

# COMPUTATIONAL METHODS FOR SPATIAL OLAP

by

Oliver Baltzer

Submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

at

Dalhousie University  
Halifax, Nova Scotia  
April 2011

© Copyright by Oliver Baltzer, 2011

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled “COMPUTATIONAL METHODS FOR SPATIAL OLAP” by Oliver Baltzer in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dated: April 12, 2011

External Examiner:

---

Research Supervisors:

---

---

Examining Committee:

---

---

Departmental Representative:

---

# DALHOUSIE UNIVERSITY

DATE: April 12, 2011

AUTHOR: Oliver Baltzer

TITLE: COMPUTATIONAL METHODS FOR SPATIAL OLAP

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: Ph.D.

CONVOCATION: October

YEAR: 2011

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions. I understand that my thesis will be electronically available to the public.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

---

Signature of Author

# Contents

<b>List of Figures</b>	<b>ix</b>
<b>Abstract</b>	<b>xiv</b>
<b>List of Abbreviations Used</b>	<b>xv</b>
<b>Acknowledgements</b>	<b>xvi</b>
<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 Outline of Thesis . . . . .	7
<b>Chapter 2 Background</b>	<b>8</b>
2.1 Data Warehousing and OLAP . . . . .	8
2.2 SOLAP Applications . . . . .	13
2.3 SOLAP System Architecture . . . . .	13
2.4 SOLAP Solutions . . . . .	14
2.5 The SOLAP Data Model . . . . .	15
2.5.1 Spatial Measures . . . . .	17
2.5.2 Spatial Feature Dimensions . . . . .	17
2.5.3 SOLAP Operations . . . . .	18
2.5.4 SOLAP View Indexing . . . . .	20
2.5.5 SOLAP Query Languages . . . . .	21
2.6 Summary and Open Questions . . . . .	21
<b>Chapter 3 Spatial Dimension Hierarchies</b>	<b>23</b>
3.1 Approaches to the Definition of Spatial Dimension Hierarchies . . . . .	23
3.2 Our Application Example . . . . .	25
3.3 Modeling the Spatial Dimension Hierarchy . . . . .	26
3.4 The Data Model . . . . .	27
3.5 ROLLUP and CUBE Queries in the Forester Example . . . . .	29



3.5.1	Querying Asymmetric Hierarchies . . . . .	29
3.5.2	Querying Multiple Alternative Hierarchies . . . . .	30
3.5.3	Querying Generalized Hierarchies . . . . .	31
3.5.4	Querying Non-Strict Hierarchies . . . . .	32
3.5.5	Complex Queries and Spatial Measures . . . . .	34
3.6	Summary . . . . .	34
<b>Chapter 4</b>	<b>A Model for the Pipelined Evaluation of Spatial OLAP Queries</b>	<b>35</b>
4.1	The Pipeline Model . . . . .	36
4.2	Data Model . . . . .	40
4.3	Mini-Engines . . . . .	43
4.3.1	Data Accessor . . . . .	44
4.3.2	Filter . . . . .	44
4.3.3	Select . . . . .	44
4.3.4	Group . . . . .	45
4.3.5	Aggregate . . . . .	46
4.3.6	Join . . . . .	47
4.3.7	Sort . . . . .	48
4.3.8	Result Stack . . . . .	49
4.4	Applications and Example Queries . . . . .	49
4.4.1	Query 1 . . . . .	51
4.4.2	Query 2 . . . . .	55
4.4.3	Query 3 . . . . .	58
4.4.4	Query 4 . . . . .	63
4.4.5	Query 5 . . . . .	68
4.5	Summary . . . . .	72
<b>Chapter 5</b>	<b>LISA – A Pipeline-Based Query Evaluation System for Spatial OLAP</b>	<b>73</b>
5.1	Design & Implementation . . . . .	74
5.1.1	Mini-Engines . . . . .	74

5.1.2	Streams . . . . .	75
5.1.3	Tracks . . . . .	75
5.1.4	Data Accessors . . . . .	77
5.1.5	Geometric Functions and Operators . . . . .	79
5.1.6	Query Strategy Definition . . . . .	80
5.2	Evaluation of LISA . . . . .	87
5.2.1	Evaluation of LISA with Respect to Traditional OLAP . . . . .	88
5.2.2	Evaluation of LISA with Respect to Spatial OLAP . . . . .	94
5.2.3	Comparison of LISA with other Systems . . . . .	102
5.3	Summary . . . . .	106
<b>Chapter 6 The geoCUBE Index</b>		<b>107</b>
6.1	Related Work . . . . .	110
6.2	Our Approach . . . . .	111
6.3	Dynamic Resolution of Hilbert Curves . . . . .	114
6.4	Exploiting Hilbert Order for I/O-Efficient Indexing . . . . .	117
6.4.1	Answering Range Aggregate Queries . . . . .	117
6.4.2	Updating OLAP Views . . . . .	119
6.5	Implementation and Experiments . . . . .	120
6.5.1	Performance Evaluation . . . . .	122
6.6	Summary . . . . .	131
<b>Chapter 7 OLAP for Moving Object Data</b>		<b>132</b>
7.1	Introduction . . . . .	132
7.2	Related Work . . . . .	137
7.2.1	Identifying Group of Moving Objects . . . . .	138
7.2.2	Extending OLAP Query Languages . . . . .	145
7.3	OLAP for Trajectories . . . . .	145
7.3.1	General Framework and Preprocessing . . . . .	146
7.3.2	Group by Overlap . . . . .	147
7.3.3	Group by Intersection . . . . .	148
7.4	Interactive OLAP for Trajectories . . . . .	152

7.5	Experimental Evaluation . . . . .	153
7.5.1	Detailed Analysis . . . . .	153
7.5.2	Robustness Against Noise . . . . .	158
7.5.3	Input Parameters . . . . .	160
7.5.4	Real World Data . . . . .	164
7.6	Summary . . . . .	167
<b>Chapter 8</b>	<b>Conclusion</b>	<b>168</b>
8.1	Contributions . . . . .	168
8.2	Future Work . . . . .	170
<b>Appendix A</b>	<b>Optimization Opportunities for LISA</b>	<b>172</b>
A.1	Query Planner and Query Optimization . . . . .	172
A.2	Implementation Language and Process Model . . . . .	173
A.3	I/O Latency and Bandwidth . . . . .	174
A.4	Storage Models, Access Methods, and Indexing . . . . .	176
A.5	Integration with Existing Data Sources . . . . .	177
<b>Appendix B</b>	<b>Implementation of Example Queries in LISA</b>	<b>178</b>
B.1	Query 1 . . . . .	178
B.2	Query 2 . . . . .	181
B.3	Query 3 . . . . .	188
B.4	Query 4 . . . . .	191
B.5	Query 5 . . . . .	197
B.6	Query 2 in optimized PostgreSQL PL/pgSQL . . . . .	208
B.7	Query 5 in optimized PostgreSQL PL/pgSQL . . . . .	209
B.8	Query 5 in PostgreSQL standard SQL . . . . .	210
<b>Appendix C</b>	<b>OLAP for Moving Object Data: Research Opportunities</b>	<b>212</b>
C.1	Alternative Grouping Operators . . . . .	212
C.2	Alternative Movement Patterns . . . . .	213
C.3	Aggregation of Trajectories . . . . .	213



## List of Figures

Figure 1.1	Hierarchy and interrelation of DBMS, GIS, OLAP and SOLAP.	2
Figure 2.1	A multi-dimensional view of data typical for OLAP. . . . .	9
Figure 2.2	Example of a <i>roll-up</i> query. . . . .	10
Figure 2.3	Star schema of a typical data warehouse. . . . .	11
Figure 2.4	Data cube lattice with dimension hierarchies. . . . .	12
Figure 2.5	High-level architecture of a SOLAP system. . . . .	14
Figure 2.6	Geometry class hierarchy for SQL. . . . .	16
Figure 3.1	The forester application example. . . . .	26
Figure 3.2	Fact constellation schema of the application example. . . . .	27
Figure 4.1	Example query decomposed into an assembly of mini-engines.	39
Figure 4.2	Example source and result tables for the query described in Section 4.4.1. They gray shading represents the records that satisfy the constraint specified by the <b>WHERE</b> clause. . . . .	52
Figure 4.3	Data flow graph of the strategy used to evaluate the query described in Section 4.4.1. . . . .	53
Figure 4.4	Example result tables for the query described in Section 4.4.2. Attribute values of <b>*</b> denote <b>NULL</b> or undefined values, which are commonly used to represent an aggregated attribute dimension.	56
Figure 4.5	Data flow graph of the query evaluation strategy described in Section 4.4.2. . . . .	57
Figure 4.6	Illustration of Query 3. Counties are represented as polygons, and individually sampled plants of different types of vegetation are represented as <b>▲</b> , <b>■</b> , and <b>●</b> respectively. The rectangular query region intersects multiple counties and contains a number of individual plant records. . . . .	59
Figure 4.7	Data flow graph of an evaluation strategy for Query 3. . . . .	60
Figure 4.8	Illustration of the trimming of spatial attributes to the extent of the query region. Here the <b>county</b> records from Query 3 are constrained to their spatial intersection with the query region.	61

Figure 4.9	Illustration of Query 4. At each level of aggregation, compute the maximum vegetation height for all member of that level that intersect with the query region. The measures are records of individually sampled plants of various species, here represented as ▲, ■, and ●. . . . .	63
Figure 4.10	Graphical representation of the results of Query 4 for each aggregation level. . . . .	64
Figure 4.11	Data flow graph of an evaluation strategy for Query 4. . . . .	65
Figure 4.12	Example of a measure that is partially associated with multiple features. . . . .	68
Figure 4.13	Illustration of Query 5. . . . .	69
Figure 4.14	Spatially extended SQL statement for Query 5, a non-strict hierarchy roll-up query. . . . .	70
Figure 4.15	Data flow graph of an evaluation strategy for Query 5. . . . .	71
Figure 5.1	Conceptual use of the MUX and DEMUX components in a data-flow. . . . .	77
Figure 5.2	Data flow graph of an evaluation strategy for Query 3. This dataflow graph is identical to that shown in Figure 4.7 of Chapter 4. . . . .	81
Figure 5.3	Query runtime by number of processors. . . . .	91
Figure 5.4	Query speed-up factor and efficiency for multiple processors with respect to single processor runs. . . . .	92
Figure 5.5	Impact of track configuration on query evaluation time. . . . .	93
Figure 5.6	Impact of dataset size on query evaluation time. . . . .	94
Figure 5.7	SQL representation of the spatial OLAP query. . . . .	97
Figure 5.8	Representation of the query region used in the spatial OLAP query. . . . .	98
Figure 5.9	Query running time by number of processors. . . . .	99
Figure 5.10	Query speed-up factor and efficiency for multiple processors with respect to single processor runs. . . . .	100
Figure 5.11	Impact of track configuration on query evaluation time. . . . .	101
Figure 5.12	Impact of dataset size on query evaluation time. . . . .	102
Figure 5.13	Distribution of complexity (i.e. number of vertices) among lulc layer objects. . . . .	103

Figure 5.14	Comparison of LISA’s and PostgreSQL’s performance on traditional OLAP queries for various dataset sizes. . . . .	104
Figure 5.15	Comparison of LISA’s and PostgreSQL’s performance on spatial OLAP queries. . . . .	105
Figure 6.1	Star schema of a typical data warehouse with facts table, three feature dimensions and two measures. The feature dimensions have been normalized within the facts table in that they are represented as keys into dimension tables. . . . .	108
Figure 6.2	Hilbert curve. . . . .	113
Figure 6.3	Increasing the resolution of the Hilbert curve preserves the order of records. . . . .	115
Figure 6.4	Records in Hilbert order are stored in consecutive blocks on disk and form multi-dimensional regions in the original space. The evaluation of a range query in breadth-first fashion only traverses part of the tree and results in a combination of sequential read and forward-seek operations at the leaf level. . .	118
Figure 6.5	Annotation of non-leaf nodes with aggregate information and references to records at the leaf level. . . . .	119
Figure 6.6	Merging the target view with the update view may result in a dynamic adaptation of the Hilbert curve resolution. . . . .	120
Figure 6.7	Overhead of pre-discretization compared to dynamic resolution adaptation. . . . .	123
Figure 6.8	Recomputing each record’s Hilbert rank for each comparison versus storing the last computed Hilbert rank of each record. .	124
Figure 6.9	Iterative versus constant-time resolution determination. . . . .	126
Figure 6.10	Batch update time for different update sizes. . . . .	128
Figure 6.11	Time to construct the index including sorting of the dataset for the geoCUBE index. . . . .	129
Figure 6.12	Query performance of geoCUBE vs. R-tree for real world data.	130
Figure 7.1	<i>OLAP For Trajectories</i> Example. (a) Input data. (b) Groups with support above the required minimum support. (c) Aggregate results reported (aggregate trajectories and counts). . . .	133
Figure 7.2	Illustration of two different version of operator <code>GROUP_TRAJEC-TORIES</code> (a) Group by Intersection, (b) Group by Overlap. . . .	136
Figure 7.3	Illustration of (a) Overlap Ratio and (b) Intersection Ratio. .	148

Figure 7.4	Screenshot of the interactive environment for <i>OLAP For Trajectories</i> . . . . .	152
Figure 7.5	Two datasets each containing three closed frequent itemsets but with different associated overlap. . . . .	154
Figure 7.6	Grouping results for dataset $A_1$ with <i>ORT</i> and <i>IRT</i> set to 0.25 respectively. . . . .	155
Figure 7.7	Grouping results for dataset $A_2$ with <i>ORT</i> and <i>IRT</i> set to 0.25 respectively. Note, the change in overlap results in a different grouping for <i>Group by Overlap</i> when compared to dataset $A_1$ . . . . .	155
Figure 7.8	Changing the <i>Intersection Ratio</i> by changing the number of parallel trajectories. . . . .	156
Figure 7.9	Grouping results for dataset $B_1$ with <i>ORT</i> and <i>IRT</i> set to 0.2, respectively. . . . .	156
Figure 7.10	Grouping results for dataset $B_2$ with <i>ORT</i> and <i>IRT</i> set to 0.2, respectively. Note, the increase of intersection size results in a different grouping for <i>Group by Intersection</i> when compared to dataset $B_1$ . . . . .	157
Figure 7.11	Groups identified by each of our algorithms from a dataset with varying levels of noise. The input parameters were set to a space resolution of 5, a minimum support of 4 and a minimum frequent itemset length of 4. . . . .	159
Figure 7.12	Synthetic dataset with 24 groups each consisting of 10 trajectories and a partial overlap of approximately 25%. . . . .	161
Figure 7.13	Groups identified by each of our algorithms at levels of resolution between 2 and 8 (left to right). Fixed parameters are $min\_support = 4$ , $min\_length = 4$ , $ORT = 0.2$ , $IRT = 0.2$ . . . . .	162
Figure 7.14	Number of groups each algorithm identifies for the given input dataset depending on the resolution. Note, <i>Group by Overlap</i> tends to identify fewer but larger groups, <i>Group by Intersection</i> , on the other hand, identifies more but smaller groups. . . . .	163
Figure 7.15	Groups (identified by color) computed by both of our methods for $ORT = IRT = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7$ ( $min\_support = 4$ , $min\_length = 4$ ). . . . .	164
Figure 7.16	Relationship between the number of identified groups and values for <i>Overlap Ratio Threshold ORT</i> and <i>Intersection Ratio Threshold IRT</i> . . . . .	165
Figure 7.17	Results obtained for the School Buses Dataset. . . . .	166



Figure C.1	Examples of different movement patterns. . . . .	213
Figure C.2	A visual generalization function for groups of trajectories. . . .	215

## Abstract

Data warehousing and On-line Analytical Processing (OLAP) are powerful tools for processing and analyzing business and analytical data. It is estimated that 80% of the data stored in data warehouses have some spatial components. It is our belief that there is a need for powerful OLAP tools that are capable of processing and analyzing spatial data. This thesis explores the design and implementation of Spatial OLAP (SOLAP) systems and describes approaches to support the characteristic features of OLAP while seamlessly integrating spatial data into the analysis process. In particular, we analyze the evaluation of OLAP queries in the presence of asymmetric, multiple-alternative, generalized, and non-strict spatial dimension hierarchies. We introduce a new pipeline-based query evaluation model that is comprehensive and powerful in that it provides a uniform approach to the expression of spatial OLAP queries that address all major dimension hierarchy types while permitting a uniform treatment of both spatial and non-spatial data. A reference implementation called “LISA” validates the objectives of our model and demonstrates favorable scalability and performance on modern multi-processor and multi-core hardware platforms. We also describe a new “geoCUBE” index, to address the fundamental problem of how to represent, index and efficiently query data that is defined by a mix of spatial and categorical attribute values. The geoCUBE index extends existing methods for indexing OLAP data to spatial data types. The effectiveness of the geoCUBE data structure is confirmed through evaluation. Lastly, we propose algorithms that facilitate OLAP-like analysis of moving object data. We introduce a new class of `GROUP BY` operators specifically targeted to the OLAP analysis of trajectories and to answering aggregate queries with respect to the spatio-temporal movement of a set of objects. Through an experimental evaluation we show our operators can be used to reliably identify groups of related trajectories when applied to synthetic and real world moving object data.

## List of Abbreviations Used

DBMS Database Management System

DW Data Warehousing

ETL Extraction, Transformation, Loading

GIS Geographic Information System

GPS Global Positioning System

I/O Input/Output

IR Intersection Ratio

IRT Intersection Ratio Threshold

OLAP Online Analytical Processing

OR Overlap Ratio

ORT Overlap Ratio Threshold

P2P Peer-to-Peer

SOLAP Spatial Online Analytical Processing

SQL Structured Query Language

WKB Well-known Binary

WKT Well-known Text

XML Extensible Markup Language

## Acknowledgements

First and foremost, I would like to thank my supervisors Andrew Rau-Chaplin and Norbert Zeh. It was their continuous guidance, inspiration, and encouragement, that allowed me to complete this long journey.

I extend thanks to the members of my examination committee Bradford Nickerson, Michael Shepherd, and Qigang Gao, for taking the time to work through this thesis and providing valuable feedback.

I thank Dalhousie University for their financial support without which this thesis would not have been possible.

I especially thank my friends and colleagues who supported and motivated me throughout my academic endeavours.

I owe my deepest gratitude to my parents Sybille and Hans-Ulrich, and my sister Kathleen, whose unconditional love and endless encouragement guided me through much more than this thesis.

# Chapter 1

## Introduction

The recent advent of larger storage solutions and the rise of disciplines such as decision support, data mining and business intelligence have led to increasing amounts of data that have to be managed and analyzed. Data Warehousing (DW) and Online Analytical Processing (OLAP) are popular technologies used by enterprises, research institutions and public organizations to store, manage and analyze high volumes of data. The data, often collected on a daily basis, is consolidated in data warehouses [81].

It is estimated that 80% of the data stored in data warehouses have some spatial component [58] and in many applications the data is or can be referenced with geo-spatial information. Most current data warehouse or OLAP systems, however, do not support the analysis of spatial components and potentially valuable information embedded in the data is not incorporated in the analysis process. There is an opportunity to enhance the analysis and decision making process by combining features of OLAP with those of Geographic Information Systems into a single tool. In 1997 Bedard, et al. proposed an online analytical processing system that incorporates spatial data into the analysis process and coined the term Spatial OLAP (SOLAP; also Spatio-Temporal OLAP). Since then SOLAP systems have been implemented for a number of specific application [20,38,118,120], but only a few general purpose systems that unify the most common features exist today [11, 24, 49, 66, 95, 100, 157, 160, 162]. A general purpose Spatial OLAP system is a system that was not designed for a specific application scenario and instead can accommodate a wide variety of applications without modification. Many of the existing solutions are based on commodity off-the-shelf components and consequently suffer from significant and growing scalability issues due to growing data volumes.

It is our belief that new systems, designed from the ground up for SOLAP, combining efficient data structures and the capacity for parallel processing, are required to deal with the challenges introduced by ever growing data volumes. From a user's

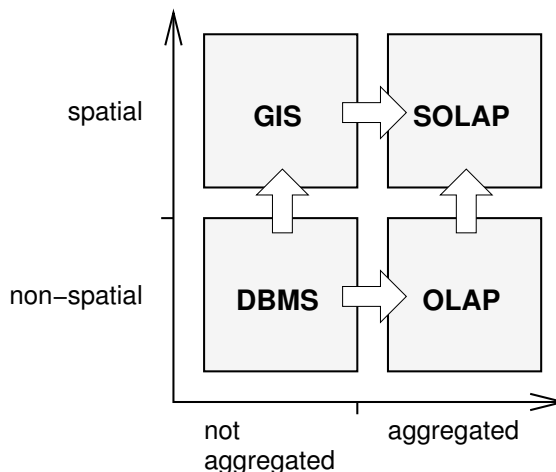


Figure 1.1: Hierarchy and interrelation of DBMS, GIS, OLAP and SOLAP.

perspective a SOLAP system appears as an intersection of GIS and OLAP tools (Figure 1.1) [153]. Its graphical user interface allows for an interactive exploration of spatially referenced data and visualizes non-spatial information in a spatial context. In addition to typical map navigation functions like pan and zoom, it provides OLAP-type operations such as roll-up, drill-down, slice and dice to aid user navigation through the data. To support such OLAP operations a representation of spatial data that is compatible with the standard OLAP data schema has to be defined.

This thesis explores the design and implementation of high performance Spatial OLAP systems, that scale well with both increasing query load and increasing data volume. The focus is on:

1. how to best integrate OLAP with spatial and GIS concepts;
2. efficiency and scalability in both sequential and parallel settings;
3. SOLAP's potential for new applications that neither OLAP, GIS, nor spatial database extensions can support alone.

At the heart of OLAP is the SQL “Cube” query:

```
SELECT  $D_1, D_2, \dots, D_d, F_1(M_1), F_2(M_2), \dots, F_k(M_k)$ 
FROM facts_table
GROUP BY CUBE( $G_1, G_2, \dots, G_g$ )
```

where  $D_1, \dots, D_d$  are categorical *feature dimensions*,  $M_1, \dots, M_k$  are numerical *measures*,  $F_1, \dots, F_k$  are numerical *aggregation functions* and  $G_1, \dots, G_g$  are categorical *dimension hierarchies*. It is our believe that a Spatial OLAP system must be capable of answering such queries where  $D_1, \dots, D_d$  are a mix of categorical and spatial feature dimensions,  $M_1, \dots, M_k$  are a mix of numerical and spatial measures,  $F_1, \dots, F_k$  are a mix of numerical and spatial aggregation functions and  $G_1, \dots, G_g$  are a mix of categorical and spatial dimension hierarchies. The challenge is how to realize these spatial extensions to OLAP concepts in a seamless and efficient manner.

To date many partial solutions to the SOLAP problem have been proposed. However, we believe that a comprehensive solution must address the integration of key aspects such as data representation, materialization, indexing, querying and user interfaces into a coherent and uniform system with well-defined interfaces. So far, approaches have either targeted only specific and very isolated problems of spatial data warehousing [23, 136, 170], without the integration of other parts into a functional system, or the system implementation was driven by a specific application, not paying much attention to a uniform data representation, general applicability and efficient algorithms [8, 20, 153, 162]. In detail we address the following problems in order to design an integrated spatial OLAP solution:

- uniform treatment of spatial and non-spatial data throughout all components of the data warehouse and its functionality;
- efficient representation of materialized spatial measures on disk;
- representation of query results in the same manner as views;
- standardized query interface;
- efficient methods for on-the-fly spatial aggregate computation;
- efficient update mechanisms for materialized spatial views;
- scalable and cost-effective computing platform as the performance of CPUs and sizes of internal memory grow slower than external storage space and data volumes.

On the basis of a simple application example encompassing four different types of spatial dimension hierarchies, namely asymmetric, multiple-alternative, generalized, and non-strict [114], we explore in Chapter 3 for each hierarchy type how it influences the evaluation of *roll-up* and *cube* queries and how the query results can be represented in a tabular form that is consistent with standard OLAP data representation of views. Asymmetric and multiple-alternative dimension hierarchies are not exclusive to spatial data and conceptually supported in traditional OLAP systems. Generalized and non-strict dimension hierarchies are very common in spatial data and not readily supported by traditional OLAP systems. These will be our focus.

In Chapter 4 we introduce a computational model for the pipeline-based evaluation of spatial and non-spatial OLAP queries. We analyze a representative set of typical spatial and non-spatial OLAP queries and propose for each query type a set of components and an evaluation strategy using our pipeline model. Our model is comprehensive and powerful in that it provides a uniform approach to the expression of spatial OLAP queries that addresses all of the major spatial hierarchy types, while permitting a uniform treatment of both spatial and non-spatial data. Although our model was evaluated against a representative set of spatial and non-spatial OLAP queries it provides the fundamental features of a general purpose spatial OLAP system. It is not limited to a specific set of queries or application scenarios and can be directly applied to a wide range of applications.

From a systems perspective, the design of our model primarily focusses on what we believe are critical requirements for a spatial OLAP system:

1. **Performance** – A modular architecture and intelligent implementations facilitate the use of efficient algorithms to achieve high performance in data processing and query evaluation.
2. **Flexibility** – The architecture enables the recombination of existing components to obtain results using different strategies and allowing for an optimal cost balance.
3. **Extensibility** – The ability to extend at a component level allows for an easy addition of new functionality and improved performance.



The proposed pipeline-based query evaluation model is both ambitious and very complex in that it attempts to present a single unified approach to the expression of spatial OLAP queries over multiple hierarchy types, while permitting a uniform treatment of both spatial and non-spatial data. Given this complexity it was important to create a reference implementation in order to address the following questions: 1) Could the model be realized in a functioning and complete system, and 2) Could that system be made fast and scalable by exploiting modern multi-core parallelism? In Chapter 5, we describe the design, implementation, and evaluation of “LISA”, a reference implementation of our pipeline-based query evaluation model. LISA is a full reference implementation of our model that supports both complex spatial and non-spatial OLAP queries. It was designed to exploit Python’s [184] dynamic type system, support for co-operative concurrent processes, and ability to integrate existing spatial libraries. LISA’s sophisticated object-oriented design and high level of abstraction provides an extensible framework which supports a wide range of data types and storage backends. It realizes the query pipeline as a highly concurrent program flow typically involving hundreds of co-operative processes. LISA is a largely unoptimized reference implementation. However, in extensive performance evaluations it demonstrates a high degree of scalability. For complex spatial queries, even in its prototype state, it outperforms PostgreSQL [146], the most widely available optimized database system that supports spatial extensions.

As with traditional OLAP, data stored in spatial OLAP views is typically multi-dimensional and represented by a number of attributes. Spatial data attributes are generally represented in continuous space rather than categorical space as is commonly the case in traditional OLAP [69, 183]. This presents a challenge for the efficient storage, indexing and updating of spatial OLAP views as traditional methods for indexing categorical data do not capture all the properties of data in continuous space. In Chapter 6, we propose a new indexing structure called “geoCUBE,” which exploits the recursive nature of the Hilbert space filling curve to index multi-dimensional data that is represented by categorical and continuous attribute values. Our approach preserves the relative locality of attribute values and records, supports an arbitrary number of categorical and continuous attribute dimensions, and provides a high query performance. In addition, our “geoCUBE” index permits efficient batch updates of

existing views, requiring only sorting complexity of the update dataset and a single sequential scan over the updated view dataset. We demonstrate the effectiveness of our “geoCUBE” index with an experimental evaluation and show its superiority in performance when compared to alternative approaches and data structures.

