replicating data, improve data quality by modeling data at different granularities, etc. Figure 3.2 shows the example schema at the logical level. Here, the addresses of individuals are actually stored in a separate table and transactions are modeled as either financial instrument transactions or money transactions.

It is easy to imagine how complex such schemas can get in a global financial institution considering varying regulatory requirements of different countries, redundancy that arises from keeping data in different granularities for performance reasons, the heterogeneity of the data sources that feed the data into the data warehouse, and with different departments asking for different kinds of reports.
3.3 Extended Metadata Graph

The data warehouse of Credit Suisse consists of base data stored in a relational database as well as metadata stored in a graph structure (such as RDF). The metadata consists of the database schema extended with DBpedia and domain ontologies (see Figure 3.3).

3.3.1 Integrated Schema

A data warehouse combines and aggregates data from many heterogeneous data sources. To handle the differences in the data sources, an integrated schema is built. To facilitate the design process, there are different levels of the schema, namely conceptual, logical and physical. The conceptual schema (business layer) serves for communication with business and contains the main entities to be modeled such as parties, transactions, and securities. The logical schema extends the conceptual one by showing inheritance, splitting entities (for instance, parties are split into individuals and organizations), etc. The physical schema contains information about database indexes or table partitioning. Typically all these schemas are designed with one modeling tool with the goal to generate the physical tables. An example of a conceptual schema as well as a logical schema is given in Figure 3.1 and Figure 3.2 respectively.

3.3.2 Domain Ontologies

In addition to the schema, our metadata consists of several domain ontologies. The domain ontologies are built by domain experts to classify data for a specific domain. These domain ontologies may be adapted to a given data warehouse, but in principal can be reused for any other data warehouses in a similar domain.

As an example, such a domain ontology could classify financial instruments or customers. At Credit Suisse, customers are divided into private and corporate customers: Private customers are implemented using an Individuals table; the information of corporate customers is stored in an Organizations table. See Figure 3.4.

Another example from our example world which classifies securities is shown in Figure 3.5. In this example, securities are divided into high tech or financial services. The values which are attached to the leaf level of a domain ontology define a filter value on an conceptual attribute. The outgoing link of a leaf node connects the domain ontology to this attribute.
Figure 3.3: Metadata Graph and Relational Data.
CHAPTER 3. SODA

Figure 3.4: Customers Domain Ontology.

Figure 3.5: Securities Domain Ontology.
3.3.3 DBpedia

The metadata also contains data from DBpedia in order to capture synonyms. Credit Suisse only maintains DBpedia entries that have direct connections to the terms stored in the integrated schema of the data warehouse. For instance, for the term “Parties” shown in our example world, the following entries have been extracted from DBpedia: customer, client, political organization, etc. As a result, when a user searches for customers then, parties would be one possible answer.

Since DBpedia is itself a graph, we could make use of more than looking for keywords which are directly connected to our metadata graph. We could, for example, look for keywords which are two or three hops away. Experiments showed, however, that by increasing the allowed distance, the number of connections increases as sharply as the quality of the results goes down. If at all, we currently only use DBpedia nodes which are connected directly to our existing metadata.

3.3.4 Base Data

As in most large-scale data warehouses, the base data of the Credit Suisse data warehouse is stored in relational databases. All the base data is implicitly connected to the metadata by the table and column names of the physical schema.

3.4 SODA in a Nutshell

In this section we will give a short explanation of the SODA approach. A more technical explanation will follow in the next chapter.

Figure 3.6 shows the main steps of the SODA approach. These steps are similar to the way systems like BANKS, DBExplorer, and DISCOVER generate SQL queries. Again, the magic of SODA lies in the use of metadata and patterns (described in Chapter 4). Starting from a list of keywords and operators, SODA computes a ranked list of executable SQL statements that are likely to meet the information needs of the user. This transformation is carried out in the following five steps:

3.4.1 Step 1 - Lookup

The lookup step matches the keywords of the input query to sets of possible entry points. A lookup of a single keyword provides us with all the nodes in the metadata graph where this keyword is found. For example, in Figure 3.7, the keywords “customers” and “Zürich” are both found once, in the domain ontology and the base data, respectively. On the other hand, the keyword “financial instruments” is found twice: once in the conceptual schema and once in the logical schema. The
output of the lookup step is a combinatorial product of all lookup terms. For this example two solutions are produced: One where “financial instruments” is found in the conceptual schema, and another one where “financial instruments” is in the logical schema. See Figure 3.8. Besides processing keywords, our algorithm also uses operator constructs (patterns) to express aggregation and filters.

3.4.2 Step 2 - Rank and top N

The next step assigns a score to every result and continues with the best N results. For the ranking, we currently apply a simple heuristic which uses the location of the entry points in the metadata graph to assign a score to a result. For example, a keyword which was found in DBpedia gets a lower score than a keyword which was found in the domain ontology. We rank the domain ontology higher, because it was built by domain experts from the financial services industry, and hence it is more likely to match the intent of our business users than the general terms found in DBpedia. There exist certainly more sophisticated ranking algorithms such as BLINKS [28], however, ranking is only a part of SODA and not the main focus of this work.
3.4. SODA IN A NUTSHELL

<table>
<thead>
<tr>
<th>Input</th>
<th>customers Zürich financial instruments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>

**Solution 1:**
- Customers (Domain ontology)
- Zürich (Basedata)
- Financial Instruments (Conceptual schema)

**Solution 2:**
- Customers (Domain ontology)
- Zürich (Basedata)
- Financial Instruments (Logical schema)

Figure 3.7: Query Classification.

Figure 3.8: Output of Lookup Step.

### 3.4.3 Step 3 - Tables

The purpose of this phase is to identify all the tables which are used in each solution and to discover the relationships between these tables. Starting at every entry point which we discovered in the lookup phase, we recursively follow all the outgoing edges in the metadata graph. At every node we test a set of graph patterns to find tables and joins. We assume, that tables found in this way, represent the entry points. In our example the output of this step are 7 tables (see Figure 3.9).

### 3.4.4 Step 4 - Filters

Filters can be found in two ways: a) by parsing the input query or b) by looking for filter conditions while traversing the metadata graph. In this step, we add the filters to the discovered tables and columns of the previous step. A filter condition consists of a column and a value such as “Zürich”. In our example, the filter conditions are used to connect “Zürich” to the city column within the addresses table. While having filters in the input query is quite common, filters stored in the metadata can be very powerful as well. An example of a filter stored in the
metadata would be “wealthy individuals” as described in the introduction.

3.4.5 Step 5 - SQL

In this final step, we take all the information that was collected earlier and combine it into reasonable, executable SQL statements. By “reasonable” statements we mean statements which take into account possible join patterns. For example, considering foreign keys and inheritance patterns in the schema. By “executable” statements we mean SQL statements that can be executed on the data warehouse.

3.5 User Interface

This section shows how the user interface looks like and explains by what means a user interacts with the system.

Figure 3.10 shows the start page of our system. Since the nature of the problem is inherently imprecise, we decided to adapt a Google-style way of interaction where the user types a query, looks at the results and depending on the results, changes the query or continues with one or several of the resulting query graphs. On the start page the user has two options: either the user enters a query and looks at the final results directly (I’m Feeling Lucky) or he walks through the algorithm step-by-step. In every step, the user may decide to continue with a selected subset of all query graphs.

Figure 3.11 shows the I’m Feeling Lucky page for the query “Find all high tech securities”. The first part of the output shows the classification of the query. Below this, the top-N query graphs are displayed. For each query graph there are buttons to show the produced N-triples and a visual representation of the query.
3.5. USER INTERFACE

Figure 3.10: Google-style Start Page of SODA.

top-20 result rows are displayed.

If a user would like to know how the query lineage of a specified entry point looks like, he may follow a link to the lineage page which is shown in figure 3.12. This figure shows all paths starting at the “high tech” entry point in the customers domain ontology and going to a logical attribute.

A prototype of SODA using the sample world presented in Section 3.2 is running at http://lukasblunschi.ch/soda and was presented in a demo at CIKM 2011 in Glasgow [8].
Figure 3.11: I’m Feeling Lucky Page of “Find all high tech securities”.
Figure 3.12: Lineage Page of the “high tech” Entry Point.
Chapter 4

Generating SQL from Patterns

This chapter provides details on the patterns and algorithms used in SODA. In particular, it shows how patterns are used to translate from a keyword-based input query to full-fledged SQL. Metadata graph patterns provide a flexible way to adapt the SODA algorithm to different data warehouses.

It is important to note that patterns described in the following probably exist in all data warehouses, but the structure depends on the modeling of the data warehouse. Here, we describe the patterns that we used for Credit Suisse. To port SODA to a different data warehouse involves adjusting the patterns to the specific structures used in that data warehouse.

4.1 Why Patterns?

SODA uses patterns in two situations:

1) In Step 1 - Lookup (Figure 3.6): Instead of trying to use natural language processing to understand the input, we have a set of so-called input patterns that SODA understands. For example, every operator is a little pattern which combines an operation (comparison or aggregation) with values or business entities.

2) In Step 3 - Tables and Step 4 - Filters (Figure 3.6): When deriving the tables and join conditions from a given set of so-called entry points (graph nodes which represent words of the input query), SODA tests for so-called metadata graph pattern matches while traversing the metadata graph. A matching pattern tells us when we arrived at a special node which could be, for instance, a table, a foreign key or an attribute with a filter condition.

Both types of patterns, input patterns and metadata graph patterns, can be adapted to work for the given application. For example, use different input operators for another set of users or modify the metadata graph patterns according to the existing schema structure. While the patterns may have to be changed
between different applications, the algorithm always stays the same. Currently, input patterns are only keywords extended with a small set of operators, whereas our metadata graph patterns can match the complexity of the Credit Suisse metadata warehouse [37].

### 4.2 Metadata Graph Patterns

In *Step 3 - Tables and Step 4 - Filters* (see Figure 3.6), we use metadata graph patterns to discover tables, joins and filters stored in the metadata.

#### 4.2.1 Pattern Descriptions

To define the patterns, SODA uses a language inspired by SPARQL [75] filter expressions: Each triple either connects two nodes or connects a node with a text label.

\[
\begin{align*}
 & ( \text{node} \text{ edge} \text{ node} ) \\
 & ( \text{node} \text{ edge} \text{ text} )
\end{align*}
\]

A node is either a static URI or a variable. Variables can be assigned any URI, but within one match, a variable keeps its URI. An edge is a static URI. A text label is simply a string. In the following, we will use italic, dark gray font for variables, put \text{t:} before text labels, and remove URI prefixes for brevity.

To match a pattern on a given graph, we assign the variable \(x\) to the current node and try to match each triple in the pattern to the graph accordingly.

#### 4.2.2 Basic Patterns

These patterns describe how tables, columns, etc. are represented in the metadata graph. SODA matches these patterns against the metadata graph to identify the tables and columns which participate in each result. Basic patterns are used at the beginning of *Step 3 - Tables* of the SODA algorithm (see Figure 3.6).

The Table pattern can be written like this:

\[
\begin{align*}
 & (x \text{ table name } \text{t:}y) \\
 & (x \text{ type } \text{physical_table})
\end{align*}
\]

This pattern matches, if the current node \(x\) has a \text{table name} attribute pointing to a text label \(t:y\). In addition, \(x\) needs to have a \text{type} attribute pointing to a node which has the URI \text{physical_table}. See Figure 4.1 for a graphical representation of these conditions.

The Column pattern could be:
4.2. METADATA GRAPH PATTERNS

The first part of the column pattern specification is similar to the table pattern. In the last line of the column pattern specification, we ensure that each column ($x$ in this example) has an incoming column edge from another node ($z$).

4.2.3 More Complex Patterns

These patterns define join relationships and inheritance structures.

The simplest implementation of a join relationship is a direct edge between a foreign key attribute and a primary key attribute. This is shown in the Foreign Key pattern:

\[
(z \text{ foreign_key } y) \& (z \text{ matches-column }) \& (y \text{ matches-column })
\]

The term “matches-column” references the Column pattern described above. Figure 4.2 shows a visual representation of this pattern. In the case of Credit Suisse, we use a more general Join-Relationship pattern which has an explicit join node with outgoing edges to primary key and foreign key.

For testing a node if it is a child in a inheritance structure, we use the Inheritance Child pattern:

\[
(y \text{ inheritance_child } x) \& (y \text{ type inheritance_node }) \& (y \text{ inheritance_parent } p) \& (y \text{ inheritance_child } c1) \& (y \text{ inheritance_child } c2)
\]
Here, $x$ needs to have an incoming \texttt{inheritance\_child} edge from an explicit inheritance node ($y$). The inheritance node, has to be of type \texttt{inheritance\_node} and needs to have three outgoing edges: \texttt{inheritance\_parent} to the inheritance parent, and two \texttt{inheritance\_child} edges to the inheritance children.

### 4.2.4 Application in SODA

The metadata graph patterns described so far are all used in \textit{Step 3 - Tables} of our algorithm (see Figure 3.6). We traverse the metadata graph starting from the entry points of a given query and recursively follow all outgoing edges. At each node, we test the Table, Column and Inheritance Child patterns. If the Table pattern matches, we collect the corresponding table name. If the Column pattern matches, we collect the corresponding column name as well as the table name. And if the Inheritance Child pattern matches, we collect the table name of the inheritance parent. We need to collect the table name of the inheritance parent because this table is needed to produce correct SQL statements.

After this first part of step 3 in the algorithm, we now have the table names for all given entry points. What remains to do in the second part of step 3, is to identify \textit{joints} that are needed to properly connect the tables. Fortunately, a similar approach as for the table names can be used: We again traverse the metadata graph starting from the entry points of a given query and recursively follow all outgoing edges. But, instead of testing the Table, Column and Inheritance Child patterns as before, we now try to match the Foreign Key pattern (or, in the case of Credit Suisse, the Join-Relationship pattern). Of all the join conditions we discovered in
this way, we now use these which are on a direct path between the entry points.
Join conditions which are only “attached” to such a path are ignored to keep the
result small and precise. A user interface, however, could make such joins available
to the user. See Figure 4.3.

4.2.5 Bridge Tables in Large Schemas
Joining the entry points as described above works fine in small data warehouses
with simple schemas, i.e. in our example world, this works well. In a large data
warehouse, as for example the Credit Suisse data warehouse, this is not enough.
In a last part of step 3 of our algorithm, we therefore look for bridge tables,
i.e. physical implementations of N-to-N relationships. Bridge tables connect two
entities by having two outgoing foreign keys. If we find a bridge table between two
of our entry points, we use it to add additional join conditions. Bridge tables—as
you might have guessed—can be described with a pattern and identifying bridge
tables therefore works similar to identifying tables and joins.

4.3 Input Patterns
Input patterns are used in Step 1 - Lookup of the SODA algorithm (see Figure 3.6).
These patterns are matched against the query terms to identify their meaning.
Input patterns allow SODA to refine the input to produce more complex queries.

4.3.1 Keywords
The first type of input patterns are keywords. To process keyword-only inputs, we
look for longest word combinations. We first try to match all the words in the
CHAPTER 4. GENERATING SQL FROM PATTERNS

input against our classification index. If we find a match, we are done. Otherwise, we recursively try smaller word combinations. In the following example, we find “Private customers” and “Switzerland”:

Private customers Switzerland

Keyword-only inputs are the simplest way to use SODA and most people are familiar with using keywords for searching. The following types of input patterns are for more advanced users. Often, one starts with keywords only and afterwards adds operators to refine the query.

4.3.2 Comparison Operators

The second type of input patterns are comparison operators. Each comparison operator is a small binary pattern where the operator is in the middle and its operands are to the left and to the right. We currently support >, >=, =, <=, < and like.

To identify operators and its operands in the input, we run our longest word combination algorithm as explained for Keywords above. This works well, because operators are simply words in the input and we can recognize them as such. The comparison operator will later on be applied to the keywords before and after itself.

4.3.3 Aggregation Operators

The last type of input patterns are aggregation operators. Here we currently use a very strict syntax, but this could be relaxed by modifying the pattern. We currently support sum and count, however, there is nothing that would prevent us from adding more when we need it. The syntax for aggregation operators is given in the next section.

4.4 Query Language

Our query language for processing keywords and comparison operators can be formalized as follows:

<search keywords> [ [ AND | OR ] <search keywords> | <comparison operator> <search keyword> ]

The optional parts are written in parenthesis, i.e. between [ and ]. The pipe sign indicates “or”. In order to express time-based range queries, the following syntax needs to be applied:
4.5 Patterns in Action - Examples

In this section we explain how we use patterns by looking at examples. We first present several queries which contain filter conditions. Afterwards we look at aggregation examples.

4.5.1 Examples with Filters

Assume that an end-user wants to find all information about Sara Guttinger—a customer of the bank. The respective SODA query as well as the SQL query are shown in Query 2.

SODA:

Sara Guttinger

SQL:

SELECT *
FROM parties, individuals
WHERE parties.id = individuals.id
   AND individuals.firstName = 'Sara'
   AND individuals.lastName = 'Guttinger'

Query 2: Keyword pattern example: SODA vs. SQL.

As we can see the SODA query is much easier to understand for a typical end-user than the SQL query where one needs to take into account the correct join and
CHAPTER 4. GENERATING SQL FROM PATTERNS

filter conditions. Furthermore, “parties” is the inheritance parent of “individuals” and is joined here, because “individuals” matches our inheritance child pattern.

In the second example we are looking for everyone who has a salary above a given value and was born on a certain date. The input query given to SODA is shown in the upper part of Query 3.

SODA:

\[
salary >= x \text{ and } birthday = \text{date}(1981-04-23)
\]

SQL:

\[
SELECT *
FROM persons
WHERE persons.salary >= x
    AND persons.birthday = 1981-04-23
\]

Query 3: Input pattern example: SODA vs. SQL.

In this query we find three input patterns: a greater-equal comparison, an equality comparison, and a \(date()\) operator. The remaining keywords are processed with our metadata graph pattern matching algorithm. Both, “salary” as well as “birthday”, would match our column pattern and we would therefore include the corresponding table (e.g. persons) as well as the two attributes (persons.salary and persons.birthday). “and” might be unknown and we therefore ignore it. For completeness, a possible SQL query which does the same is also shown in Query 3.

In this example, the SODA query looks almost as complicated as the SQL input. But, this is only due to the three input operators we wanted to show. Usually, one simply types a few keywords to start with and incrementally refines the SODA query.