The analysis of moving object databases is a field of research that has received significant attention in recent years [18,61,62,67,75,103,107,169,192]. Typical applications of this discipline are location-based services [155], traffic control [136], transport logistics [48], wild life tracking [104] and epidemiology [20,168]. With the large adoption of Global Positioning Systems (GPS), Radio Frequency Identification (RFID) and mobile devices in everyday life, an increasing amount of data is being collected by such applications, and there is a growing need for the analysis of aggregated information about moving objects. In Chapter 7, we propose algorithms that facilitate OLAP-like analysis of moving object data. Our algorithms focus on the identification of aggregate groups among trajectories at varying levels of resolution. The underlying conceptual ideas may be applicable to other analysis scenarios as well.

We propose a new class of **GROUP BY** operators specifically targeted to the OLAP analysis of trajectories and to answering aggregate queries with respect to the spatio-temporal movement of a set of objects. The main problem studied is how to identify aggregation groups with respect to a feature dimension representing trajectories. It is very unlikely that any two trajectories are exactly the same. Hence, standard aggregation of records based on groups with equivalent trajectory values is not very useful in most cases. Instead, we propose to partition the given trajectories into groups of trajectories using a new **GROUP BY** operator, which we term **GROUP\_TRAJEC-TORIES**. This operator returns a group identifier for each trajectory, and then OLAP can proceed with standard aggregation according to the group identifiers instead of the trajectories themselves.

The solution proposed is a novel extension of the frequent pattern mining methods that are already available in today’s data warehousing solutions with new algorithms that are more appropriate for identifying groups of moving objects. It is specifically designed to capture relationships between movement patterns and thus allows the identification of groups of objects that exhibit a complex behavior. At the same time it integrates well with existing OLAP models by using established data representations

and query languages, as well as allowing the user to interactively browse the results at varying levels of resolution and aggregated information. The challenge here was how to define effectiveness of a novel operator like `GROUP_TRAJECTORIES` when there is no obvious benchmark. Our approach was to take data with known group movements, obscure them with noise, and then evaluate the algorithms' capacity to extract the original group movements. Using various generated and real-world moving object data sets we tested how well the `GROUP_TRAJECTORIES` operator worked for the OLAP analysis of trajectories in the context of different application scenarios.

## 1.1 Outline of Thesis

The remainder of this thesis is organized as follows. Chapter 2 reviews fundamental concepts of OLAP and how they translate into the context of Spatial OLAP. It individually addresses the components of characteristic OLAP queries, reviews how they can be applied to spatial data and identifies issues surrounding their implementation. In Chapter 3 we investigate the impact of spatial dimension hierarchies on OLAP specific *roll-up* and *cube* queries. In Chapter 4 we introduce a pipeline-based query evaluation model for OLAP in the presence of spatial and non-spatial dimension hierarchies and measures, followed by an in-depth discussion and evaluation of its implementation "LISA" in Chapter 5. Chapter 6 focusses on the efficient storage and retrieval of data for query evaluation and introduces a novel approach for indexing data in a SOLAP system. A new application, combining SOLAP and data mining is introduced in Chapter 7. It suggests a new class of `GROUP BY` operators and addresses the mining of group movements from moving object data. This thesis is concluded in Chapter 8 with a summary of its contributions and directions for future research.

## Chapter 2

### Background

In this chapter we provide an overview of the concepts underlying OLAP and how they translate into the context of Spatial OLAP. We begin with describing a typical OLAP architecture and define terms common to OLAP. We review related work and explore the interplay between general OLAP concepts and the specific demands of spatial application. The chapter concludes with an identification of implementation-specific issues and lays the groundwork for the proposed research.

#### 2.1 Data Warehousing and OLAP

Data warehousing is the consolidation of historical data from various sources into a single large data repository under a unified schema [31]. It provides online analytical processing to aid in subject-driven decision making, strategic planning and data mining [37]. Hence, a data warehouse can be described as a subject-oriented, integrated, time-variant and nonvolatile collection of data [81] that is distinct from transactional data used for operational application-oriented processing.

Commonly a data warehouse is constructed by integrating data from a number of sources, such as relational databases, flat files, spreadsheets and legacy systems. During this integration relevant information is extracted, noise and invalid records are removed and the consistency of the data within the schema is ensured. This process of populating a data warehouse is referred to as the extraction, transformation and load (ETL) stage and is paramount to the quality of the data warehouse [149,186].

Online Analytical Processing (OLAP) was first proposed in 1993 by E.F. Codd [37] and has since been very successful in business intelligence [142]. OLAP is the subject-oriented analysis of multi-dimensional *measures* (subjects) in relationship to a number of *feature dimensions* (attributes) at different levels of granularity. It is an important tool used in decision support and data mining and allows business executives and knowledge workers to extract knowledge hidden within the data of a

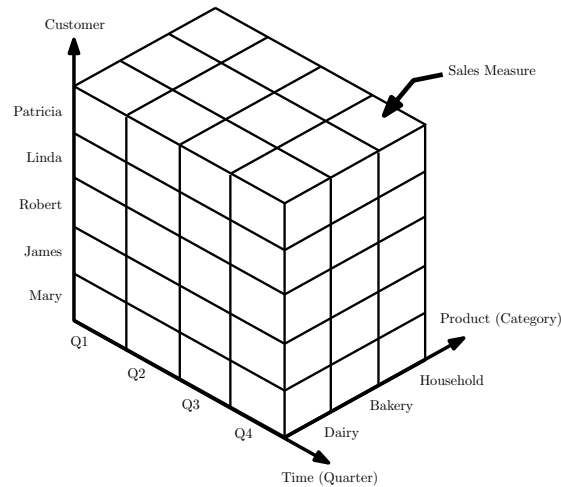


Figure 2.1: A multi-dimensional view of data typical for OLAP.

data warehouse [37,81]. OLAP functionality is typically provided by data warehouses, which may be augmented with specialized OLAP engines or OLAP servers to enhance performance.

Figure 2.1 illustrates a typical multi-dimensional view on data based on a selected set of feature dimensions at a specific granularity. The subject of this view is the measure *Sales* and each cell in the represented view contains an aggregated *Sales* value. The feature dimensions are *Customer*, *Product* and *Time*, while the levels of granularity are set to *Category* for the *Product* dimension and *Quarter* for the *Time* dimension.

Feature dimensions in a data warehouse provide individual *concept hierarchies* that allow the analysis at different levels of granularity. A *Time* dimension, for example, might have the concept hierarchy *Day*  $\rightarrow$  *Month*  $\rightarrow$  *Year*, with level *Month* providing a more general view of the measure than level *Day* and similarly *Year* a more general view than *Month*. Every feature dimension has at least two implicit hierarchy levels: the finest level of granularity, that is the base data, and the *any* level, which is the most general level of granularity of a dimension. In fact, the *any* level of a dimension hierarchy is implicitly used when its attribute values do not affect the analysis and the dimension can be ignored [170].

Based on this multi-dimensional data model, OLAP provides a number of operations to manipulate the view of the data. The *roll-up* operation is used to increase

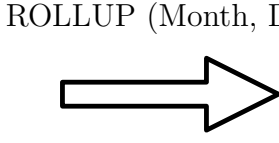
Time (Day)	Sales	ROLLUP (Month, Day)	Time (Month)	Sales
06-01-14	3500		06-01	5040
06-01-25	1240		06-02	2470
06-01-30	300		06-03	3830
06-02-04	1000			
06-02-20	1470			
06-03-07	830			
06-03-14	3000			

Figure 2.2: Example of a *roll-up* query.

the granularity of a view for a specific set of feature dimensions. It causes records to be combined and their measures aggregated. Typical aggregation functions used are *sum*, *average*, *min*, *max* and *count*, while there are a variety of others depending on the application. For example, as illustrated in Figure 2.2, when analyzing a measure *Sales* with respect to the *Time* dimension at the *Day* level, all *Sales* from the same day might be summed together. Similarly, when rolling-up the *Time* dimension to the *Month* level all *Sales* of the same month will be summarized, leading to a more general view of the data.

As the converse of the *roll-up* operation, there exists a *drill-down* operation, which increases the granularity of a dimension and thus specializes the analysis of a measure. Other common OLAP operations are *slice* and *dice*, which select records based on specific values at one or more feature dimensions [185], as well as various range and range-aggregate queries [5].

To conceptually model multi-dimensional data in a data warehouse a star or snowflake schema is used [31]. These schemas are called “subject-oriented” as they facilitate the subject-driven analysis of data, where the measures are the subjects and the attribute dimensions are the parameters of the analysis. Traditional entity-relationship models, on the other hand, are called “application-oriented” as they capture the relationships between entities as they are modeled by the application accessing the data [81]. Star and snowflake schemas are composed of a central *fact table* and a number of *dimension tables*. The fact table is a large table representing the central subjects in the data warehouse. Each record in the fact table consists of one or more measures and a foreign key into each dimension table. Dimension tables

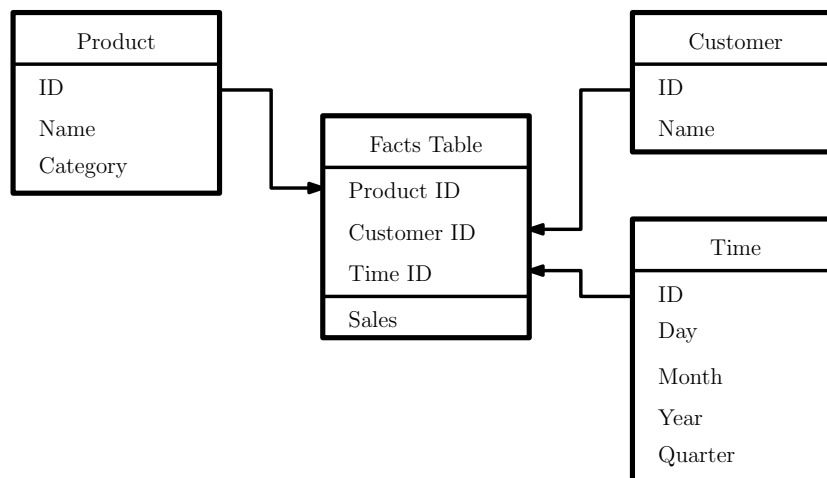


Figure 2.3: Star schema of a typical data warehouse.

are small and contain the attribute values of the feature dimensions. Figure 2.3 shows an example star schema.

OLAP systems are typically used interactively. During interactive online analysis the aggregated measures of a roll-up operation are required to be computed in a timely fashion for the user to maintain his chain of thoughts [130]. Also data mining algorithms may access particular views multiple times and require the data warehouse to respond quickly to such queries. OLAP queries, however, typically involve the aggregation of thousands or millions of records, which may render the fully online computation of views impractical. Preprocessing in the form of offline materialization addresses this problem and has become a common approach to minimize query time. During materialization, views of various combinations of dimensions and their hierarchy levels are precomputed using `GROUP BY` queries and stored to support faster access at a later time. For example, the `GROUP BY` query to compute the view shown in Figure 2.1 from the star schema presented in Figure 2.3 would be expressed in SQL [1] as follows:

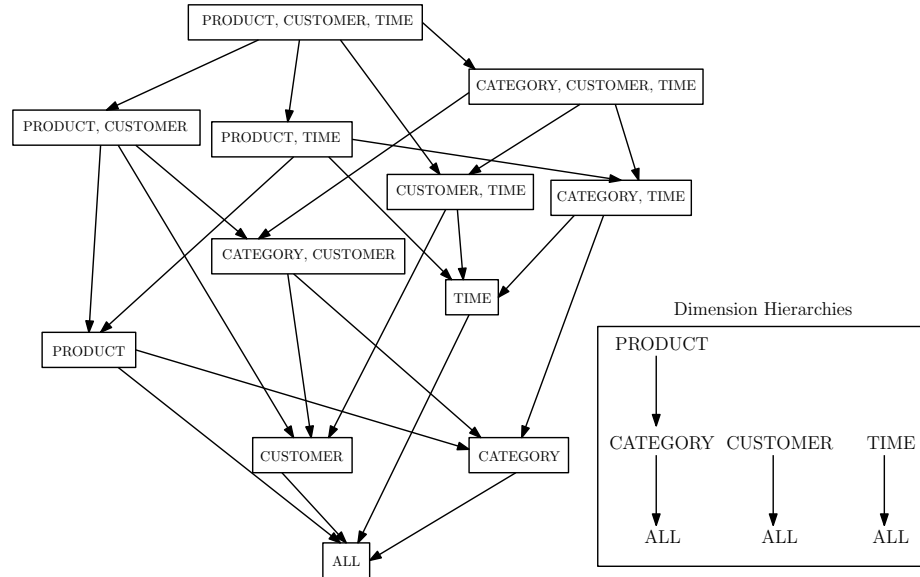


Figure 2.4: Data cube lattice with dimension hierarchies.

```

SELECT
    Customer.Name AS Customer,
    Time.Quarter AS Time,
    Product.Catergy AS Product,
    SUM(FactsTable.Sales) AS Sales
FROM
    FactsTable
LEFT JOIN Customer ON CustomerID = Customer.ID
LEFT JOIN Time ON TimeID = Time.ID
LEFT JOIN Product ON ProductID = Product.ID
GROUP BY Customer.Name, Time.Quarter, Product.Category

```

The saving in query time when utilizing precomputed views, however, comes at the cost of additional storage requirements. The entire set of views that can be constructed from the data in the data warehouse is commonly referred to as a *data cube* and can be represented by a lattice as illustrated in Figure 2.4 [71]. As the number of views increases exponentially with the number of dimensions and hierarchy levels in the data warehouse, it is evident that materializing all views in the data cube is infeasible due to storage space constraints. Specifically, if a data warehouse has  $d$  dimensions and the number of hierarchy levels in dimension  $i$  is  $H_i$ , then the total number of possible views is  $\prod_{i=1}^d H_i$ . Also, some of the views in the data cube may not be significant or do not represent an interesting analysis subject. Hence,



the general approach is to materialize a *partial data cube* which contains a selected subset of views from the *full data cube* [5, 31, 44]. The process of selecting views for materialization is referred to as *view selection* and algorithms typically use cost models based on view size, access frequency and benefit for other materializations. A further reduction of storage requirements for the materialization of data cubes can be achieved by computing Iceberg cubes [21, 34], which contain only those data values that satisfy a given constraint, such as minimum value.

## 2.2 SOLAP Applications

Recently there has been a growing interest in spatial OLAP [24, 68, 136, 150, 153, 161, 201]. Spatial OLAP (SOLAP) builds on the concepts introduced in OLAP systems but adds the ability to define and query over spatial dimensions and measures. Land management and urban planning in coastal regions, for example, may use aggregated historical data to obtain information about probable flooding areas based on spatial aggregation of coverages [190]. Transportation and logistic businesses can use pre-computed line aggregates to obtain instant routing information [30, 135] or optimize their allocation of resources in specific regions based on historical data analysis. Emergency services can be equipped with mobile low-power computing devices to access areal information about the emergency site for decision support [2]. Environmental modeling approaches can benefit from pre-computed spatial information that can be quickly accessed during simulation [9], such as sets of nearest neighbors. A chain of department stores can perform an interactive analysis of demographic data to determine a potentially profitable site to build a new store [111]. In health care, environmental epidemiology analyzes data aggregated in field studies and relates them to potential contaminant occurrences in the environment using areal information [131]. A similar method can be used in health surveillance to track contagious diseases, even on a global scale.

## 2.3 SOLAP System Architecture

Based on a typical high-level view of an OLAP architecture [81], Figure 2.5 illustrates the SOLAP analog. The system provides a graphical user interface to visualize and

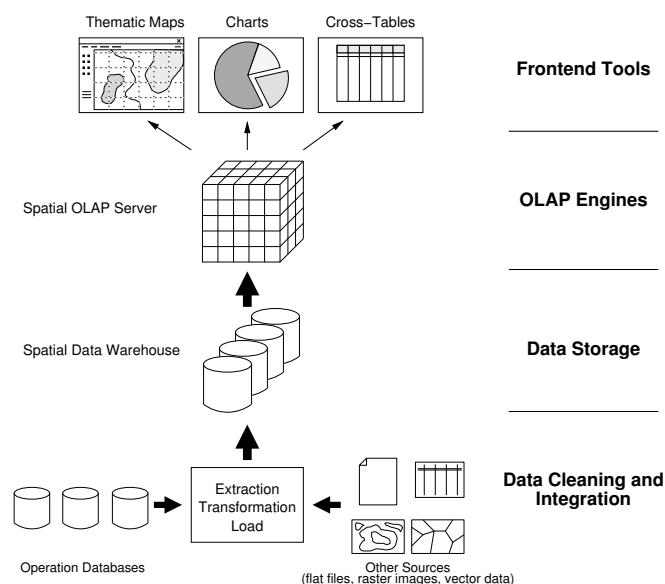


Figure 2.5: High-level architecture of a SOLAP system.

interactively explore spatial information and to perform OLAP operations on spatial and non-spatial data in a timely fashion. To do so, it is connected to a spatially enabled OLAP server – the SOLAP server. This server provides the technology for the user to execute SOLAP queries, processes the data accordingly and if necessary performs appropriate materialization of views. The data itself is stored in a spatial data warehouse which is accessed by the SOLAP server. The data warehouse is populated with data from transactional databases and/or other sources of spatial and non-spatial information. When entered into the data warehouse, the data passes through an extraction, transformation and loading (ETL) process, where noise and invalid records are removed from the dataset and its consistency is ensured. Based on this architecture a conceptual model for SOLAP is described in the following section.

## 2.4 SOLAP Solutions

There exists a number of Spatial OLAP solutions [24,49,95,100,157,161] that integrate the analysis of spatial data with OLAP functionality. Many of these systems provide extensive tool sets for integration with existing ETL processes, business intelligence and database solutions. However, this integration is typically provided at a very high level. JMap [100], SOVAT [161], SAS Web OLAP Viewer [157] and Map4Decision [95]

provide Spatial OLAP functionality by combining existing OLAP solutions such as Microsoft SQL Server Analysis Services [121], Mondrian [92] or SAS Enterprise BI Server [157] with GIS solutions such as ESRI ArcGIS [54] or spatial database systems like Oracle Spatial [134]. The complexity of this integration is hidden from the user through a unified user interface. However, this high level of integration relies on a single component to ensure the integration of two separate subsystems that do not communicate directly. In addition, spatial and non-spatial data are stored by different subsystems and the architecture does not provide a uniform representation of spatial and non-spatial data at the storage level. This architecture is limited by the interfaces each subsystem provides and does not scale well with increasing data volume, query load and changing application requirements. Other solutions, such as GeoMondrian and GeWolap provide integration of spatial and non-spatial data types at a lower level. They directly interface with the Mondrian OLAP server [92] and extend the capabilities of the OLAP server to handle spatial data. The OLAP server, in turn, uses a spatially enable database system, e.g., PostGIS [151] or Oracle Spatial [134], as storage for data and to execute data queries. In this architecture spatial and non-spatial data is represented in a uniform manner at the database level and queries involving spatial operators or functions are evaluated by the database engine. The OLAP server's responsibility is to translate Spatial OLAP queries issued by the user into sequences of SQL queries that are being evaluated by the underlying database engine. In addition, the OLAP server can build pre-materialized views of the data cube that are stored in the database system for faster access. Mondrian is a powerful OLAP server originally developed for traditional OLAP applications. The extensions that GeoMondrian and GeWolap provide only enable support for spatial data types but do not change the strategies used by Mondrian to evaluate OLAP queries. This prevents these systems from efficiently answering queries on complex spatial dimension hierarchies, such as non-strict dimension hierarchies.

## 2.5 The SOLAP Data Model

In order to put SOLAP on a firm foundation it is important to define the spatial extensions to the OLAP data model. While there have been many partial proposals, there is no single widely agreed-on Spatial OLAP data model. Such a model would

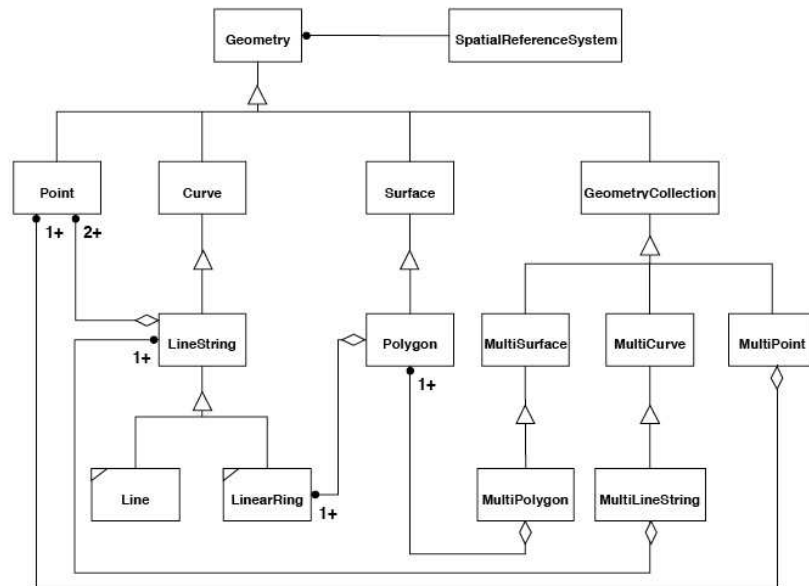


Figure 2.6: Geometry class hierarchy for SQL.

have to address spatial data types, measures, feature dimensions and operations.

In 1997 Stefanović described in his thesis [170] a basic model for a spatial OLAP system which allows feature dimensions and measures to contain spatial components. Stefanović did not provide formal definitions of the spatial components that can be handled by such a system. His thesis focussed on the treatment of arbitrary spatial regions represented as 2-dimensional polygons. Two years later in 1999 the Open GIS Consortium proposed the OpenGIS Simple Feature Specification for SQL [132], an extension to SQL for the handling of spatial data in a relational database system. This specification provides formal definitions for spatial data types commonly used in spatial databases [73, 114, 152]. These basic data types are:

- point
- line
- polygon
- any collection of the above.

The complete geometry class hierarchy proposed by the Open GIS Consortium in [132] is shown in Figure 2.6.

### 2.5.1 Spatial Measures

With measures now being spatial objects, aggregation functions for spatial data types are required to compute spatial aggregate measures. In [170] only *union* was considered as a spatial aggregation function, while [73] and [132] also considered *intersection*. Additionally, Güting described the aggregation functions *mindist* and *maxdist* [73]. However, it is not clear whether these functions return spatial objects as a result, i.e. the two closest respectively farthest objects in the input, or a scalar measure, i.e. the distances between the two closest or farthest objects. Furthermore, similar to the *average* function in conventional databases, Güting proposes a number of spatial algebraic aggregation functions that are combinations of other functions: *convex-hull*, *area*, *perimeter* and *centre*.

### 2.5.2 Spatial Feature Dimensions

A spatial data warehouse may, similarly to spatial measures, also contain spatial feature dimensions, i.e. the dimension values may be spatial objects. An example for such a spatial feature dimension could be a set of small land coverages each associated with the amount of vegetation in its area. In combination with concept hierarchies of feature dimensions it has been suggested in [114] and [17] that spatial data warehouses must support the following types of dimension hierarchies:

- **non-spatial dimension hierarchy** – concept hierarchy as defined in conventional data warehousing;
- **spatial to non-spatial hierarchy** – concept hierarchy that has some number of lower levels that are spatial and remaining higher levels that are categorical as in conventional data warehousing;
- **spatial to spatial hierarchy** – concept hierarchy where all levels are spatial.

While concept hierarchies in conventional data warehouses are generally represented by a surjective mapping between hierarchy levels, this is not always possible in spatial dimensions. Due to the extent of spatial objects, such as lines and polygons, relationships between two hierarchy levels may be described by a mapping where the value of a lower hierarchy level maps to one or more values of a higher hierarchy

level. The mappings between levels of explicit dimension hierarchies can be arbitrary and are pre-determined by the user. The hierarchy  $City \rightarrow County \rightarrow Province$ , for example is explicit, since the knowledge of which cities belong to which county and which counties belong to which province is pre-determined. Although it is not a requirement, in most cases of explicit spatial dimension hierarchies a topological relationship between the hierarchy levels also exists. The Dimensionally Extended 9 Intersection Model (DE-9IM) [36], introduced by Clementini and Di Felice, defines a set of standard topological relationships between different types of spatial objects. When providing predicates to determine such relationships, one can use these predicates to automatically generate topological hierarchies [117].

In some applications it may not always be possible to define an explicit concept hierarchy for a given spatial dimension. However, to still allow the analysis of data at different levels of granularity, implicit hierarchies can be generated by using, for example, clustering methods, data structures or topological operators [136, 173]. Implicit hierarchies are not pre-determined by the user and their structure depends on the algorithms used to generate them. An example for the use of an implicit hierarchy can be found in land management. When analyzing different types of land coverages their arrangement does not follow a particular pattern or a known hierarchical structure, such that generating an implicit hierarchy is necessary to support analysis at different levels of resolution.

### 2.5.3 SOLAP Operations

A spatial OLAP system must be capable of performing analogs of standard OLAP operations such as:

- roll-up
- drill-down
- slice & dice

on all types of dimension hierarchies as described above [153, 170]. The selection of records by a *roll-up* or *drill-down* query is determined by the concept hierarchies of the involved dimensions. The selection of records for a *slice* or *dice* query, on the

other hand, is selected by predicates, implying that a spatial data warehouse must provide appropriate predicates that can be applied to spatial objects. A set of typical topological predicates is described in [36] and [132].

The conceptual schema-based modeling of spatial data warehouses was addressed in detail by Malinowski and Zimányi [114]. They proposed an ER-based (entity relationship) approach to modeling spatial dimensions and spatial dimension hierarchies and provided a notation to express relationships among those. This approach permits us to describe a spatial data warehouse with a star or snowflake schema and apply conventional data warehousing and OLAP methodologies to it.

One important feature of their methodology is pre-materialization of the data cube and the schema representation of the data warehouse allows for an easy construction of the data cube lattice. The materialization and aggregation of measures, independently of their types, is generally constrained by the summarizability of the data and the space requirements of the aggregates. Pedersen and Tryfona studied the summarizability problem for spatial data types and provided the results of their analysis in [139]. The problem of space constraints for the aggregation of spatial data is addressed similarly as in conventional data warehouses, by using view selection. The additional types of measures available in spatial data warehouses, however, are much more complex than scalar measures in conventional data warehouses, such that partial cube materialization may not lead to a sufficient reduction of the required space. To overcome this dilemma, Stefanović [170] and later Han, et al. [83] propose selective materialization to materialize only selected records of a view. Another approach for the reduction of space requirements is the storage of approximate spatial aggregates using multi-resolution amalgamation as proposed in [201] and [147], and to compute exact aggregates on demand.

To optimize the computation of aggregate values in a spatial data warehouse several algorithms and data structures have been proposed. López et al. survey a number of them and provided formal definitions for various classes of aggregation functions, including spatial aggregation [112]. Many of the discussed approaches for optimizing the aggregation process are based on underlying indexing data structures and allow the efficient retrieval of aggregate values based on range queries. The indexing structures used are commonly tree-based and as such generate an implicit hierarchy on

the spatial dimension. Papadias et al. have proposed one such data structure, called the aggregation R-tree (aR-tree) [136]. The aR-tree annotates intermediate nodes of an R-tree with the aggregate value of the measures that are enclosed in a node's minimum bounding rectangle (MBR). For range aggregate queries this data structure can be very efficient in practice as it does not always require one to traverse the entire tree down to the leaf level in order to answer a query. Other, similar data structures, focussing on different aggregation functions were proposed in [199] and [173]. While all these approaches support efficient range aggregate queries, they fail to address explicit concept hierarchies or the indexing of categorical data. Hence, they are not directly applicable to query evaluation in a spatial data warehouse that combines categorical with spatial data and defines spatial concept hierarchies. However, during the pre-materialization of views, similar algorithms and data structure may be used to speed up the materialization process. Furthermore, as López et al. pointed out, these discussed approaches do not represent the query result in the same context as the original data making *sequenced queries* impractical. The representation of query results as a new relation is, however, essential to online analysis as one query may build on the result of a previous query.

#### 2.5.4 SOLAP View Indexing

Pre-materialization by itself is not sufficient to speed up query evaluation. Only in combination with effective indexing techniques it is possible to query pre-materialized views efficiently. Conventional database systems commonly use B-tree based data structures [15] for indexing views, while in spatial databases R-tree based approaches are very common [16, 76, 152, 164]. One advantage of R-trees is that they can also be used to index non-spatial (categorical) data [25]. Specifically for relational OLAP, where each materialized view of a data cube is stored as a multidimensional table, Dehne et al. proposed the RCUBE index, an indexing structure combining the benefits of space-filling curves, B-tree indices and parallel processing [45]. None of these data structures, however, support efficient on-the-fly evaluation of OLAP operations such as *roll-up*. This becomes necessary in the case of partial cube materialization when non-materialized views have to be computed from already existing views on-the-fly at query time. Additional requirements for indexing structures in spatial



data warehouses include the ability to uniformly deal with continuous data as well as discrete data and to provide efficient index construction, update and query techniques.

### 2.5.5 SOLAP Query Languages

With respect to interfacing with external applications such as user interfaces or report generators, many conventional OLAP systems have been adopting query languages similar to SQL [71] or XML for Analysis [125]. However, none of these query languages provide sufficient support for the analysis of spatial data and need to be extended to be used in spatial data warehouses. The OpenGIS Simple Feature Specification for SQL [132] could provide the directions for such extension as it extends SQL with spatial analysis features. In addition to a query language, a Spatial OLAP system may also provide a language for the description of how query results should be represented. The Graphical Presentation Language proposed in [51] provides one approach that combines both, a spatial query language and a presentation language.

## 2.6 Summary and Open Questions

In summary, a number of issues related to the design and implementation of a spatial data warehousing and OLAP system can be identified. The most dominant issue is the integration of different components such as data representation, materialization, indexing, querying and user interfaces into a coherent and uniform system with well defined interfaces. So far, approaches have either targeted only specific and very isolated problems of spatial data warehousing [136, 170], without the integration of other parts into a functional system, or the system implementation was driven by a specific application, not paying much attention to a uniform data representation, general applicability and efficient algorithms [153, 160]. Other open problems that must be addressed in order to design an integrated spatial OLAP system are:

- uniform treatment of spatial and non-spatial data throughout all components of the data warehouse and its functionality;
- efficient representation of materialized spatial measures on disk;
- representation of query results in the same manner as views;

- standardized query interface;
- efficient algorithms for on-the-fly spatial aggregate computation;
- efficient update mechanisms for materialized spatial views;
- scalable and cost-effective computing platform as the performance of CPUs and sizes of internal memory grow slower than external storage space and data volumes.

In the remainder of this thesis we address these issues. Chapter 3 discusses the evaluation of OLAP queries in the presence of spatial dimension hierarchies. In Chapter 4 we introduce a computational model that facilitates the evaluation of non-spatial and spatial OLAP queries in an efficient scalable manner that uses a consistent data model. Chapter 6 then discusses a new multidimensional indexing mechanism that allows a uniform representation of spatial and non-spatial data. Finally, in Chapter 7 we propose an application of spatial OLAP in combination with spatial data mining and define a new class of `GROUP BY` operators for the use in spatial OLAP systems.

## Chapter 3

### Spatial Dimension Hierarchies

Key to the definition of a coherent and uniform spatial OLAP system is a coherent and uniform approach to the definition and querying of spatial dimension hierarchies. This chapter contains no algorithms, implementations or experiments, rather it provides a detailed analysis of different types of dimension hierarchies in spatial OLAP systems and their impact on the evaluation of spatial `ROLLUP` and `CUBE` queries. Based on a specific example, it explores different types of spatial dimension hierarchies and determines how typical OLAP queries can be evaluated in their presence. The proposed representation of hierarchies is expressive, in that it supports a wide range of common SOLAP applications, and as will be demonstrated in the remainder of this thesis, it admits an efficient implementation.

In the following section we describe existing approaches to the definition of spatial dimension hierarchies. In Section 3.2 we introduce our application example which serves as a basis for our analysis. We describe the modeling of spatial dimension hierarchies in Section 3.3 and their representation in a data model in Section 3.4. Section 3.5 then analyzes `ROLLUP` and `CUBE` queries for each spatial dimension hierarchy in our example and Section 3.6 concludes the chapter with a summary of our findings.

#### 3.1 Approaches to the Definition of Spatial Dimension Hierarchies

There exists a significant amount of work addressing dimension hierarchies in the context of Spatial OLAP. Most of this work focussed on how to construct spatial dimension hierarchies. The approaches can be loosely categorized into those based on ad-hoc hierarchies and those based on well defined topological relationships.

Ad-hoc hierarchies are commonly derived from spatial indexing data structures and most prominently use R-trees [137, 150]. They are primarily used when spatial objects have to be analyzed at various levels of granularity, but there does not exist

an underlying concept structure within the data. Hence, their structure is driven by the data and changes along with the data. Ad-hoc hierarchies are not structured by fixed concepts known to the user.

The second approach is the construction of hierarchies based on topological relationships. This approach assumes there exists some concept structure within the data that can be described by topological relationships [117, 139]. Most explicit hierarchies that are predefined by the user are based on topological relationships, e.g. *Municipality*  $\subset$  *County*  $\subset$  *Province*, and could, if necessary, be generated implicitly.

Also, there exist spatial dimension hierarchies that can only be explicitly defined and cannot be constructed as an ad-hoc or topological hierarchy. An example of such a hierarchy is *Postal Code*  $\rightarrow$  *County*, as there may exist an arbitrary mapping between a postal code and a county, depending on the location of the postal distribution centre.

Malinowski and Zimányi provided a first model for the representation of spatial hierarchies independent of their implementation [114]. This model is derived from the Entity-Relationship Model [33] and seamlessly integrates with the traditional modeling of concept hierarchies. The authors introduced a symbolic notation for the description of spatial dimension hierarchies. They failed to provide a formal definition of this notation. By considering a number of examples, the authors furthermore suggested a classification of spatial dimension hierarchies and distinguished between the following significant types:

- asymmetric
- generalized
- non-strict
- multiple-alternative.

Malinowski and Zimányi addressed more types of spatial hierarchies in [114]. However, these hierarchies are constructed from the above types.

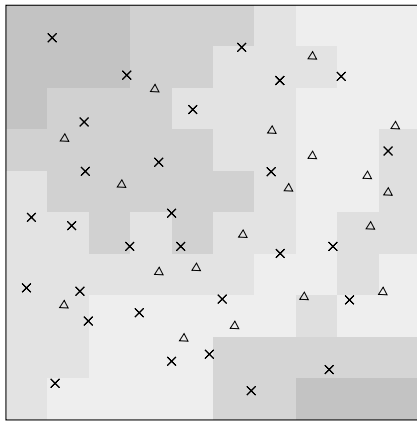
### 3.2 Our Application Example

To explore the evaluation of OLAP queries in the presence of spatial dimension hierarchies we will use an example. By carefully looking at some typical SOLAP applications and trying to extract common requirements, we constructed a small and simple example application that captures these requirements. We largely adopt the core concepts of Malinowski and Zimányi [114] but operationalize these concepts in specific example queries.

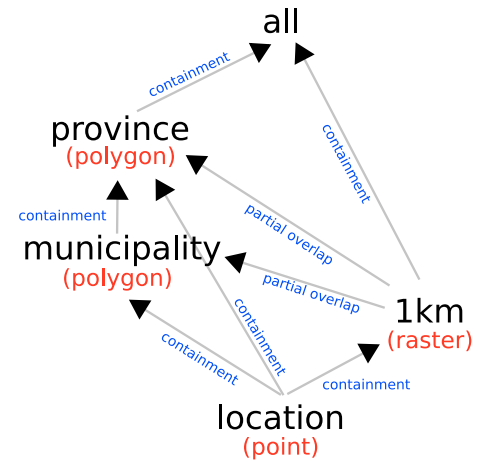
In our example, a forester is doing a survey of vegetation cover. He collects observations at random points that include the vegetation type and the approximate vegetation density. He also has access to a satellite image providing a raster of elevation data at a 1 km resolution. Based on this data, the forester wishes to answer queries such as:

- What is the most common plant species grouped by province?
- What is the average plant density relative to the elevation?
- What type of vegetation can be found in a specific municipality?

The forester example relates to a common scenario in land cover analysis or forestry and involves two measures. The first measure is a satellite raster image covering the entire area of interest. Each cell (pixel) of the raster is associated with a value corresponding to the average elevation in this cell. Hence, the raster image can be seen as an elevation map. The second measure is a set of sample locations distributed throughout the area of interest, each recording the primary type of vegetation and the approximate vegetation density at the location. Figure 3.1a shows a section of a map representing both measures at their finest level of granularity. The measures have very different spatial characteristics. The elevation measure uses a partitioning of the space and associates each partition with a measure value. Each vegetation density measure, on the other hand, is associated with a single point location and the locations are randomly distributed. Now the question is how to model the spatial dimension hierarchies implemented in the forester example.



(a) Example dataset including elevation map and sample locations.



(b) Spatial dimension hierarchies.

Figure 3.1: The forester application example.

### 3.3 Modeling the Spatial Dimension Hierarchy

Since the point locations of the vegetation density measure are lower-dimensional objects than the partitions of the raster, a hierarchy relationship between the raster and the sample locations can be defined. Additionally other hierarchy levels, such as *Province* and *Municipality*, can be defined on the spatial dimension to allow OLAP analysis of the data in different contexts. The complete hierarchy of the spatial dimension in our example is shown in Figure 3.1b. For this hierarchy, the membership between its levels can be determined using topological relationships. For most relationships full containment is a logical choice, as elements of one level of the hierarchy are each well contained in one element of a higher hierarchy level. This is the case, for example, for the relationship between municipalities and provinces. On the other hand, for the relationships between raster cells and municipalities or provinces, full containment is not applicable. This is due to the fact that the space partitioning of the raster does not “align” with that of the *Municipality* or *Province* hierarchy levels. This means that a single raster cell may not be well contained within a municipality or province and instead only partially overlap with them. In cases of partial overlap a membership function has to be defined to determine the membership of a lower level element in a higher level element. In the simplest case this function is a surjection,

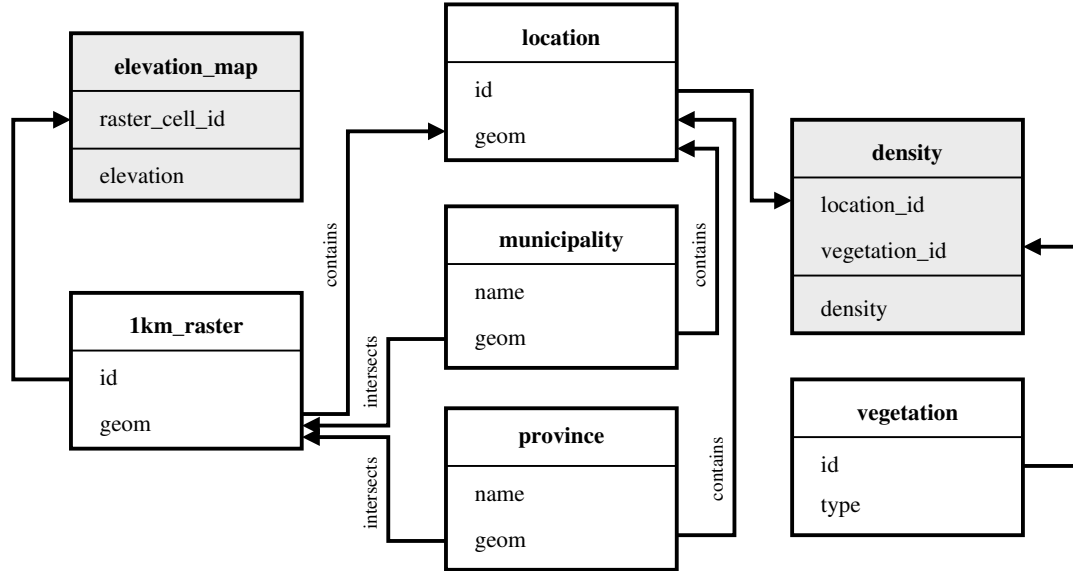


Figure 3.2: Fact constellation schema of the application example.

which matches each element of the lower level to only one element of the higher level. A typical example of such a function is *max-overlap*, which associates the lower-level element with the most overlapping higher level element. Such a function, however, may not always be desirable, as it affects the correctness of the result by not considering other existing relationships. Alternatively, a partial membership function can be defined to associate a lower-level element with all higher-level elements it intersects with. This approach, however, requires special care when evaluating **GROUP BY** aggregate queries, as shown in Section 3.5.4.

### 3.4 The Data Model

Spatial dimension hierarchies can, similarly to conventional concept hierarchies, be modeled using snowflake or constellation schemas. The constellation schema for our example is shown in Figure 3.2 and is used as the basis for answering queries in the following. For simplicity, however, all dimensions in the example have been denormalized for each measure, such that complex **JOIN** clauses can be avoided when formulating the queries. To denormalize dimensions with respect to the vegetation density measure, the following SQL query, extended by the OpenGIS Simple Feature for SQL [132], can be used on the above constellation schema:

```

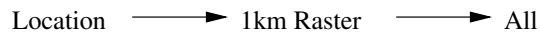
CREATE VIEW vegetation_density AS
SELECT
    province.name AS province,
    municipality.name AS municipality,
    location.id AS location_id,
    vegetation.type AS vegetation_type,
    1km_raster.id AS raster_cell_id,
    density.density AS density
FROM density
LEFT JOIN location ON density.location_id = location.id
LEFT JOIN vegetation ON density.vegetation_id = vegetation.id
LEFT JOIN 1km_raster ON CONTAINS(1km_raster.geom, location.geom)
LEFT JOIN municipality ON CONTAINS(municipality.geom, location.geom)
LEFT JOIN province ON CONTAINS(province.geom, location.geom)

```

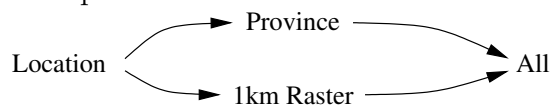
This query computes a constrained Cartesian product across all entities, where records are only combined if they satisfy the constraints provided in each of the `ON` clauses. For example, a `density` record is only combined with a `location` record, if the `density` record's `location_id` attribute is equal to the `location` record's `id` attribute.

The spatial dimension hierarchies chosen in the example can be classified into the following types as identified by Malinowski and Zimányi [114]:

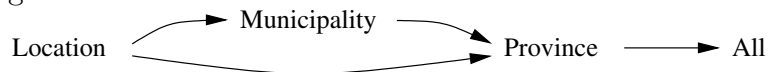
- asymmetric



- multiple-alternative



- generalized



- non-strict



We will now focus on each of these spatial hierarchies and examine the evaluation of `ROLLUP` and `CUBE` queries.

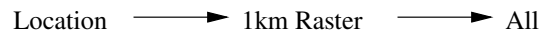


### 3.5 ROLLUP and CUBE Queries in the Forester Example

In the following we discuss the formulation and expected results for ROLLUP and CUBE queries for each of the hierarchy types in our example.

#### 3.5.1 Querying Asymmetric Hierarchies

In the forester example every location (i.e., observation of vegetation type and density) is within a 1 km raster cell defined by the elevation map. Not every raster cell contains a location.



This type of relationship is what characterizes an *asymmetric* hierarchy.

A ROLLUP query on this hierarchy can be formulated as:

```
SELECT raster_cell_id, location_id, count(*) AS count
FROM vegetation_density
GROUP BY raster_cell, location WITH ROLLUP
```

Since the lowest level in this ROLLUP query is *Location*, only records that have in fact a location and an associated measure will be considered. The reason for this is that any other record that is not associated with a location does not contribute a measure value to the query result. The result of this query is

raster_cell_id	location_id	count
3	7	1
5	5	1
8	3	1
8	4	1
11	6	1
11	2	1
11	8	1
15	1	1
3	*	1
5	*	1
8	*	2
11	*	3
15	*	1
*	*	8

and only contains those records for which *location\_id* is defined (not NULL). The symbol \* denotes NULL or undefined values and indicates in this representation an aggregated attribute dimension.

Similarly, a CUBE query is formulated as:

```

SELECT raster_cell_id, location_id, count(*)
FROM vegetation_density
GROUP BY raster_cell, location WITH CUBE

```

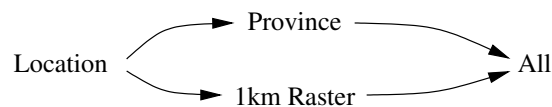
and has the result

raster_cell_id	location_id	count
3	7	1
5	5	1
8	3	1
8	4	1
11	6	1
11	2	1
11	8	1
15	1	1
3	*	1
5	*	1
8	*	2
11	*	3
15	*	1
*	1	1
*	2	1
*	3	1
*	4	1
*	5	1
*	6	1
*	7	1
*	8	1
*	*	8

We observe that ignoring all records that do not associate with a valid location is equivalent to using a `WHERE` clause that explicitly selects those records for which `location_id` is not `NULL`.

### 3.5.2 Querying Multiple Alternative Hierarchies

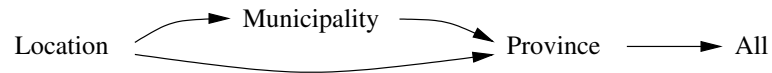
Multiple alternative hierarchies are combinations of other types of hierarchies where the user selects which hierarchy path is considered in the analysis at runtime. In the forester example a multiple alternative hierarchy is



and for any query the user has to choose either  $Location \rightarrow Province \rightarrow All$  or  $Location \rightarrow 1km\ Raster \rightarrow All$ . Choosing either path leads to an *asymmetric* hierarchy that can be evaluated as in Section 3.5.1.

### 3.5.3 Querying Generalized Hierarchies

In generalized hierarchies an element of a child level may be associated with either a parent at a directly adjacent level or a parent at some other higher level of the hierarchy. That is, the immediate parent element of some child element may be the grandparent element of some other child. For example, a location in downtown Halifax is in the Halifax Regional Municipality while a location in Kejimikujik National Park is in Nova Scotia, but not within any municipality.



For the evaluation of ROLLUP and CUBE queries on generalized hierarchies, place holders have to be introduced to act as virtual parents for child level elements. That is, in our example we need to introduce virtual municipalities for all those locations that are directly associated with provinces. Introducing such virtual parents allows for the evaluation of queries similar to:

```

SELECT province, municipality, location_id, count(density) as count
FROM vegetation_density
GROUP BY province, municipality, location WITH ROLLUP
  
```

resulting in

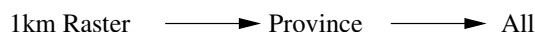
province	municipality	location_id	count
Nova Scotia	virtual	7	1
Nova Scotia	virtual	8	1
Nova Scotia	Halifax	3	1
Nova Scotia	Halifax	2	1
New Brunswick	virtual	4	1
New Brunswick	Fredericton	6	1
New Brunswick	Saint John	1	1
Newfoundland	virtual	5	1
Nova Scotia	virtual	*	2
Nova Scotia	Halifax	*	2
New Brunswick	virtual	*	1
New Brunswick	Fredericton	*	1
New Brunswick	Saint John	*	1
Newfoundland	virtual	*	1
Nova Scotia	*	*	4
New Brunswick	*	*	3
Newfoundland	*	*	1
	*	*	8

In this result, the municipality “virtual” has been introduced to provide an intermediate association for locations that are not directly associated with “real” municipalities. This allows for intuitive query results, since all locations that provide a measure

value can still be considered at a hierarchy level with which they may not be directly associated.

### 3.5.4 Querying Non-Strict Hierarchies

Non-strict hierarchies are hierarchies in which a child level element may not be mapped to only one parent level element but to multiple. This is commonly the case when the child level element intersects with multiple elements at the parent level and a partial membership function is used. It arises frequently in practice when dealing with spatial subdivisions that have been independently defined. For example, a single postal code may intersect two counties and the borders of a physical element, such as a forest, may not follow municipal boundaries.



In these cases the challenge is how to handle partial membership with respect to `GROUP BY` queries. For example, if a 1km raster cell falls in two provinces, how should a location count associated with the raster cell contribute to each province's aggregate count? There are two obvious approaches: 1) Allocate the raster's count measure to both provinces, or 2) attempt some kind of proportional distribution based on the degree of overlap. The problem with the first method is double counting. The sum of the aggregate location counts across all provinces is greater than the sum of all locations. The problem with the second method is the introduction of approximation in that just because the raster overlaps one province by 75% does not necessarily mean that 75% of the locations are in that province. In many applications approximation is an appropriate solution.

A `ROLLUP` query assuming absolute membership between a raster cell and the provinces it intersects may produce the following result:

```

SELECT province, raster_cell_id, COUNT(*) as count
FROM height_map
GROUP BY province, raster_cell WITH ROLLUP

```

province	raster_cellId	count
Nova Scotia	2	1
Nova Scotia	3	1
Nova Scotia	5	1
Nova Scotia	1	1
New Brunswick	4	1
New Brunswick	5	1
Newfoundland	2	1
Nova Scotia	*	4
New Brunswick	*	2
Newfoundland	*	1
*	*	7

It can be easily seen that some raster cells are accounted for multiple times, once for each province they intersect. In the example there exist only 5 distinct raster cells matching the query, but the count evaluated by the query is 7. To prevent this problem of double counting raster cells that partially overlap with multiple provinces, we can annotate the aggregation function, i.e. COUNT, to incorporate the amount of partial membership into the computation of the aggregate. An aggregation function PARTIAL\_COUNT( $R$ ,  $M_{\text{overlap}}$ ) can be introduced, for raster cells  $R$ , given a membership function  $M_{\text{overlap}}$ , which only considers the partial contribution of a record to an aggregation as follows:

$$\text{PARTIAL\_COUNT}(R, M_{\text{overlap}}) = \text{COUNT}(R) \cdot M_{\text{overlap}}(R).$$

In the given example we can define the membership function  $M_{\text{overlap}}$  as:

$$M_{\text{overlap}}(R) = \frac{\text{AREA}(\text{INTERSECTION}(\text{parent}(R), R))}{\text{AREA}(R)}.$$

Modifying the previous query to use PARTIAL\_COUNT will then produce the correct result for the ROLLUP query:

```
SELECT province, raster_cell, PARTIAL_COUNT(*, M)
FROM height_map
GROUP BY province, raster_cell WITH ROLLUP
```

province	raster_cellId	count
Nova Scotia	2	0.60
Nova Scotia	3	1.00
Nova Scotia	5	0.25
Nova Scotia	1	1.00
New Brunswick	4	1.00
New Brunswick	5	0.75
Newfoundland	2	0.40
Nova Scotia	*	2.85
New Brunswick	*	1.75
Newfoundland	*	0.40
*	*	5.00

### 3.5.5 Complex Queries and Spatial Measures

All types of queries shown above can be combined and additional predicates such as `WHERE` and `HAVING` clauses can be specified. Conceptionally, however, these can be treated independently of the `GROUP BY` clause, as the `WHERE` clause selects the candidate records before the `GROUP BY` is evaluated and the `HAVING` clause is applied independently to each group that is found.

When spatial measures are present, they are treated similarly to scalar measures. In the case of non-strict hierarchies, however, the membership function provided to the extended aggregation function must return the partial spatial measure's component that corresponds to the partial membership, e.g., the intersection of the child with the parent object.

## 3.6 Summary

In this chapter we have presented a simple example which nevertheless gives rise to four different types of spatial dimension hierarchies, namely *asymmetric*, *multiple-alternative*, *generalized*, and *non-strict*. We have explored how each hierarchy type influences the evaluation of `ROLLUP` and `CUBE` queries, and how the query results can be expressed in a tabular form that is consistent with standard OLAP representation. In Chapter 4 we introduce a computational model that allows for the evaluation of both spatial and non-spatial OLAP queries and includes support for the evaluation of such queries on non-strict hierarchies.

## Chapter 4

### A Model for the Pipelined Evaluation of Spatial OLAP Queries

Much of the existing and current research on the design and implementation of spatial OLAP systems focussed on the integration of already existing OLAP and GIS components into a single application using specialized middleware software [115, 154]. This approach is often cost-effective, quick to implement and allows the user to perform spatial OLAP operations, while hiding the complexity of the interactions between the components. However, with the continuous growth of spatial and non-spatial data stored in data warehouses and used for analysis, the middleware approach quickly reaches its limitations with respect to efficiency and scalability. Existing middleware solutions are not designed to utilize current and future computer hardware well, which is evolving toward multi-core architectures that facilitate concurrent processing with limited main memory resources. This creates a demand for spatial OLAP systems that are scalable and efficient.

In the database community these issues of efficiency, scalability and concurrent processing are often addressed by evaluating database queries in a pipelined manner. The fundamental idea of this concept is to split the work associated with the evaluation of a database query into multiple tasks (e.g., retrieval, join, aggregation). Each task is performed by a specialized component, and data between the components is transferred using data streams. Additionally, the pipeline approach allows different components to process independent intermediate results concurrently, thus interleaving the processing of individual tasks and improving the utilization of modern multiprocessor and multicore hardware architectures.

In this chapter we adopt the pipeline approach for the evaluation of spatial OLAP queries. We analyze a set of typical spatial OLAP queries and propose for each query a set of components and an evaluation strategy using the pipeline model.

These typical queries include queries against *asymmetric* and *non-strict* hierarchies

as discussed in the previous chapter. Queries against *multiple-alternative* hierarchies reduce to the other hierarchy types as soon as the user selects the branch to be taken. As described in Chapter 3, the queries against *generalized* hierarchies can typically be handled by the introduction of “virtual” values and are not addressed as a special case here.

During our investigation we focus primarily on what we believe are critical requirements for a spatial OLAP system:

1. **Performance** – The use of a modular architecture and intelligent implementations facilitate the use of efficient algorithms to achieve high performance in data processing and query evaluation.
2. **Flexibility** – The architecture enables the recombination of existing components to obtain results using different strategies and allowing for an optimal cost balance.
3. **Extensibility** – The ability to extend at a component level allows for an easy addition of new functionality and improved performance.

In the following sections we provide an in-depth discussion of how the pipeline model can be applied to spatial OLAP queries. Specifically, Section 4.1 provides an overview of the pipeline model and describes the concepts it relies on. In Section 4.2 we describe the underlying data model that enables the exchange of data between the different components involved in a pipelined spatial OLAP query. Section 4.3 provides details about different types of components that are required to implement the pipeline model for spatial OLAP queries. Section 4.4 provides an in-depth discussion of representative OLAP and spatial OLAP queries and their realization in the pipeline model. The chapter concludes with a summary of our findings in Section 4.5. In Chapter 5 we describe a proof-of-concept implementation of the pipeline model for spatial OLAP, provide a performance analysis of our implementation and a comparison with other approaches.

## 4.1 The Pipeline Model

The pipeline approach for the evaluation of database queries was initially proposed by Boral and DeWitt in 1980 [26]. Since then, the model has been adopted by a



number of database engines, commercial and academic ones. Many of the available traditional OLAP systems have also grown out of database management systems that employ a pipelining approach to query evaluation.

The fundamental idea of the pipelining model is to divide the evaluation of a complex query into a number of smaller and simpler tasks, each an independent component consuming a set of input data and producing some set of output data. Between these components data is transferred using data streams, and each component can pass its results on to the next component as soon as they become available. This stream-oriented approach to data transfers allows an interleaving of the execution of the individual components as well as a concurrent execution on multiple processors or processing cores if those are available. Both cases are beneficial for the utilization of available processing resources, as some components can be executed while others are, for example, waiting on disk I/O or other slow resources.