4.5.2 Examples with Aggregation

Assume that a business analyst wants to find the top \(n\) customers with the highest trading volume in 2010. Ideally, the input query would look like

```
Top 10 trading volume customer
between January 2010 and December 2010
```

Unfortunately, the first problem is that the given date range could refer either to trading volume or to customer. To resolve this ambiguity, one could write the query as a)
4.5. PATTERNS IN ACTION - EXAMPLES

Top 10 trading volume customer transaction date
between date(2010-01-01) date(2010-12-31)

In this case, we would find customers with top trading volume which executed transactions in a certain time frame. On the other hand, if we wanted to find young customers with high trading volumes, we would write the query as b)

Top 10 trading volume customer birth date
between date(1980-01-01) date(1990-01-01)

For this problem, it suffices to show both results to the business analyst and let her choose the result that matches her intent. The second problem, however, is more difficult to tackle: How to infer from “trading volume” to “aggregation of transaction amount”? One way to solve this problem is to let the user write a more precise query with explicit aggregation operator, e.g.

Top 10 sum(amount) customer transaction date
between date(1980-01-01) date(1990-01-01)

Since this is rather unintuitive, another way to handle such cases is to introduce a domain ontology. A domain ontology in our case defines a classification for a given domain. Here we need a domain ontology for trading applications with an explicit aggregation operator such as sum(transaction amount) (see Figure 4.4). Given this domain ontology, “trading volume” can be translated automatically.

To give a concrete example, we assume that we are interested in the amount of the transactions per trading day. The corresponding SQL statement and its SODA counterpart are shown in Query 4.

<table>
<thead>
<tr>
<th>SODA:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum (amount) group by (transaction date)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT sum(amount), transactiondate</td>
</tr>
<tr>
<td>FROM fi_transactions</td>
</tr>
<tr>
<td>GROUP BY transactiondate</td>
</tr>
</tbody>
</table>

**Query 4:** Aggregation pattern example: SODA vs. SQL.

The advantage of the SODA aggregation approach over standard SQL becomes even more apparent for aggregation queries that require a multi-table join. Since
Figure 4.4: Domain Ontology for Trading Applications.
SODA automatically identifies the join predicates, the end-user does not need to worry about writing full SQL, which is often hard for typical non-tech savvy business analysts and hence SODA takes over that burden to enable user-friendly data warehouse search.

Consider, for instance, the example where we want to rank the organizations by trading volume. Query 5 shows the SODA query and the corresponding proper SQL statement. From the point of view of a business analyst, the SODA query is more intuitive, easier and much faster to write.

```
SODA:
   count (transactions) group by (company name)

SQL:
   SELECT count(fi_transactions.id), companyname
   FROM transactions, fi_transactions, organizations
   WHERE transactions.id = fi_transactions.id
   AND transactions.toParty = organizations.id
   GROUP BY organizations.companyname
   ORDER BY count(fi_transactions.id) desc
```

**Query 5**: Organizations ranked by trading volume example: SODA vs. SQL.
Chapter 5

Experiments & Results

In this chapter we report on the experiments that we carried out with SODA on Credit Suisse’s central data warehouse, which is among the largest and most complex data warehouses in the financial industry. Our experimental results demonstrate that SODA’s keyword search algorithm generates executable SQL queries with high precision and recall compared to the manually written gold standard queries. (Gold standard queries have been manually written by domain experts.) SODA reveals ambiguities of the query keywords by understanding different patterns of the schema graph and by searching the base data using an inverted index. In addition to simple keyword search, SODA also supports conjunctive range queries as well as aggregations. As we will see in this section, the results clearly show that SODA not only works well for small schemas with several tables but also for large schemas with complex inheritance and join relationships of a modern enterprise data warehouse from the financial services industry. Moreover, we will highlight our experiences and challenges we faced when working with large and complex data sets.

5.1 Experimental Setup

5.1.1 The Credit Suisse Data Warehouse

The Credit Suisse data warehouse is an enterprise data warehouse that consists of three main layers, namely integration layer, enrichment layer, and analysis layer [24]. See Figure 5.1.

The integration layer receives data from some 2,500 different source systems covering all areas of the bank such as information about customers, investment products, trades, etc. The total data volume covering three test environments and one production environment fully replicated over physically separated data
centers is currently around 700 terabytes. The unintegrated data comprises several thousands of tables with some 30,000 attributes. The main goal of the integration layer is to take the data from the heterogeneous data sources and integrate them into a carefully modeled enterprise data warehouse with bi-temporal historization [11]. In other words, the data warehouse is a temporal database system with time dimensions covering the validity time and the system time [74].

Once the data is fully historized and quality controlled, the enrichment layer is used for storing so-called reusable measures and dimensions that are calculated based on previously integrated data. A typical example of data enrichment are SOX, Basel II, and Basel III calculations that compute complex base key figures that are materialized for efficiency reasons. The actual data analysis takes place in the analysis layer which consists of several business specific physical data marts fed either from the integration layer or the enrichment layer. Typical examples of these business applications are dedicated data marts for risk calculations, legal and compliance assessments or profitability calculations.

In addition to the actual data warehouse, Credit Suisse also has a metadata warehouse [37] that allows navigating and searching the complex schema of the various data warehouse layers. This metadata warehouse enables business users, requirements engineers, and software architects to get a better understanding of the complex relationships between the various data items. SODA builds on top of
the metadata warehouse and provides additional functionality, namely automatic SQL generation and hence interactive data exploration based on keywords.

### 5.1.2 Software and Hardware Used

SODA is implemented in Java version 1.6 with a generic database back-end that we tested with three different database systems: Derby, MySQL, and Oracle. Our experiments with real data were executed using Oracle 11gR1 on a Sun M5000 shared memory machine with 32 cores, 128 GB of main memory, and an enterprise-scale storage back-end that is attached to several data warehouse servers. The operating system is Solaris 10. Our data set is based on the full schema of the integration layer consisting of 472 tables with a reduced and anonymized data volume of 220 GB. The top 10% of these tables have above $10^7$ records with the largest table comprising $6.7 \times 10^8$ rows. Moreover, the complex schema of the data warehouse consists of dozens of inheritance relationships with several levels.

Since our test environment is shared with other applications, our experiments were restricted to 4 cores and a maximum of 32 GB of main memory. Building up the inverted index for all 472 base data tables took 24 hours on a single core. The total size of the inverted index over all base tables is 9.5 GB comprising $1.4 \times 10^8$ non-unique records. Note that the inverted index is only built on table columns of data type “text”. In other words, base data table columns with numerical data types are not contained in our inverted index.

Table 5.1 shows the complexity of the schema graph in terms of the number of entities and attributes. Note that the cardinality of conceptual entities which represents the business world is 226, while the cardinality for logical entities and physical tables is 436 and 472, respectively. These numbers indicate that the complexity of the technical world increases with respect to the business world. Also note that the total number of attributes increases from 985 for the business world, that is, the conceptual attributes, to 2700 and 3181 for the technical world, that is, logical attributes and physical columns, respectively. The total size of schema graph is 37 MB which is relatively small compared to the total size of the base data which is 220 GB.

The great challenge that every business analyst faces who wants to query the data warehouse, is to understand the meaning of all these 436 entities and 472 tables and to correctly correlate them to each other (i.e. to understand the relationships among each other). As we can see in Section 5.3.2, SODA is considered as an important step to enable end-users to explore the data warehouse in an intuitive way.
<table>
<thead>
<tr>
<th>Type</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Conceptual entities</td>
<td>226</td>
</tr>
<tr>
<td>#Conceptual attributes</td>
<td>985</td>
</tr>
<tr>
<td>#Conceptual relationships</td>
<td>243</td>
</tr>
<tr>
<td>#Logical entities</td>
<td>436</td>
</tr>
<tr>
<td>#Logical attributes</td>
<td>2700</td>
</tr>
<tr>
<td>#Logical relationships</td>
<td>254</td>
</tr>
<tr>
<td>#Physical tables</td>
<td>472</td>
</tr>
<tr>
<td>#Physical columns</td>
<td>3181</td>
</tr>
</tbody>
</table>

Table 5.1: Complexity of the Schema Graph.

5.1.3 Queries

Our query workload consists of a mix of queries taken from the query logs, queries proposed by our business users and synthetic queries to cover corner cases of our algorithms — such as complex aggregations with joins. Inspired by the 20 queries for astrophysics proposed by Jim Gray et al. [77], this paper shows the results for the 10+ most interesting queries we executed against one of the Credit Suisse data warehouses. These queries that are shown in Table 5.2 cover various query types such as queries against base data (B), against the schema (S) or domain ontology (D). Others handle inheritance (I), predicates (P) or aggregates (A). The abbreviations for the query types are shown in the column “Comments”. Note that these different queries types are later on used for comparison with other systems (see Table 6.1).

Column 2 shows the queries expressed in terms of keywords and column 3 gives additional comments about the queries. For instance, query Q1.0 is answered by finding a match of “private customers” in the customer domain ontology as well as finding a match of “family name” in the schema graph. Query Q2.1 is evaluated by using “Sara” as a filter criterion on the base data. Queries Q2.2 and Q2.3 are additional refinements of query Q2.1 to yield more precise answers. Queries Q3.1 and Q3.2 show the ambiguity of the query “Credit Suisse”. The user could either be interested in Credit Suisse as an organization or as an entity that is part of an agreement. Query Q5.0 is interesting since it needs to correctly identify inheritance relationships (by applying the inheritance pattern), while queries Q9.0 and Q10.0 are aggregation queries.

For each of these queries we manually wrote so-called gold standard queries (see column 4), i.e. executable SQL statements that return the best results for the given queries. The gold standard queries serve as the yard stick for measuring precision and recall of the SODA queries. Due to privacy reasons of Credit Suisse’s
<table>
<thead>
<tr>
<th>Q</th>
<th>Keyword</th>
<th>Comment</th>
<th>Gold standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>private customers family name</td>
<td>Use customer domain ontology (D) and combine with attribute from schema (S).</td>
<td>3-way join incl. inheritance (I).</td>
</tr>
<tr>
<td>2.1</td>
<td>Sara</td>
<td>Use base data (B) as a filter criterion.</td>
<td>3-way join incl. inheritance (I) with where-clause on firstname.</td>
</tr>
<tr>
<td>2.2</td>
<td>Sara given name</td>
<td>Same as for Q 2.1 + restriction on given name (S).</td>
<td>Same as for Q 2.1.</td>
</tr>
<tr>
<td>2.3</td>
<td>Sara birth date</td>
<td>Restriction on birth date to focus on specific table (S).</td>
<td>Same as for Q 2.1.</td>
</tr>
<tr>
<td>3.1</td>
<td>Credit Suisse</td>
<td>Use base data (B) as a filter criterion to find the organization.</td>
<td>Single table containing information about organizations with where-clause on org_name.</td>
</tr>
<tr>
<td>3.2</td>
<td>Credit Suisse</td>
<td>Use base data (B) as a filter criterion to find Credit Suisse agreements.</td>
<td>Single table containing information about deals with where-clause on agreement_name.</td>
</tr>
<tr>
<td>4.0</td>
<td>gold agreement</td>
<td>Use base data (B) as filter and match with schema attribute (S).</td>
<td>2-way join.</td>
</tr>
<tr>
<td>5.0</td>
<td>customers names</td>
<td>Identify inheritance relationships (I) and use names domain ontology (D).</td>
<td>Two separate 3-way join queries for private and corporate clients.</td>
</tr>
<tr>
<td>6.0</td>
<td>trade order period &gt; date(2011-09-01)</td>
<td>Time-base range query (P) on given column (S).</td>
<td>3-way join with where-clause incl. inheritance (I).</td>
</tr>
<tr>
<td>7.0</td>
<td>YEN trade order</td>
<td>Use base data (B) filters and schema (S).</td>
<td>5-way join with 2 where-clauses incl. inheritance (I).</td>
</tr>
<tr>
<td>8.0</td>
<td>trade order investment product Lehman XYZ</td>
<td>Base data (B) + schema (S).</td>
<td>5-way join with where-clause incl. inheritance (I).</td>
</tr>
<tr>
<td>9.0</td>
<td>select count() private customers Switzerland</td>
<td>Base data (B) + domain ontology (D) + aggregation (A)</td>
<td>5-way join + aggregation incl. inheritance (I).</td>
</tr>
<tr>
<td>10.0</td>
<td>sum(investments) group by (currency)</td>
<td>Aggregation (A) with explicit grouping and schema (S).</td>
<td>5-way join + aggregation + group by.</td>
</tr>
</tbody>
</table>

Table 5.2: Experiment Queries.
5.2 Results

5.2.1 Precision and Recall

For each input query $Q_i$, we compute precision and recall for all $j$ SQL statements $R_{ij}$ SODA produces. To compute precision, we compared the result tuples of a produced SQL statement of SODA #$_{ij}$ with the result tuples of the Gold Standard query #$_{Gi}$. A precision of 1.0 means, that a SQL statement produced by SODA returned only tuples that also appear in the Gold Standard result #$_{Rij}$ ⊆ #$_{Gi}$. Similarly, a recall of 1.0 means, that a SQL statement produced by SODA returned all tuples of the Gold Standard result #$_{Gi}$ ⊆ #$_{Rij}$. In Table 5.3 we show precision and recall of the best SQL statement produced by SODA. We also calculated the number of SODA results with precision and recall greater than 0 and equal to 0. Table 5.3 shows the results.

We can see that for a majority of the queries, SODA produces a precision of 1.0 while the recall is either 1.0 or 0.2 (see Q2.1 and Q2.2). The reason for the sometimes low recall is due to the fact that the data warehouse uses bi-temporal historization where the actual join keys are not properly reflected in the schema graph. In order to mitigate this problem, the schema graph needs to be annotated with join relationships that reflect bi-temporal historization. Note that SODA provides a very flexible way of incorporating these changes that are typically required for modern data warehouses that constantly evolve over time—both in size and complexity.

Another interesting observation is that for some queries the precision is close to 0 or even 0 (see Q5.0 and Q9.0). These results are due to the complex nature of the data model with several bridge tables where SODA is not able to identify the correct join conditions.

5.2.2 Query Complexity and Runtime

After measuring the precision and recall of the queries, we now analyze the query complexity and the runtime.

The query complexity is defined as the number of combinations that can potentially lead to a query result. For instance, recall the query “customers Zurich financial instruments” shown in Figure 3.7. This query has a complexity of $1 \times 1 \times 2 = 2$ which is explained as follows: The term “customers” occurs 1 time in the domain ontology, the term “Zurich” occurs 1 time in the base data and the term
5.2. RESULTS

<table>
<thead>
<tr>
<th>Q</th>
<th>Best Result</th>
<th>#Results</th>
<th>#Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision (P)</td>
<td>Recall (R)</td>
<td>P,R &gt; 0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>1.00</td>
<td>0.20</td>
<td>1</td>
</tr>
<tr>
<td>2.2</td>
<td>1.00</td>
<td>0.20</td>
<td>1</td>
</tr>
<tr>
<td>2.3</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>3.1</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>3.2</td>
<td>1.00</td>
<td>1.00</td>
<td>3</td>
</tr>
<tr>
<td>4.0</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>5.0</td>
<td>0.12</td>
<td>0.56</td>
<td>1</td>
</tr>
<tr>
<td>6.0</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>7.0</td>
<td>0.50</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>8.0</td>
<td>1.00</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>9.0</td>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>10.0</td>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.3: Precision and Recall for Experiment Queries.

"financial instruments" occurs 2 times (once in the conceptual and once in the logical schema). In general, the number of results after the lookup phase grows quickly due to the combinatorial product of all entry points. The remaining steps, however, are all linear in the size of the meta-data.

The end-to-end execution time of a SODA query is split up into time fractions that correspond to the algorithmic steps: (1) lookup, (2) rank, (3) tables, (4) SQL and (5) grouping. The total time to execute these 10+ queries was roughly one hour where the majority of the time was spent on executing the generated SQL queries. The time for SODA to analyze the query and to produce proper SQL is in the range of seconds. A summary of the query complexity, the number of query results along with the runtimes for SODA (in seconds) and the end-to-end query execution time (in minutes) is given in Table 5.4. We can see that the SODA runtimes are between 0.73 and 7.31 seconds while the total runtime for executing the SQL query on the database ranges between 1 and 40 minutes. Note that query Q10.0 has the largest total runtime of 40 minutes due to the aggregation and group by operations that need to be performed along with a 5-way join on large tables. These numbers indicate that the overhead for the SODA query processing is a small fraction compared to the total query execution time.
<table>
<thead>
<tr>
<th>Q</th>
<th>Complexity</th>
<th>#Results</th>
<th>SODA runtime (sec)</th>
<th>Total runtime (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>3</td>
<td>1</td>
<td>1.54</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>4</td>
<td>4</td>
<td>0.81</td>
<td>1</td>
</tr>
<tr>
<td>2.2</td>
<td>12</td>
<td>2</td>
<td>1.60</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>12</td>
<td>3</td>
<td>1.69</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>12</td>
<td>6</td>
<td>3.78</td>
<td>2</td>
</tr>
<tr>
<td>3.2</td>
<td>12</td>
<td>6</td>
<td>3.78</td>
<td>2</td>
</tr>
<tr>
<td>4.0</td>
<td>16</td>
<td>4</td>
<td>4.89</td>
<td>4</td>
</tr>
<tr>
<td>5.0</td>
<td>4</td>
<td>4</td>
<td>1.24</td>
<td>6</td>
</tr>
<tr>
<td>6.0</td>
<td>5</td>
<td>2</td>
<td>0.73</td>
<td>1</td>
</tr>
<tr>
<td>7.0</td>
<td>20</td>
<td>4</td>
<td>4.94</td>
<td>1</td>
</tr>
<tr>
<td>8.0</td>
<td>8</td>
<td>4</td>
<td>2.94</td>
<td>2</td>
</tr>
<tr>
<td>9.0</td>
<td>30</td>
<td>6</td>
<td>7.31</td>
<td>1</td>
</tr>
<tr>
<td>10.0</td>
<td>25</td>
<td>6</td>
<td>2.83</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.4: Query Complexity and Runtime Information.

5.3 War Stories

5.3.1 Experience and Challenges

In this section we discuss our experience and challenges we faced when working with the enterprise data warehouse of Credit Suisse. Perhaps one of the biggest challenges in a real data warehouse is data quality. On the one hand, the number of source systems and the constantly changing business requirements make it almost impossible to have a perfectly matching schema description that 100% reflects the physical database implementation. Moreover, data warehouses are actually never really finished, since new feeder files are ingested into the data warehouse on a regular basis. These, in turn, need to be modeled and integrated into the existing enterprise data model.

Another real challenge is the complexity of the schema which includes inheritance relationships of several levels. The complexity increases when there are several relations (physical bridge tables) between siblings of an inheritance relationship (see entity “associate_employment” Figure 5.2). These bridge tables between siblings are common in several areas of the schema. Hence, automatic generation of SQL that takes into account these complexities is non-trivial—especially when some of the primary/foreign key relationships are not always implemented or the data is not populated yet. These issues often lead to queries with low precision
and recall as we have seen for query Q5.0.