The interaction between the individual components of a pipeline model is typically illustrated using a data flow graph, where the nodes of the graph represent the components that execute tasks and the edges represent the data streams. In database terminology such a data flow graph may also be referred to as a query plan or strategy and is subject to query optimization.

In the following we will refer to the components that are responsible for the execution of tasks as mini-engines. We choose this term as it is less ambiguous and captures the nature of the components, which is to autonomously act as part of a larger query evaluation engine. For mini-engines to be interoperable with each other, the interfaces between them have to be well-defined. To define these interfaces, mini-engines that have a similar purpose (e.g., indexed storage) but varying implementations (e.g., B-tree or hash table) are grouped into a class for which the interfaces to other classes of mini-engines are defined. The interface definitions for each class of mini-engines can be considered the rules by which a data flow graph can be constructed.

The abstraction of query evaluation tasks into mini-engines allows for the encapsulation of their implementation. A single mini-engine can be developed independently from other mini-engines and optimized with respect to specific requirements such as performance, cost, or accuracy. A set of mini-engines that together can be used to answer a specific query may form a unit that itself can be considered a mini-engine.

Thus, it is possible to describe query evaluation strategies as recursive constructions of mini-engines. Mini-engines that cannot be constructed by combining other mini-engines are called elemental mini-engines. In Section 4.3 we provide details of the elemental classes of mini-engines that are required to evaluate typical spatial OLAP queries.

The concept of mini-engines provides an unparalleled flexibility for query optimization by allowing the system to choose and combine mini-engines in such a way that query costs are minimized for any kind of query. Leaning on object-oriented design concepts, the mini-engines additionally allow for the extension of an existing system at a mini-engine component level. It restricts the implementation of extensions to small components with well-defined interfaces and avoids complex relationships and dependencies with other components which can often be observed in monolithic designs. This reduces the overhead and costs associated with adding new functionality to the system and makes the design attractive for systems that are required to be available over extended periods of time and continuously adapt to changing environments, such as new data types, different storage subsystems, or access patterns.

To illustrate the mini-engine concept, consider the following OLAP query as an example:

```
SELECT a1, a2, a3, AGGREGATE(f)
FROM View
GROUP BY ROLLUP(a1, a2, a3)
```

It retrieves all records from view `View` and performs aggregation on the facts attribute `f` for each group of records with respect to the attribute sets provided in the `GROUP BY` clause. In this example the `ROLLUP` function generates the attribute sets `{a1,a2,a3}`, `{a1,a2}`, `{a1}`, and `{}`, each representing a separate group-by aggregation. The results from each group-by aggregation are then merged into a combined result view. In the context of the pipelining model, the strategy for evaluating the above query can be represented as a data flow graph as shown in Figure 4.1. The data flow graph is a decomposition of the evaluation steps into distinct tasks that are mapped to mini-engines. The query `Q` is used to filter the records in the view. In this example it selects all records available in the input view, that is, the `DATA ACCESSOR` mini-engine must retrieve all records from the view. Generally, the `DATA ACCESSOR` mini-engine

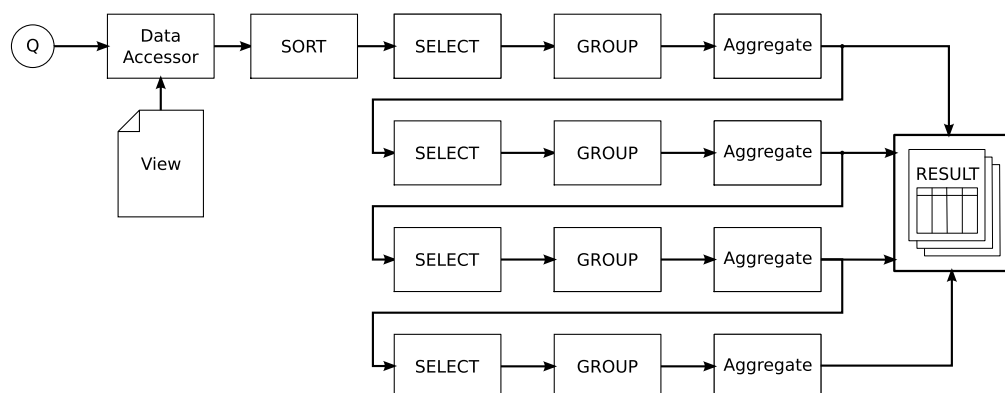


Figure 4.1: Example query decomposed into an assembly of mini-engines.

provides an abstraction of one or more data structures that provide access to data stored in the view. These data structures are typically indexing data structures that allow efficient access to data given selection criteria. In this example, however, this can be any data structure, even the simplest one, as records are not required to be filtered by any given predicate or expected to be processed in any particular order. These implementation details, however, are intentionally hidden in the context of the data flow graph, as it represents a conceptual query strategy. The records retrieved by the **DATA ACCESSOR** mini-engine are passed to the next mini-engine in the data flow graph. Depending on the sort order of the incoming record stream, the **SORT** mini-engine rearranges the input records according to the grouping set specified in the **GROUP BY** clause. The sort order information of each data input stream is provided as part of the metadata associated with the stream. In the next step the **SELECT** mini-engine selects from the incoming record stream only those record attributes that are relevant for the further processing of the query. In the first instance of the **SELECT** mini-engine for this example query, the attributes **a1**, **a2**, **a3**, and **f** are selected, and a new record stream containing only these attributes is emitted. A **GROUP** mini-engine is then used to obtain a grouping of the records in the data stream, which corresponds to the first most detailed level of aggregation groups. The grouped records are then passed to an **AGGREGATE** mini-engine which performs the actual aggregation of facts for each group. Note that the **ROLLUP** statement generates a number of grouping sets, each representing a different level of aggregation such that multiple sets of **SELECT**, **GROUP**, and **AGGREGATE** mini-engines are required to evaluate the query, one set for

each aggregation level. Since the `ROLLUP` function constructs grouping sets in such a way that each consecutive grouping set is a subset of the previous grouping set, while the attributes remain in the same order, it is possible to use the aggregated results from the previous grouping set as an input for the next grouping set. This property is being utilized in the shown strategy by splitting the result stream of one level of aggregation into a stream that is considered a result output stream and a stream which is used as input to the next level of aggregation. This process is repeated for each grouping set generated by the `ROLLUP` function. The result output by each aggregation is accumulated in a result stack that combines the results into a single result representation.

The property of the `ROLLUP` function to construct grouping sets such that a subsequent grouping set is a subset of the previous grouping set is often exploited during the pre-materialization of OLAP views. The materialization of views, that contain a prefix subset of the dimension attributes in an already materialized view, is typically much more efficient as it does not require the resorting of records. Our model uses a similar approach, by reusing aggregate values computed for a higher dimensional view to calculate aggregate values for appropriate subviews without resorting the records. Thus, the model also lends itself to be used during pre-materialization of views and cost models employed by view selection methodologies can take into account the strategies our model can support to facilitate efficient construction of subviews.

Due to the non-transactional nature of OLAP and the amount of data that is processed, OLAP systems typically operate in either a loading mode for efficient insertion of large amounts of data or a query mode for efficient query evaluation. The remainder of this chapter focuses on the query evaluation in OLAP systems and thus considers only read-only queries.

## 4.2 Data Model

The data model describes the format and properties of the data that can be passed between mini-engines and forms the basis of the data-flow-oriented approach to query evaluation. A data flow graph, which defines a particular query evaluation strategy, is composed of two types of distinct building blocks: streams and mini-engines.

Formally a stream  $s = (S, R, O)$  is defined by three properties: its schema  $S$ , a sequence of records  $R$ , and an order  $O$ . The schema  $S$  of a stream defines the format of the data records the stream transports and is described by a tuple of attributes  $S = (A_1, A_2, \dots, A_k)$  each of which is associated with a data type  $\mathcal{A}_i$ . The sequence of records  $R$  that is transported by a stream can contain two types of records: proper records and stop words. A proper record  $r$  is a data record that is composed of a set of attribute values  $r = (a_1, a_2, \dots, a_m)$  corresponding to schema  $S$  of the stream with  $r \in \mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_m$ . Thus, each attribute value  $a_i$  of  $r$  is of type  $\mathcal{A}_i$ . A stop word record is a special control record used only internally to the query evaluation to differentiate partitions of the data stream into continuous substreams. A stop word record indicates that the records preceding the stop word have, based on some criteria, certain properties in common, whereas the records following the stop word have different properties in common, based on the same criteria. For example, stop word records are used when grouping records together. Each group of records is separated from another group in the stream using a stop word record. A stop word record does not possess a schema and we denote such a record with the symbol  $\square$ . Thus  $R = \langle r_1, r_2, \dots, r_n \rangle$  with  $r_i \in \mathcal{A}_1 \times \mathcal{A}_2 \times \dots \times \mathcal{A}_m \cup \{\square\}$ .

The order  $O$  of a stream  $s$  with schema  $S$  is a potentially empty ordered list of attributes that specifies the order of records in the stream. The attributes in  $O$  must be a subset of the attributes in  $S$ . For a non-empty order  $O = (A'_1, A'_2, \dots, A'_p)$  the records transported by  $s$  can be assumed to be sorted in such a way that without loss of generality the predicate  $\text{leq}_k(O, r_i, r_j)$  defined recursively as follows is true for any pair of records  $r_i, r_j$  with  $r_i, r_j \in R$ , and  $i \leq j$ :

$$\text{leq}_k(O, r_i, r_j) = \begin{cases} \text{true}, & \text{if } k > |O| \text{ or } a'_{k,i} < a'_{k,j}; \\ \text{false}, & \text{if } a'_{k,i} > a'_{k,j}; \\ \text{leq}_{(k+1)}(O, r_i, r_j), & \text{if } a'_{k,i} = a'_{k,j}, \end{cases}$$

where  $a'_{k,i}$  is  $r_i$ 's value of attribute  $A'_k$ , the  $k$ -th element in  $O$ , and  $a'_{k,j}$  is  $r_j$ 's value of attribute  $A'_k$ . If the order of a stream is empty ( $O = \emptyset$ ),  $R$  is still an ordered list, but it may not be sorted in any particular order. In this case no assumptions about the order of records in  $R$  should be made.

The mini-engines of a data flow graph can generally be differentiated into unary and binary mini-engines. Unary mini-engines can be considered functions that take one input stream and produce one output stream. The input and the output streams are not required to have any properties in common, can both have different schemas and transport a different number of records. Formally a unary mini-engine  $ME_u$  can be defined as:

$$s_{out} = ME_u(s_{in}).$$

Similarly, binary mini-engines can be considered functions that take two input streams and produce one output stream. In the general case, there are no interdependencies between the input and output streams; however, as discussed in Section 4.3, specific mini-engines have concrete requirements with respect to input streams. Formally a binary mini-engine  $ME_b$  is defined as:

$$s_{out} = ME_b(s_{in_1}, s_{in_2})$$

A special type of mini-engine is the **RESULT STACK**. It is an n-ary mini-engine that can have an arbitrary number of input streams, but itself does not produce any output streams. It is used exclusively to terminate record streams. The **RESULT STACK** mini-engine is discussed in detail in Section 4.3.8.

In addition to input stream arguments, mini-engines may be parameterized depending on their function and implementation. A **FILTER** mini-engine (see Section 4.3.2), for example, may be parameterized with a predicate function defined on the records of the input stream to determine for each record whether it is to be included in the output stream. Other types of parameters may be constant constraints or modifiers that are known prior to the query evaluation.

The process for the evaluation of a query is described as an evaluation strategy or a query plan and can be represented as a data flow graph. An evaluation strategy for a query is typically determined by a query optimizer, which uses a cost model to identify the most suitable evaluation strategy given a set of cost constraints. A query evaluation strategy that was determined by a query optimizer already contains information on what mini-engines are involved in the evaluation process and what types of streams connect the mini-engines. The only information that is not known

after compiling the evaluation strategy are the actual records that are processed by this query.

The following section describes in detail a set of mini-engines that can be used to design query strategies for common traditional OLAP and spatial OLAP queries. The design of such query strategies is discussed in detail in Section 4.4.

### 4.3 Mini-Engines

Mini-engines are self-contained computational units that communicate with their environment through well defined interfaces. This section introduces a number of different conceptual types of mini-engines, each specialized to perform a specific type of task. The types of mini-engines that are being addressed in this section are those required for the evaluation of most spatial OLAP queries. This section describes in detail the following mini-engines:

- DATA ACCESSOR
- SELECT
- FILTER
- GROUP
- AGGREGATE
- JOIN
- SORT
- RESULT STACK

Note, the mini-engines described in this section are abstract and no concrete assumptions about their implementation beyond the scope of the data model described in Section 4.2 are made. An implementation of the pipelined query evaluation model for spatial OLAP is discussed in Chapter 5.

### 4.3.1 Data Accessor

The DATA ACCESSOR mini-engine is a unary mini-engine used to retrieve records from a data store. As input the mini-engine accepts a stream of query records. For each of these query records, the mini-engine retrieves those records from the data store that satisfy a predefined predicate with respect to the query record. The retrieved records are reported via an output stream  $s_{out}$ , and each query record from the input stream yields a partition of records in the output stream separated from the next partition by a stop word record. Depending on the implementation of the DATA ACCESSOR mini-engine, the retrieval of records from a data store may be supported by a particular access method or data structure (e.g., B-tree or R-tree). If the access method to retrieve the records can guarantee a particular order of these records, then the mini-engine can use this information, as well as information about the order of the query input stream, to derive the order of the output stream.

Typical implementations of the DATA ACCESSOR mini-engine would use a disk storage system and utilize data structures such as B-trees or R-trees to efficiently retrieve records. As an example, a DATA ACCESSOR mini-engine that can retrieve points contained in a polygonal region, may use an R-tree over the spatial properties of the points to efficiently answer containment queries.

### 4.3.2 Filter

A FILTER is a unary mini-engine used to remove records from an input stream that do not satisfy a given predicate, while adding those records that satisfy the predicate to the output stream. Stop word records satisfy any predicate and are always passed through to the output stream.

The FILTER mini-engine does not modify the schema of the records it processes and does not change their relative order, it may only reduce the number of records in the output stream compared to the number of records in the input stream.

### 4.3.3 Select

The SELECT mini-engine is a unary mini-engine used to transform input records with a given input schema into output records with a potentially different schema. To do



so, the **SELECT** mini-engine is parameterized with a function

$$f : S_{in} \times O_{in} \rightarrow S_{out} \times O_{out}$$

which maps a record from the input stream with schema  $S_{in} = \{A_1, A_2, \dots, A_p\}$  to an output record with a schema  $S_{out} = \{B_1, B_2, \dots, B_q\}$ . The output record is then inserted into the output stream. The **SELECT** mini-engine does not remove or create records, and each proper output record is a transformation of an input record. Stop word records are exempt from transformation and are passed unmodified to the output stream. Thus, the total number of output records is equal to the total number of input records. Due to the potential change in schema between the input and the output stream, the sort order of the output stream may not correspond to the sort order of the input stream. The sort order of the output stream  $O_{out}$  contains at least those attributes from the input sort order that are also included in the output schema. If possible, the function  $f$  above may derive a more precise specification of the output sort order by taking into account the nature of the transformations from input attributes to output attributes.

#### 4.3.4 Group

The **GROUP** mini-engine is a unary mini-engine that is parameterized on a set of grouping attributes  $G = \{A_1, A_2, \dots, A_p\}$ . It is used to partition the records received from an input stream so that all records within one partition share the same values with respect to the grouping attributes  $G$ . The group mini-engine receives records from an input stream and compares each pair of consecutive records with respect to the grouping attributes. If both records share the same values, they are considered to be within the same partition and are written consecutively to the output stream. If the grouping attribute values of the two records differ, the records are written to the output stream separated by a stop word record to indicate that the records belong to different partitions. Thus, to obtain a partitioned output stream that guarantees that each partition corresponds to a distinct equivalence class with respect to the grouping attributes, the input stream must be sorted with respect to the grouping attributes so that the attributes in  $G$  form a prefix of the input stream's sort order  $O$ .

When the **GROUP** mini-engine receives a stop word record from the input stream its default action is to immediately discard this record and not to process it further. This allows the mini-engine to remove any prior partitioning from the stream. However, this functionality can optionally be disabled and any input stop word record is passed through as is. This permits the **GROUP** mini-engine to add a refined partitioning on top of an already partitioned stream.

The output of the **GROUP** mini-engine is a stream with the same properties as the input stream and containing the same number of proper records. Depending on its configuration, the **GROUP** mini-engine may discard all stop word records it receives from the input stream and potentially injects a new set of stop word records to separate the partitions of records it has identified.

### 4.3.5 Aggregate

The **AGGREGATE** mini-engine is a unary mini-engine to compute one or more aggregate values for each partition of records in the input stream. The parameter to the mini-engine is a set of aggregate functions  $\Phi = \{\phi_1, \dots, \phi_k\}$  with  $\phi_i : \mathcal{A}_i \times \mathcal{A}_i \rightarrow \mathcal{A}_i$ , where  $\mathcal{A}_i$  is the type of the  $i$ th attribute of the input stream schema  $S_{in}$ . The mini-engine produces an output stream  $s_{out}$  with the same schema  $S_{in}$  as the input stream, with each partition of the input stream represented by only one proper record in the output stream. Each output record's attribute values are the aggregation values computed for the corresponding partition of input records. For each attribute  $A_i \in S_{in}$ , its corresponding value  $a_{i,k}$  in record  $r_k$  and each partition of  $n$  input records  $(r_1, \dots, r_n)$ , the aggregation value  $\phi_i^*(r_1, \dots, r_n)$  is defined as

$$\phi_i^*(r_1, \dots, r_n) = \begin{cases} a_{i,1}, & \text{if } k = 1; \\ \phi_i(\phi_i^*(r_1, \dots, r_{k-1}), a_{i,k}), & \text{if } k > 1. \end{cases}$$

The stop word records that separate partitions in the input stream are retrieved by the mini-engine and passed on to the output stream unmodified. This ensures that the partitioning information of the input stream is retained in the output stream, as this information may be required for subsequent operations such as the **JOIN** operation. This is discussed in Section 4.3.6.

The order of the output stream depends on the order of the input stream and the aggregation functions defined for each attribute. The relative order of the output records is the same as the relative order of the partitions of the input stream. The output records may not be sorted in any particular order with respect to their attribute values depending on the aggregation functions that were used. In some cases the aggregation functions applied to the attributes in  $O_{in}$  may yield aggregation values that are in the same sorted order as the corresponding attribute values of the input records. For these cases the AGGREGATE mini-engine can derive the sort order of the output stream from the sort order of the input stream under consideration of the aggregation functions applied to the sorting attributes in  $O_{in}$ . For example, if the aggregation functions used on the attributes in  $O_{in}$  of the input stream each emit the first occurring attribute value, the sort order of the output stream is equivalent to that of the input stream and can be characterized by the same attributes. If, on the other hand, the aggregation function is, for example, a sum, the aggregation values for each of the ordering attributes may no longer be sorted as their final values depend on the values of the attributes of each input record as well as the number of records in each partition.

An AGGREGATE mini-engine typically occurs in some combination with a GROUP mini-engine; the GROUP mini-engine identifies and marks the aggregation groups, and the AGGREGATE mini-engine performs the actual aggregation.

#### 4.3.6 Join

The JOIN mini-engine is a binary mini-engine used to combine one stream of records with another stream of records. In particular, it creates the Cartesian products of record partitions received from the first input stream with corresponding record partitions received from the second input stream. Thus, it processes each input stream one partition after another and expects both input streams to have the same number of partitions. The stop word records that separate individual partitions in the input streams are merged into a single stop word record that is emitted as an output after the Cartesian product of the corresponding partitions has been output. The Cartesian product between two partitions  $P_1$  and  $P_2$ , with  $P_1$  containing  $m$  records and  $P_2$  containing  $n$  records, is an output partition  $P_{out}$  with  $m \times n$  records.

Each record in  $P_{out}$  is constructed by concatenating the attribute values from the corresponding records in  $P_1$  and  $P_2$ . The schema of the output stream to which the result records are written is  $S_{out} = S_{in_1} \circ S_{in_2}$ , where  $\circ$  is the concatenation operator. After writing all records of an output partition  $P_{out}$  to the output stream, the JOIN mini-engine merges the stop word records that conclude the input partitions into a single stop word record that is output to separate the partitions in the output stream. Thus, the mini-engine emits stop word records after each Cartesian product of the input partitions it computes.

The sort order of the output stream of the JOIN mini-engine is at least that of the first input stream, as its records are processed by the outer loop of the join process. If all of the first input stream's attributes were also ordering attributes of the first input stream, then the order of output stream additionally inherits the order of the second input stream as a secondary order in addition to the primary order provided by the first input stream.

### 4.3.7 Sort

The SORT mini-engine is used to reorganize the records within a stream so that they satisfy a given sort order constraint. This sort order is provided to the mini-engine as a parameter in the form of a list of ordering attributes  $O = (A_1, \dots, A_k)$ , where  $\{A_1, \dots, A_k\} \subseteq S_{in}$ . The sorting of the records removes any partition of the input stream that may be present by discarding all stop word records that may be present in the input stream. To sort the proper records received from the input stream, the mini-engine may utilize temporary external storage and may need to consume the entire input stream before producing an output. The SORT mini-engine's output stream has the same schema as the input stream and the number of output records matches the number of proper input records. In comparison to the input stream, the sort order of the output stream is characterized by the sort order attributes given to the mini-engine as a parameter and may not be the same as the input stream's sort order.

### 4.3.8 Result Stack

The RESULT STACK mini-engine is a special kind of mini-engine in that it may have multiple input streams but does not have an output stream. The purpose of the RESULT STACK mini-engine is to act as a container for query results that are passed to it in form of record streams. As such, the RESULT STACK mini-engine terminates record streams and prepares the collected records for presentation to the application. Such representation may be a table, a view or some kind of data visualization. The presentation of query results to the application is not in the scope of this thesis, and for the remainder of this chapter we consider the result stack as a termination of one or more record streams.

## 4.4 Applications and Example Queries

In this section we demonstrate the applicability, flexibility and strength of the pipeline model for spatial OLAP. For this purpose we describe an application scenario and various example queries. We begin with example queries for traditional OLAP to demonstrate the applicability of the pipeline approach not only to spatial OLAP but also to traditional OLAP. Then, the remainder of this chapter focuses on the evaluation of spatial OLAP queries. For the illustration of each of the example queries, we first provide a representation of the query in SQL and then describe a translation of the query into an interacting set of mini-engines.

As an application scenario, consider a modern national forestry department conducting a comprehensive survey of the country's vegetation coverage and storing all of the collected data in a large data warehouse. The data the department collects during the survey may be

- For a large number of individually sampled plants, recordings of each plant's longitude and latitude coordinates, family, genus, and species, estimated age and height.
- Areas of vegetation types derived from high-resolution satellite imagery.
- Additional auxiliary information from 3rd party providers, such as political regions, emergency services district regions, postal code regions, as well as historic

meteorological data.

To analyze the data in the data warehouse, an analyst may execute the following traditional and spatial OLAP queries:

**Query 1:** What is the maximum height of all plants between 10 and 50 years of estimated age, for each species?

Without processing any spatial data, this query represents a typical `GROUP BY` aggregation query often used in traditional OLAP.

**Query 2:** What is the maximum height of all plants between 10 and 50 years of estimated age, for each species, genus, and family?

This query is an extension of the first query and introduces a traditional OLAP roll-up hierarchy to obtain aggregate results at different levels of hierarchical groupings.

**Query 3:** What is the average vegetation height in the region between 42.975N 69.351W and 48.228N 59.480W, grouped by county?

This query can be considered the spatial equivalent of the first query. It queries for a set of plant records whose spatial *location* attribute is contained within the given spatial query region, just as Query 1 queries for a set of plant records whose non-spatial *age* attribute is within a numeric range. The plant records are then grouped by the county that contains their location, and their *height* measure values are aggregated for each such group.

**Query 4:** What is the maximum vegetation height in the region between 42.975N 69.351W and 48.228N 59.480W, for each county, province, and country?

This query is the spatial equivalent of Query 2 and extends Query 3 with a spatial roll-up hierarchy to obtain results at different levels of spatial resolution.

**Query 5:** Report the total area covered by forest for each postal code, fire district, and county?

This query is an example of a spatial OLAP query that uses a non-strict roll-up hierarchy, i.e. a hierarchy for which the members of one level are not well contained within the members of the levels above.

Queries 1 and 2 are typical categorical OLAP queries and can be evaluated using existing traditional OLAP systems. Queries 3, 4, and 5, on the other hand, involve spatial dimension attributes and require a spatial OLAP system to be evaluated efficiently.

#### 4.4.1 Query 1

What is the maximum height of plants between 10 and 50 years of estimated age, for each species?

This query is a typical group-by query, a fundamental concept of traditional OLAP and often used when querying a database for aggregate information. For this query, we assume no spatial components are involved in the evaluation of the query, and the `species` entity is represented as a categorical attribute, e.g., a name. Hence, the query deals with data types that are readily available in standard SQL compatible database management systems.

In SQL, the query can be formulated as

```
SELECT species.name, MAX(plant.height)
FROM species
LEFT JOIN plant ON plant.species_id = species.id
WHERE plant.age >= 10 AND plant.age <= 50
GROUP BY species.name
```

The result of this query is a view (or table) that lists, for each species that has recorded plants within the given age range, the maximum height among these plants. The query considers only plants with an estimated age between 10 and 50 years; thus, the `plant` records are filtered to include only those that satisfy this condition. The association between a `species` entity and a `plant` entity is established through a `LEFT JOIN` statement, which determines the Cartesian product between the `species` table and the `plant` table and filters the result to include only records that satisfy the join predicate `species.id = plant.species_id`. These records are then grouped by the `name` attribute of each `species` entity. Finally, the aggregated fact of each group is computed by the `MAX` function, which, in this example, determines the maximum value of the `height` attribute for each `plant` record in the group. Figure 4.2 shows an example result for this query. Note, the `LEFT JOIN` statement combines the `species`

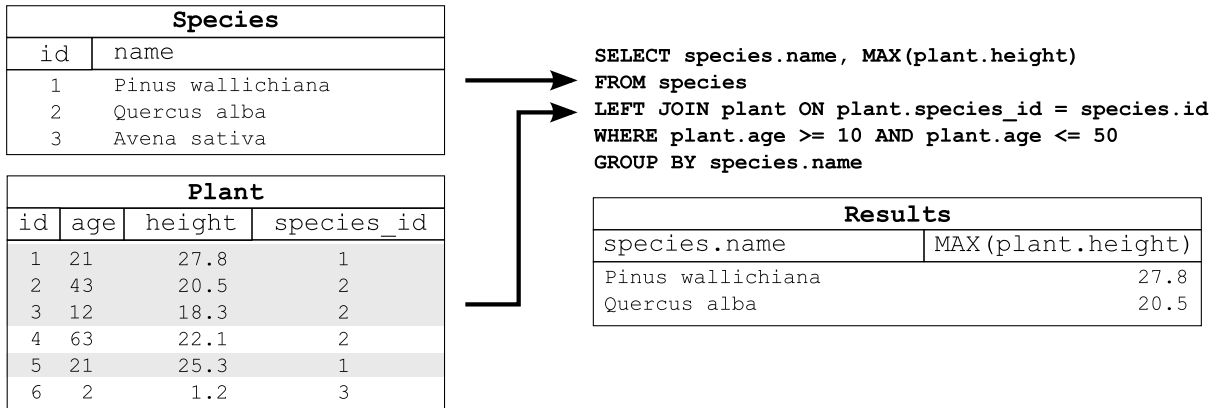


Figure 4.2: Example source and result tables for the query described in Section 4.4.1. They gray shading represents the records that satisfy the constraint specified by the `WHERE` clause.

table and the `plant` table based on the unique identifier of each species to which one or more plants belong.