The strength of SODA is that these data quality issues or schema complexities can be mitigated by annotating the schema graph or by extending the graph patterns. For instance, if we know from—let’s say the Testing Team—that some database tables that are part of a bridge tables between siblings are not populated yet, the schema can be annotated indicating that the respective relationship should be ignored. Once the underlying database tables are populated, the annotation can be updated so that SODA can take this relation into account when generating queries.

Another open issue of SODA is how to deal with temporal aspects of the data warehouse, i.e. bi-temporal historization. At the moment, SODA has no special support for temporal operators. Time is processed by SODA just like any other dimension. That is, SODA can generate queries that involve time (via range predicates on “valid-date” in a bi-temporal database or by restricting a year or a quarter), but SODA does not support, say, temporal aggregations or history joins. We plan to implement those as part of future work if business users ask for it.

Finally, SODA does not blindly produce all theoretically possible join paths, but rather combines a directed graph traversal with a given set of patterns to find useful tables and joins. While this has the advantage of being less computationally intensive and usually still leads to the intended results, there is no guarantee that we are not missing a required join path. E.g. we might not be able to find a join path between two entities which are too far apart in the schema graph. In this situation, “far-fetching” patterns might help. In other situations, however, “far-fetching” patterns might produce so many results that even ranking them becomes infeasible.
5.3.2 Feedback from Various Audiences

We demonstrated SODA to various people inside and outside of Credit Suisse to get feedback about our system. The people were both computer scientists as well as business users. One of the interesting observations is that different users see potentially completely different usage scenarios for SODA.

One group of people is impressed by the feature of the inverted index on the base data which allows identifying data items spread across several tables in the data warehouse that they were not even aware of. The reason for the data items to be located in different tables is due to the different data semantics.

Another group of people sees the potential of using SODA as an exploratory tool to analyze the schema and learn patterns in the schema in order to find out which entities are related with others. These types of users would issue a query and get a table as a result. Next, they would use the SODA schema browser to dive deeper. By an interactive approach of generating automatic queries based on keywords and analyzing the schema, they would identify potential flaws in the schema design or data quality issues.

A third group of users would use SODA to help creating SQL statements. They appreciate the feature that SODA automatically discovers join relationships between tables. Typically, end-users would just say “Give me tables X, Y and Z and show me the differences in calculations with respect to the previous months”. These types of business users are not willing to define the complex join conditions by themselves. In fact, within Credit Suisse we incorporated some of the SODA functionality in the so-called Adjustment-Engine—a system that enables business users to adjust data in the data warehouse by themselves.

Finally, a forth group of people is looking into the possibility of using SODA as a way to help document legacy systems by reverse engineering the conceptual, logical and physical schema based on the existing physical implementation of the data warehouse. After the reverse engineering is completed, the RDF schema graph can be generated and annotated accordingly. SODA would give them the possibility to explore legacy systems where documentation is very scarce or does not even exist.

As we can see from the various types of feedback illustrated above, SODA can be used for different tasks that were originally not even foreseen when we designed SODA.
Chapter 6
Related Work

In this chapter we look at related work of SODA. We first look at how search in relational databases evolved and how it compares to SODA. Section 6.2 provides a qualitative comparison with related systems. And finally, we compare SODA to similar studies in the field.

6.1 Search in Relational Databases

The design of SODA is based on the experience gained with a number of related systems that were developed over the last decade. The first systems to support keyword search in relational databases were DBExplorer [1], DISCOVER [32], and BANKS [6]. The key idea of these systems was to build an inverted index on the base data and to consider key/foreign key relationships when building query results. The inverted index is used to find all occurrences of the keywords of a query in tuples of the database. The key/foreign key relationships are used to compute join paths to construct business objects from the tuples that match different keywords of the query. The results of DISCOVER and BANKS are in the granularity of specific instances (i.e., individual business objects assembled from individual tuples that match the keywords). DBExplorer generates results in the granularity of sets of business objects. All three approaches differ in the way they generate the join paths.

Based on the foundations laid with this early work on keyword search in relational databases, a number of more sophisticated systems have been developed in the recent past. Keymantic [5] shows how to support search on the “Hidden Web”. In the “Hidden Web”, no inverted indexes can be constructed because the base data is not crawlable. The only information that is known to Keymantic is metadata such as the names of input fields from, e.g., crawling the Web forms of a “Hidden Web” database. So, a keyword query is processed as follows: First, all
keywords that correspond to metadata items (e.g., field names) are extracted. The remaining keywords are considered as possible input fields. Second, the likelihood of a remaining keyword to a metadata item is computed in order to rank different options to execute the keyword query on the “Hidden Web” database.

The work that is most closely related to SODA is SQAK [82]. SQAK is the only system that we are aware of that is able to generate aggregate queries. It is, therefore, well suited for data warehouses. SQAK has, furthermore, a special way to compute join paths that respects the direction of key/foreign key relationships. Unfortunately, all these techniques are hard-coded into the SQAK approach. As a result, SQAK is not able to process any queries that go beyond the pre-defined SQAK pattern of \texttt{SELECT-PROJECT-JOIN-GROUP-BY} queries. Furthermore, SQAK is not able to integrate metadata in the flexible and general way that SODA can.

6.2 Evaluation of Related Systems

Table 6.1 gives an overview of which features are supported by the related systems described in the previous section. It shows the features that are supported by the individual systems and which benchmark queries involve these features. Keymantic was the only system that we could evaluate experimentally because executable binaries were available from the authors; for the other systems, the overview of Table 6.1 is based on the description from the papers.

Simple queries that involve keywords found in the base data (e.g., “Sara” or “Credit Suisse”) are obviously well supported by DBExplorer, DISCOVER, and BANKS, as shown in the first line of Table 6.1. Since SQAK specifically targets aggregate queries, it cannot handle simple keyword queries; such simple \texttt{SELECT} queries just do not match SQAK’s predefined pattern. In principle, Keymantic can handle such simple keyword queries, but for complex schemas with thousands of columns like that of the Credit Suisse data warehouse, Keymantic is not able to select the right columns to query even when given all the available metadata. It should be noted that DBExplorer as well as DISCOVER cannot handle even simple queries if the schema involves cycles. So, these two systems sometimes have issues for simple keyword queries on base data (indicated by a check mark in parenthesis in Table 6.1).

The advantages of SODA only become apparent for more complex queries and for queries that mix several features and involve metadata. While DBExplorer, DISCOVER, and BANKS do support look up of keywords in base data, these systems are nevertheless not able to process, e.g., Query 9 because that query also involves the right treatment of inheritance, domain ontologies, and group-by / aggregation. As a result, each of the systems listed in Table 6.1 (except SODA)
6.2. EVALUATION OF RELATED SYSTEMS

<table>
<thead>
<tr>
<th>Query type</th>
<th>Experiment Queries</th>
<th>DBExplorer</th>
<th>DISCOVER</th>
<th>BANKS</th>
<th>SQAK</th>
<th>Keymantic</th>
<th>SODA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base data</td>
<td>2.<em>, 3.</em>, 4, 7, 8, 9</td>
<td>✓</td>
<td>✓</td>
<td>NO</td>
<td>(NO)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Schema</td>
<td>1, 2.2, 2.3, 4, 6, 7, 8, 10</td>
<td>NO</td>
<td>(✓)</td>
<td>✓</td>
<td>NO</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Inheritance</td>
<td>1, 2.*, 5, 6, 7, 8, 9</td>
<td>NO</td>
<td>NO</td>
<td>No</td>
<td>NO</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Domain ontology</td>
<td>1, 5, 9</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>(✓)</td>
<td>✓</td>
</tr>
<tr>
<td>Predicates</td>
<td>6</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregates</td>
<td>9, 10</td>
<td>NO</td>
<td>NO</td>
<td>✓</td>
<td>NO</td>
<td>NO</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Qualitative Comparison.
can handle only a few of the benchmark queries (and those with caveats).

The only other system that is able to integrate metadata beyond key/foreign key relationships is Keymantic. To some extent it can handle queries that involve synonyms and homonyms (i.e., queries that involve a domain ontology or DBpedia data). But, even Keymantic cannot handle any queries that involve inheritance. Modeling inheritance involves the modeling of mutually exclusive relationships. Even within Credit Suisse such inheritance relationships are not modeled in a consistent way; that is why a flexible pattern matching approach is needed as used in SODA in which different patterns can be specified for the same concept. Flexible pattern matching is even more important in a generic search tool that is supposed to be used in different organization with highly varying modeling conventions.

It is worth mentioning that in data warehouses such as those found at Credit Suisse, physical column and table names never correspond to those documented as part of a conceptual or logical schema. At Credit Suisse, for example, “birth date” is shortened to “birth dt”; furthermore, entity names (such as agreement or investments) are suffixed with “td”. The best way to discover such matches is to keep metadata at multiple schema levels and to apply pattern matching across those levels as done in SODA (Figure 3.3).

SODA is also the only system that can properly deal with predicates. While it is conceivable that some of the systems be extended to deal with certain kinds of range predicates (e.g., date ranges), SODA is the only system that is able to handle predicates induced by the metadata (e.g., \textit{wealthy customers} as customers that have an annual income that is higher than a certain threshold defined as part of the domain ontology or other metadata).

6.3 Other Related Work

The systems discussed in the previous two sections are not the only related work. Various aspects of generating SQL from keywords have been studied in the literature. For instance, [28] studies alternative ranking algorithms; [48] addresses physical database optimization by using more efficient index structures; [72] supports complex queries by a more sophisticated approach to process natural language; and [65] provides tuple reduction. Other works use principles from information theory and statistics to summarize the relational schemas [88].

Another line of research studies the use of query refinement and query disambiguation approaches [61, 17, 18]. Ortega-Binderberger et al. [61] studies the importance of user subjectivity and achieves query refinement through relevance feedback. Similarly, SODA presents several possible solutions to its users and allows them to like (or dislike) each result. Elena Demidova et al. [17, 18] use query disambiguation techniques to process keyword queries automatically ex-
6.3. OTHER RELATED WORK

SnipSuggest [39] is a system that enables context-aware auto completion for SQL by taking into account previous query workloads which are, in turn, represented as workload DAG. When a user types a query, possible additions are suggested based on the highest ranked node in the DAG. Query suggestions include tables, views, functions in the FROM-clause, columns in the SELECT and GROUP BY clauses as well as predicates in the WHERE clause. The main difference to our approach is that SnipSuggest makes it easier for end-users to interactively build and improve SQL statements while SODA does not require any SQL knowledge at all. Moreover, SODA does also not rely on query logs.

Keyword search [49] and natural language processing [47] have also been applied to XML databases. [49] presents a survey that classifies search methods into four categories: a) Tree-based methods, where the result is based on the notion of LCA (lowest common ancestor). b) Statistics-based approaches, which work with statistics on the data distribution. c) Graph-based methods, which look for connecting subgraphs containing all keywords. d) Methods on RDF graphs, where the additional semantics of an RDF graph are utilized. SODA is closely related to methods that fall into categories (c) and (d). While the presented approaches can work with RDF graphs, they are not really making use of the additional semantics. The patterns in SODA, however, allow us to capture this information. The NaLIX system [47] takes natural language query as input and translates it into XQuery. One of the strengths of the system is that it provides feedback to the user if the query terms cannot be classified and hence translated. In these cases, the system suggests different ways of reformulating the queries.
Chapter 7

Conclusions

In this work we demonstrated that SODA (Search Over DA.ta Warehouse) is one step towards enabling end-users to interactively explore large data warehouses with complex schemas in a Google-like fashion. The key idea of SODA is to use a graph pattern matching algorithm to generate SQL based on simple key words. Our experiments—with both synthetic data as well as with a large data warehouse of a global player in the financial services industry—show that the generated queries have high precision and recall compared to the manually written gold standard queries. One of the strengths of SODA is that it can disambiguate the meaning of words by taking into account join and inheritance relationships among the matching tables. Moreover, SODA allows mitigating inconsistencies in the schema or data as well as data qualities issues by updating the respective metadata graph or by extending the graph pattern match algorithm.

As part of our future work we will evaluate the impacts of using DBpedia for matching keyword queries against various synonyms found in our classification. Since the use of DBpedia will naturally increase the number of possible query results, we will study more advanced ranking algorithms. Finally, we plan to use additional metadata graph patterns, for example, to better cope with bi-temporal historization or data lineage across different layers of the Credit Suisse data warehouses.
Part II

Moving Objects Indexing
Chapter 8

Introduction

Indexing support for moving objects is a crucial requirement in domains such as car tracking [35], airplane surveillance [84], mobile phone tracking [2], emergency services [23], social networking [51], and gaming engines [87]. In these applications, an update may be a car/airplane/phone sending a message on its new position. A query may be a range query, a nearest-neighbor query, or a time-parametrized range query asking for predicted positions of moving objects at a future time \( t_q \). Queries are issued either by car/air traffic control or by users themselves. All of the above applications face a principal problem: how to support efficient query processing under high update rates.

Considerable effort has been made in the past to develop efficient indexing support for this problem. Existing work can be classified into two groups:

The first group focuses on time-parameterized queries on top of external memory structures [64, 69, 79, 78, 63, 34, 62, 42, 89, 14]. The hidden assumption of these methods is that the data plus indexes would not fit into the available main memory. However, with the rise of large main memories to dozens of gigabytes, this assumption may be questioned [76]. For instance, the number of cars/trucks/motorcycles in Germany is currently about 58,000,000 [43]. Assume we come up with a space-efficient data and index representation. This data plus indexes can then fit into a few GBs of main memory.

The second group makes the opposite assumption that all data does fit into main memory [38, 90, 54, 15]. These methods, in contrast to the first group, are not general spatial-temporal indexes and only support continuous, present-time, monitoring queries. In particular, they do not support time-parameterized queries. In addition, none of these works presents any scalability solution to exploit the aggregate main memory of modern architectures with multiple cores and nodes.

Consequently, in this work we tackle the question: Can we come up with a method that solves the issues of both groups? Our research provides a positive answer.
Traditionally, index creation has been considered an extremely costly process. For that reason, research on moving object indexes has been centered around creating sophisticated index structures. These indexes are created once, kept, and then modified according to incoming updates. This has led to a plethora of complex index structure proposals in the past. However, with the rise of large main memories and fast multi-core CPUs, this “natural law” of maintaining a moving object index can be questioned.

We present a novel main-memory method termed MOVIES (MOVing objects Indexing using frEquent Snapshots) that supports time-parameterized (predictive) queries and is at the same time space-, query-, update-, and multi-CPU-efficient. At its core, MOVIES resembles the approach taken by a cinematographer: as it is impossible to capture continuously moving data with any camera in one image, a cinematographer has to take a series of still images at a given frame rate. As long as the frame rate exceeds the inertia of the human eye (i.e., at least 24 frames per second), an illusion of continuous movement is created. We follow exactly the same approach: We try to provide as many still index images of the data as possible. For a very short period of time, we use a given index image to answer incoming queries. After that, we throw that index away. As long as the index build rate is high, an illusion of a continuously up-to-date index will be created.

We will show that—surprisingly—index creation can be a matter of subseconds even for datasets comprising hundreds of thousands of moving objects. For instance, we will demonstrate that index creation for 1 million moving objects (a common data set size used in recent moving objects studies, see Section 16.4) takes as little as 0.16 seconds on a single computing core, allowing for six index rebuilds per second. The price we have to pay for these features is slightly out-of-date (stale) query results, i.e., even though queries are executed immediately in our approach, query results may not consider the most recent updates. However, we will show that even for massive data sets this query result staleness may be reduced to (sub-)seconds. This meets by far the demands of real applications. For instance, state-of-the-art flight control in Europe currently works with a staleness of 5 seconds [73].
Chapter 9

MOVIES

With the exponential growth of moving objects data to the Gigabyte range, it has become critical to develop effective techniques for indexing, updating, and querying these massive data sets. This chapter presents MOVIES [19, 20], a scalable and space-efficient indexing method designed for main memory and tailored for multicore and multi-node environments.

To meet the high update rate as well as low query response time requirements of moving object applications, we take a novel approach in moving object indexing. In our approach, we do not require a sophisticated index structure that needs to be adjusted for each incoming update. Rather, we construct conceptually simple short-lived index images that we only keep for a very short period of time (subseconds) in main memory. Our approach resembles the one taken by a movie-camera which takes a series of still images per second: we take a series of still index images per second. Our approach is feasible as long as index construction is cheap which is true for a main memory database. Today, tens of millions of moving objects may easily be kept in a few GB of main memory. As a consequence, the resulting technique MOVIES supports at the same time high query rates and high update rates, trading this property for query result staleness.

9.1 Preliminaries

9.1.1 Problem Statement

We consider a data set of \( N \) moving objects in a two-dimensional space of data with domain \( |X| \times |Y| \), where \( |X| \) (resp. \( |Y| \)) represents the number of different positions in the horizontal (resp. vertical) dimension. Extending our technique to more dimensions is straightforward and is explained in more detail in Section 15.2.3. Similarly to Jensen et al. [34], we assume a discrete space of \( 2^{16} \times 2^{16} = 2^{32} \).
different positions. Each moving object is identified by a unique key termed an
OID. In addition, each moving object emits updates on its current location \((x, y)\)
and its speed vector \(\vec{s}\) by sending an \((x, y, \vec{s}, \text{OID})\) tuple to central indexing
server(s). Again as Jensen et al. [34], we assume that objects travel at a maximum
speed \(S_{\text{max}}\) and are guaranteed to send updates at least every \(t_{\Delta_{\text{max}}}\) seconds.
Furthermore, we assume that indexes are queried with two-dimensional predictive
range queries \(Q(r, t_q)\) specifying a range \(r = [x_{\text{low}}; x_{\text{high}}] \times [y_{\text{low}}; y_{\text{high}}]\) and a query
time \(t_q\). Note that other query types such as predictive k-nearest-neighbor may
easily be derived from predictive range queries as discussed in [34]. As a summary,
we will work with the following two operations:

\[
\begin{align*}
\text{Updates} : & (x, y, \vec{s}, \text{OID}) \\
\text{Queries} : & Q(r, t_q) \\
& r = [x_{\text{low}}; x_{\text{high}}] \times [y_{\text{low}}; y_{\text{high}}]
\end{align*}
\]

9.1.2 Formal Argument

In this section, we provide a formal argument to illustrate the core benefit of
our approach. We do not strive to provide a full-blown cost model but rather
focus on the key aspects. For realistic moving objects scenarios, the amount of
updates will be in the tens of millions per second. We will develop a method
that does not trade query performance for update performance as done in several
existing methods. Consider a simple index structure organizing a sorted mapping
spatialposition \(\mapsto\) OID (binary range search on a B+-tree or any cache-optimized
tree). We assume that the spatial position is linearized by a linearization function
(see Section 9.4). The cost for both querying and updating in-place are of the
order \(O(\log N)\), where \(N\) is the number of entries. An update in a positional
index consists of deleting the old entry and creating a new entry. Thus in the
worst case we need two logarithmic traversals. We derive a cost formula

\[
C_{\text{update-in-place}} = 2 \cdot c_1 \cdot \log_2 N
\]

where \(c_1\) is a hardware-dependent constant.

Similarly, the initial cost for bulkloading an index is of the order \(O(N \log N)\),
which translates to a cost formula

\[
C_{\text{bulkloading}} = c_2 \cdot N \log_2 N
\]

where \(c_2\) is a hardware-dependent constant.

Now, let us assume that instead of performing updates in-place, we collect \(W\)
updates in a separate structure with \(O(1)\) complexity per update, in other words,
9.1. PRELIMINARIES

$C_{\text{array update}} = c_3$. We will periodically rebuild a new index from that structure. The cost for this is

$$C_{\text{collect and rebuild}} = W \cdot C_{\text{array update}} + C_{\text{bulkloading}}$$

When will this strategy be cheaper than update-in-place? We obtain

$$W \cdot c_3 + c_2 \cdot N \log_2 N \leq W \cdot 2c_1 \cdot \log_2 N$$
$$c_2 \cdot N \log_2 N \leq W \cdot (2c_1 \log_2 N - c_3)$$
$$W \geq (c_2 \cdot N \log_2 N)/(2c_1 \log_2 N - c_3)$$

We may estimate upper bounds for the constants assuming a single core and the index to be limited to 16 million elements as $c_1 = 73.6\,\text{ns}$, $c_2 = 8.9\,\text{ns}$, and $c_3 = 112.5\,\text{ns}$. Thus we get

$$W \geq (8.9 \cdot N \log_2 N)/(2 \cdot 73.6 \cdot \log_2 N - 112.5)$$

For an index of size $N = 1,000,000$, the collect and rebuild approach will already be cheaper when the number of updates per second reaches $W = 62,872$. Also note that the query processing costs in both approaches are exactly the same. We just argued on how to improve update cost without touching query cost. On the contrary, the collect and rebuild approach could even be improved to build read-optimized indexes. That would additionally improve the query response time over an update-in-place approach.

Let us examine the maximum number of updates supported by the different methods. How many updates can we expect to support in a collect and rebuild approach? We may rebuild the index every $T_{\text{frame time}}$ seconds. This can be rewritten as follows:

$$C_{\text{collect and rebuild}}/T_{\text{frame time}} \leq 1$$
$$W \cdot C_{\text{array update}} + C_{\text{bulkloading}} \leq T_{\text{frame time}}$$
$$W \leq (T_{\text{frame time}} - C_{\text{bulkloading}})/(C_{\text{array update}})$$

The maximum number of updates processed per second can then be computed as

$$U_{\text{max}} = W/T_{\text{frame time}}$$

which is limited by the upper bound $1/C_{\text{array update}}$.

Assuming we allow for a $T_{\text{frame time}}$ of $3 \cdot C_{\text{bulkloading}}$, we obtain the function displayed in Figure 9.1. The figure shows that we may expect collect-and-rebuild
to *gain* an order of magnitude over update-in-place. The *price* we pay for that is query result staleness, limited by $2 \cdot T_{frame time}$. Figure 9.2 shows that even for an index of 1,000,000 elements being rebuilt by a single computing core, staleness will remain below a second.

### 9.2 The MOVIES Algorithm

This section presents the MOVIES algorithm (MOVing objects Indexing using frEquent Snapshots). As stated above, our method resembles the approach taken by a cinematographer: we try to create as many still index images as possible. This generates the illusion of a continuously up-to-date index.

#### 9.2.1 Algorithmic Walkthrough

The MOVIES algorithm is based on *index frames*. Each index frame is active during a short time interval called the *frame time* $T_i = [t_i; t_{i+1})$, where $i$ denotes the ID of the frame and $t_i$ denotes the moment in time when frame $i$ started. During each index frame, e.g., time interval $T_{45}$ for Frame 45 in Figure 9.3, we use a read-only index, $I_{44}$, to answer all incoming queries. We also keep an update buffer $U_{45}$ collecting all updates arriving during $T_{45}$. In addition, we build a new read-only index $I_{45}$ based on the updates collected in update buffer $U_{44}$ during $T_{44}$ (see arrow $\rightarrow$). Depending on whether the update buffers contain updates for all OIDs, this index build has to consider information available in index $I_{44}$ (see
9.2. THE MOVIES ALGORITHM

As soon as the new index $I_{45}$ is built, we start a new frame, e.g., Frame 46. In this frame, we use the newly built read-only index $I_{45}$ from Frame 45 to answer all incoming queries. We keep an update buffer $U_{46}$ to collect incoming updates. In addition, we build a new read-only index $I_{46}$ based on the updates collected in $U_{45}$ during $T_{45}$. Again, depending on whether the update buffers contain updates for all OIDs, this index build has to consider information available in index $I_{45}$.

As soon as the index is built, we start a new frame, e.g., Frame 47 (not shown) which is similar to Frame 45. Note that the number of entries collected in the update buffers may be considerably larger than the actual entries available in the index.

9.2.2 Comparison to Differential Files

The idea of collecting updates in a separate space and applying them in a batch was first used in the context of relational databases more than 30 years ago [70]. The idea of that paper was to collect changes in a separate differential file and merge that file regularly with the existing external memory index. Since then differential files were extended in multiple ways [57, 33, 56] and became state-of-the-art for read-mostly environments such as data warehouses (DWH) [46] as well as desktop [52], enterprise [46], and web search [25] engines.

In contrast to all of these approaches, MOVIES differs as follows:

1. For a moving objects application the query result staleness of a file-based method as followed in other applications [46, 52, 25] would be unacceptable.
For moving object indexing we require query result staleness to be below a few seconds (e.g., for an aircraft surveillance scenario it should be below 5 seconds [73]). Therefore, we have to optimize our algorithm for keeping staleness low. This can only be achieved by keeping the data entirely in main-memory.

2. In our scenario, moving objects are guaranteed to send an update at least every $t_{\Delta \text{max}}$ seconds. This is a standard assumption in similar studies [34]. Thus for certain situations, e.g. $t_{\Delta \text{max}} < T_i$, we may completely ignore the information available in the old index: We simply need to build an index image from the update buffer. Therefore, in contrast to differential file-based approaches [57, 33, 56], in MOVIES there is no need to perform a costly merge with the previous index. An index merge will only be used as a fallback.

3. In a moving objects scenario, the amount of updates is so immense that the number of updates collected in an update buffer may be by a factor larger than the number of entries in the current read-only index. This means the majority of the most recent values are available in the update buffer and not in the index. As a consequence, effective techniques for organizing the update buffers are required. In addition, if we detect that updates have been issued for all entries in the index, then again an index merge is obviated.
9.3. UPDATE BUFFERS

9.2.3 MOVIES Core Algorithm

The MOVIES core algorithm is displayed in Algorithm 1. It takes as input a stream of updates \( \text{StreamU} \), a stream of queries \( \text{StreamQ} \), and a bootstrapTime interval. The algorithm starts by creating a new update buffer \( U_0 \) (Line 1). Then the stream of updates \( \text{StreamU} \) is routed to that buffer (Line 2). An empty index \( I_0 \) is created in Line 3. Then the algorithm waits for a certain amount of time specified by a bootstrapTime. During that time, however, incoming updates are collected in update buffer \( U_0 \).

Lines 5–15 show the indexing loop used to create the sequence of index frames. This loop will be repeated until a global flag \texttt{should\_terminate} is set to true (Line 5). The loop keeps a counter \texttt{currentFrameID} for the current frame ID. Inside the loop an index frame starts by creating a new update buffer \( U_{\text{currentFrameID}} \) (Line 6). The stream of updates is routed to the new update buffer (Line 7). The stream of queries is routed to the index created in the previous iteration (Line 8). For the first loop iteration this index will be the empty index \( I_0 \). In Line 9, we check whether the \texttt{currentFrameID} is two or higher. If that is the case, we destroy the index built in index frame \( I_{\text{currentFrameID} - 2} \) (Line 10). After that we check whether we may ignore the data available in the old index (Line 12), e.g., this may happen if all elements in the old index became outdated by elements in the update buffer. If that check succeeds, we simply call the \texttt{build\_Index} operation on the update buffer (Line 13); otherwise, we create a new index \( I_{\text{currentFrameID}} \) also using information from the old index (Line 15). We then destroy the update buffer filled in the previous frame (Line 17). Finally, we increase the \texttt{currentFrameID} counter by one and continue looping (Line 18).

9.3 Update Buffers

9.3.1 Logging or Aggregation?

We now turn to the data structures used to represent update buffers. As we want to handle high update rates, we have to make sure that the update buffers do not exceed the available main memory. We solve this as follows.

For high update rates, the update buffers may contain several updates for the same \texttt{OID}, i.e., the update buffer may be considerably larger than the original index. As the aggregate of updates to an \texttt{OID} is sufficient for query processing (e.g., the most current position of a moving object), it makes sense to implement update buffers \( U_i \) by aggregation buffers \( \hat{U}_i \), organized using \texttt{OIDs} as keys. For each \texttt{OID}, \( \hat{U}_i \) only keeps the last update received for this moving object. In our approach, updates are written to aggregation buffers \( \hat{U}_i \) immediately when they arrive.
Algorithm 1: MOVIES Core Algorithm

Input: Stream of updates $StreamU$
Stream of queries $StreamQ$
Time Interval $bootstrapTime$

1. $U_0$.create();
2. $StreamU$.setDestination($U_0$);
3. $I_0$.create();
4. wait($bootstrapTime$);
5. for $(Integer \ currentFrameID = 1; \neg should\_terminate;)$ do
6. \hspace{1em} $U_{currentFrameID}$.create();
7. \hspace{1em} $StreamU$.setDestination($U_{currentFrameID}$);
8. \hspace{1em} $StreamQ$.setDestination($I_{currentFrameID-1}$);
9. \hspace{1em} if $currentFrameID \geq 2$ then
10. \hspace{2em} $I_{currentFrameID-2}$.destroy();
11. \hspace{1em} end
12. \hspace{1em} if (“may ignore old index”) then
13. \hspace{2em} $I_{currentFrameID} \leftarrow buildIndex(U_{currentFrameID-1})$;
14. \hspace{1em} else
15. \hspace{2em} $I_{currentFrameID} \leftarrow buildIndex(U_{currentFrameID-1}, I_{currentFrameID-1})$;
16. \hspace{1em} end
17. \hspace{1em} $U_{currentFrameID-1}$.destroy();
18. $currentFrameID = currentFrameID + 1$;
19. end
9.3. UPDATE BUFFERS

We implemented aggregation buffers using arrays of size $N$ where the slot at position $i$ stores the aggregate of object $i = OID$. Figure 9.4 shows an example. Figure 9.4(a) shows updates as represented in a log. Figure 9.4(b) shows the data as stored in an aggregation buffer: Only the last update for any given $OID$ is stored, e.g., for $OID = 5$ only the update at timestamp 15:23:12:008 is kept. Note that other implementations are possible, e.g., using a hash table. This ensures constant insert time for updates. We refer to this variant as Aggregated MOVIES. The algorithm based on FIFO update buffers is termed Logged MOVIES.

### 9.3.2 Data Entry Layout

For an implementation of the data structures to collect updates, we need to define what a single entry $(x, y, \vec{sv}, OID)$ looks like. As stated in Section 9.1.1, we assume a discrete space of $2^{16} \times 2^{16} = 2^{32}$ different positions. $OID$s have the domain $0, \ldots, 2^{27} - 1 \approx 134$ million and we work with $2^5 = 32$ different speed values. Using 27 bits for the $OID$s is actually only needed if 134 million moving objects are indexed by a single MOVIES instance. So one could also use less bits for the OID and use more bits for the position or speed vectors. This choice solely depends on the application’s needs. Thus, we represent both the position, speed vectors, and the $OID$ of a moving object using 8 bytes in total; the first four bytes store the $(x, y)$ position and the last four bytes store the speed vector $\vec{sv}$ and the OID. Figure 9.5 illustrates the data layout used.

It is interesting to note, that both updates and index entries can use exactly the same layout. Furthermore, this packed representation leads to more efficient
transfers in the cache hierarchy and better utilizes space in main memory. And finally, note that we could also assume updates containing a delta to the old position. However, this will only work in case the application guarantees that updates are never lost on the network. However, it is straightforward to modify our algorithms to support delta updates.

9.4 Read-Only Indexes

9.4.1 Spatial Indexing Method

As read-only index we use a state-of-the-art spatial indexing method. We focus on a technique that is at the same time simple and efficient. Therefore, we have chosen linearized kd-tries \[85, 60\]. This index was used in many papers in different variants and was shown to outperform competing approaches \[66, 59, 22, 34\].

The core idea of a linearized kd-trie is to simulate a pointer-based index structure. This is achieved by assigning each node of a virtual kd-trie a unique identifier termed a locational code. Locational codes are based on a space-filling curve such as the z-curve \[85, 60\] (see Figure 9.6(b)) or the Hilbert curve \[29\]. These recursive space-filling curves enumerate a multi-dimensional space (i.e., the nodes of the kd-trie, see Figure 9.6(a)) with a one-dimensional curve (see Figure 9.6(b)). For each object that needs to be managed by an index, it then suffices to compute its locational code, i.e., the virtual node it belongs to in the kd-trie. This calculation is independent of the locational codes of other objects. Therefore, at no point in time it is required to actually create a pointer-based kd-trie. As the locational codes are one-dimensional, they are ordered to provide efficient query processing. The resulting codes plus the data are then stored in a sorted index (see Figure 9.6(c), using \(w = 2\) bits per dimension).

One important observation, developed in Chapter 10, is that locational codes may be inlined with the data to avoid extra storage cost. Both point and range queries are efficiently supported. The latter are crucial for our scenario as several other types of queries such as nearest neighbor queries may be based on range queries \[34\]. Moreover, kd-tries are not limited to two dimensions but also work

\[\begin{array}{c|c|c|c}
32\text{ bits} & 5\text{ bits} & 27\text{ bits} \\
(x, y) & sv & OID \\
\end{array}\]

Figure 9.5: Data Layout in an Index.

\[1\] Following the terminology of Donald Knuth \[41\] a trie partitions the data space whereas a tree partitions the data.
9.4. READ-ONLY INDEXES

well for high dimensional spaces. Also recall that in contrast to grid-based indexes (e.g., as used by Yu et al. [90]), which would need to store an exponential number \((1/\text{grid length})^d\) of pointers to inclusion lists, an approach based on locational codes will for all \(d\) only require \(N\) indexing slots. Thus, our method can easily be extended to higher dimensional data (see 15.2.3). Note, however, that as with most spatial indexing methods, we expect linearized kd-trees to be affected by the curse of dimensionality and perform well only for a modest number of dimensions.

9.4.2 Index Layout

For positional indexing as described above, we need a one-dimensional index implementing a mapping from a locational code \(z\) to a set of \((x, y, \vec{s}_v, OID)\) entries. As range queries require indexed sequential access, we also need a structure that keeps entries sorted w.r.t. their locational codes. However, as we never modify an index, we do neither require tuple-at-a-time insert nor update functionality on the index structures. So we choose a simple sorted array in main memory, leading to fast random access by binary search and fast sequential operations (query, index merge, sorting).

Note that the sorted index which stores the linearized kd-trie may be optimized for cache performance in main memory. This is facilitated in our approach given that the index does not need to support updates. For example, it could be implemented by a simple sorted array as suggested above or a more complex read- and cache-optimized structure such as a CSS-tree [68]. We take the former option for implementation simplicity, given that our analysis in Section 9.1.2 indicates that better read-optimized structures will only improve query processing time in favor of MOVIES when compared to an update-in-place approach.
9.4.3 MOVIES buildIndex algorithm

Algorithm 2 shows how MOVIES builds a new index. It takes as its input an update buffer $U_i$, the old index $I_i$, and the number of moving objects $N$. Lines 1–4 initialize index $I_{i+1}$ by setting all its entries to empty. Line 6 resets the $OIDcounter$. Lines 7–15 merge update buffer $U_i$ into the new index $I_{i+1}$.

In more detail, while update buffer $U_i$ still contains updates (Line 7), the next update $u$ is retrieved and removed from this update buffer (Line 8). Line 9 retrieves the $OID$ from $u$. Line 10 computes the locational code for the given update. Line 11 checks whether we already received an update for this $OID$. If that is the case, we increase the $OIDcounter$ by one (Line 12). Finally, Line 14 writes update $u$ to the new index $I_{i+1}$ at position $OID$, i.e., the array is used as an index on $OIDs$.

After that, Line 16 checks whether the $OIDcounter$ surpasses the number $N$ of different moving objects. If that is the case, we simply ignore the information available in the old index (Line 17) as all entries in $I_i$ are guaranteed to have become outdated by entries in update buffer $U_i$. Otherwise, i.e., $OIDcounter < N$, we use the fallback strategy to merge possible stale entries from the old index $I_i$ into the new index $I_{i+1}$ (Lines 18–26).

Finally, Lines 27–28 sort the index w.r.t. locational codes. For better indexing performance, we use a radix sort algorithm in our implementation that only considers the $z$-code prefix of a data entry (see Chapter 10).

Note that all loop operations performed by our algorithm (Lines 2, 7 and 20) are implemented by sequentially accessing the data which is very cache-efficient. Only Lines 11&14, and 22&23 perform random accesses in $I_{i+1}$. An interesting extension of our algorithm would be to implement Lines 7–15 and 20–25 by a cache-optimized hash-join. We leave this to future work. Also note that in Lines 7–15 a sort-based merge is not possible as entries have to be merged w.r.t. their $OIDs$ and not w.r.t. their positions.
Algorithm 2: buildIndex

Input: UpdateBuffer $U_i$, Index $I_i$, Integer $N$.

Output: Index $I_{i+1}$.

1 // initialize new index:
2 forall the Entry $e \in Index I_{i+1}$ do
3 \hspace{1em} $e = \text{empty}$.
4 end
5 // use updates to build new index:
6 Integer $OIDcounter = 0$
7 while $U_i$.size() > 0 do
8 \hspace{1em} $u = U_i$.removeNext()
9 \hspace{1em} $OID = \text{getOID}(u)$
10 \hspace{1em} $u = \text{computeLocationalCode}(u)$
11 \hspace{1em} if $I_{i+1}[OID] = \text{empty}$ then
12 \hspace{2em} $OIDcounter = OIDcounter + 1$
13 \hspace{1em} end
14 \hspace{1em} $I_{i+1}[OID] = u$
15 end
16 if $OIDcounter \geq N$ then
17 \hspace{1em} // SKIP: ignore old index!
18 else
19 \hspace{1em} // fallback: MERGE old index into new index:
20 forall the Entry $e \in I_i$ do
21 \hspace{2em} $OID = \text{getOID}(e)$
22 \hspace{2em} if $I_{i+1}[OID] == \text{empty}$ then
23 \hspace{3em} $I_{i+1}[OID] = e$
24 \hspace{2em} end
25 end
26 end
27 // sort index on locational codes:
28 $I_{i+1}$.sort()
29 // return new index:
30 return $I_{i+1}$
Chapter 10

Tuning Spatial Indexing

MOVIES is a spatial indexing method. To make MOVIES scalable and space-efficient, we applied several tricks and techniques. In this chapter we work on the indexing part of MOVIES. We first explain how to make indexing in MOVIES space-efficient and afterwards, we show how to make it fast.