In the context of the pipeline model, the evaluation strategy for this query is represented as a data flow in Figure 4.3. The structure of the graph can be divided into the following steps:

**Step 1:** Determine all species of plants. This task is performed using a `DATA ACCESSOR` mini-engine that retrieves all records from the `species` table. We assume that each species is unique and that the `DATA ACCESSOR` mini-engine returns only one record per species. This property can be enforced during the creation of the `species` table using a unique key constraint. The output of the `DATA ACCESSOR` mini-engine is a stream of `species` records with the schema  $S_{out} = \{A_{name}, A_{id}\}$ .

**Step 2:** Generate query records. Given the `species` records, the following steps will identify the `plant` records that are associated with each species. This association is done based on the primary key of the `species` records, which is `species.id`. All remaining attributes of the `species` records are not important for this part of the query and are removed from the records. The resulting stream of `species.id` records is used as a query stream in Step 3.



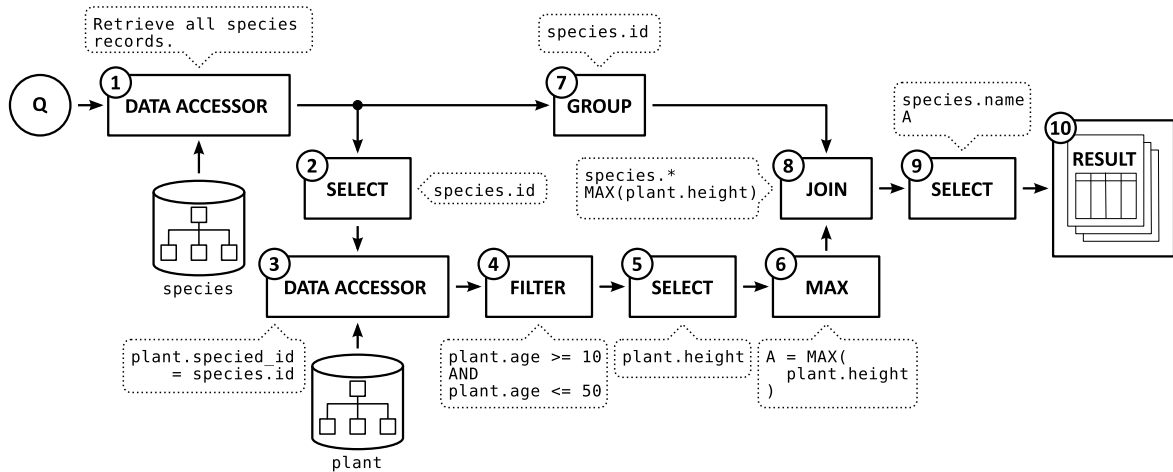


Figure 4.3: Data flow graph of the strategy used to evaluate the query described in Section 4.4.1.

**Step 3:** Retrieve all **plant** records for each species. Using the unique ID associated with each species, the **DATA ACCESSOR** mini-engine can query the **plant** data store for records that are associated with each species. To perform this query efficiently, the **DATA ACCESSOR** mini-engine may use an index (e.g., B-tree) on the **plant.species\_id** attribute of the **plant** records. For each **species.id** received as an input to the mini-engine, the **DATA ACCESSOR** returns a partition of **plant** records that are associated with that corresponding species. If no such **plant** record exists, the partition returned by the mini-engine is empty.

**Step 4:** Filter **plant** records by age. Since the subject of the analysis are plants with an estimated age between 10 and 50 years, this constraint is being enforced by filtering the retrieved **plant** records and discarding all records for which the constraint is not satisfied.

**Step 5:** Select attributes relevant for aggregation. Since the aggregation only requires information about each plant's height, the **height** attribute is the only attribute that is retained in the **SELECT** mini-engine's output stream.

**Step 6:** Compute the maximum vegetation height for each partition. After Step 3, each partition of the **plant** record stream corresponds to a particular species in the **species** record stream produced in Step 1. Hence, the maximum vegetation

height of plants with an estimated age of between 10 and 50 years for each species can be determined by aggregating each partition of `plant` records. The aggregation only needs to be performed on the `plant.height` attribute, which will be appropriately annotated with the corresponding species' name in Step 8. The output of the AGGREGATE mini-engine is a partitioned stream of records where each partition has either no records, (i.e., the partition was empty before the aggregation) or one record representing the aggregate value.

**Step 7:** Generate a partition for each unique species. Since the query compiler is aware that species are unique and that the stream from Step 1 contains only one record per species, it is not required to sort the record stream before partitioning it. The GROUP mini-engine simply inserts stop word records between the individual species records to mark the partitions.

**Step 8:** Annotate aggregated records with species name. After the aggregation, each partition of the aggregated record stream is joined with its corresponding counterpart partition of the `species` record stream. The partitions of the `species` record stream each contain only one record per species and partitions of the aggregation record stream may contain either no or one record specifying the maximum vegetation height determined for that partition. Thus, the JOIN mini-engine produces one output record for each pair of `species` record partition and aggregation record partition if an aggregate record exists. The output stream of the JOIN mini-engine is a stream of records with schema  $S_{out} = \{A_{name}, A_{id}, A_{height}\}$ , where the  $A_{name}$  attribute is the name of the species,  $A_{id}$  its unique identifier, and the  $A_{height}$  attribute the maximum height of plants of the given species between 10 and 50 years of estimated age.

**Step 9:** Remove irrelevant attributes. The result of the query only needs to contain the name of each species and the maximum height that was determined. The SELECT mini-engine removes all other attributes before the results are passed to the RESULT STACK in Step 10.

The evaluation strategy described above is not the only strategy that can be used to evaluate the given query. There exist alternative strategies which may, depending

on the properties of the data, evaluate the same query less or more efficiently. Given additional information about the nature of the underlying data, the strategy above may need to be modified to evaluate the query differently and more efficiently.

For a system to choose the most suitable of several candidate evaluation strategies, a query optimizer should be employed. A query optimizer analyzes, for each query, the strategies that can potentially evaluate the query and assigns an estimated cost to each mini-engine in a strategy's data flow. This cost depends on the type of mini-engine and is based on the amount of data it has to process, what data structures are used and what operations are performed on the data. The total estimated costs of all strategies are then compared, and the strategy with the lowest cost is chosen to evaluate the query. The design and implementation of an appropriate cost model and a corresponding query optimizer are not in the scope of this thesis, and the reader is referred to [97, 163] for in-depths discussions of query optimization.

#### 4.4.2 Query 2

What is the maximum height of a plant between 10 and 50 years of estimated age, for each species, genus, and family?

This example query is a typical OLAP roll-up query. It computes results at multiple levels of aggregation. In this example, the different levels of aggregation from the most detailed to the most aggregated level are the species, genus, and family of a plant's ontological classification. In SQL, this query may be formulated as

```
SELECT family.name, genus.name, species.name, MAX(plant.height)
FROM family
LEFT JOIN genus ON genus.family_id = family.id
LEFT JOIN species ON species.genus_id = genus.id
LEFT JOIN plant ON plant.species_id = species.id
WHERE plant.age >= 10 AND plant.age <= 50
GROUP BY ROLLUP(family.name, genus.name, species.name)
```

and Figure 4.4 shows an example of the query result. This query is structurally similar to Query 1; however, note the keyword `ROLLUP`. The `ROLLUP` keyword indicates that, for the attributes listed as parameters, multiple group-bys are computed such that each group-by has the last element of the previous group-by removed. In this example the following group-by sets are generated: `(family.name, genus.name, plant.name)`,

<b>Results</b>			
<code>family.name</code>	<code>genus.name</code>	<code>species.name</code>	<code>MAX(plant.height)</code>
Cupressaceae	Cupressus	Cupressus gigantea	17.6
Pinaceae	Pinus	Pinus strobus	27.4
Pinaceae	Pinus	Pinus monticola	31.7
Fagaceae	Quercus	Quercus alba	22.5
Fagaceae	Castanea	Castanea sativa	25.1
Cupressaceae	Cupressus	*	17.6
Pinaceae	Pinus	*	31.7
Fagaceae	Quercus	*	22.5
Fagaceae	Castanea	*	25.1
Cupressaceae	*	*	17.6
Pinaceae	*	*	31.7
Fagaceae	*	*	25.1
*	*	*	31.7

Figure 4.4: Example result tables for the query described in Section 4.4.2. Attribute values of  $\star$  denote NULL or undefined values, which are commonly used to represent an aggregated attribute dimension.

(`family.name`, `genus.name`), (`family.name`) and  $\emptyset$ . The query evaluation needs to compute the aggregation for each of these group-bys. As they are each a prefix of the previous group-by, it is possible to use the aggregation results of a preceding group-by as input for the next aggregation. This allows OLAP systems to use intelligent algorithms for the efficient evaluation of ROLLUP queries.

Figure 4.5 shows a data flow graph based on the pipeline evaluation model that can be used to evaluate the ROLLUP query described above. One can see that the structure of parts of the data flow graph is very similar to that of Query 1, but the multiple levels of joins and aggregation can be easily identified. The underlying concept of this strategy is that records from a more detailed level of aggregation are associated with records from a less detailed level of aggregation, and this relationship is transitive across all levels of aggregation. Hence, the evaluation of the query first identifies records at the least detailed level of aggregation (Step 3) and then uses those to query associated records at the next more detailed level of aggregation (Step 7). This process continues with each level of aggregation until the level of measure records (`plant`) has been reached. Following Step 11, the result is a stream of `plant` records, partitioned in such a way that each partition contains `plant` records that are associated with a particular species. Since the query is not required to report

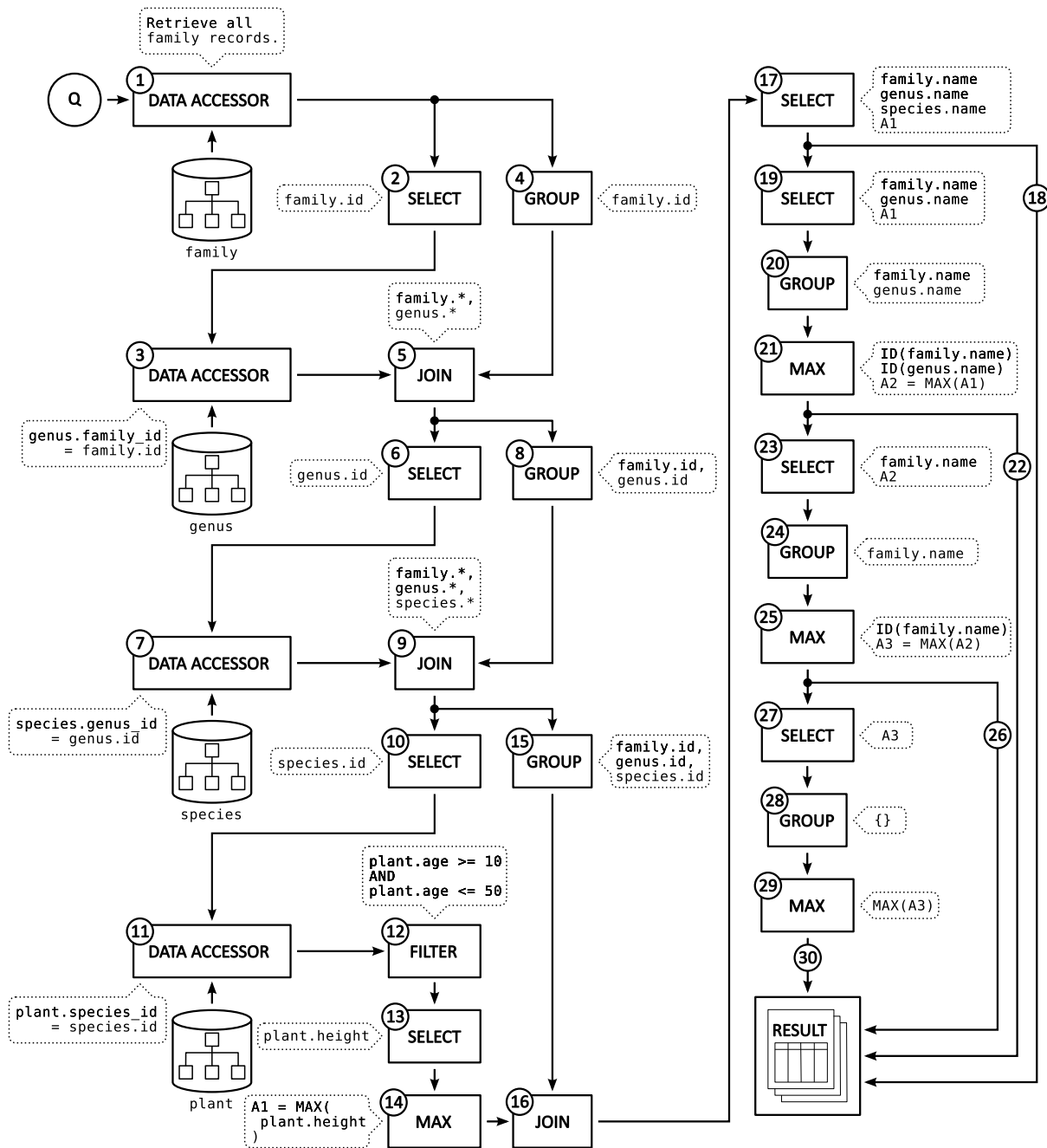


Figure 4.5: Data flow graph of the query evaluation strategy described in Section 4.4.2.

any results at the resolution of the plant records and the aggregation function `MAX` is associative, the strategy immediately aggregates the partitions of the plant records to obtain aggregation values at the `species` level (Step 14). Step 16 then joins the aggregate values with the corresponding records representing the lowest level of the `ROLLUP` hierarchy to produce the results for the (`family.name`, `genus.name`, `species.name`) aggregation level. The results for the remaining aggregation levels are then derived from the (`family.name`, `genus.name`, `species.name`) results by removing unwanted attributes and aggregating the records based on the desired group attributes (`family.name`, `genus.name`) in Step 21, (`family.name`) in Step 25, and () in Step 29. The results of each aggregation are forwarded to the `RESULT STACK` mini-engine, where they are prepared for consumption by the application.

Note that due to the order in which records are retrieved from the data stores, the strategy can evaluate the `ROLLUP` query without sorting any data. This may allow for a very efficient evaluation of the query but depends strongly on properties of the data that is underlying the query and the design and layout of the data warehouse. There may exist other strategies that may or may not evaluate the same query more efficiently depending on these factors.

#### 4.4.3 Query 3

What is the average vegetation height in the region between 42.975N 69.351W and 48.228N 59.480W, grouped by county?

Query 3 determines the average height of vegetation within the given rectangular query region for each county in this region. The query region is specified in the form of two points: 42.975 degrees north / 69.351 degrees west and 48.228 degrees north / 59.480 degrees west. This query is our first example of a spatial query, as it involves the processing of spatial properties such as the query region and the location and region attributes of the queried records. The structure of the query is very similar to the structure of Query 1 and, in fact, Query 3 can be considered the spatial equivalent of Query 1. In contrast to Query 1, some components involved in the evaluation of this query exclusively process spatial attributes. Figure 4.6 illustrates this query. Using a spatial extension to SQL [132], the query can be expressed in SQL as

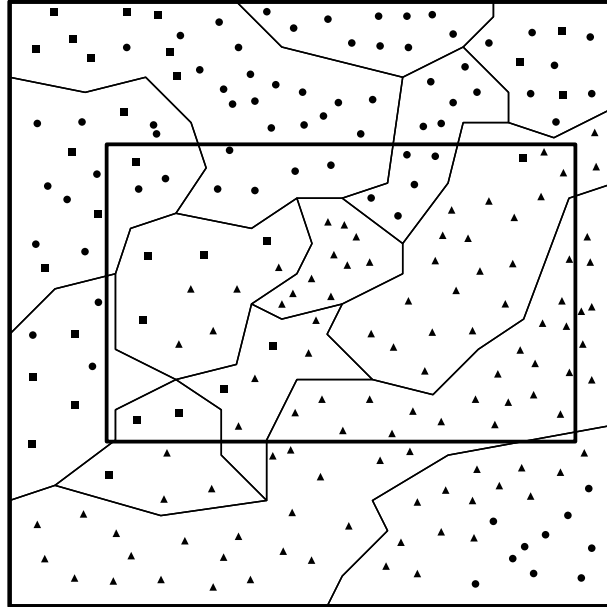


Figure 4.6: Illustration of Query 3. Counties are represented as polygons, and individually sampled plants of different types of vegetation are represented as ▲, ■, and ● respectively. The rectangular query region intersects multiple counties and contains a number of individual plant records.

```

SELECT county.name, AVERAGE(plant.height)
FROM county
LEFT JOIN plant ON CONTAINS(county.region, plant.location)
WHERE CONTAINS(
    MakeBox2D(
        MakePoint(42.975, -69.351),
        MakePoint(48.228, -59.480)
    ),
    plant.location
)
GROUP BY county.name

```

To evaluate the query, the system needs to identify which locations are contained within the given query region. Furthermore, it needs to associate each location with the county that contains it and group the annotated locations by the counties they belong to. Finally it applies the aggregation function `AVERAGE` to all records within a group to compute the average height of the `plant` records within that group.

Figure 4.7 shows the data flow graph of a strategy to evaluate the query. One can see that the structure of the strategy is similar to the structure of the strategy discussed for Query 1, and both strategies have the same number of steps, often with

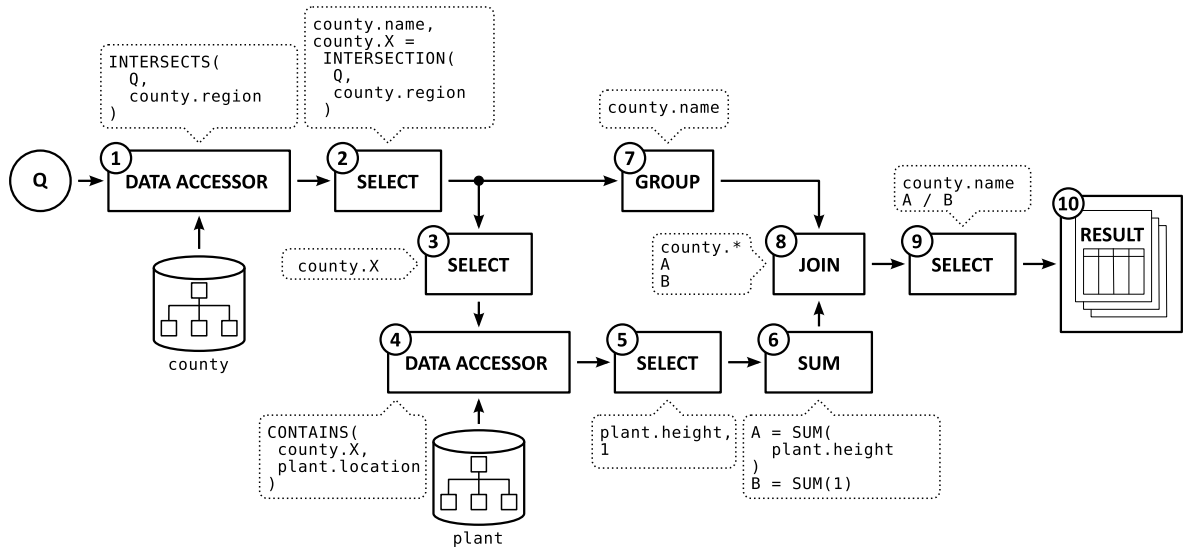


Figure 4.7: Data flow graph of an evaluation strategy for Query 3.

similar functions:

**Step 1:** Determine affected counties. Since, for this query, we are only interested in locations that are contained within the query region, and each location is associated with only one county, we are also only interested in counties that intersect with the query region. To retrieve those counties, Step 1 uses a **DATA ACCESSOR** mini-engine that retrieves records from a data store depending on whether their spatial properties intersect with a given spatial object, in this case the query region. To perform this task efficiently, the **DATA ACCESSOR** mini-engine may for example utilize a spatial index, such as an R-tree, over the spatial attributes of the **county** records.

**Step 2:** Constrain county extents to query region. To identify which **plant** records are associated with a county, the strategy uses the region of each county that intersects with the query region as a query region for a subsequent query across the **plant** records. Thus, the retrieved **county** records' extents are trimmed to the query region by computing the spatial intersection between each county's total region and the query region, and all other attributes that are irrelevant to the query are removed. The **county** records are transformed in that way to obtain query regions that constrain subsequent query results to the corresponding



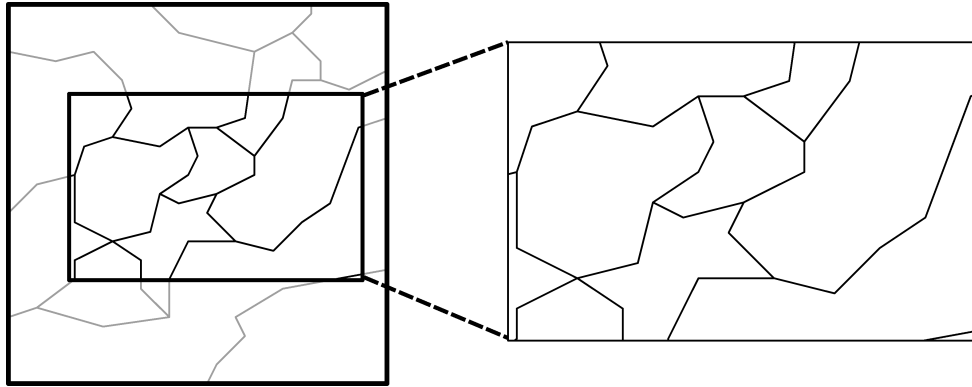


Figure 4.8: Illustration of the trimming of spatial attributes to the extent of the query region. Here the **county** records from Query 3 are constrained to their spatial intersection with the query region.

county as well as the original query region at the same time. The trimming is illustrated in Figure 4.8.

**Step 3:** Transform county records into query records. For the trimmed **county** record extents to be used as queries over the **plant** records, they need to be extracted as isolated query records. Thus, the output of Step 2 is subsequently transformed into a stream of query records using a **SELECT** mini-engine. Each query record corresponds to a county record that was retrieved in Step 1 and transformed into a record with a single spatial attribute representing the spatial intersection of the county's region with the query region.

**Step 4:** Retrieve plants within the query region for each county. By utilizing the query records created in Step 3, a **DATA ACCESSOR** mini-engine for polygonal range queries can now be used to retrieve the plant records that are enclosed by each county's individual query region. In practice, this may be implemented using an R-tree index, which can facilitate the efficient retrieval of these records. For each county that intersects the query region, the **DATA ACCESSOR** mini-engine then produces a partition of its output stream containing plant records with location attributes that are both enclosed by the county's region and the query region.

**Step 5:** Select aggregation attributes. The result of the query is expected to be the

average height of plants. Since the average function is a non-distributive function, the aggregation function is decomposed into a set of distributive operations (accumulation) and a final non-distributive operation (division). However, this approach requires not only the accumulation of each plant's height measure but also the determination of the number of records in each partition. For this purpose, the `SELECT` mini-engine injects an additional temporary attribute with value 1 to be accumulated as the number of records. This allows the strategy to later compute the final average height of plants for each partition of records.

**Step 6:** Compute the sum of the plants' heights and the number of records for each partition. Only the `height` attribute of the plant records and the temporary attribute are subject to the aggregation; all other attributes of the plant records are discarded.

**Step 7:** Generate partitions of the `county` records.

**Step 8:** Annotate aggregate records with `county` record attributes. Each of the partitions aggregated in Step 6 correspond to a `county` record that was retrieved in Step 1. In this step the aggregate record of each partition is annotated with the attributes of the corresponding `county` record. This establishes the association between a `county` and its corresponding aggregate values for the given query region.

**Step 9:** Compute final aggregation results. The records generated by the `JOIN` mini-engine in Step 8 combine all the information required to derive the final result records. The `SELECT` mini-engine constructs these final records by selecting the `county.name` attribute and computing the associated average vegetation height from the sum of heights (attribute A) and the number of plants in the corresponding region (attribute B).

As was the case for Query 1, the evaluation strategy discussed above is not the only strategy that can be chosen. Depending on the nature of the dataset, a different strategy may be required to evaluate the same query and to obtain results more efficiently. In particular, the evaluation of spatial OLAP queries allows for a broad variety of different evaluation strategies, as there exist a large number of data

access methods for spatial data and a variety of algorithms to process spatial data efficiently [43,156]. Similar to traditional OLAP queries, the best strategy to evaluate a query may be chosen by a query optimizer, which uses a cost model to rate each strategy and determine the most suitable one.

This example query demonstrates that the pipeline model is suitable to model and express evaluation strategies for simple spatial OLAP queries. In the next section we demonstrate that it is also suitable for more complex spatial OLAP queries.

#### 4.4.4 Query 4

What is the maximum vegetation height in the region between 42.975N 69.351W and 48.228N 59.480W, for each county, province, and country?

Query 4 is a spatial roll-up query to determine the maximum vegetation height within a predefined query region, for each county, province and country. The query

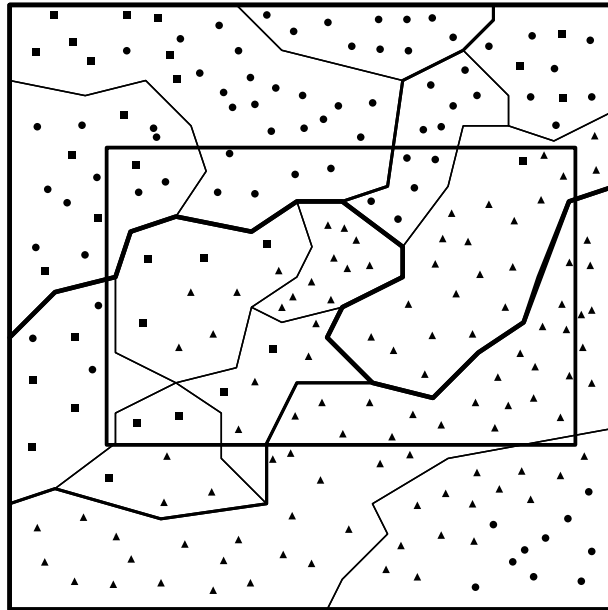


Figure 4.9: Illustration of Query 4. At each level of aggregation, compute the maximum vegetation height for all member of that level that intersect with the query region. The measures are records of individually sampled plants of various species, here represented as ▲, ■, and ●.

is illustrated in Figure 4.9, and the corresponding query result is show in Figure 4.10. Figure 4.10 is a pictorial representation of the query results, where the colour intensity of each polygon is proportional to the average height value for the corresponding

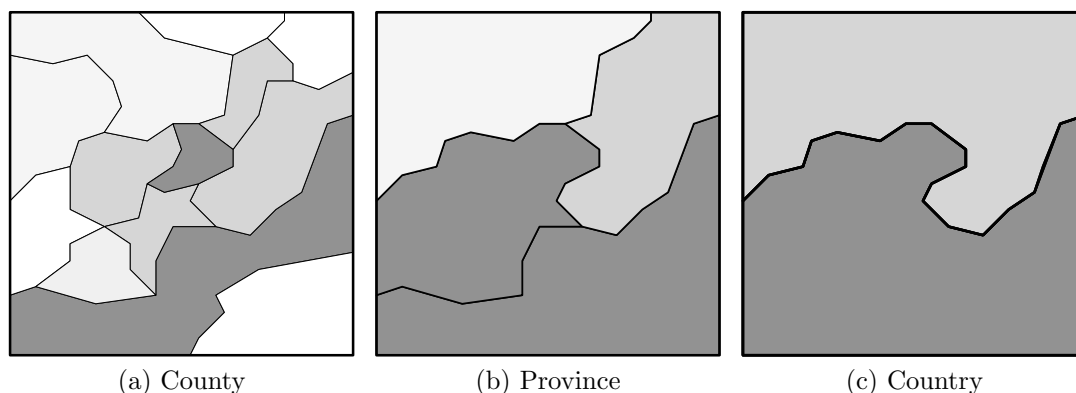


Figure 4.10: Graphical representation of the results of Query 4 for each aggregation level.

region within the query region. There may be a number of other representation methods (e.g., tabular or overlays) that are suitable for presenting the same query results; however, the choice of the method depends on the application.