Sections 9.3 and 9.4 already present the data structures and data layout of the update buffer and the read-only index respectively. Now we look at location codes in general and z-codes in special and show how these can be used to create a space-efficient index.

10.1 Linearized kd-tries

For MOVIES we are looking for an indexing technique which is at the same time simple and space efficient and still has good query performance. Linearized kd-tries fulfill these criterias and they have been shown to outperform competing approaches [66, 59, 22, 34].

The main idea of kd-tries is to simulate a pointer-based index. This is achieved by computing a virtual kd-trie and assigning a location code to each virtual node in the trie. Such location codes are based on space-filling curves such as the z-curve [85, 60] or the Hilbert curve [29].

To actually store a position of an object, first the virtual node in the kd-trie is computed (see Figure 9.6(a)). For this node, we then get the location code (see Figure 9.6(b)) and store it in a sorted array for efficient query processing (see Figure 9.6(c)). Computing this location code is independent of other objects. Also, at no point in time it is required to actually create a pointer-based index. In theory, the location code requires an additional 4 bytes for each object. However, we can reduce this to 0 bytes per entry with a technique we term in-place locational codes, introduced below.
10.1.1 Z-Codes

A z-curve [85, 60] is a function which maps multidimensional data to one dimension while preserving locality of the data points. A z-code of a point in multidimensional space is calculated by interleaving the binary representations of its coordinate values. An example is shown in Figure 9.6(b).

Since z-curves preserve locality quite well, but are still fast to compute, they are a good option for MOVIES to compute location codes. We can directly use z-codes as location codes.

10.2 Partial In-Place Location Codes

To make our index space-efficient we store locational codes inline which is similar to operations proposed by Orenstein and Merrett [60]. However, in contrast to their work, we consider partial z-codes and partial reverse z-codes in order to reduce the cost for reversing a z-code at query time.

This works as follows: Assume for example an \((x, y)\)-position pair of size \(2 \times 3\) bits = 6 bits. A locational code of order \(w\) is defined to combine \(w \leq 3\) bits of each dimension of \((x, y)\) into a locational code of size \(2w\) bits. For instance, if \((x, y) = (011, 101)\) and \(w = 2\), then the z-code of order 2 of \((x, y)\) is \(0110\).

**Definition 1 (Z-Code).** The z-code of a tuple

\((x, y) = (x_0..x_{n-1}, y_0..y_{n-1})\)

where \(x_i\) and \(y_i\) represent individual bits is

\[z\text{-code}(x, y) = \langle x_0y_0x_1y_1...x_{n-1}y_{n-1} \rangle\]

The z-code of order \(w < n\) is defined as

\[z\text{-code}_w(x, y) = \langle x_0y_0x_1y_1...x_{w-1}y_{w-1} \rangle\]

As we have seen, if we assume an \((x, y)\)-pair of size \(2 \cdot 16\) bits and compute \(z\text{-code}_w(x, y)\) of order \(w\), we require only \(2w\) bits to store the actual z-code. The original pair \((x, y)\) can be reconstructed using \(z\text{-code}_w(x, y)\) and bits \(x_w, ..., x_{n-1}\) and \(y_w, ..., y_{n-1}\). Thus, we obtain partial in-place z-codes.

**Definition 2 (Partial In-Place Z-Code).** The partial in-place z-code of order \(w \leq n\) of a tuple

\((x, y) = (x_0..x_{n-1}, y_0..y_{n-1})\)
where \( x_i \) and \( y_i \) represent individual bits is

\[
\text{pipz-code}_w(x, y) = < x_0x_1 \ldots x_w-1x_w \ldots x_{n-1}y_w \ldots y_{n-1} >
\]

Note that the length of a partial in-place z-code is always 2\( n \) for all \( w \). Therefore, we may store z-codes in-place in the data entry (see Figure 10.1). We do not require additional space for z-codes, again making main memory representation more compact.

An additional advantage of this scheme is that the sort time may be reduced. This is because we only need to sort entries with respect to the first 2\( w \) bits. To make use of this, we implemented a radix-sort algorithm that performs 2\( w \) passes over the data where the \( i \)th pass considers bit \( i \). Obviously, if \( w \) is small, less radix-passes are required. On the other hand, queries will need to scan larger intervals of the partially sorted array in order to return correct results. Thus, the sort time decreases and query execution time increases and vice-versa. Therefore, this mechanism allows us to trade index rebuild performance for query performance in a fine-granular manner.

### 10.2.1 Precomputed Snippets

An additional improvement we make to traditional z-codes are partially precomputed z-codes. Assume a position pair

\[
(x, y) = (< x_0 \ldots x_{n-1} >, < y_0 \ldots y_{n-1} >)
\]

where any subsequence of bits

\[
x_i x_{i+1} x_{i+2} = < 011 >
\]

and

\[
y_i y_{i+1} y_{i+2} = < 101 >
\]

\((0 \leq i < n-2)\)

Then z-code\((x, y)\) has a subsequence

\[
z_{2i} \ldots z_{2(i+3)} = < 011011 >
\]
As we have to compute many z-codes, similar patterns of z-code subsequences will reoccur. Therefore, we conclude that it makes sense to pre-compute mappings from \((<x_{i..i+p}>,<y_{i..i+p}>) \mapsto <z_{2i..2(i+p)+1}>\) for a given \(p > 0\). In our implementation we chose \(p = 8\) requiring only 131 KB of main memory when stored in an array. Note that similar pre-computations may be defined for reverse z-codes.

### 10.2.2 Reversing Partial Z-Codes

To perform post-filtering operations on data entries, we define reverse z-codes.

**Definition 3 (Partial Reverse Z-Code).** The partial reverse z-code of order \(w \leq n\) of a partial in-place z-code \(z\) is

\[
pipz-code^{-1}_w(z) = (x, y)
\]

i.e.

\[
pipz-code^{-1}_w(pipz-code_w(x, y)) = (x, y)
\]

Note that reverse z-codes cause some computational effort at query time depending on \(w\). Thus we trade memory efficiency for the small overhead needed to reverse z-codes at query time (see Section 11.3.2).

### 10.3 Experiments

The following two experiments highlight the techniques described above.

Both experiments were performed on a single server having two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. Our testdata contains 6.4 million objects. Their position is given by the start position of our moving object traces used in Chapter 14.

Computing z-codes and sorting them afterwards is responsible for most of the time needed to rebuild an index. Therefore, what we see here is mainly the index rebuild cost.

#### 10.3.1 Partial In-Place Z-Codes Computation

Figure 10.2 shows the cost for computing 6.4 million partial in-place z-codes depending on the precision of a z-code (length of the z-code = \(2w\) bits, see Definition 2). The figure shows two different methods: \texttt{z-code \(T_x\)} is the naive loop-based implementation where \(x\) specifies the number of threads used. \texttt{z-code \(PREC\ T_x\)} is our method based on precomputed snippets as presented in Section 10.2.1. \texttt{z-code \(T_x\)}
10.3. EXPERIMENTS

Figure 10.2: Costs of Z-Code Computation [loop-based versus precomputed].

T1 requires up to 0.98 seconds. z-code T4 uses four threads and requires up to 0.27 seconds. In contrast, our method z-code PREC Tx requires at most 0.16 seconds on a single core and 0.06 seconds on four cores. Thus, our approach is up to 4.2 times faster than the loop-based approach.

10.3.2 Sorting Depending on Z-Code Precision

Figure 10.3 shows the costs for sorting 6.4 million data entries depending on the precision of a z-code. The curve shows two different methods: qsort Tx is the sort algorithm as provided by Java 5, i.e., quicksort for large partitions and insertion-sort for small partitions. rsort Tx is a linear radixsort performing $2w$ passes over the data, i.e., only the first $2w$ leading bits have to be considered for the sort.

qsort Tx shows a constant behavior independent of $w$. qsort T1 requires about 1.1 seconds; qsort T4 requires 0.4 seconds. In contrast, the performance of rsort Tx depends on $w$. For z-code length smaller than 16, rsort T1 is always faster than qsort T1; for z-codes longer equal 16, it is slightly slower. The same holds for rsort T4: For a z-code length smaller than 16, rsort T4 is always faster than qsort T4. If the z-code length is below 12, the improvement of rsort T4 over qsort T4 is at least 40%. For $2w \leq 6$ it is more than a factor 2.

We conclude that if $w$ is chosen small, the sort time, and therefore the reindexing time can be decreased. However, a small $w$ will have a negative impact on query performance. This trade-off is explored in the next chapter.
Figure 10.3: Costs for Sorting [quicksort versus radixsort].
Chapter 11

Quality-Aware Range Query Processing

This chapter presents how to process range queries when using positional indexing. First we will review the standard, textbook approach for processing range queries. Afterwards, we introduce a technique we call quality-aware query processing.

The standard approach either executes range queries without partitioning and then has to postfilter invalid results, or it completely partitions a range query into many small range queries and executes them all without having to postfilter very much. Both strategies have clear disadvantages which we will highlight below.

11.1 Standard Range Query Processing

In this section we concentrate on range queries of the form $Q = [x_{low}; x_{high}] \times [y_{low}; y_{high}]$ as other query types such as k-nearest-neighbor may easily be derived from range queries [34]. The result set obtained by executing a query $Q$ on data set $X$ is denoted as $Q(X)$.

In general, a two-dimensional range query $Q = [x_{low}; x_{high}] \times [y_{low}; y_{high}]$ is processed by transforming it into a one-dimensional interval query $Q_z = [z_{low}; z_{high}] = [z\text{-code}(x_{low}, y_{low}); z\text{-code}(x_{high}, y_{high})]$. The result of $Q_z$ is then obtained as follows:

1. perform a binary search on the index using $z_{low}$ as the key;

2. as long as the current index entry is smaller than or equal $z_{high}$: continue reading the index sequentially and returning the current entry.

As identified in previous work [85, 60, 66], this naive rq method is very expensive as the interval $[z_{low}; z_{high}]$ may contain a considerable number of false positives
not residing inside the query range defined by $Q$, having to be post-filtered. This issue is illustrated in Figure 11.1, where $Q = [(2, 2) \times (5, 4)]$. In the example, z-code(2, 2) = 12 and z-code(5, 4) = 50. Therefore, the interval query $Q_z$ would select the entire interval [12; 50] (depicted red in the figure). Previous approaches have solved this issue by partitioning query $Q_z = [z_{low}; z_{high}]$ recursively into multiple subqueries $Q^i_z = [z^i_{low}; z^i_{high}] \subset [z_{low}; z_{high}]$ ($0 \leq i \leq n$), satisfying the conditions

1. $[z^i_{low}; z^i_{high}] \cap [z^j_{low}; z^j_{high}] = \emptyset \forall i \neq j$,

2. $\bigcup_i Q^i_z(X) \subseteq Q_z(X)$, and

3. $\bigcup_i Q^i_z(X) \supseteq Q(X)$.

The idea is to minimize the number of false positives by issuing multiple probes. In order to explain how subqueries are computed, we introduce z-code partitioning [85, 60].

**Definition 4 (Z-Code Partitioning). Let**

$$Q_z = [z_{low}; z_{high}] = [< z_0 \ldots z_{n-1}>; < z'_0 \ldots z'_{n-1}>]$$

**be an interval query and bp be a bit position. Then, a partitioning part($Q_z$, bp) of $Q_z$ is defined as follows.** Set $b$ ($bp \leq b < n$) to the bit position marking the first differing bit of $z_{low}$ and $z_{high}$, i.e.,

$$z_i = z'_i \forall i < b, z_b \neq z'_b$$
If no such $b$ exists, $\text{part}(Q_z,bp)$ returns $Q_z$ otherwise $\text{part}(Q_z,bp)$ is

$$\text{part}(Q_z,bp) = (\left\lfloor z_{\text{low}}; z_{\text{lowsplit}} \right\rfloor, \left\lceil z_{\text{highsplit}}; z_{\text{high}} \right\rceil, b)$$

where

$$z_{\text{lowsplit}} = (z_{\text{high}} \text{ setting } z'_b = 0 \text{ and } z'_{b+2k} = 1 \forall k \geq 1)$$

$$z_{\text{highsplit}} = (z_{\text{low}} \text{ setting } z_b = 1 \text{ and } z_{b+2k} = 0 \forall k \geq 1)$$

Based on Definition 4, a recursive partitioning of $Q_z$ is obtained as follows: The recursion starts with $\text{part}(Q_z,0) = (Q_{\text{left}}, Q_{\text{right}}, b_0)$. Then in the second recursion step we compute both $\text{part}(Q_{\text{left}}, b_0 + 1)$ and $\text{part}(Q_{\text{right}}, b_0 + 1)$ and so on. Finally all recursive steps will terminate if no $b < n$ may be obtained anymore.

**Example 1 (Z-Code Partitioning).** Suppose we wish to query the window

$$Q = [(2, 2) \times (5, 4)]$$

as in Figure 11.1. As we have seen,

$$z\text{-code}(2, 2) = 12 = (001100)_2$$

$$z\text{-code}(5, 4) = 50 = (110010)_2$$

where we indicate the binary representation of the z-codes in parentheses. We start by computing $\text{part}(Q_z,0)$. The first differing bit in the two z-codes of $Q_z$, starting from bit zero, is bit zero itself. So

$$z_{\text{lowsplit}} = (011010)_2 = 26$$

$$z_{\text{highsplit}} = (100100)_2 = 36$$

Consequently, we obtain $\text{part}(Q_z,0) = ([12, 26], [36, 50], 0)$. By inspecting Figure 11.1, the reader may observe that we have divided the query window into two sub-windows by partitioning the space in the $x$ dimension. The two sub-queries improve precision by eliminating false positives, e.g., values 27 to 35 were originally returned by $Q_z$, but now are filtered out from the more refined sub-queries. In order to improve precision even further, we may partition the two sub-queries by reapplying the procedure above to recursively compute $\text{part}([12, 26], 1)$ and $\text{part}([36, 50], 1)$. □

A severe problem of Definition 4 is that the recursion may generate many subqueries [60]. The cost of executing these subqueries may in fact be higher than executing the initial unpartitioned query. In contrast to previous work, we solve this by steering that partitioning process based on the quality of the partitioning, as explained in the next section.
11.2 Quality-Aware Range Query Partitioning

Our goal here is to maximize query performance. As full query-partitioning would be inefficient, we compute a partial partitioning that tries to minimize the number of false hits retrieved. Our core idea is to partition queries based on benefit. This results in a method termed quality-aware rq, which works as follows: For each potential query partitioning step we compute its benefit, i.e., the length of the interval (dead space) that would be pruned by performing the partitioning step. We keep a priority queue \( PQ \) of potential recursive steps ordered by the benefits of the split in descending order, i.e., we obtain the benefit of the split before performing the split. We then recurse using \( PQ \) as follows: We remove the top element from \( PQ \), i.e., the partitioning step providing the highest benefit, and perform this partitioning. This process is performed recursively until a global error \( \lambda \) falls below a certain threshold \( \text{maxError} \). Here \( \lambda \) describes the ratio of the area covered by the query window (not the naive range query) versus the area covered by the current query partitioning. For instance, if we set a partitioning error \( \text{maxError} = 50\% \), this means that the region covered by the intervals of the query partitions may be at most 50\% larger than the region covered by the query window.

**Example 2** (Quality-Aware Range Querying). Consider one more time the query window \( Q \) depicted in Figure 11.1.

\[
Q = [(2, 2) \times (5, 4)]
\]

The area of the query window in this example is equal to 12. The length of the corresponding z-code representation of the query window \( Q_z = [12, 50] \) is 39. Thus, the error of the current z-code partitioning \( Q_z \) with respect to \( Q \) is

\[
\frac{(39 - 12)}{12} = 225\%
\]

Note that an error of zero would mean that the z-code partitioning exactly covers the query window. Assuming a value of \( \text{maxError} = 50\% \), we should proceed with z-code partitioning and partition \( Q_z \). As described in Example 1 above, this partitioning results in

\[
Q_z^1 = [12, 26] \\
Q_z^2 = [36, 50]
\]

Now, the length of the z-code representation dropped to 30, yielding an error with respect to \( Q \) of 150\%. As the error still exceeds our threshold, we should continue partitioning the intervals. However, we choose to partition first the interval
that will bring us the largest reduction in length. This can be computed by first deriving one level of partitioning of both $Q_1^z$ and $Q_2^z$.

\[
Q_1^z \rightarrow Q_{11}^z = [12, 15]; Q_{12}^z = [24, 26] \\
Q_2^z \rightarrow Q_{21}^z = [36, 39]; Q_{22}^z = [48, 50]
\]

Both candidate partitionings reduce their original interval lengths from 15 to 7. So, we may arbitrarily move on and partition $Q_1^z$ into $Q_{11}^z$ and $Q_{12}^z$. The length covered by $Q_{11}^z$, $Q_{12}^z$, and $Q_2^z$ equals 22, yielding an error of 83%.

\[
(22 - 12)/12 = 83\%
\]

The next z-code partitioning decision will then choose the interval among $Q_{11}^z$, $Q_{12}^z$, and $Q_2^z$ that most reduces the original interval lengths. Since $Q_{11}^z$ and $Q_{12}^z$ are already very good—and only little further reduction is possible there, the next interval to partition is $Q_2^z$. Partitioning $Q_2^z$ reduces the total length from 22 to 14, yielding an error of 16%. Our error is now below the maxError threshold of 50% and we are done.

\[
(14 - 12)/12 = 16\%
\]

\[\square\]

## 11.3 Experiments

The following three experiments study the effect of quality-aware query partitioning.

As in the previous chapter, we used a single server having 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. Our testdata contains 6.4 million objects. Their position is given by the start position of our moving object traces used in Chapter 14. The queries we use are also the same as in Chapter 14. See Section 14.1.2 for more details. The average number of query results is 39 for a query window of size 1,000 m × 1,000 m and 2,967 for a query window of size 10,000 m × 10,000 m.

We will see that quality-aware query partitioning works well and may gain up to 1.5 orders of magnitude depending of the maximum allowed error compared to a naive query partitioning as described in Section 11.1. We will also see how a quality-aware range query performs in comparison to a simple scan or a naive range query approach depending on the index precision—the z-code length used.
11.3.1 Quality-Aware Query Partitioning

The goal of this experiment is to understand the impact of quality-aware partitioning. We used a large query window of 10,000 m × 10,000 m, keep the index size constant at 6.4 million elements and vary maxError of the quality-aware query partitioning quality-aware rq for a batch of 500 queries.