Using SQL with a spatial extension, the query may be expressed as follows:

```

SELECT country.name, province.name, county.name, MAX(plant.height)
FROM country
LEFT JOIN province ON CONTAINS(country.region, province.region)
LEFT JOIN county ON CONTAINS(province.region, county.region)
LEFT JOIN plant ON CONTAINS(county.region, plant.location)
WHERE CONTAINS(
    MakeBox2D(
        MakePoint(42.975, -69.351),
        MakePoint(48.228, -59.480)
    ),
    plant.location
)
GROUP BY ROLLUP(country.name, province.name, county.name)

```

Note the structural similarity of this query with Query 2, which also is a roll-up query with three aggregation levels. The evaluation of Query 4 involves the processing of spatial attributes and the relationships between the members of different aggregation levels are determined based on the spatial properties of these members.

One strategy for the evaluation of this query is depicted as a data flow graph in Figure 4.11. At a conceptual level, the steps in this data flow graph correspond to those used in the strategy for Query 2, yet they are fundamentally different with

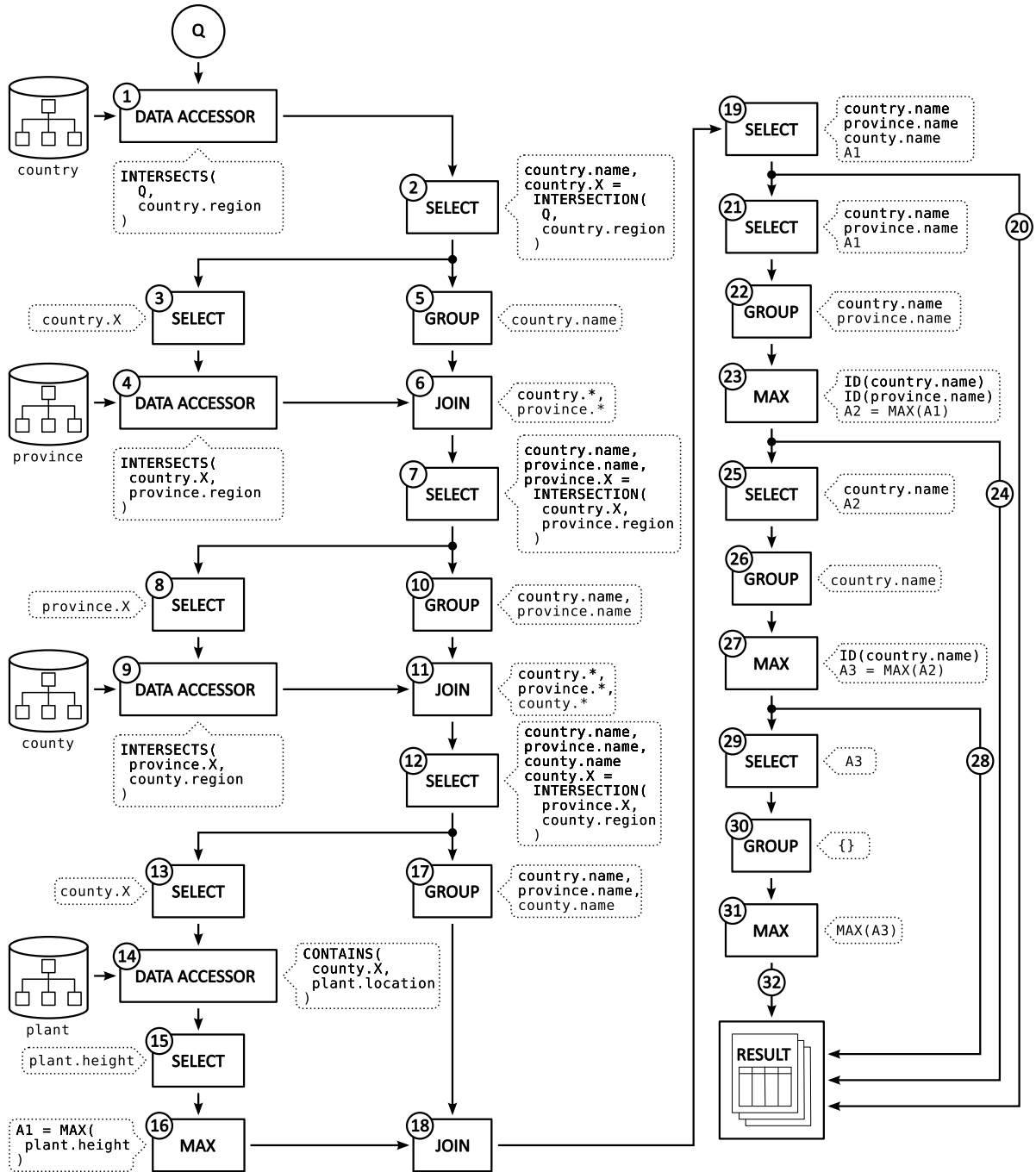


Figure 4.11: Data flow graph of an evaluation strategy for Query 4.

respect to the data types that are processed and the functions that are executed due to the spatial nature of this query.

**Steps 1, 4, 9, and 14:** The **DATA ACCESSOR** mini-engines that are used to fetch records from the corresponding datasets are retrieving these records based on their spatial properties. To perform this task efficiently, these **DATA ACCESSOR** mini-engines may utilize spatial data access methods or indexing structures that are defined and available for each of the given datasets. One such indexing structure is an R-tree, which is a suitable indexing structure in all four cases of this example query. Other strategies for this query or other queries may employ different access patterns on the data and may thus require different indexing structures. Depending on the number of supported data access method there may be a number of different **DATA ACCESSOR** mini-engines, each supporting a different access method.

**Steps 2, 7, and 12:** In these steps, the spatial attributes of records that represent members of an aggregation level are spatially constrained. The spatial constraint is defined by the matching member of the previous aggregation level as well as the query region, as the query is only concerned about aggregation values within that region. The transformation of the records maintains only those attributes of the records that are further relevant to the query; all other attributes are discarded. After each transformation step, the constrained spatial attributes are converted into query records in Steps 3, 8, and 13, respectively, to determine the members of the next level of aggregation or the measure values that are to be aggregated.

**Steps 6, 11, and 18:** These “join” steps combine records from one level of aggregation with their respectively associated records of the aggregation level below. Both input streams to the **JOIN** mini-engine are appropriately partitioned. The “parent” record stream is partitioned using one of the **GROUP** mini-engines in Steps 5, 10, or 17 and the “child” record stream is partitioned by the corresponding **DATA ACCESSOR** mini-engine that was used to retrieve the “child” records. The output of each of the **JOIN** mini-engines is a stream of records containing one record for each “child” record that was identified, combined with

the attributes of its corresponding “parent” record.

**Steps 15, 16, 19 – 32:** As input to the JOIN operation in Step 18 the records from the lowest aggregation level are aggregated for each partition generated in Step 14. This can be done because the query is only required to determine aggregate values at the lowest level of aggregation and thus can aggregate the detailed records immediately after retrieving them in Step 14.

After Step 18, all aggregate values from the lowest level of aggregation have been associated with attributes of records from all aggregation levels above. This record stream represents the most detailed level of aggregation and, after removing unwanted attributes in Step 19, is passed to the RESULT STACK mini-engine in Step 20 as part of the query result. The same results are also regrouped in Step 22 to obtain partitions of records such that the records in each partition share the same `country` and `province` attributes. These partitions are then regarded as aggregation groups in Step 23, which produces the query results for the second level of aggregation that is requested by the query (Step 24). Using the results of the second level of aggregation, the regrouping in Step 26 and subsequent aggregation in Step 27, as well as Steps 30 and 31, are analogously performed to obtain the results for the two coarsest levels of aggregation requested.

Observe that the structure of the evaluation strategy for Query 4 is very similar to the structure of the strategy for Query 2. Individual mini-engines in the strategy have been replaced to provide capabilities necessary to process spatial data. This fact emphasizes the flexibility of the pipeline model in that the type of query and thus the high-level evaluation strategy are independent of the data types that are being processed. Thus, to apply an existing strategy to a different set of data types, it often suffices to replace the mini-engines responsible for processing the data with ones that implement the same operations for the new data types.

Simply replacing the mini-engines of an existing evaluation strategy to support different data types may not always result in an efficient strategy. The specific characteristics of certain data types may require different access methods to retrieve records of these types efficiently. Thus, a query optimizer should be used to assess multiple

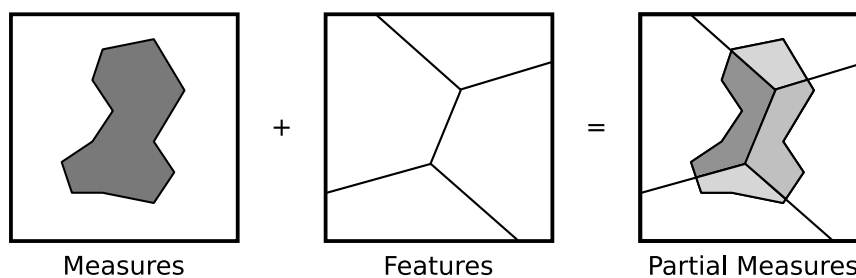


Figure 4.12: Example of a measure that is partially associated with multiple features.

candidate strategies and, using a cost model, determine which of the candidate strategies may be the most cost-effective. Here, cost may be measured in a number of ways: for example, the time it takes to answer the query, the number of disk operations required to evaluate the query, the electrical power used, etc. or any combination of those.

#### 4.4.5 Query 5

Report the total area covered by forest, for each postal code, fire district, and county?

Query 5, described in this section, is one of the most general spatial roll-up aggregation query, as it does not make any assumptions about the relationships between the aggregation levels, except that members of two aggregation levels relate to each other if their spatial properties intersect. This type of query can be used with non-strict roll-up hierarchies, for which the members of a child aggregation level do not map to unique members of the parent aggregation level. In the example query, this may be the case for the aggregation of “fire districts” and “postal codes”, where fire district regions may not necessarily line up with postal code regions. Since there may be a legitimate reason for relating these otherwise disjoint layers in a roll-up query, e.g., for taxation or reporting purposes, a spatial OLAP system must be capable to evaluate such queries efficiently.

When evaluating non-strict hierarchy queries, measure values may not be associated with unique “parent” records and instead partially overlap more than one “parent” record. These types of measures are called *partial measures*, as they partially contribute a value to each “parent” they are associated with (see Figure 4.12). Their values are typically subject to scaling, which is derived from the amount of their partial contribution to a given “parent” record.



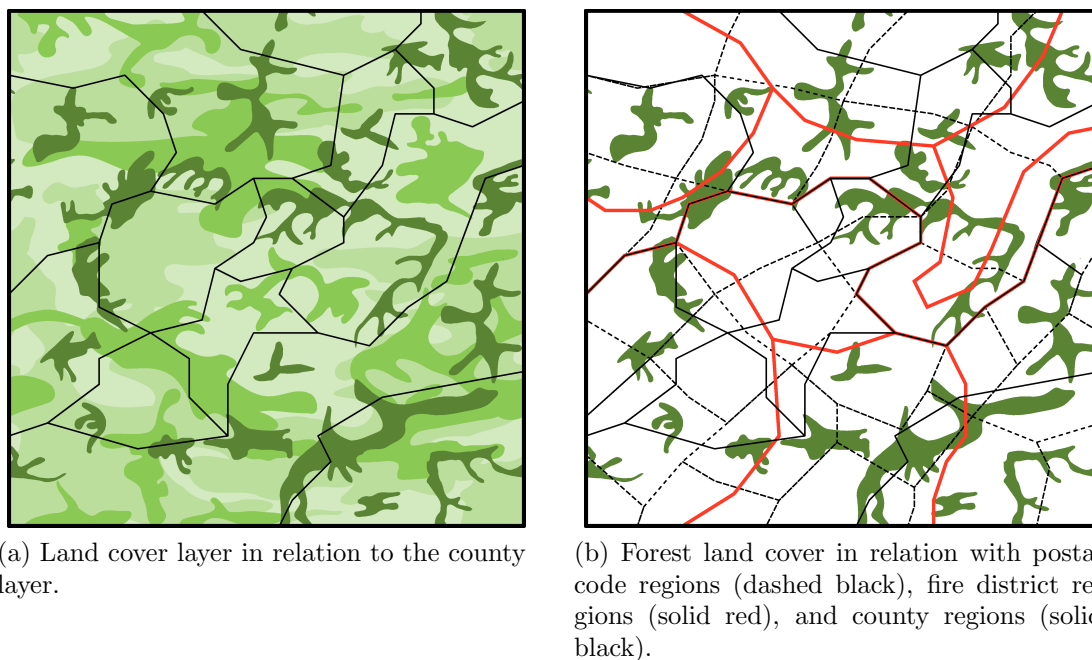


Figure 4.13: Illustration of Query 5.

Figure 4.13 shows an illustration of Query 5 and how the different aggregation levels of this query relate to each other. Note that the query restricts the land cover layer to only forest cover and that the regions represented in one layer do not align with the region objects stored in another layer. Thus, the measure records retrieved from the land cover layer are partial measures. Using a spatially extended SQL syntax, the query can be formulated as shown in Figure 4.14.

Figure 4.15 shows a query evaluation strategy that may be used for the evaluation of Query 5. Note, the structure of the data flow graph is very similar to that of Query 4. Only a `FILTER` mini-engine was added to the conceptual structure to constrain the measure records to those that represent forest areas (Step 15) and the aggregation of the measures was moved after the last `JOIN` mini-engine (Step 19). The aggregation can only be performed after the `JOIN` operation because measures can only be aggregated once their partial contribution to the final aggregate value has been determined. To determine that partial contribution, both measure and “parent” information must be available. In the case of Query 5, the area of the forest land cover only contributes partially to the final aggregate value, proportional to the amount of overlap between the forest land cover and the “parent” record of the level above. In fact, the spatial region of the “parent” record is likely also only a part of

```

SELECT
    county.name,
    fire_district.id,
    postal_code.code,
    Area(
        Intersection(
            Intersection(
                Intersection(
                    county.region,
                    fire_district.region
                ),
                postal_code.region
            ),
            land_cover.region
        )
    )
FROM county
LEFT JOIN fire_district
    ON INTERSECTS(county.region, fire_district.region)
LEFT JOIN postal_code
    ON INTERSECTS(
        Intersection(
            county.region,
            fire_district.region
        ),
        postal_code.region
    )
LEFT JOIN land_cover
    ON INTERSECTS(
        Intersection(
            Intersection(
                county.region,
                fire_district.region
            ),
            postal_code.region
        ),
        land_cover.region
    )
WHERE
    land_cover.type = 'forest'
GROUP BY ROLLUP(county.name, fire_district.id, postal_code.code);

```

Figure 4.14: Spatially extended SQL statement for Query 5, a non-strict hierarchy roll-up query.

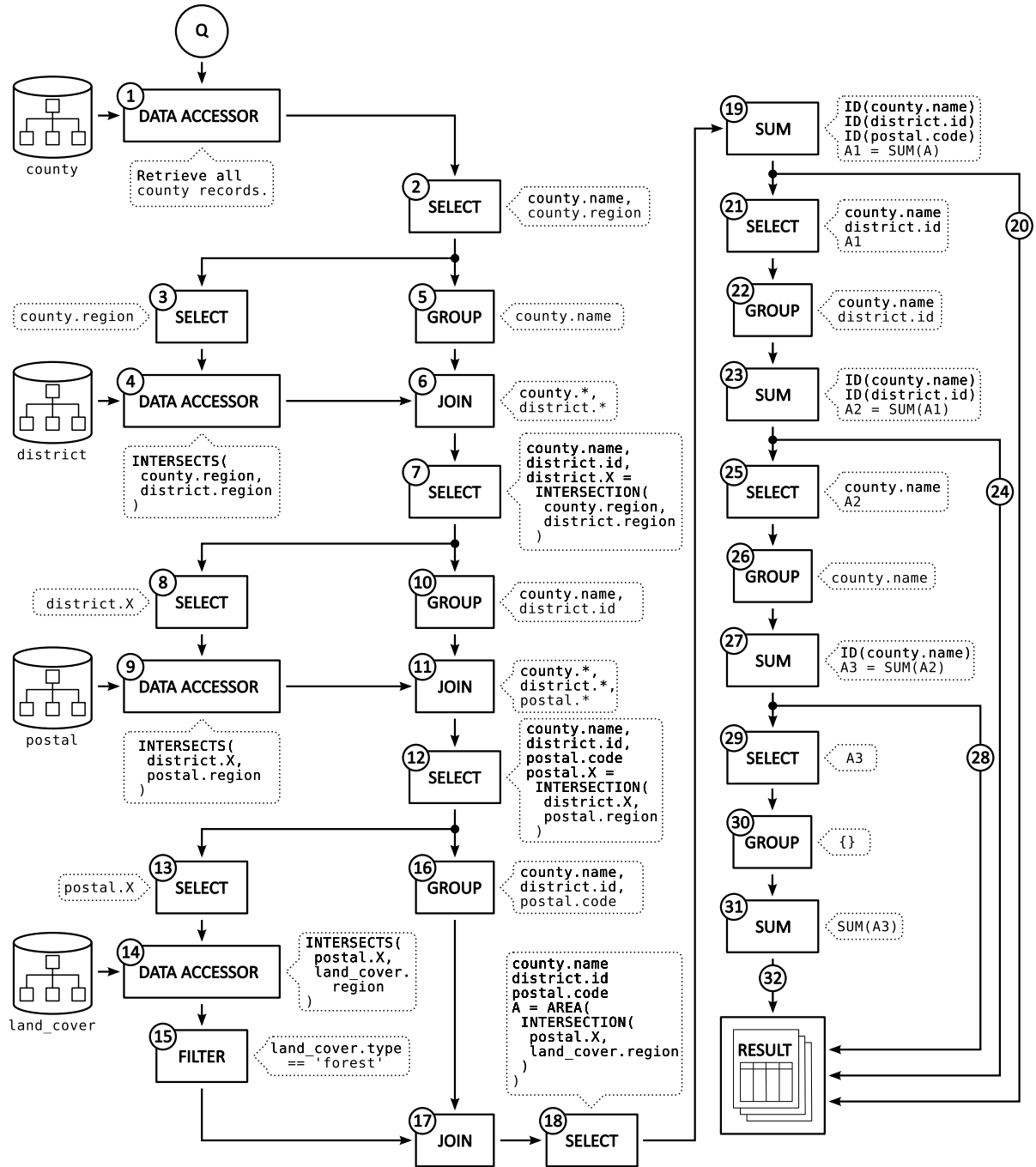


Figure 4.15: Data flow graph of an evaluation strategy for Query 5.

the full region of the corresponding postal code. Specifically, it is the intersection region between the corresponding fire district, postal code and county respectively. The partial contribution of the measure in this query is determined by computing the spatial intersection of the forest land cover region with the corresponding spatial extent of the “parent” record and then computing the area of this intersection. Since this computation requires access to the “parent” record of a measure, the computation of the partial contribution can only be performed after the JOIN operation.

#### **4.5 Summary**

In this chapter we investigated the applicability of pipeline-based query evaluation of spatial and non-spatial OLAP queries. We introduced the concept of mini-engines and described the interactions between different mini-engines as a data flow graph. The analysis of representative example queries shows that this model is expressive enough to represent the widest range of spatial OLAP applications and provides a conceptual framework which allows for a modular architecture of such systems to easily adapt to changing requirements. In the next chapter we will discuss a complete implementation of the pipeline-based query evaluation model for spatial OLAP and highlight both its flexibility and its ability to effectively utilize multi-core processors.

## Chapter 5

# LISA – A Pipeline-Based Query Evaluation System for Spatial OLAP

In this chapter we describe the design, implementation, and evaluation of the pipeline-based approach to the evaluation of spatial OLAP queries discussed in the previous chapter. We introduce the Location Intelligence and Spatial Analysis (LISA) system and describe how to translate the model from Chapter 4 into an effective system design. The discussion of LISA is divided into two parts:

1. **Design and implementation of LISA as a pipeline-based query evaluation system.** This part describes the design decisions that were made during the implementation of LISA in depth and focusses on the implementation-specific technical details of the LISA framework.
2. **Evaluation of LISA for its applicability and performance in the context of traditional and spatial OLAP.** This second part of the chapter focusses on the experimental evaluation of LISA and investigates how its implementation performs with respect to various use cases that are typical in traditional and spatial OLAP. Additionally, LISA's performance will be compared to that of other systems which use different approaches to query evaluation but provide similar functionality.

The model described in the previous chapter is ambitious in that it attempts to present a single unified approach to the expression of spatial OLAP queries that address all major spatial hierarchy types, while permitting a uniform treatment of both spatial and non-spatial data. The open questions addressed in this chapter include: 1) Can the model be realized in a functioning and complete system, and 2) Can that system be made fast and scalable by exploiting multi-core parallelism?

Section 5.1 describes the design and implementation of LISA and provides insight into how the individual components of the pipeline-based model are translated into

software components. It further explains how pipeline-based query evaluation strategies are converted into programs within the framework. Section 5.2 then focusses on the experimental evaluation of LISA with respect to performance and scalability, and provides a comparison of LISA with a spatially enabled database system. The chapter is concluded in Section 5.3 with a summary of our findings.

## 5.1 Design & Implementation

The basis for the design and implementation of the LISA system is formed by the pipeline-based query evaluation model discussed in Chapter 4. As such, LISA can be considered a reference implementation which captures every concept described by the model and provides concrete implementations for each of its building blocks. The main objective of this initial implementation was to focus on completeness and flexibility to underline the suitability of our model in practice. The implementation supports traditional OLAP queries as well as spatial OLAP queries, and it can be used for the evaluation of all the example queries discussed in Chapter 4. The absolute performance of this implementation was not our primary concern. As we will see in Section 5.2, however, the implementation still provides good results for scalability and performance when compared with other systems. The main programming language of our reference implementation is Python [184], with some components using external libraries implemented in C and C++. In the following sections we describe how the different concepts and components of our model are represented within LISA and provide some insight into the way system issues are addressed.

### 5.1.1 Mini-Engines

LISA implements mini-engines as independent processes using Python's `multiprocessing` library. This library interacts with the operating system (OS) and delegates the management and scheduling of processes to the operating system's scheduler. This approach significantly reduces the development effort, as the application does not need to provide scheduling mechanisms and can instead leverage the mechanisms provided by the operating system. At the same time the application can make use of all the resources available to the operating system, and the operating system synchronizes access to them. Additionally, with the application being both input/output and

computationally intensive, it is expected for our multi-processing implementation to gain a better utilization than a single-process implementation, as the operating system can adjust the process schedule according to I/O requirements and performance.

Mini-engines in LISA provide a common application programming interface (API), which allows them to interact with each other and other components of the data flow graph. Different types of mini-engines generally only differ in the way they are constructed to account for different parameters that are required by their specific implementations. Once a mini-engine is constructed, it has a well defined interface for reading input data as well as providing output. The construction of mini-engines and their arrangement into a data flow in an automatic manner is typically the responsibility of a query optimizer. The design and implementation of a query optimizer is not in the scope of this chapter, and the data flows for queries used in later sections of this chapter were manually constructed.

### 5.1.2 Streams

As defined by the model, records are transported between mini-engines using streams. In LISA, these streams are represented by queue data structures. In fact, each stream is represented by an array of queues, one for each endpoint of the stream. When a record is inserted into a stream, the stream implementation automatically appends the record to each endpoint's queue. The endpoints, typically other mini-engines, can then asynchronously retrieve their next record from the stream. Note that mini-engines connected to the same stream will likely operate at different speeds and, thus, a single record may be consumed by each mini-engine at a different time. The queues additionally act as buffers for records, which allow mini-engines to access record streams in an asynchronous fashion and thus reduce the risk of a slower mini-engine blocking the progress of faster mini-engines. To control the overall memory allocation for stream buffers, the size of each queue is limited and, when a queue is full, the sender is blocked until the receiver removes records from the beginning of the queue.

### 5.1.3 Tracks

The concept of tracks is not part of the description of the model in Chapter 4. Instead it is introduced by LISA to achieve a better utilization of multi-processor and

multi-core hardware. The interleaving of the processing of different independent mini-engines in a data flow can greatly benefit from the availability of multiple processors. As mini-engines vary in complexity, less complex and faster mini-engines may be slowed down by more complex mini-engines that require more computation time to process. In these cases it may be beneficial to create multiple instances of the more complex mini-engines and distribute the independent tasks among them evenly. On multiple processors this allows the complex mini-engines to run in parallel and perform the same amount of work in less time, thus reducing the wait-time for faster mini-engines. In LISA these parallel processing paths are called tracks and are not limited to individual mini-engines but allow the parallelization of entire sections of the data flow. To accomplish this, LISA implements two additional components that can be part of a data flow: a multiplexer and a demultiplexer.

The multiplexer or **MUX** component is an adaptor for an ordinary record stream that acts like an endpoint with respect to the input stream. It also provides the same interfaces as a stream, and mini-engines can register with it as endpoints. When transferring a record from the input stream to an endpoint, the record is not being sent to all endpoints but only to a single endpoint. Thus, each endpoint receives a different subset of the records in the input stream, and mini-engines attached to these endpoints process disjoint subsets of the records. Consequently the records of the original input stream are distributed among all endpoints, with each such endpoint being the first mini-engine of an independent parallel track (see Figure 5.1).

The demultiplexer or **DEMUX** component is a mini-engine which reverses the effects of the **MUX** component and combines multiple independent record streams into a single record stream. All of the input streams are required to have the same record schema. The **DEMUX** mini-engine is typically used to combine the output of streams of multiple parallel tracks into a single output stream, which can then be processed further. It does not, however, provide any guarantees about the order in which these records are combined.

The combination of the **MUX** stream adaptor and the **DEMUX** mini-engine is used by LISA to realize multi-track processing, which improves the utilization of multi-processor and multi-core systems. Figure 5.1 shows a schematic representation of the use of the **MUX** and **DEMUX** components.



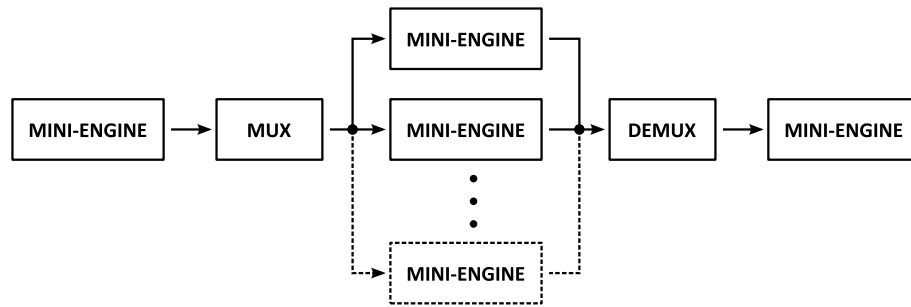


Figure 5.1: Conceptual use of the MUX and DEMUX components in a data-flow.

#### 5.1.4 Data Accessors

Data Accessors in LISA are defined in a generic manner to provide access to data without exposing a particular implementation. The specific implementations of data structures and access methods are encapsulated in classes of data sources and access methods, respectively. Thus, each instance of a **Data Accessor** is passed a reference to a data source instance and an access method instance.

LISA currently provides a number of data source implementations:

1. Comma-separated values (CSV) flat file: the simplest type of data source available in LISA. It is represented by a plain text file where each line corresponds to a record and the attribute values of each record are delimited by commas. The CSV file data source does not provide any indexing, nor does it guarantee the records are sorted in any particular manner. Thus, the access method implementations that are using this data source are required to scan over the records until the result of a query can be identified.
2. SQLite database table: SQLite [90] is a popular embedded relational database system which allows to store structured data in tables and provides a rich SQL-based query language. Additionally it can provide B-tree indices on sortable table attributes and thus significantly speed up the retrieval of records. SQLite databases are single files and the SQLite engine to access these database files is directly linked into the client application. LISA provides a data source interface to individual tables stored within a SQLite database through Python's native SQLite bindings [85]. Access methods using such a data source can utilize any indices defined on the table and, thus, can retrieve records efficiently without

scanning the entire record set.

3. R-tree: The R-tree data source in LISA provides efficient access to spatial data. The data source can be divided into two parts: the dataset and the index. The dataset is represented as an unordered sequence of geometric objects represented in the “well-known binary” (WKB) format [132], a binary representation of spatial objects regulated by the Open Geospatial Consortium. The dataset provides random access to any location within the sequence, and the entire dataset file is mapped into virtual memory for convenient access and improved performance. The index part of the data source is implemented as an R-tree using the Spatial Index Library [77] and its associated R-tree Python bindings [63]. The index is created during the construction of the dataset, and for each geometry in the dataset, it uses its rectilinear bounding box as a key. The value associated with a key is the position of the geometry’s binary representation within the dataset. The index can then be used to locate geometry records within the dataset based on the intersection of their bounding boxes with the query region. The geometries can be retrieved from the dataset based on the location reference returned by the R-tree query. Geometric objects are stored in WKB format on disk and converted into an internal representation when retrieved. The internal representation of geometric objects in LISA is provided by the GEOS library [41], an open source geometry engine, and its corresponding Python bindings called Shapely [64].

The implementation of the R-tree data source intentionally separates the data from the index, as the geometries stored in the dataset have variable complexity and their binary representations vary in size. These variable size binary objects cannot efficiently be stored as values in the R-tree. One level of indirection through the position within the dataset sequence solves this problem.

The R-tree index returns all geometries whose bounding boxes are contained within or intersect with the query region’s bounding box. This may include geometries whose actual boundaries are not within or do not intersect with the query’s actual boundary. To address this, the R-tree data source provides the additional functionality to filter all retrieved records based on their geometries’

relationships to the query region. This approach is not optimal, and in unfavorable cases, a query may retrieve many more records than are ultimately reported. In the general case, spatial objects are often arranged in such a way that the overhead is proportional to the size of the query region and, thus, approximately constant with respect to the number of result records.

In addition to these data source implementations, LISA provides two access method implementations: an identity query and a range query. Given an attribute name and a query value, the identity query returns the first record it can identify in the corresponding data source for which the specified attribute has the same value as the query value. This identity query is typically used to query records by their primary key. The range query access method, on the other hand, is used to find records whose values for a given attribute fall “within” the query range. The query range must resemble some sort of extent. For numeric attributes, the query range is represented by a range data type holding an upper limit and a lower limit. Polygonal geometry attributes do not require a special range data type, as they already represent extents. The identity and range query access methods are the main access methods used during the evaluation of LISA. However, based on application requirements, other access methods can be easily implemented.

### **5.1.5 Geometric Functions and Operators**

LISA uses the Geometry Engine Open Source (GEOS) library [41] for its internal representation of geometric objects. GEOS is a C++ port of the Java Topology Suite [42] and is used by a number of commercial and open-source software packages. It provides an object-oriented representation of all geometry types defined by the Open Geospatial Consortium [132] and implements various functions and operators that can be used with these geometries. Examples of such functions and operators that are frequently used in typical LISA queries are “contains”, “intersects” and “intersection”. To use GEOS’s C++ API from within Python, our implementation uses the Shapely library, which implements a Python wrapper on top of the GEOS C++ library.

### 5.1.6 Query Strategy Definition

This section describes the definition of query strategies using the LISA framework. It allows the evaluation of a wide range of queries with varying complexity. This includes traditional database and OLAP queries as well as spatial OLAP queries. At this point, the LISA framework does not provide an automatic query compiler or query optimizer, and query evaluation strategies that follow the pipeline model described in Chapter 4 are manually defined using the Python programming language and LISA's query evaluation framework.

To illustrate an example of the definition of such a query evaluation strategy, consider Query 3 discussed in Chapter 4:

```
SELECT county.id, AVERAGE(plant.height)
FROM county
LEFT JOIN plant ON CONTAINS(county.boundary, plant.location)
WHERE CONTAINS(
    MakeBox2D(
        MakePoint(42.975, -69.351),
        MakePoint(48.228, -59.480)
    ),
    plant.location
)
GROUP BY county.id
```

Chapter 4 describes in detail how this query can be translated into the query evaluation strategy data flow graph shown in Figure 5.2. Based on this dataflow, the strategy can be defined in Python using the LISA framework as described in the following:

1. **Definition of data sources:** The first step is the definition of data sources. This includes the definition of a mini-engine that generates the stream of queries as well as the data sources to retrieve records from. Both spatial object layers are defined as R-tree data sources.

```
1 ##### Definition of Data Sources #####
2 # Definition of the query schema. The target attribute for
3 # the query is the location attribute of the plants dataset.
4 query_schema = Schema()
5 query_schema.append(Attribute('plants.location', Geometry))
```

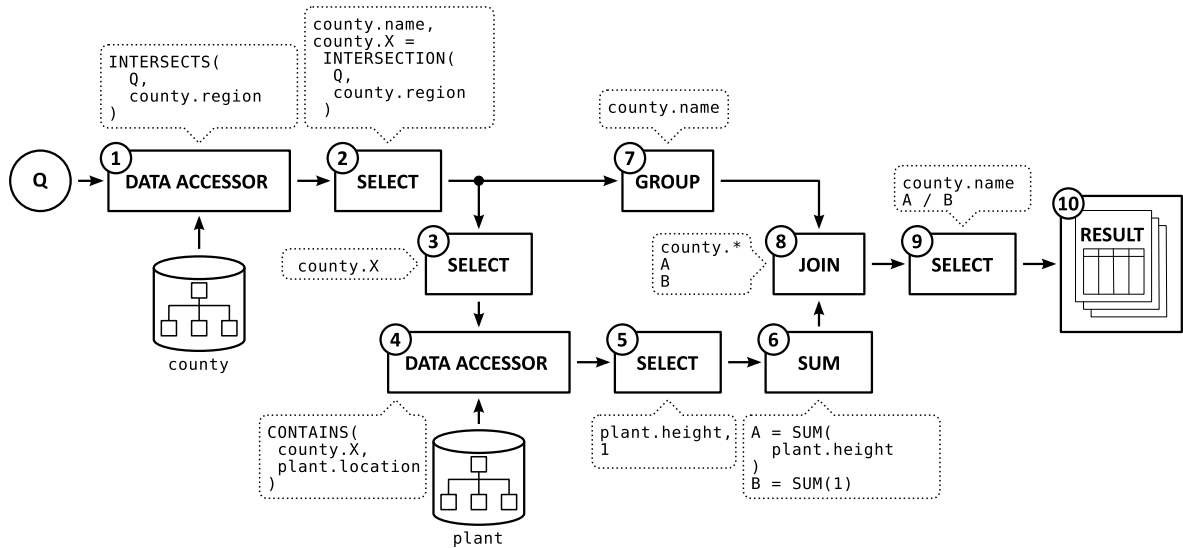


Figure 5.2: Data flow graph of an evaluation strategy for Query 3. This dataflow graph is identical to that shown in Figure 4.7 of Chapter 4.

```

6
7 # Define the query region.
8 query = Geometry(Polygon((
9     (-69.351, 48.228),
10    (-59.480, 48.228),
11    (-59.480, 42.975),
12    (-69.351, 42.975)
13 )))
14
15 # ArrayStreamer is a mini-engine that provides a record
16 # stream filled with records provided as arguments.
17 query_streamer = ArrayStreamer(query_schema, [
18     # A single record containing the query region.
19     (query,),
20 ])
21 engines.append(query_streamer)
22
23 # Specify the file datasets
24
25 counties_file = 'data/spatial/counties'
26 counties_source = Rtree(counties_file, 'county.boundary')
27
28 plants_file = 'data/spatial/plants'
29 plants_source = Rtree(plants_file, 'plant.location')

```

2. **Query records from the county layer:** The second step is the creation

of a `DataAccessor` for the counties' data source. The queries used by the `DataAccessor` are retrieved from the output stream of the `query_streamer` mini-engine created in the previous step. As we are interested in all counties intersecting with the query region, we are using a range query (`FindRange`) access method in conjunction with the counties' R-tree data source. The final `engines.append(counties_accessor)` appends the mini-engine to the list of mini-engines that are to be started when the query evaluation begins. For each mini-engine in this list an individual process will be created once the construction of all mini-engines is completed. The processes are then executed concurrently to perform the query evaluation.

```

1 ##### Query records from county layer #####
2 # the FindRange access method. The range queries are the output
3 # of the query stream.
4 counties_accessor = DataAccessor(
5     query_streamer.output(),
6     counties_source,
7     FindRange,
8 )
9 engines.append(counties_accessor)

```

3. **Demultiplexing of the counties record stream:** To allow for a better utilization of processors in the system, the record stream of counties is demultiplexed for multiple records to be processed asynchronously in parallel. Each processing track attaches to an output channel of the `Demux` component. The `Demux` component is a LISA specific extension of the model described in Chapter 4 and not shown in the dataflow graph illustrated in Figure 5.2. For this query strategy the `Demux` component is inserted between steps 1 and 2 shown in the dataflow graph and allows the execution of steps 2 through 8 by multiple parallel tracks. The output of these tracks, following the `Join` mini-engine in step 8, is then multiplexed into a single record stream.

```

1 ##### Demultiplexing county record stream #####
2 # Setup a demultiplexer for the county record stream
3 # across a number of independent computation tracks.
4
5 demux = Demux(counties_accessor.output())

```

```

6
7 for i in range(tracks):
8     # Obtain the current track's demultiplexed county
9     # record stream.
10    channel = demux.channel()

```

4. **Prepare query ranges for plants layer:** The queries executed on the plants layer are derived from the intersections of the original query region and the counties that intersect with it. Using this approach, each query at the plants layer will only return records that are within the corresponding county *and* within the query region. The stream of county records is provided by the `channel` object, which is an output channel of the previously created `Demux` component. The `Select` mini-engine then applies a transformation to each record, which computes the intersection between the retrieved county record's geometry and the original query's geometry. Note, although the resulting spatial attribute is a geometry with extent, it is being renamed to `plant.location`, as our LISA implementation requires the attribute name of the query record to match the name of the attribute that is being queried.

```

1     ##### Prepare queries for plants layer #####
2     # Calculate the query range over locations at the plant
3     # layer. Trim the retrieved counties to the boundary of
4     # the original query.
5     counties_select = Select(
6         channel,
7         UniversalSelect(
8             channel.schema(),
9             {
10                # Rename resulting attribute to match the
11                # attribute that is being queried.
12                'plant.location': {
13                    'type': Geometry,
14                    'args': ['counties.boundary'],
15                    'function': lambda v: intersection(v, query),
16                }
17            }
18        )
19    )
20    engines.append(counties_select)

```

5. **Query records from the plants layer:** Using the query ranges computed by

the `Select` mini-engine from the previous step, the plant layer's `DataAccessor` is being used to query plant records that are contained within each query. The output of the `DataAccessor` mini-engine is a stream of plant records partitioned based on the result sets obtained from each input query. Thus, plant records within one partition are all contained within the same county.

```

1     ##### Query records from plants layer #####
2     plants_accessor = DataAccessor(
3         counties_select.output(),
4         plants_source,
5         FindRange
6     )
7     engines.append(plants_accessor)

```

6. **Select and aggregate measures:** The objective of the query is to report, for each county, the average height of plants that intersect with the query region. Thus, the query must compute the average height of all plant records that are within the same partition of the previous `DataAccessor` mini-engine's output stream. The computation of the average is a typical aggregate function, and we are using the `Aggregate` mini-engine to implement this aggregation. However, as a preprocessing step, we remove all attributes except the `height` attribute from the plant records. To do this, we employ a `Select` mini-engine which transforms all plant records retrieved in the previous step into records with only one `height` attribute. During this process, the `Select` mini-engine passes through any stop words included in the input stream, so that the partitioning of the transformed plant records is preserved. The output of the `Select` mini-engine is then passed to the `Aggregate` mini-engine, which executes the `AverageAggregator` function with the records in each partition of the input stream. The output of the `Aggregate` mini-engine in turn is a record stream partitioned in the same manner as the input stream. Each output partition contains only one record with the average height associated with its `height` attribute.

```

1     ##### Select and aggregate measures #####
2     plants_select = Select(
3         plants_accessor.output(),

```



```

4     UniversalSelect(
5         plants_accessor.output().schema(),
6         {
7             'height': {
8                 'type': float,
9                 'args': ['plants.height'],
10                'function': lambda v: v
11            }
12        }
13    )
14 )
15 engines.append(plants_select)
16
17 plants_aggregate = Aggregate(
18     plants_select.output(),
19     AverageAggregator(plants_select.output().schema(), 'height')
20 )
21 engines.append(plants_aggregate)

```

7. **Select county ID and prepare for JOIN:** For the final report produced by the query, each computed average plant height for a county is to be associated with that county's ID. Thus, each record produced by the previous step's **Aggregate** mini-engine must be associated with the ID of the county which generated the corresponding query for the plants layer. As the order of partitions in the **Aggregate** mini-engine's output stream is the same as the order of the track's county record stream, it is possible to use a **Join** mini-engine to combine the records of both streams into a stream of combined records. For the **Join** mini-engine to work correctly, both input streams must be partitioned. This step transforms each county record processed by the track into a record that only contains the county's ID, using a **Select** mini-engine. Subsequently, the stream of transformed county records is partitioned using the **Group** mini-engine. As each county has a unique ID, the **Group** mini-engine uses the `id` attribute of each input record as the criterion for each equivalence class that forms an output stream partition. Consequently, the output of the **Group** mini-engine is a stream of records partitioned by the `id` attribute containing one record per partition.

```

1     ##### Select county ID and prepare for JOIN #####
2     select = Select(

```

```

3         channel,
4         UniversalSelect(
5             channel.schema(),
6             {
7                 'id': {
8                     'type': int,
9                     'args': ['id'],
10                    'function': lambda v: v
11                },
12            }
13        )
14    )
15    engines.append(select)
16    counties_grouper = Group(
17        select.output(),
18        {'id': lambda a, b: a == b}
19    )
20    engines.append(counties_grouper)

```

8. **Join county IDs with aggregate values:** To prepare the final result output, the `Join` mini-engine combines the partitioned stream of county ID records with that of average plant height records. The partitions of both streams each contain only a single record, so that the resulting output stream contains as many records as there were counties selected in Step 2. Each output record has two attributes: `id` from the stream of county ID records and `height`, the average plant height, from the `Aggregate` mini-engine's output stream. The `Join` mini-engine's output stream is then appended to a list of output streams that need to be multiplexed to combine the results from independent processing tracks into a single output stream.

```

1     ##### Join county IDs with aggregate values #####
2     joiner = Join(
3         counties_grouper.output(),
4         plants_aggregate.output()
5     )
6     engines.append(joiner)
7
8     mux_streams.append(joiner.output())

```

9. **Combine tracks and report results:** The final step in the query evaluation strategy is the multiplexing of all tracks' output streams and the reporting

of the results. The `Mux` mini-engine is used to combine the output stream of each track's `Join` mini-engine into a single record stream. The `mux_streams` parameter to the `Mux` mini-engine is an array of record streams that are to be combined. Each track inserts its corresponding output stream into this array. As with the `Demux` mini-engine, the `Mux` mini-engine is a LISA specific extension to the model described in Chapter 4 and not show in the dataflow graph in Figure 5.2. Conceptionally, the `Mux` mini-engine is inserted into the dataflow graph between steps 8 and 9. Finally, the `ResultFile` mini-engine is used to write the records from the `Mux` mini-engine's output stream to a text file using comma-separated values for reporting.

```

1 ##### Combine tracks and report results #####
2 mux = Mux(*mux_streams)
3 engines.append(mux)
4
5 result_stack = ResultFile(
6     'results.txt',
7     mux.output(),
8 )
9 engines.append(result_stack)

```

After the definition of the query evaluation strategy and the construction of the mini-engines, a managing task executes the query by starting the individual processes associated with the mini-engines. While results of the query are being emitted as they are available, the evaluation of the query is completed once all mini-engine processes terminate.

Query 3 presented here only utilizes a subset of LISA's capabilities and more complex query evaluation strategies can be implemented analogous to the strategy implemented for Query 3. The experiments discussed in the following sections, for example, require more sophisticated query evaluation strategies whose implementations within the LISA framework can be found in Appendix B.

## 5.2 Evaluation of LISA

This section presents a comprehensive evaluation of our LISA system and analyzes the performance, runtime behaviour and applicability to real-world applications of

our LISA implementation. As our LISA reference implementation provides support for different types of application scenarios, we evaluate each scenario independently. In conclusion we then compare our observations with those for other software systems that provide similar functionality.

The evaluation of LISA can be divided into the following experiments:

1. Evaluation of LISA with respect to traditional OLAP
2. Evaluation of LISA with respect to spatial OLAP
3. Comparison of LISA with other systems

For each of these experiments we investigate the performance of our LISA implementation with respect to its main feature, the parallel execution of individual queries, as well as the size of the data being queried.

All of the experiments were conducted within the same experimental environment. It consisted of a cluster of compute nodes, each equipped with two Quad-Core Intel<sup>®</sup> Xeon<sup>®</sup> E5430 2.66 GHz processors (Hyperthreading turned off), 8 GB of RAM, and two 147 GB SAS hard drives in RAID 0 configuration. The operating system used on the compute nodes was CentOS 5.3 Linux. Access to the compute nodes was dedicated, so that no other applications other than operating system services interfered with the experimental evaluation.

### **5.2.1 Evaluation of LISA with Respect to Traditional OLAP**

This experiment evaluates our LISA reference implementation with respect to traditional OLAP queries. As an example for a traditional OLAP query we have chosen a hierarchical roll-up query. Unlike a CUBE query, the roll-up query does not incur any additional complexity for cube selection or efficient cube generation. Nevertheless, it is a critical sub-task of the cube generation process. Thus, the roll-up query is a good representative choice for a traditional OLAP query. Specifically, we have chosen a query similar to Query 2 discussed in Section 4.4.2 of Chapter 4.

The dataset for the evaluation of LISA with respect to traditional OLAP roll-up queries is a synthetic dataset which represents a data warehouse of customer-order

relationships and is derived from the well known TPC-H data warehouse benchmark [176]. The entities of the TPC-H dataset that are of interest for our experiments are:

- nation,
- customer,
- orders, and
- lineitem.

Within the TPC-H dataset, these entities form a hierarchical relationship of the following form: `nation`  $\rightarrow$  `customer`  $\rightarrow$  `orders`  $\rightarrow$  `lineitem`. This hierarchy will be used as the roll-up hierarchy by our experimental evaluation.

The datasets used in this evaluation were generated using the dataset generator “DBGGEN” provided by the Transaction Processing Performance Council (TPC). The size of a dataset is measured by a scaling factor  $SF$ , and the following table shows the number of records for each entity for different scaling factors:

<b>Entity</b>	$SF = 1$	$SF = 0.5$	$SF = 0.1$
nation	25	25	25
customer	150000	75000	15000
orders	1500000	750000	150000
lineitem	6001215	2999671	600572

For our experiments, we prepared datasets with sizes between  $SF = 0.1$  and  $SF = 1$ . Each dataset was stored in a dedicated SQLite database file and organized in tables representing a normalized star schema [81]. The `lineitem` entity stores, in addition to the primary and foreign keys, also a `quantity` attribute and a `price` attribute, which were used in our test query.

The query used in the following experiments was similar to that described as Query 2 in Chapter 4 and can be formulated using SQL as:

```

SELECT nation.id, customer.id, orders.id, MAX(lineitem.price)
FROM nation
LEFT JOIN customer ON customer.nation_id = nation.id
LEFT JOIN orders ON orders.customer_id = customer.id
LEFT JOIN lineitem ON lineitem.order_id = orders.id
WHERE lineitem.quantity >= 10 AND lineitem.quantity <= 15
GROUP BY ROLLUP(nation.id, customer.id, orders.id)

```

The structure of this query is identical to that of Query 2 discussed in Chapter 4. For this experiment we used the evaluation strategy that was described in the context of Query 2 (see Figure 4.5) and adapted it to the entity and attribute names used in the experiment’s dataset. The translation of the evaluation strategy for this experiment’s query into Python code using the LISA framework can be found in Appendix B.2.

### Parallel Speed-Up

The first metric we wanted to evaluate is how the parallel execution of mini-engines influences the running time of the query. For this purpose, the dataset was fixed in size at a scaling factor of  $SF = 0.5$ , and the query was executed for different processor configurations. Each test platform provided a maximum of 8 processing cores, and for different processor configurations individual processing cores were disabled or enabled using interfaces provided by the operating system. For each processor configuration, the query was also executed with different configurations of the number of parallel tracks within the query’s data flow. Parallel tracks, as described in Section 5.1.3, are copies of a section of the dataflow that are executed in parallel on independent sets of records. The metrics measured during the execution of the run were “wall time”, i.e., the total amount of time required to complete the query, “user time”, i.e., the amount of CPU time spent by the query performing internal calculations and executing the program flow; and “system time”, i.e., the amount of CPU time spent by the query waiting for operating system calls, such as disk I/O. For our evaluations, however, we focus only on “wall time” as a practical measure. Each measure was averaged over a minimum of 3 runs to compensate for variance in measuring the metrics.

The graph in Figure 5.3 shows the running time of the query with respect to the number of processing cores enabled in the system. For each individual processor configuration, we also chose the best configuration of parallel tracks, i.e., the number

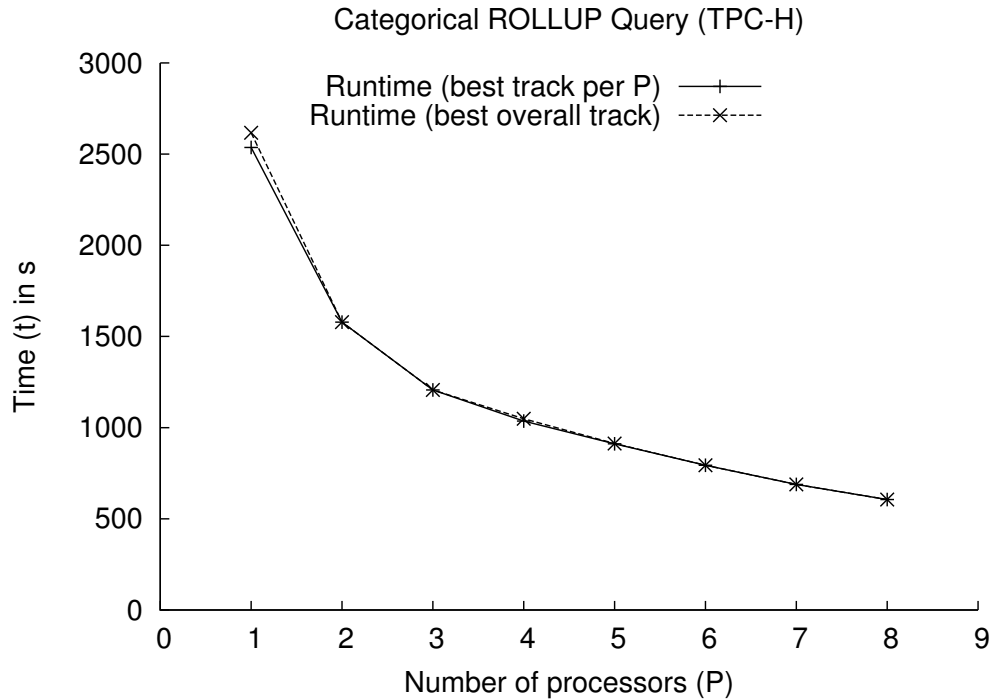


Figure 5.3: Query runtime by number of processors.

of parallel tracks that resulted in the best performance. We can observe that an increasing number of available processing cores allows for more parallelism during query evaluation, and that the query running time is notably reduced when more processors are available. The graph also suggests that a duplication of computational resources does not reduce the query running time by an equivalent factor. This can also be observed in Figure 5.4, which shows the factor of speed-up and the efficiency of configurations using multiple processing cores compared to runs on a single processor. It indicates that the speed-up obtained by increasing the number of processors is nearly linear. However, each additional processor appears to contribute only about 50% of its capacity. This observation is further supported by the graph showing the overall efficiency based on the number of processors, which shows the efficiency converging to approximately 50% for 8 processors. This is a rather unexpected behaviour. Typical reasons for a degradation of processor performance are bandwidth limitations or synchronization overhead. Often in these situations the overhead would increase as more processors are added and more processors compete for the same resources. The experiments, however, do not indicate a further degradation of the processor

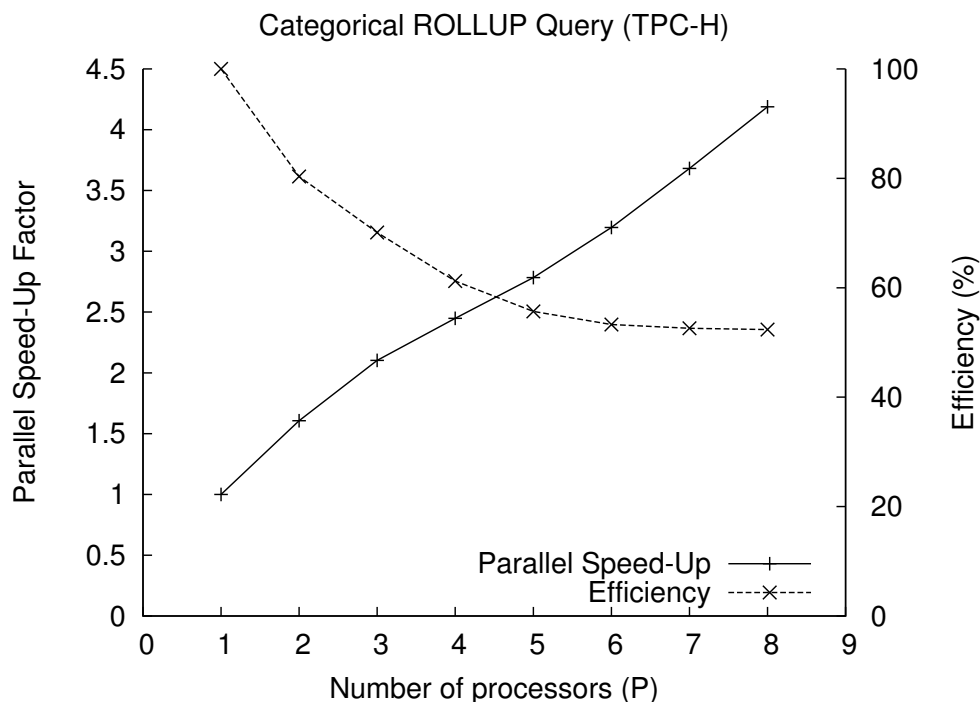


Figure 5.4: Query speed-up factor and efficiency for multiple processors with respect to single processor runs.

performance and suggest that any additional processors would contribute a constant amount of approximate 50% of their capacity. This behaviour can be explained with a deeper analysis of the query evaluation strategy: The evaluation strategy used for this query was implemented so that it distributes the queries on the `lineitem` layer over multiple parallel tracks. Each of these queries incurs a fixed overhead of disk I/O, which the operating system can mask by allocating processor resources to other mini-engines while the `lineitem` query waits for the disk subsystem to provide data. In this experiment, this is most effectively done in the case of a single processor. For more than one processor, the amount of computation that is available to be processed while waiting for disk I/O to complete becomes proportionally smaller as the number of processors increases. This causes additional processors to become less utilized and thus spending approximately 50% of their capacity in an idle state waiting for disk I/O to complete. This behaviour does not indicate a disk I/O bottleneck, as the time spent waiting for disk I/O does not increase with an increasing number of processors; it merely shows a high fixed cost or latency for disk accesses. If the number



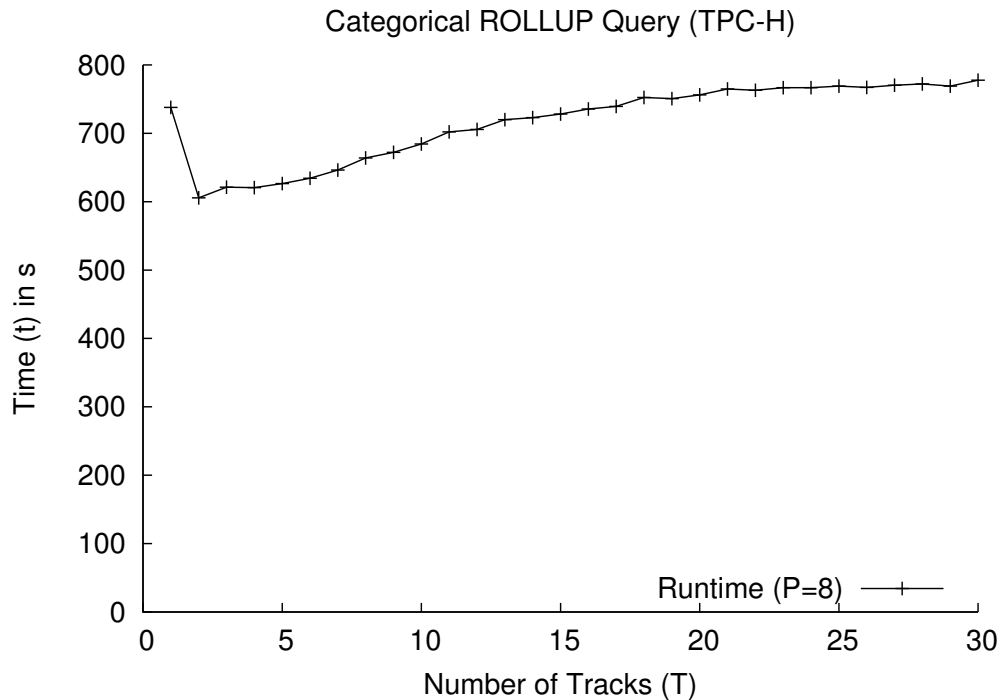


Figure 5.5: Impact of track configuration on query evaluation time.

of processors was to increase even further, a typical I/O bottleneck behaviour would eventually present itself as processes start to compete for the limited I/O bandwidth of the disk subsystem.