Figure 11.2 shows the average execution time for a single query when varying maxError. Obviously, for naive rq the execution time remains constant as it does not partition. For quality-aware rq and maxError = 0.1, the execution time is an order of magnitude worse than the one of naive rq. For maxError = 1, the execution times are the same. For a bigger maxError, quality-aware rq improves up to 1.5 orders of magnitude over naive rq and reaches the best value at maxError = 50. After that the execution times slightly rise again.

Figure 11.3 shows the average number of partitions created. For a small maxError, we observe that a considerable number of partitions is created, e.g., 1000 partitions for maxError = 1. If we increase maxError, the number of partitions decreases sharply to about 8 partitions at maxError = 50. Due to these results, we chose maxError = 50 for the experiments in Chapter 14.

11.3.2 Query-Method Comparison

The goal of these experiments is to understand the trade-offs of the different query processing techniques. Figures 11.4 and 11.5 show the cost for querying 6.4 million data entries. As determined above, we use maxError = 50. The curves show three different methods:
11.3. EXPERIMENTS

Figure 11.3: Number of Created Partitions while Varying Maximum Error.

1. **scan**: a simple scan of non-zcoded entries with a post-filter.

2. **naive rq**: unpartitioned range query as described in Section 11.1.

3. **quality-aware rq**: quality-aware range query as described in Section 11.2.

Figures 11.4 and 11.5 vary the query window size from $1,000 \times 1,000$ m to $10,000 \times 10,000$ m. The two figures show that the query performance of the **scan** method is independent of the window size. Furthermore, as no z-codes are used, it is also independent of the z-code precision and remains constant at 0.05 seconds. Figures 11.4 and 11.5 also show query methods (2) and (3). For small and big query windows these methods outperform **scan** by up to three orders of magnitude. Both figures illustrate a similar tendency: **quality-aware rq** is at least as fast as **naive rq**. For a certain $w \geq w_1$ (e.g., $2w_1 = 16$ in Figure 11.5), **quality-aware rq** outperforms **naive rq** by one order of magnitude. These results show the expected trade-off: For longer z-codes, query execution time decreases whereas indexing time increases. Thus, the length of a z-code may be used to trade indexing performance for query performance in a fine-granular manner.

Another effect that can be observed from these experiments is that the magnitude of the differences among the three methods decreases when we use large query windows, as in Figure 11.5. For small query windows, the query cost tends to be dominated by the cost of index probes, which are optimized by our quality-aware range querying strategy. In contrast, the query cost for large query windows is dominated by the enumeration of all the results that match the window. Consequently, as query windows increase, the intrinsic cost of query processing super-linearly approaches the scan cost upper bound.
Figure 11.4: Query Performance \( [qw = 1,000m \times 1,000m] \).

Figure 11.5: Query Performance \( [qw = 10,000m \times 10,000m] \).
11.3. EXPERIMENTS

11.3.3 Query-Method Scalability

The goal of this experiment is to understand the scalability of different query processing techniques. We evaluate the scalability of the different query methods w.r.t. the index size by scaling it up to 25.6 million elements.

Figure 11.6 shows that quality-aware rq executes 41,200 queries per second for 25.6E6 elements given a query window size of 1,000 m × 1,000 m. This is more than three orders of magnitude better than scan and 350 times better than naive rq. For qw =10,000 m × 10,000 m (Figure 11.7), we again observe that the performance differences between the methods diminishes, as query processing cost gets increasingly dominated by the cost to enumerate the results that match the query window. Nevertheless, the figure shows that scan has very high cost (up to 0.15 seconds), whereas the best query based method quality-aware rq needs at most 0.0026 seconds to query 25.6E5 elements. This is more than two orders of magnitude faster than scan and 45 times better than naive rq.

As we see in Figure 11.7, quality-aware rq achieves query processing costs in the range of milliseconds even for large query windows. This result warrants an additional observation: The use of such an optimized query processing method in main memory may lead us to revisit techniques from the literature for query optimization of spatial queries [80]. For example, selectivity estimation will only pay off for queries that execute in such little time if it can be made to fit into microseconds.

For the remainder of this paper, we will always use quality-aware range querying. In addition, we restrict our attention to queries of size 1,000 m × 1,000 m.
Figure 11.7: Query Scalability \[qw = 10,000 \text{ m} \times 10,000 \text{ m}\].
Chapter 12

Time-Parameterized Query Processing

In this chapter, we discuss how to evaluate time-parameterized queries. A time-parameterized query $Q(t_q)$ asks for object positions in a given range at a given time $t_q$:

$$Q(t_q) = ([x_{low}; x_{high}] \times [y_{low}; y_{high}])$$

In order to answer these queries, we must transport objects to their predicted positions at time $t_q$. This can be done at indexing time, which we term Predictive MOVIES (PI MOVIES), or at querying time, which we term Non-Predictive MOVIES (NPI MOVIES). Both strategies work for both Logged and Aggregated MOVIES resulting in a total of four different combinations.

Section 12.1 and Section 12.2 will explain the two approaches in more detail. Experiments comparing the four different combinations of MOVIES will follow in our main experiments section in Chapter 14.

12.1 Predictive MOVIES

This section introduces the algorithm Predictive MOVIES (PI MOVIES).

12.1.1 Indexing Strategy

For each index build, we index all data w.r.t. a single point in time $t_{\text{index}} > t_u$. We term $t_{\text{index}}$ the index time. Thus, for every incoming update $u$ we index the moving object at a predicted position $(x, y) + \vec{s}(t_{\text{index}} - t_u)$. Here, we may avoid any extra storage space by translating objects immediately when an update arrives.
However, for each incoming update we have to compute the predicted position—which may be costly. After that, the timestamp for the update may be dropped. If during fallback an object is encountered that has not received an update (Line 22 of the buildIndex algorithm), that object is simply translated to a new position using the new index time.

12.1.2 Query Strategy

We follow the strategy of enlarging query windows to account for the difference between query and index time [34]. As $t_q$ may be either larger or smaller than $t_{\text{index}}$, we have to consider three cases:

1. $t_q < t_{\text{index}}$: the objects have to be translated to an earlier time,
2. $t_q > t_{\text{index}}$: the objects have to be translated to a later time,
3. $t_q = t_{\text{index}}$: the objects do not have to be translated.

For cases (1) and (2), we rewrite $Q(t_q)$ to consider the maximum distance $\varepsilon$ an object may have traveled relative to the index time.

$$\varepsilon = S_{\text{max}} \cdot |t_q - t_{\text{index}}|$$

Every $Q(t_q)$ is rewritten to $Q(t_q)'$ and is then postfiltered w.r.t. $t_q$ and their respective speed vectors $\vec{sv}$ as obtained from the index.

$$Q(t_q)' = [x_{\text{low}} - \varepsilon; x_{\text{high}} + \varepsilon] \times [y_{\text{low}} - \varepsilon; y_{\text{high}} + \varepsilon]$$

12.1.3 Choice of Index Time

One important decision for Predicted MOVIES is which index time to choose. Let

$$\bar{Q} = \{Q_1(t_1^q), \ldots, Q_k(t_k^q)\}$$

be the set of queries arriving during a single indexing frame of MOVIES. To minimize query enlargements and therefore the performance penalty for large range queries, we wish to minimize the sum of query window areas added due to enlargement:

$$\sum_{i=1}^{k} |t_i^q - t_{\text{index}}|^2$$

Assume, for simplicity, that all queries ask for a fixed time into the future, i.e., all queries ask for $t_{\text{now}} + \Delta t$. 
Now, for every index it holds, that after being rebuilt, an index receives queries during one frame time $T_{\text{frame time}}$. To produce balanced query enlargements, the index time should be $\Delta t + T_{\text{frame time}}/2$ ahead of the time the index is ready for querying. In order to achieve that, we must set the time of the update buffer that will be used to build a new index appropriately. An update buffer is used to collect updates two frames before being used to build a new index (see Algorithm 1 and Figure 9.3). Therefore, the index time to be used for an update buffer collecting updates in the next frame should be set to

$$t_{\text{index}} = t_{\text{now}} + 2T_{\text{frame time}} + (\Delta t + T_{\text{frame time}}/2)$$

$$= t_{\text{now}} + 2.5T_{\text{frame time}} + \Delta t$$

### 12.2 Non-Predictive MOVIES

This section introduces the algorithm *Non-Predictive MOVIES*.

#### 12.2.1 Indexing Strategy

For each index build, we index each index entry w.r.t its timestamp $t_u$. In order to do this, we need to keep for each update its corresponding timestamp. Thus, we require slightly more storage space (See Figure 12.1), but do not have to compute predicted positions at indexing time.

#### 12.2.2 Query Strategy

We use query window enlargement as with Predictive MOVIES. However, as objects are indexed with different last update times, we must perform query enlargement with respect to the time of the oldest update considered in the index, i.e.,

$$\varepsilon = S_{\text{max}} \times |t_q - t_{\text{min}}|$$
As calculating $t_{\text{min}}$ by scanning all update timestamps has prohibitive cost, we provide a bound on $t_{\text{min}}$. When an update arrives, it takes at most two times the frame time $T_{\text{frame time}}$ for it to appear in the index used for querying (see Algorithm 1 and Figure 9.3). As an update for each object must arrive within $t_{\Delta_{\text{max}}}$, then

$$t_{\text{min}} \geq t_{\text{now}} - (t_{\Delta_{\text{max}}} + 2T_{\text{frame time}})$$
Chapter 13

MOVIES Parallelization

Most moving object applications, such as flight control, can tolerate staleness as long as it is kept below a few seconds [73]. In order to keep staleness within these constraints, MOVIES takes advantage of the parallelism available in modern hardware architectures. We use both intra-node and inter-node parallelism to speed up MOVIES frames. To exploit intra-node parallelism, we implement update, index merge, and sorting with multiple threads. For inter-node parallelism, we leverage data partitioning to limit the size of the index partitions allocated to each processing node in a commodity cluster of shared-nothing machines.

13.1 Intra-node Parallelization

This section discusses how to effectively parallelize MOVIES for a single computing node. The simplest strategy to parallelize MOVIES in a single multi-processed node is to apply the exact same strategy developed for inter-node parallelization and run several virtual MOVIES instances [76]. Instead, we develop below a strategy better tailored to our method.

13.1.1 General Trade-Off

In general, one may either (1) increase query performance or (2) reduce staleness by increasing reindexing performance. (1) amounts to adding additional query threads. As these threads are read-only, no locking is required. (2) amounts to adding more threads to the indexing process, i.e., by parallelizing the sorting and the index merge process of Algorithm 2. We tuned our implementation of MOVIES experimentally for best performance on our hardware (see Section 14.1.1). The best variant we found uses two threads to process queries, one thread to receive and process updates, and four threads for index merge and sorting.
13.1.2 Parallel Update and Index Merge

The update merge (Lines 7–15, Algorithm 2) as well as the optional index merge (Lines 15–21) are changed to partition the OIDs into disjoint equal-sized domains. Each domain is handled by a separate thread each processing the same load. This method does not require any locking on OIDs.

13.1.3 Parallel Sorting

The sorting in Line 24 (either radix- or quicksort) is improved to partition the array of size $N$ in-place using a single thread into two partitions based on a radix partitioning. Each partition is then treated recursively by a separate thread (resp. by the same thread if no more threads are available). This scheme scales almost linearly in the number of CPUs for a small number of CPUs (see Section 10.3.2). For a larger number of CPUs other sorting algorithms may be used [71].

13.2 Inter-node Parallelization

This section discusses how to effectively parallelize MOVIES for multiple computing nodes. Experiments which scale the number of processing nodes are shown in the next chapter in Section 14.3.

13.2.1 OID-partitioning

If the number of moving objects is too high to be handled by one machine, the data may be horizontally partitioned based on OIDs using hash or range partitioning. Thus, if we distribute the data to $G$ processing nodes, the number of elements on each processing node will be at most $\lceil N/G \rceil$. In turn, each processing node will have less indexing cost. Furthermore, query result staleness will decrease. Updates may be partitioned by a separate machine or by the network itself, routing updates based on their OID to the appropriate machine, i.e., each update has to be sent to exactly one machine. Queries, however, have to be sent to all processing nodes computing subsets of the query result, which are then unioned. Our current implementation uses this strategy. It is both simple to implement and suited to a scenario in which we wish to achieve scalability in the number of updates, exactly the main requirement of a moving objects application.

13.2.2 Spatial Partitioning

Scalability on data size, number of updates, and number of queries calls for partitioning data by region. Each region would then be handled by a separate pro-
cessing node. This strategy requires more effort than OID-partitioning, as moving objects may have to be redistributed from one node to another while traveling. On the other hand, queries only have to be sent to the nodes overlapping the query window. A challenge here is to keep at the same time both balanced and disjoint partitions. This cannot be achieved with a static partitioning, as some regions will contain much more data than others. A viable strategy would be to use a coarse-granular seeded R*-tree [4, 50] that partitions the data into regions of a target size $\approx \lceil N/G \rceil$ moving objects. Each of these regions could then be assigned to a separate processing node running MOVIES. We leave this strategy to be explored as part of future work.
Chapter 14

Experiments

In this chapter, we study the performance of MOVIES using a large, realistic dataset. The scenario we target is “Index all cars in Germany”. We used a commercial dataset which contains the complete road network of Germany and simulate cars traveling along these roads.

We explore various aspects of our approach and compare it with state-of-the-art approaches. The goals of our experiments are:

1. Determine the maximum supported update rate of MOVIES when scaling the index size (Section 14.2.1).
2. Determine query throughput of MOVIES when scaling the update rate (Section 14.2.2).
3. Determine query throughput of MOVIES when scaling the update rate and scaling the index size. (Section 14.2.3)
4. Determine the main memory consumption of MOVIES (Section 14.2.4).
5. Determine the performance of MOVIES when implemented on a cluster of shared-nothing machines (Section 14.3).

Section 14.1 presents the experimental setup in more detail. Section 14.2 and Section 14.3 contain the experiments.

14.1 Experimental Setup

This section presents the setup used for our experimental analysis of MOVIES.
14.1.1 Hardware and Implementation Parameters

All experiments were performed on servers having each two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e., four cores in total, and 6 GB of main memory. We used two separate servers, M1 and M2, to generate updates and queries. These updates and queries were sent over the network and received by servers termed processing nodes (PN1–PN4) that did the actual indexing. M1 and M2 were each connected to the switch by one network cable and in addition with one network cable each directly to PN3 (PN4 respectively). All network links supported 1 Gbit/s. For the single instance experiment, we used one processing node PN1. For the parallelization experiment, we used up to four processing nodes PN1–PN4. All code used for the experiments was implemented in Java 5. In our implementation, we avoided complex object types whenever possible and used primitive Java types. We also avoided Java generics as current Java implementations do not translate generics into efficiently typed code. To make maximal use of the four cores provided by each machine, we implemented a multi-threaded variant of MOVIES. For the experiments in Sections 14.2.1 to 14.3 evaluating MOVIES, we waited until at least 8 re-indexing phases were completed. Then we measured at least 10 re-indexing phases and report the average.

14.1.2 Data and Queries

Our experiments were inspired by the scenario ‘index all cars in Germany’ which comprises 58M cars [43]. We obtained a commercial data set containing the complete road network of Germany [83]. This data set consists of 38 million nodes and 40 million road segments. The geography of Germany covers 640 km x 863 km. We assumed cars to travel at a maximum speed $S_{\text{max}} = 60 \text{m/s} = 216 \text{km/h}$. As in similar studies [54, 55, 63], we initially used the Brinkhoff moving object generator [10]. However, it turned out that that generator does not scale for the massive workloads considered in this paper. In particular, large number of nodes and road segments are not possible. Therefore, we developed our own generator based on the ideas of Brinkhoff [10]. This means that we used the same moving object placement strategy network-based approach (NB). It places cars using the same skewed distribution as the roads and nodes themselves. We then moved cars on the network by assigning each car a constant random direction that it would try to follow on the network. This avoids the overheads of doing Dijkstra-computations for each car. We also allowed cars to share similar itineraries on the network at different times. This generates similar traces as in the original generator [10], but at much lower cost. Our trace generator is open source and available from sourceforge at http://moto.sourceforge.net/. If not mentioned otherwise, a data set of 6.4 million moving objects was used on MOVIES experiments with a single processing node.
14.2 MOVIES Experiments

We compare MOVIES to baseline index structures, including binary search trees and B+-trees. Furthermore, we compare MOVIES against a state-of-the-art moving object index: the B*-tree [34].

As all tree-based methods are hard to parallelize without considerably sacrificing performance (locking), we parallelized all tree-based methods to obtain lock-free methods as follows: We partitioned the data by OID into four disjoint partitions, and used a separate tree and thread to index each partition as suggested by Stonebraker et al. [76]. Thus, the tree-based methods could make maximal use

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>index size $N$</td>
<td>$100,000 \ldots 6,400,000 \ldots 100,000,000$</td>
</tr>
<tr>
<td>update rate $V$</td>
<td>$0 \text{ /s} \ldots 58,000,000 \text{ /s}$</td>
</tr>
<tr>
<td>query rate</td>
<td>$0 \text{ /s} \ldots 1,000 \text{ /s} \ldots 10,000 \text{ /s}$</td>
</tr>
<tr>
<td>query window size $qw$</td>
<td>$1 \text{ km} \times 1 \text{ km} \ldots 10 \text{ km} \times 10 \text{ km}$</td>
</tr>
<tr>
<td># road network segments</td>
<td>39,509,805</td>
</tr>
<tr>
<td># road network nodes</td>
<td>37,967,339</td>
</tr>
<tr>
<td>data region</td>
<td>$640 \text{ km} \times 863 \text{ km}$</td>
</tr>
<tr>
<td>index granularity</td>
<td>$26.3 \text{ m} \times 26.3 \text{ m}$</td>
</tr>
<tr>
<td>$S_{\text{max}}$</td>
<td>60 m/s</td>
</tr>
<tr>
<td>$t_{\Delta_{\text{max}}}$</td>
<td>$\leq N/V$</td>
</tr>
</tbody>
</table>

Table 14.1: Settings Used in the Experiments.

For scalability experiments on a single processing node, we used up to 25 million moving objects (raw size = 200MB). For the parallelization experiment, we used up to 100 million moving objects (raw size = 800MB). As outlined in the Introduction, we have placed all data and indexes for all methods into main memory. We evaluate time-parameterized predictive range queries. Note again that other query types, such as time-parameterized predictive k-nearest-neighbors, may be inferred from predictive range queries (see [34]). We used a query window size corresponding to the size of a small town center like Oldenburg, which has an extension of roughly $1,000 \text{ m} \times 1,000 \text{ m}$. Bigger query windows did not substantially change our results. We report on experiments with a large query window of $10,000 \text{ m} \times 10,000 \text{ m}$ in our raw query performance evaluation of Section 11.3.2. Query centers were chosen using the NB strategy [10], thus creating a skewed distribution on queries. If not mentioned otherwise, we set a query rate of 1,000 queries per second and a predictive query time $t_q = t_{\text{now}} + 1.5t_{\Delta_{\text{max}}}$. Table 14.1 summarizes the settings; Figure 14.1 shows the road network used in the experiments.
Figure 14.1: Road Network of Germany.
14.2. MOVIES EXPERIMENTS

14.2.1 Scalability in Index Size

The goal of this experiment is to understand the maximum update rates supported by MOVIES when scaling the size of the data set.

Figure 14.2 shows the results of a scalability experiment where the index size is varied up to 25.6 million moving objects. We kept a fixed query rate of 1,000 time-parameterized queries/s and display the maximum update rate supported by each indexing method. The results show that both variants of MOVIES outperform all other methods. Figure 14.2 shows that all tree-based methods degrade sharply with growing index sizes. The binary search tree was not able to scale beyond 3.2 M objects as then it could not meet the query rate anymore. For the B+-tree, we experimented with several values of $k$ and $k^*$. However, the best B+-tree we could devise ($k = k^* = 16$) was not able to scale beyond 12.8M moving objects. For 12.8M the B+-tree could only handle 0.6M updates/s. Similarly, for the B*-tree we
performed a separate experiment varying the number of phases $n$ and display only the best version of the $B^x$-tree we could devise ($k = 16$ and $n = 2$). Interestingly, the $B^x$-tree was also not able to scale beyond 12.8 M moving objects. The $B^x$-tree even performed slightly worse than the best $B^+$-tree except for $N=12.8$ M. This is due to the fact that the $B^x$-tree incurs overhead compared to the $B^+$-tree as it has to compute predictions for each incoming update at indexing time.

In the experiment, in contrast to all other methods, MOVIES Logged with Non-Predictive Indexing (NPI) shows update rates of around 14 million updates/s for index sizes up to 25.6 M objects. This value is close to the network limit of 14.7 million updates/s. MOVIES Aggregated NPI shows on average a slightly smaller update rate ranging from 11 M to 14 M updates/s. However, MOVIES Aggregated NPI still performs better than all tree-based methods. For an index of size of 12.8 M objects, the improvement of the best MOVIES variant over the best $B^+$-tree and $B^x$-tree is of a factor of 15. For an index size of 25.6 M only MOVIES was able to index the data meeting the query rate. Interestingly, in this experiment the binary search tree performs even better for small indexes than the $B^x$-tree. This is due to the high cost for computing predictions for each incoming update. This is also evidenced when we compare the four different variants of MOVIES using different indexing methods.

Figure 14.3 displays the update throughput for MOVIES when using different indexing schemes, i.e., either Aggregated or Logged MOVIES (Section 9.3), and either non-predictive (NPI) or predictive indexing (PI) (Chapter 12). The figure shows that Logged MOVIES NPI has the best performance. In contrast, Logged MOVIES PI achieves only half of the throughput. This is due to the fact that for predictive indexing each update has to be translated to a new position. This is
14.2. MOVIES EXPERIMENTS

CPU-intensive and also explains why the B*-tree performs worse than a standard B+-tree: At extreme update rates, the computational cost of predictions offsets the gain obtained by smaller query window enlargements.

Figure 14.4 shows the average staleness observed in the scaling experiment. The results show that the staleness grows for larger index sizes. That is expected as an important component of frame time is the time to sort the data in a new index. For MOVIES Logged NPI the staleness grows up to 19 sec, for MOVIES Aggregated NPI it increases up to 7 sec. If the staleness has to be reduced, this can be achieved by scaling out on multiple processing nodes. This is explored in Section 14.3.

14.2.2 Scalability in Update Rate

The goal of this experiment is to understand the maximum query rate supported by MOVIES when scaling the update rate. We keep the index size constant at 6.4M objects and vary the update rate. Figure 14.5 shows the result.

The figure shows that the binary search tree is only able to sustain a very low query rate. For update rates above 2.1M updates/s this method is not able to execute any more queries and thus fails to scale beyond this point. Similarly, we observe that for update rates between 0.1M to 1M the best B+-tree is able to execute between 3,000 and 2,000 queries/s, respectively. For higher update rates, however, the B+-tree degrades sharply: For an update rate of 4M updates/s, the best B+-tree is only able to execute a small amount of queries and thus fails to scale beyond this point. The B*-tree has better query performance than the B+-tree for update rates up to about 3M updates/s, but also fails to scale beyond 4M
The MOVIES variants show an interesting behavior: The predictive variants outperform the non-predictive variants in terms of query performance up to an update rate of 4M updates/s, exhibiting query rates around 3,000 queries/s. Above 4M updates/s, however, the query performance of the non-predictive variants sharply increases up to 9,200 queries/s. This behavior can be explained by analyzing the trade-off between indexing predictions and performing query window enlargements. For modest update rates, non-predictive methods must perform bigger query enlargements to compensate for the relatively large value of $t_{\Delta_{\text{max}}}$ These enlarged queries impose significant computational overhead. Predictive methods, on the other hand, aggressively reduce query window enlargements by computing predictions for each update applied to the index. At high update rates, however, we observe the opposite effect: Predictive methods pay a high computational cost for predicting every update applied to the index. The gain in query window enlargements is not enough to offset these costs, because as $t_{\Delta_{\text{max}}}$ is relatively low, the enlargements performed by non-predictive methods are also relatively small. In addition, non-predictive methods have lower computational cost for collecting updates.

Another effect may be observed for the non-predictive variants: At an update rate of about 10M updates/s, the query rate drops to around 6,000 queries/s. This slight drop in the query rate may be explained by the fact that at very high update rates, the cost to collect updates starts to become significant, draining CPU resources from both query processing and index rebuilding.

The staleness for Logged MOVIES NPI stayed constant around 3 sec up to 7M updates/s. For higher updates rates it increased linearly up to 7.2 sec. For
Aggregated MOVIES NPI the staleness was constant around 2.5 sec. The predictive variants could not be scaled beyond 8M updates/s and their average staleness stayed between 2 to 3 sec. The relatively high staleness for low update rates can be explained as follows: If during one index rebuild MOVIES receives only few updates, then MOVIES has to retrieve the old data for many objects from the old index. This leads to many random accesses to the old index and therefore hurts rebuild performance. The time needed to look up old data goes down as the update rate increases and reaches zero around 5M updates/s. Even though this effect would lead to decreasing staleness, the staleness stays about constant, because processing the updates becomes more expensive. In summary, this experiment shows that the MOVIES variants scale well for high update rates. Of all methods, only MOVIES was able to scale up to 14M updates/s. Note again that all methods completely resided in main memory.

14.2.3 Scalability in Update Rate and Index Size

The goal of this experiment is to understand the maximum query rate supported by MOVIES when scaling both the update rate and the index size.

So far we only varied either index size or update rate. In the following we vary both. This experiment measures the maximum query rate and varies index size and update rate simultaneously. The update rate is always set to index size divided by 2, i.e. for an index size of 100,000 we use an update rate of 50,000 updates per second. So we model a scenario in which cars report their positions on average every two seconds.

As expected from our discussion in Section 14.2.2, for a given value of $t_{\Delta \text{max}}$, 

Figure 14.6: Maximum Query Rate when Scaling Update Rate and Index Size.
either predictive indexing or non-predictive indexing dominates, due to the trade-off between prediction costs for each update versus window enlargements for each query. In addition, we have seen in previous sections that MOVIES largely dominates tree-based methods. These observations are confirmed by Figure 14.6. The predictive variants of MOVIES are the dominant methods overall in this experiment, given that $t_{\Delta_{\text{max}}}$ is kept roughly constant. These variants achieve up to a factor of two to three better query performance than the best tree-based method. For all methods, the maximum attainable query rate decreases with scenario size, as in larger scenarios more of the server’s resources are dedicated to update processing and index rebuilds.

14.2.4 Main Memory Consumption

The goal of this experiment is to understand the main memory consumption of the different methods.

Figure 14.7 shows the main memory consumption for the different methods when varying the index size. The results show that the memory consumption for the Logged MOVIES variants are about 3.5 GB. This is due to the fact that our current implementation of Logged MOVIES tries to allocate as much main memory for buffers as it can in order to support a maximum update rate. This, however, could be improved by not letting Logged MOVIES allocate all available memory. Our method would still work then, but would support lower update rates. Several strategies could be used to limit the buffers employed to collect updates. These strategies will be presented elsewhere.

In contrast, the Aggregated MOVIES variants have considerably less memory
requirements of about only 1 GB for NPI and 0.7 GB for the PI variant when the index size is 25.6M elements. The binary search tree has very high main memory requirements of about 3.7 GB for an index size of 12.8M. It even fails to scale to 25.6 M due to excessive main memory requirements. The $B^+$-tree requires up to 1.1 GB for an index of 25.6 M. This is 10% more than Aggregated MOVIES NPI which requires 1 GB. The $B^+$-tree requires up to 0.9 GB. Aggregated MOVIES PI was the best method as it required only up to 781 MB. This is because predictive indexing obviates the need to store timestamps for each update. In summary, this experiment shows that the memory requirements of the best MOVIES method are comparable (or sometimes less) than the tree-based methods even though we keep two indexes and two update buffers.

14.3 Shared-Nothing Scale-Out

The goal of this experiment is to examine how MOVIES scales when increasing the number of processing nodes. In order to adapt the different methods to a shared-nothing landscape, we horizontally hash-partitioned the data by OID. We keep the index size constant at 25.8M and vary the number of processing nodes PN from one to four. As our experiments with a single processing node have shown, the transfer limit imposed by the network is a serious bottleneck. Therefore, we required a special network setup as described in Section 14.1.1. With that setup we could transfer up to 58M updates/s to four processing nodes while still being able to distribute queries. This corresponds to roughly linear scalability in the transfer limit, as shown in Figure 14.8. This linear scalability of the transfer limit comes from each processing node having a separate network card to communicate with the generating nodes. Note that this scaling may be limited in larger setups by the ability of the network switch to distribute network packets to several nodes at maximum bandwidth.

14.3.1 Maximum Update Rate

Figure 14.8 also compares how the variants of MOVIES scale with increasing numbers of processing nodes. The NPI MOVIES variants Aggregated and Logged scale up to 47M and 54M updates/s respectively. This is explained as follows: As observed above for a single MOVIES index, it holds that the smaller the index the closer we get to the network limit. As we keep the index size constant and increase the number of processing nodes, we see two effects: first, the benefit of having smaller indexes on each processing node (see Figure 14.1 and Section 14.2.1), and second, the linear scale-up due to data partitioning; hence the super-linear scale-up. Figure 14.8 also shows that the NPI variants perform consistently better than
Figure 14.8: Maximum Update Rate when Scaling to Four Processing Nodes.

Figure 14.9: Average Staleness when Scaling to Four Processing Nodes.
Figure 14.10: Maximum Update Rate on Four Shared-Nothing Servers.


14.3.2 Shared-Nothing Scalability in Index Size

In another experiment, we used all four processing nodes for indexing. Figure 14.10 shows the results. Similarly to the single instance experiment, MOVIES outperforms all other methods. All tree-based methods, including the $B^\times$-tree, degrade sharply for growing index sizes. The tree-based methods fail to scale beyond an index of size 51M, i.e., 12.8M moving objects per processing node. In contrast, MOVIES scales up to 102M moving objects. Furthermore, for index sizes up to 51M, Logged MOVIES sustains an update rate close to the network limit of 58M updates/s. For 51M moving objects, the improvement of MOVIES over the best $B^+$-tree is factor 15; the improvement over the best $B^\times$-tree is factor 11. The average staleness of all the MOVIES variants is the same as shown for the single instance experiment, but the index size is four times larger (see Figure 14.4). For example, the staleness of MOVIES Logged NPI is 21 seconds for an index with 102M elements.
Chapter 15

Possible Extensions of MOVIES

This work is not meant to close the book on short-lived index images, but rather opens the book for several important extensions of our approach. We list below a number of extensions that are possible interesting avenues for future work:

15.1 Algorithmic Extensions

In this section we look at extensions to our core algorithm. E.g. the index organization, the merge strategies, or support for adding and removing moving objects.

15.1.1 Fuzzy Aggregated MOVIES

A variant of keeping two aggregation indexes is to keep only one fuzzy aggregation index. Thus in total we keep three indexes. This works as follows: All incoming updates are at all times directly written to an aggregation index $\hat{I}_1$ organizing items based on their OID. At the beginning of Frame 1, index $\hat{I}_1$ is copied to index $I_2$. Then $I_2$ is created by inverting $I_1$ on the appropriate key. At the beginning of Frame 2, aggregation index $\hat{I}_1$ is copied to index $I_1$. Then $I_1$ is created by inverting $I_2$ on the appropriate key. This process is repeated until the flag should_terminate is set to true. In contrast to Logged and Aggregated MOVIES, Fuzzy Aggregated MOVIES requires item-level locking on index $\hat{I}_1$, as the index copy and update operations may access the same entry of $\hat{I}_1$ concurrently. An additional drawback of this approach is that no copy of $\hat{I}_1$ will be consistent with respect to a point in time $t_i$ anymore. This is due to the fact that $\hat{I}_1$ is modified concurrently during the non-atomic copy operation. Thus if the copy operation takes $t_\delta$ seconds and starts at time $t_i$, it follows that only a subset of the updates arriving in the time interval $[t_i; t_i+t_\delta]$ will be considered in buildIndex.
15.1.2 Velocity-Dependent MOVIES

Moving objects may be partitioned into groups based on quality of service (QoS) requirements of an application. For instance, moving objects may be grouped by velocities, e.g., planes go into a small MOVIES index with fast index build (milliseconds), high-speed trains go into a medium-sized MOVIES index with frequent builds (subseconds), cars go into a big index with infrequent builds (seconds). As query rewrites have to consider the maximum velocity $S_{max}$ of all moving objects (see Section 9.1.1), a MOVIES index for cars would rewrite queries with a much smaller value of $S_{max,cars}$ than a MOVIES index for airplanes.

15.1.3 Adaptive Index Rebuild

If the distribution of updates is highly skewed in time, e.g., at certain time intervals many updates arrive, whereas at other time intervals only few or even no updates arrive, it may be counterproductive to rebuild the index during certain intervals. For these situations, it may make more sense to start the next index build phase only after a minimal number of updates have arrived. Thus, instead of performing as many frames as possible, we may pause after each frame. The decision whether to start the next frame could be based on QoS-requirements of the application, e.g., desired query throughput and maximum allowed query staleness.

15.1.4 Keeping Multiple Index Frames

As mentioned above, for cases where $t_{\Delta max}$ is slightly bigger than the reindexing phase, it could make sense to still skip the merge with the previous index. Correctness is then obtained by keeping $k = \lceil t_{\Delta max}/T_i \rceil$ read-only indexes. Queries will be sent to all of these indexes. Query results obtained from a more recent index will then supersede results from older indexes.

The drawbacks of this variant are higher memory requirements due to more indexes as well as additional costs at query time for querying multiple indexes. The advantages are that we save the cost for merging with an old index when only a small subset of the information has not been updated.

15.1.5 Supporting Insertions and Deletions

The core method presented in Section 9.2 handles continuous updates emitted by moving objects. We can extend the method to deal with changes in the moving object population itself, i.e., to support not only position updates but also insertions and deletions of moving objects.
First, we should store insert and delete operations in the update buffers used by MOVIES. During the merge algorithm, we can incorporate these operations into the new index being built from scratch.

However, there may be some challenges in either collecting updates or performing the merge that stem from the representations chosen for indexes and update buffers. For example, in Aggregated MOVIES we represent OIDs implicitly in update buffers organized as arrays. In order to deal with fragmentation resulting from deletions, we can keep a free list of OIDs to be reused in subsequent insertions.

### 15.2 Indexes

In this section we look at modifications to the indexes. For example, would index compression be beneficial or how to use cache-optimized index structures. We also look at changes to MOVIES if we want to support more-dimensional data.

#### 15.2.1 Index Compression

At the moment we are using a lightweight compression scheme, i.e. inlined z-codes. However, it would make sense to try out other compression techniques. As a consequence, more data could be indexed by a single machine. In addition, the bandwidth between main memory and CPU could be used more effectively. Our algorithms may also be extended to directly work on compressed data.

#### 15.2.2 Cache-Optimized Indexes

In addition to index compression, we can also exploit the fact that we rebuild indexes every frame to create data structures that are highly optimized for queries, e.g., by making indexes cache-aware. One simple extension to our method would be to use a CSS-Tree to store z-coded entries instead of simply a sorted array [67]. Hildenbrand points out that this choice of representation may bring a factor of two gain in query processing performance at almost no additional build cost [30]. The same types of optimizations would be less helpful to update-in-place approaches. For example, we would have to employ a CSB++-Tree instead of a read-optimized CSS-Tree for update-in-place methods [68].

#### 15.2.3 Extension to High-Dimensional Data

Up to now we assumed that moving objects send two-dimensional \((x, y, s_v, OID)\) updates to the server. We now assume that moving objects are positioned in a
$d$-dimensional space and send updates of the form $(x_1, \ldots, x_d, \vec{sv}_d, OID)$ tuples to the server where $\vec{sv}_d$ is a $d$-dimensional speed-vector.

Algorithmically no changes to MOVIES are required to support this data. Algorithms 1 & 2 are used unchanged. However, a few adaptations w.r.t. the data layout are required.

1. The z-code indexing and in-place z-codes need to be extended to consider additional dimensions. In particular, a multi-dimensional partial in-place z-code is defined as follows:

**Definition 5** ($d$-dimensional Partial In-place z-code). The $d$-dimensional partial in-place z-code of order $w \leq n$ of a tuple $(x_1, \ldots, x_d)$ is $\langle x_1, 0 \ldots x_1, n-1, x_d, 0 \ldots x_d, n-1 \rangle$ where $x_{i,j}$ represents the $j$-th bit of the $i$-th dimension.

2. Definitions for $d$-dimensional partial reverse z-codes and precomputed snippets for $d$-dimensional linearizations may be extended analogously.

3. In terms of data layout, we so far considered a 64-bit entry and update size. As more dimensions require more space, a longer entry will be required for higher dimensions to accommodate both the longer position and speed vector.

4. Similarly, the query processing techniques presented above are adapted by considering additional dimensions. Our query processing techniques work on the linearized data proceeding bitwise from the leftmost to the rightmost bit. Therefore, a higher dimensionality will not effect the processing strategy.

## 15.3 Recovery

In this last section of extensions, we look at different recovery methods.

### 15.3.1 Recover by Wait

The simplest strategy to recovery is to not provide any mechanisms for recovery. Instead, as any moving object will report its new positions within at most $t_{\Delta_{\text{max}}}$ seconds, we proceed as follows. After a processing node has been restarted, we simply initialize all entries to empty and collect updates for $t_{\Delta_{\text{max}}}$ seconds. After that, MOVIES performs its first reindexing phase to obtain a read-only index. As soon as this indexing phase is finished, MOVIES is guaranteed to be in a consistent state.
15.3.2 Recover by System-based Redundancy

An alternative approach to recover would be to use system-based redundancy, e.g., we could set up two indexing servers that index the same data concurrently. If one of the servers fails, the other could request a copy of the data from the other server. This was recently evaluated by Lau and Madden [44].

15.3.3 Log-based Redundancy

As we receive update entries with the most recent position of an object, we could simply log these entries to disk and in case of recovery interpret them as redo-log entries. Note that as we assume that every moving object sends an update at least every $t_{\Delta_{\text{max}}}$ seconds, we only have to keep a log of updates that arrived in the past $t_{\Delta_{\text{max}}}$ seconds. Therefore the log of updates on disk may be organized as a ring-list constantly overwriting entries that are older than $t_{\Delta_{\text{max}}}$ seconds. Note that this approach may limit update processing unless disk bandwidth is large enough. In addition, it only makes sense if the time to replay the log from disk is expected to be smaller than $t_{\Delta_{\text{max}}}$.
Chapter 16

Related Work

Considerable work has been done in the area of moving objects. The existing methods can be classified into two groups: methods with or without time-parameterized (TP) queries. General design issues for moving object indexes can be found in [58].

16.1 With Time-Parameterized Queries

16.1.1 External Memory

Many approaches are centered around extending external memory structures such as the B$^+$-tree, R-tree[27], or R*-tree[4]. All of these methods assume that data would not fit into main memory. Examples include the TR-tree and TB-tree [64], the TPR-tree [69], the TPR*-tree [79], the STP-tree [78], and the R$^{PPF}$-tree [63].

The most relevant work to our work is the B$^x$-tree [34] as, conceptually, it has some similarities to the MOVIES indexing strategy. The core idea of the B$^x$-tree is to map three-dimensional data (two spatial and one temporal dimension) to a one-dimensional space. This is done by using a recursive space-filling curve and mapping data to a B$^+$-tree very similarly to [60]. However, in contrast to the latter approach, the B$^x$-tree also partitions data into phases corresponding to future time intervals. For each phase, it uses a separate subtree to index moving objects and predicted positions. As a consequence, prediction queries are supported. As the B$^x$-tree is based on a B$^+$-tree, it is very easy to integrate it into existing database management systems. The B$^x$-tree was shown to outperform competing methods such as the TPR-tree [69]. However, in contrast to MOVIES, the B$^x$-tree does not rebuild the index based on updates buffers but rather follows an update strategy similar to update-in-place. Also the partitioning into phases used by the B$^x$-tree leads to relatively high query cost (as observed in our experiments), something avoided by MOVIES. Other methods index moving objects by transforming them
to a higher dimensional space. These methods include STRIPES [62] and the dual method of [42], which transform $d$-dimensional space to $2d$-dimensional Hough-X space [31]. The recently proposed $B^{\text{dual}}$-tree uses the same idea [89]; however, it maps the Hough-X space back to a one-dimensional space using a Hilbert curve. Tao and Xiao present a study on dual methods concluding that if query efficiency is required (as required in this paper), dual methods are not competitive [81]. Interestingly, in their concluding remarks, the authors suggest that it could be beneficial to rather reconstruct a non-dual method periodically. Exactly this approach is followed by MOVIES. Finally, some more recent developments show how to make moving object indexes more resilient to variance in data distribution over time [14] and object moving patterns in prediction queries [36, 91]. All of these methods are orthogonal to our technique of frequent index rebuilds. It is straightforward to carry the improvements applicable to the $B^*$-tree also to MOVIES.

16.1.2 Main Memory

The approach of Cui et al. partitions data into sets of active objects that stay in a main memory buffer and inactive objects that reside on external memory [16]. Therefore, that work is more of a buffering scheme for moving object indexing. It is orthogonal to the techniques presented here and can be applied on top of any moving objects index.

16.2 Without Time-Parameterized Queries

16.2.1 Main Memory

Relevant to our work are methods that use main memory for monitoring queries. The method of Kalashnikov et al. [38] uses a fix-sized grid where the grid-size is chosen w.r.t. the average query window sizes. Each grid cell maintains pointers to two lists with query results. Query results are periodically reevaluated and query results are delivered with a time delay $\Delta t$. Yu et al. extend the approach of Kalashnikov et al. to $k$-NN queries [90]. Mouratidis et al. improve the approach of Yu et al. to only update grid-cells that are affected by an incoming update [54]. Recently, these studies have also been subject of even further refinements [15]. However, none of the former methods provides any support for time-parameterized queries. Also these approaches do neither provide any means how to scale for cases when the main memory is exhausted nor provide any parallelization scheme. In contrast, MOVIES provides solutions for all of these issues. In a more recent work, Mouratidis et al. focus on $k$-NN in road networks where the distance among objects is not the euclidean distance but rather the length of the shortest path
on the network [55]. Therefore, the latter method will not work for objects not following roads, e.g., planes, ships, people’s phones. In contrast, MOVIES supports all of these scenarios.

16.3 Extensions for Efficient Updates

16.3.1 External Memory

Frequent update handling in R-trees was the topic of previous studies [45, 7]. A general survey on how to optimize B-trees for high update rates was recently presented by Graefe [26]. Several of these optimizations may be traced back to Lars Arge’s buffer tree [3]. Graefe also mentions differential files [70] as an effective means to trade query performance for update performance. However, Graefe does not mention that one could trade query result staleness and keep both queries and updates efficient as in MOVIES. QU-Trade adapts an R-Tree type moving object index to trade update for query performance in specific index regions according to the workload [86]. This idea is similar in spirit to [21]. Likewise, an interesting extension of our approach would be to take advantage of frequent rebuilds to adapt the index structure to the workload observed during the last index frame. This could be achieved by the selective use of the partial in-place z-code technique discussed in this paper.

16.3.2 Main Memory

Batching updates in a similar way to Lars Arge’s buffer tree [3] was also considered for main memory optimized trees such as [92, 21], however trading query for update performance. In contrast, MOVIES does not trade query performance for update performance. Other cache-efficient trees are the CSS-tree [68] and the FPB+-tree [12]. An interesting challenge would be to extend both the B*-tree and MOVIES to include these optimizations. However, as pointed out in Section 9.1.2, the query processing performance is not affected by MOVIES. Therefore, the general trade-off of update-in-place versus collect and rebuild as used by MOVIES will remain unchanged. Rather, as MOVIES may build read-only indexes at each index frame, MOVIES could even improve overall query performance by building read-only cache-aware indexes.

16.4 Experimental Studies

Moving object scenarios comprise a large number of objects and a large number of updates. As mentioned above, the number of cars in Germany is about
58,000,000 [43]. Assuming every car sends an update on its position every 2 seconds, then this boils down to 29,000,000 updates per second. If we were to index not only cars but also planes, people’s cellular phones, etc., we would face even higher data and update volumes. In this work, we are interested in supporting these large-scale scenarios. Therefore, we are considering data sets of up to 100,000,000 moving objects. This is 10 times larger than in the biggest study available [45] and by at least two orders of magnitude larger than in all other studies we are aware of, e.g., [54, 55, 34, 38, 90, 16]. Even the recent benchmark of Chen et al. also only targets small- to medium-scale scenarios [13]. We think it is important to scale to such large data sets in order to understand the limits of the different methods. In addition, our evaluation on such a large, realistic scenario also demonstrates that it is now feasible to index moving objects employing a method that completely operates out of main memory.

Table 16.1 shows the number of moving objects used in the moving objects studies mentioned in Sections 16.1 to 16.3. Note that some of these studies do not even mention the update rates being evaluated. In contrast, we process update rates of up to 58,000,000 updates per second. We conclude that our study is the first to present an experimental evaluation for moving object indexing for comparably massive workloads.

<table>
<thead>
<tr>
<th>Dataset Size [# moving objects]</th>
<th>Relative Size [%]</th>
<th>Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.001</td>
<td>[64]</td>
</tr>
<tr>
<td>10,000</td>
<td>0.010</td>
<td>[78]</td>
</tr>
<tr>
<td>25,000</td>
<td>0.025</td>
<td>[63]</td>
</tr>
<tr>
<td>100,000</td>
<td>0.100</td>
<td>[7]</td>
</tr>
<tr>
<td>128,000</td>
<td>0.128</td>
<td>[79]</td>
</tr>
<tr>
<td>200,000</td>
<td>0.200</td>
<td>[54, 55]</td>
</tr>
<tr>
<td>500,000</td>
<td>0.500</td>
<td>[42]</td>
</tr>
<tr>
<td>800,000</td>
<td>0.800</td>
<td>[89]</td>
</tr>
<tr>
<td>900,000</td>
<td>0.900</td>
<td>[69, 62]</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.000</td>
<td>[34, 38, 90, 16]</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10.000</td>
<td>[45]</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100.000</td>
<td>this paper</td>
</tr>
</tbody>
</table>

Table 16.1: Sizes of Datasets in the Literature.
Chapter 17

Conclusions

This part of the thesis has proposed a novel approach to time-parameterized moving object indexing of massive data sets under very high update rates. Our approach is based on frequently building short-lived index images. This keeps at the same time query throughput high, query response time low, and update performance high.

The price we have to pay is slightly out-of-date (stale) query results, which is acceptable in several applications including aircraft control [73]. We have shown that this price can be reduced to be as small as a few seconds even for very large data sets of up to 100,000,000 moving objects.

Our experiments have demonstrated the feasibility of our approach even for massive realistic data sets. We have presented results of an experimental study using the entire road network of Germany: a network size unmatched by any previous work. In our study, we scale up to 100,000,000 moving objects and 58,000,000 updates per second. MOVIES shows order-of-magnitude improvements over state-of-the-art approaches, such as the B*-tree [34] as well as several baseline methods, with respect to supported update rates and query rates. One general conclusion is that the popular pattern of keeping and modifying an index should be dropped for moving object scenarios.

Another surprising conclusion of our study is that the idea of indexing predictions for time-parameterized queries as done by some external memory indexes does only work well in main memory for low update rates.

In terms of future work, we plan to examine the trade-off of scalability and staleness in more detail. For some applications, such as computer games, staleness may not be a concern altogether, given that these systems already execute following a tick model [87]. So we plan to explore how well MOVIES would perform when we need to rebuild an index every tick of the game.

Another research direction would be to extend our approach to consider cache-aware B*-trees as read-optimized structures [68]. However, as shown by our formal
analysis, the general trade-off of update-in-place versus collect-and-rebuild would even be improved in favor of MOVIES. Finally, we are planning to apply MOVIES to the more general problem of indexing data streams.
Chapter 18

Summary and Conclusions

18.1 Summary

In this thesis we looked at two problems from the data analysis domain: Keyword search on data warehouses and indexing of moving objects—or more general—indexing of rapidly-changing data.

We started by highlighting the difficulties of both problems:

The main difficulty in keyword search on data warehouses is to produce results in the appropriate size, containing all the data related to a keyword, but still keeping the result small and concise. Since results are produced by joining small information tuples, this leads to an optimization problem, where the appropriate number of joins in each result needs to be found. Further difficulties in keyword search on data warehouses are to infer the correct meaning of a keyword, to understand metadata which is available besides the base data, and to process graph patterns which understand the semantics of the data warehouse.

To overcome these difficulties, we implemented a prototype system that has some similarities to a graph processing engine: We model the metadata and the base data as a graph. To infer the meaning of keywords, we look for graph nodes which contain a similar word. Starting at these nodes, we call them entry points, we follow outgoing edges and try to match graph patterns wherever we go. As soon as one of our graph patterns matches, we understand the semantic meaning of a node, because we know what the graph pattern is meant to represent. This approach allows us to identify tables and joins related to a given keyword.

The main difficulty in moving objects indexing is to keep up with processing all the incoming updates and queries. In our experiments, we scaled up to 100,000,000 moving objects and 58,000,000 updates per second—numbers unmatched by any previous work in the scientific literature.

To be able to process updates and queries in such numbers, we first devised an
CHAPTER 18. SUMMARY AND CONCLUSIONS

indexing algorithm, which allows updates and queries to run in parallel without blocking each other. The important idea here was to trade neither in update nor query performance, but to allow for a small amount of staleness in the result. We do that by creating short-lived indexes at a high rate—seconds or even subseconds—and so produce an illusion of a continuously up-to-date index. Furthermore, we developed several improvements to moving objects indexing, e.g., partial in-place z-codes, quality-aware range query processing, or time-parameterized query processing in main memory.

After presenting possible solutions to both problems, we evaluated our systems:

To evaluate SODA we run experiments on synthetic data as well as on the data warehouse of Credit Suisse. The data warehouse of Credit Suisse receives data from some 2,500 source systems covering all areas of the bank. Its total data volume is around 700 terabytes. Our experiments were executed on the integration layer of the data warehouse which consists of 472 tables and has a data size of 220 gigabytes. Besides running these realistic experiments, we also qualitatively compared SODA against related systems. The experiments and the comparison showed that SODA not only works well for queries from the banking industry, but also is the most versatile system to handle metadata and complex schemas.

To evaluate MOVIES we implemented several spatial indexing methods, including two baseline approaches (binary search trees and B+-trees) and one state-of-the-art moving object index (Bx-tree). We used a commercial dataset which contains the complete road network of Germany and simulated cars traveling along these roads. A variety of experiments showed the amazing scalability and speed of MOVIES.

18.2 Ongoing and Future Work

Keyword search on data warehouses is interesting not only in the financial industries domain, but also in many other domains. We are currently applying SODA to a biology-related domain. Interestingly, in this new use-case, not only the metadata, but also the base data is stored as a graph. Furthermore, the preferred query language is not SQL, as in our Credit Suisse use-case, but SPARQL. In a first approach, we are trying to reduce the new problem to a problem which we already understand: We convert the base data from a graph structure to a relational database and use SQL to query the data. In a second approach, we would like to extend SODA to index graph data directly and produce SPARQL queries.

A second extension to SODA is better classification and ranking methods. At the moment we are using a very strict classification algorithm. As soon as we relax this, we will naturally get many more results. To be able to process these results, a better ranking method is needed. Here we can adapt techniques developed for
keyword search on text documents.

As already mentioned in Chapter 15, MOVIES did not close the book on short-lived indexes, but rather opened a path to interesting future work. First, the core MOVIES algorithm can be modified in different interesting ways. For example, the indexes could be partitioned into groups according to the velocity of the moving objects. Second, the internal data structures of MOVIES could be extended to better support caches or to work with high-dimensional data. Third, MOVIES could be extended by different recovery methods to make sure that MOVIES can be used in a production environment. And fourth, we showed how to use MOVIES to index and query location-based moving objects. However, the indexing technique of MOVIES is far from limited to this area. It should be well possible to use MOVIES in all kinds of domains, where data changes rapidly and queries have to be executed simultaneously.
List of Tables

5.1 Complexity of the Schema Graph. ...................................... 42
5.2 Experiment Queries. ........................................................ 43
5.3 Precision and Recall for Experiment Queries. ......................... 45
5.4 Query Complexity and Runtime Information. .......................... 46

6.1 Qualitative Comparison. ..................................................... 51

14.1 Settings Used in the Experiments. .................................... 101

16.1 Sizes of Datasets in the Literature. ................................... 122
# List of Figures

1.1 Staleness Dimension. ................................. 3
3.1 Sample World: Conceptual Schema. .................. 14
3.2 Sample World: Logical Schema. ...................... 15
3.3 Metadata Graph and Relational Data. ................. 17
3.4 Customers Domain Ontology. ........................ 18
3.5 Securities Domain Ontology. ........................ 18
3.6 SODA Overview. ..................................... 20
3.7 Query Classification. ................................ 21
3.8 Output of Lookup Step. ............................... 21
3.9 Output of Tables Step (join relationships not shown). 22
3.10 Google-style Start Page of SODA. ................... 23
3.11 I’m Feeling Lucky Page of “Find all high tech securities”. 24
3.12 Lineage Page of the “high tech” Entry Point. ...... 25
4.1 Table Pattern. ........................................ 29
4.2 Foreign Key Pattern. .................................. 30
4.3 Joins on Direct Path. .................................. 31
4.4 Domain Ontology for Trading Applications. ........... 36
5.1 Credit Suisse Data Warehouse Architecture. ............ 40
5.2 Complex Schema Hierarchy. ............................ 47
9.1 Expected Maximum Update Rate (the gain). .......... 64
9.2 Expected Staleness (the price). ........................ 65
9.3 Two Index Frames of MOVIES Core Algorithm. ....... 66
9.4 Alternate Organization of Data in Update Buffers. .... 69
9.5 Data Layout in an Index. ............................... 70
9.6 kd-Trie Mapped to a Pointer-Free Linearization. ..... 71
10.1 Improved Data Layout in an Index. .................... 77
10.2 Costs of Z-Code Computation [loop-based versus precomputed]. 79

131
10.3 Costs for Sorting [quicksort versus radixsort] . . . . . . . . . . . . . . 80
11.1 Naive Range Query . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
11.2 Query Execution Time Varying Maximum Error . . . . . . . . . . . . . 86
11.3 Number of Created Partitions while Varying Maximum Error . . . . 87
11.4 Query Performance $[qw =1,000 m \times 1,000 m]$ . . . . . . . . . . . . 88
11.5 Query Performance $[qw =10,000 m \times 10,000 m]$ . . . . . . . . . . 88
11.6 Query Scalability $[qw =1,000 m \times 1,000 m]$ . . . . . . . . . . . . 89
11.7 Query Scalability $[qw =10,000 m \times 10,000 m]$ . . . . . . . . . . 90
12.1 Data Layout for Non-Predictive MOVIES . . . . . . . . . . . . . . . . . 93
14.1 Road Network of Germany . . . . . . . . . . . . . . . . . . . . . . . . 102
14.2 Maximum Update Rate when Scaling the Index Size . . . . . . . . . . 103
14.3 Maximum Update Rate Comparison of Four MOVIES Variants . . . . 104
14.4 Average Staleness when Scaling the Index Size . . . . . . . . . . . . . 105
14.5 Maximum Query Rate when Scaling the Update Rate . . . . . . . . . . 106
14.6 Maximum Query Rate when Scaling Update Rate and Index Size . . 107
14.7 Main Memory Consumption . . . . . . . . . . . . . . . . . . . . . . . . 108
14.8 Maximum Update Rate when Scaling to Four Processing Nodes . . . 110
14.9 Average Staleness when Scaling to Four Processing Nodes . . . . . . 110
14.10 Maximum Update Rate on Four Shared-Nothing Servers . . . . . . . . 111
Bibliography


[73] Personal communication with Skyguide Flight Control.


[75] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.