The impact of the number of parallel tracks chosen for the evaluation of the query is presented in Figure 5.5. It shows that a choice of two parallel tracks notably improves the running time of the query. A larger number of tracks increases the overhead incurred by managing the additional number of mini-engine processes for each track. In fact, for 17 (a total of 143 processes) or more tracks, the overhead of process management starts to dominate the computation required for query evaluation, and the execution of the query begins to slow down compared to a single track configuration.

### Data Size

The impact of the dataset size on the running time of the query was evaluated by using a fixed processor configuration and varying the scaling factor of the input dataset between  $SF = 0.1$  and  $SF = 1.0$ . The metrics measured in this experiment were the same as for the evaluation of the parallel speed-up, and running time values presented

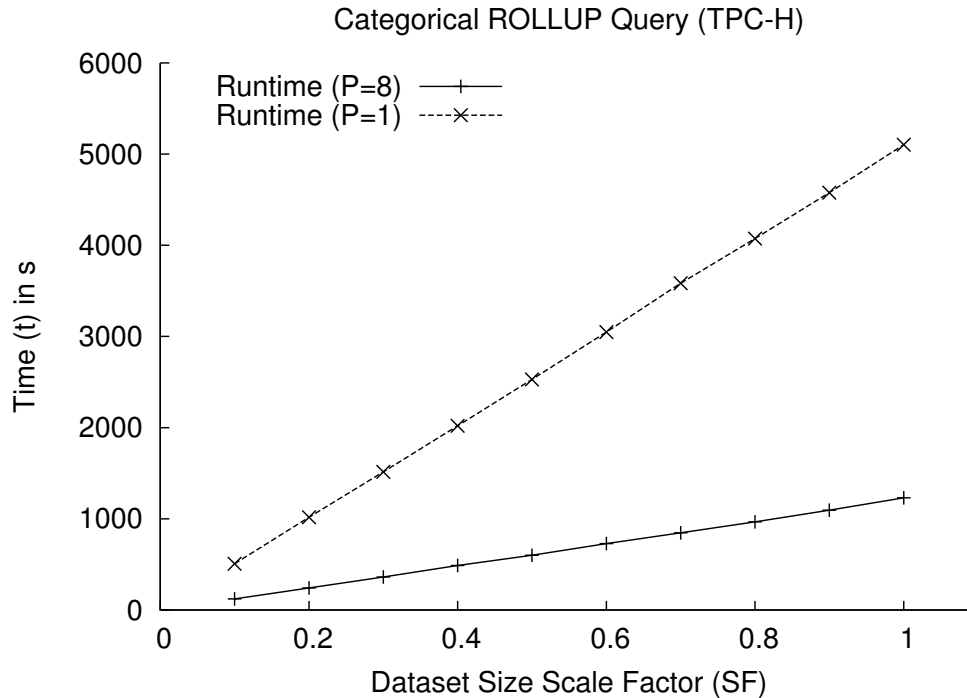


Figure 5.6: Impact of dataset size on query evaluation time.

here refer to “wall time”.

The graph in Figure 5.6 shows the influence of the dataset size on the query time for two different processor configurations:  $P = 1$  and  $P = 8$ . We can observe that the dataset size has a linear influence on the running time of the query, which causes the query evaluation time to increase proportional to the increase in the dataset’s size. This behaviour is expected, as the roll-up hierarchy in the TPC-H dataset is a strict hierarchy, which implies there exists only one path from a record at the `lineitem` level to the `nation` level of the hierarchy. An increase of the scaling factor increases the size of all hierarchy levels proportionally and with it the number of roll-up paths. Our query evaluation traverses these roll-up paths and, consequently, its running time increases proportionally to the number of paths. The graph also shows that this observation is true for both, runs with a single processor and with multiple processors.

### 5.2.2 Evaluation of LISA with Respect to Spatial OLAP

For the evaluation of the LISA reference implementation with respect to spatial OLAP, we chose a roll-up query over a non-strict hierarchy of spatial layers. We

believe this type of query is representative of typical spatial OLAP queries, as it involves a combination of traditional OLAP concepts, such as roll-up aggregation of measures, as well as processing of spatial data. Furthermore, the query addresses challenges associated with processing spatial hierarchies, such as non-strict associations between hierarchy levels and partial measures. Specifically, we chose a query similar to Query 5 discussed in Section 4.4.5.

The dataset used for the evaluation of spatial OLAP queries with LISA was a real-world dataset derived from data retrieved from the United States Geological Survey (USGS) and the United States Census Bureau (USCB). It was composed of the following layers:

Layer Name	Description	Number of Objects	Source
<b>states</b>	Boundaries of U.S. federal states	51	[180]
<b>counties</b>	Boundaries of U.S. counties	3140	[179]
<b>zip5</b>	Boundaries of U.S. 5-digit ZIP codes	50065	[178]
<b>lulc</b>	Land Use and Land Cover	361202	[181]

The query used in this experiment used the following non-strict hierarchical relationship between these layers: **counties**  $\rightarrow$  **states**  $\rightarrow$  **zip5**  $\rightarrow$  **lulc**<sup>1</sup>.

In addition to the full-size dataset described above, parts of this experiment were conducted with smaller subsets of the same dataset. Similar to the dataset size notation we used in the evaluation of LISA with respect to traditional OLAP, we use a scaling factor  $SF$  to denote different sizes of datasets in this evaluation, where  $SF = 1.0$  specifies the full-size dataset. The construction of smaller subsets of the original dataset was done by selecting a random subset of spatial objects from the lowest layer of the hierarchy, i.e., **lulc**. The number of objects selected from that layer was equal to the total number of objects in the layer times the scaling factor. We chose to apply the dataset scaling only to the lowest layer of the hierarchy as

---

<sup>1</sup>The **counties** layer was chosen above the **states** layer. Although counties are administratively well contained within federal states, the resolution of the underlying data caused some **counties** object boundaries to cross **states** object boundaries, generating a non-strict relationship. Placing the **counties** layer above the **states** layer resulted in a larger fan-out of the roll-up hierarchy tree at the top level of the hierarchy. For the evaluation of our query, this had no significant impact on the query performance, as both layers were relatively small compared to the other layers in the hierarchy. Also, the way the query evaluation strategy was defined, the order of the upper layers had no impact on the queries executed at the lower levels of the hierarchy.

the higher levels only have small numbers of objects. Their reduction in combination with the query’s regional constraint may result in a significantly skewed query load.

For our experiments, we prepared datasets with sizes between  $SF = 0.1$  and  $SF = 1$ . For the use by our LISA reference implementation, each layer of each dataset was stored in LISA’s own data format and indexed using an R-tree index as described in Section 5.1.4 earlier in this Chapter.

The query used in the following experiments was similar to that described as Query 5 in Chapter 4 and can be formulated using SQL as shown in Figure 5.7. A visual representation of the query region and the top two layers of the roll-up hierarchy, `states` and `counties`, is shown in Figure 5.8.

The structure of this query was identical to that of Query 5 discussed in Chapter 4 and we used the evaluation strategy shown in Figure 4.15 for its evaluation. The translation of the evaluation strategy into Python code using the LISA framework can be found in Appendix B.5.

In this experiment, we evaluated the same aspects of the query evaluation as we did for LISA with respect to traditional OLAP queries.

### Parallel Speed-Up

As for LISA with respect to traditional OLAP, the first metric we evaluates was how the parallel execution of mini-engines influences the running time of the query. For that we used the full-size dataset with a scaling factor of  $SF = 1.0$ . The query was executed for different processor configurations, and we measured the “wall time” running time of each query execution. The measured values were averaged over a minimum of 3 runs.

The graph in Figure 5.9 shows the running time of the query with respect to the number of processing cores enabled in the system. For each processor configuration, we chose the best configuration of parallel tracks. We observe that with an increasing number of processing cores, the running time of the query execution was reduced. In addition, in the graph in Figure 5.10, we can see that the parallel speed-up from additional processors was nearly linear with a significantly higher efficiency when compared to results we have obtained from the evaluation against traditional OLAP queries. This can be attributed to the larger amount of processor-bound computation

```

query := CAST(MakeBox2d(
  MakePoint(-93.88, 24.22),
  MakePoint(-65.39, 49.81)
) AS geometry);
SELECT
  states.id,
  counties.id,
  zip5.id,
  SUM(
    Area(
      Intersection(
        Intersection(
          Intersection(
            Intersection(
              states.geom,
              query
            ),
            counties.geom
          ),
          zip5.geom
        ),
        lulc.geom
      )
    ) / Area(lulc.geom)
  )
FROM states
INNER JOIN counties
  ON INTERSECTS(
    Intersection(
      states.geom,
      query
    ),
    counties.geom
  )
INNER JOIN zip5
  ON INTERSECTS(
    Intersection(
      Intersection(
        states.geom,
        query
      ),
      counties.geom
    ),
    zip5.geom
  )
INNER JOIN lulc
  ON INTERSECTS(
    Intersection(
      Intersection(
        Intersection(
          states.geom,
          query
        ),
        counties.geom
      ),
      zip5.geom
    ),
    lulc.geom
  )
WHERE
  INTERSECTS(states.geom, query)
GROUP BY ROLLUP(states.id, counties.id, zip5.id);

```

Figure 5.7: SQL representation of the spatial OLAP query.

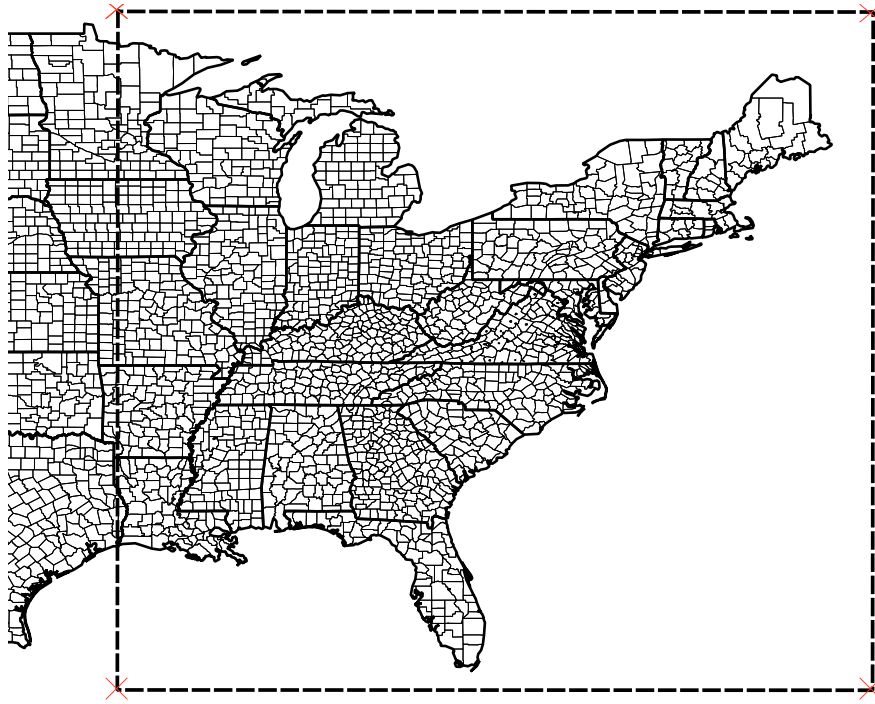


Figure 5.8: Representation of the query region used in the spatial OLAP query.

that is involved in the evaluation of spatial queries. Operations on spatial objects, such as `INTERSECTION` and `INTERSECTS`, require a significant amount of computation, which in turn results in a better utilization of processors. However, we can also observe that the processor efficiency slightly decreases as the number of processors increases. Although, the query is bound by computation for a smaller number of processors, but for a larger number of processors, limitations in memory and/or disk bandwidth start to manifest themselves. This is not surprising, as the increase of the number of processors also increases the overall throughput of computations and, thus, the amount of data that flows through the calculations. This data flow is limited by the bandwidths of the memory and disk subsystems. As the number of processors increases even further, multiple concurrent processes share access to the memory and disk subsystems, which can execute some operations only sequentially. This causes processors to enter idle states while they wait for memory or disk operations to be completed. We can see that the processor efficiency steadily decreased to approximately 85% for 8 processors. One method to address this degradation of processor efficiency is to employ different subsystems for memory and disk I/O. Unless

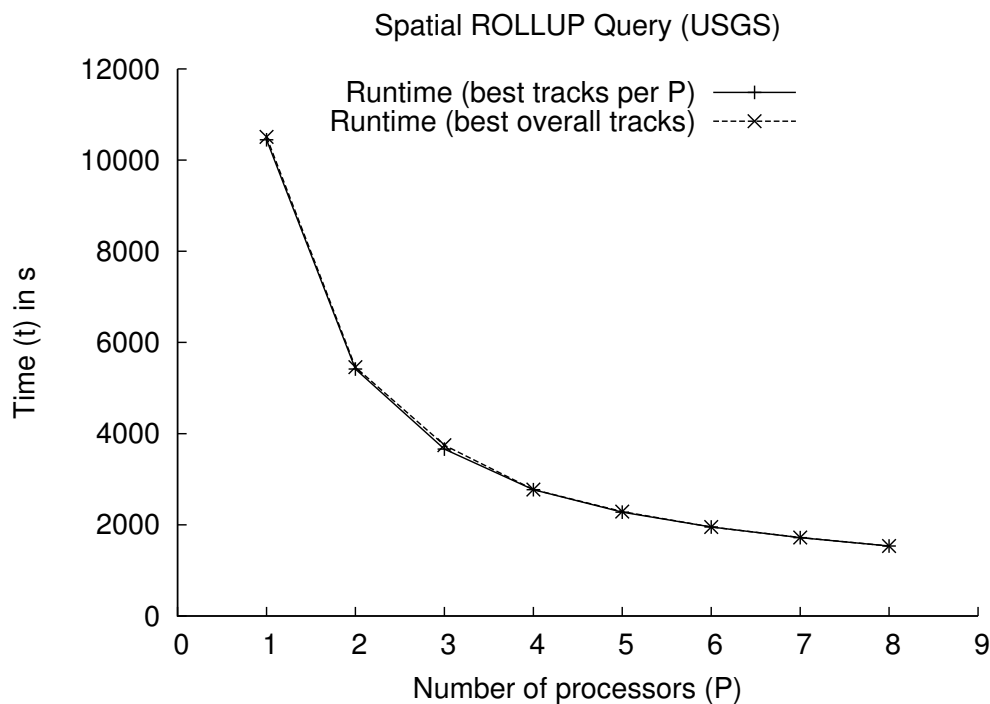


Figure 5.9: Query running time by number of processors.

the bandwidth of these systems can scale proportionally to the number of processors, we expect the efficiency to always degrade for an increasing number of processors.

Figure 5.11 shows how the number of parallel tracks affected the query evaluation time when 8 processing cores were available. This graph shows that the evaluation time was significantly reduced by using more than one track, and the shortest evaluation time was achieved with 11 or more tracks. Unlike the graph for traditional OLAP queries, the evaluation time did not appear to deteriorate for a larger number of tracks and remained relatively constant up to 30 tracks. This can be explained with the higher computational requirements of this query compared to that for traditional OLAP; the overhead for managing the additional processes is not as high relative to the actual computation done by the query. With a further increase of the number of tracks, we expect the management overhead of a large number of processes to eventually dominate the time spent on actual computation and, thus, increase the query evaluation time.

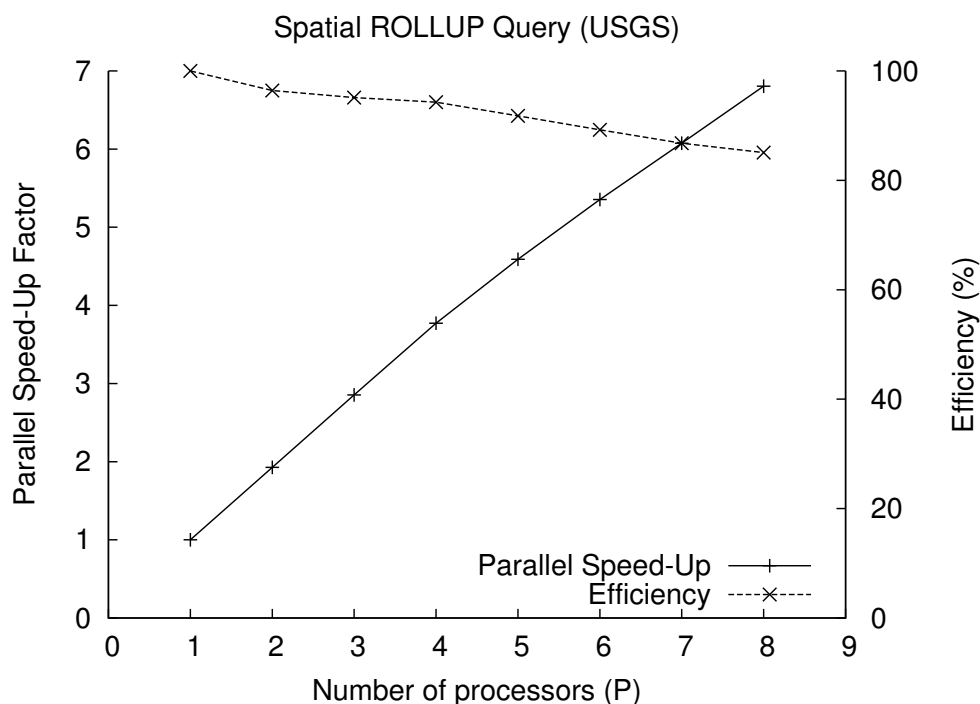


Figure 5.10: Query speed-up factor and efficiency for multiple processors with respect to single processor runs.

### Data Size

For the evaluation of the influence the dataset size has on the running time of a spatial OLAP query, a fixed processor configuration was chosen and the dataset size was varied using the scaling factor described earlier. For the evaluation, we chose input datasets with scaling factors between  $SF = 0.1$  and  $SF = 1.0$ . As the metric to measure the runtime of the query we chose the “wall time.”

The graph in Figure 5.12 shows the influence of the dataset size on the query time for two different processor configurations:  $P = 1$  and  $P = 8$ . As expected, we can observe that the query time increased as the input data grew in size. However, the progression of the graph is not consistent, and the increase in running time from one dataset size to the next appears to be arbitrary. This behaviour is due to the nature of the spatial objects at the `lulc` layer. The spatial objects at this layer represent the land cover or land usage over a specific region, and the objects vary greatly in geographic size and complexity. Especially the complexity, i.e., the number of vertices and edges describing the polygons, of the spatial objects has a significant



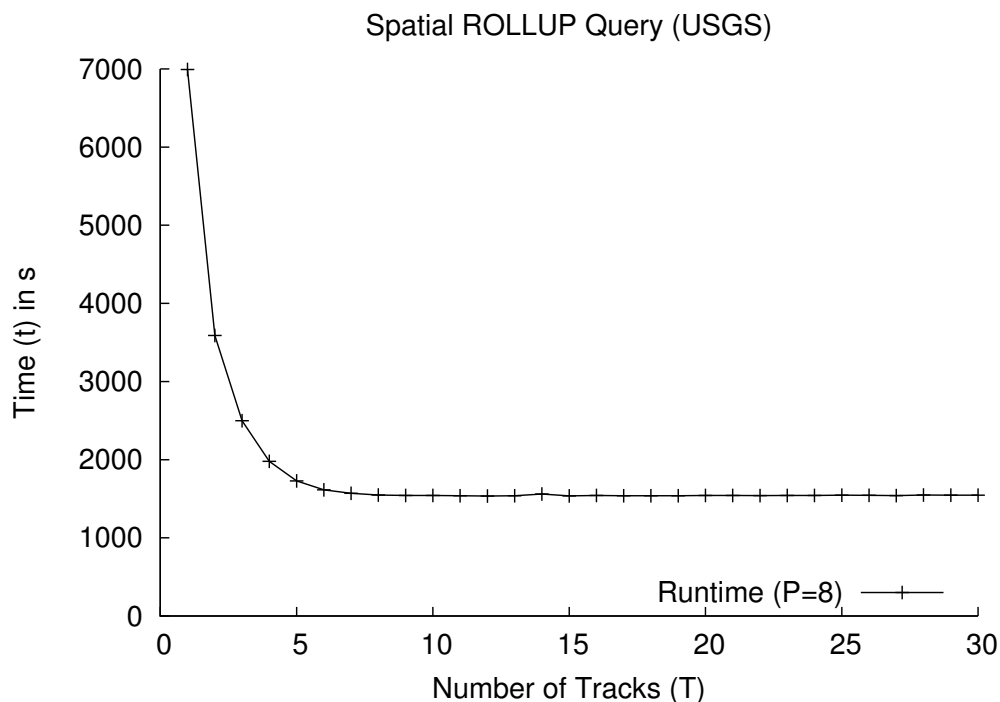


Figure 5.11: Impact of track configuration on query evaluation time.

impact on the running time of individual spatial operations, such as computing the intersection. The random selection of objects to generate smaller datasets only takes into account the number of objects selected but not their size or complexity. The graph in Figure 5.13 shows the distribution of `lulc` objects over the spectrum of complexity in terms of vertices. We can see that the majority of objects at this layer have well below 100 vertices with a median of  $\mu_{1/2} = 25$ . However, there are a number of objects that have more than 10,000 vertices with up to 93,592 vertices. This skew in the distribution of complexity among the spatial objects can cause a few very complex objects to be selected during the dataset scaling process, and those few objects may have a significant influence on the amount of computation that is required during query evaluation. Thus, for some subsets of the data, although they contain a smaller number of objects, the query running time might be much higher than for other similarly sized or even larger subsets.

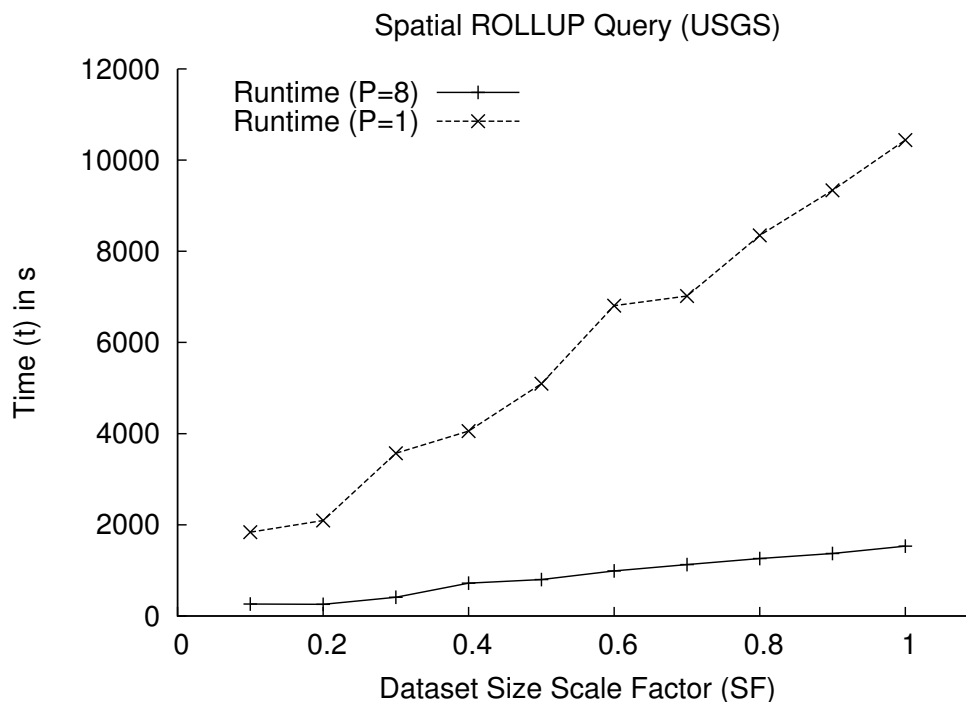


Figure 5.12: Impact of dataset size on query evaluation time.

### 5.2.3 Comparison of LISA with other Systems

For the comparison of LISA with other existing systems, we chose the PostgreSQL relational database system [146]. PostgreSQL is a free open-source relational database system with a large feature set that is comparable to that of established commercial products such as Oracle, Microsoft SQL Server, or IBM's DB2. PostgreSQL also provides extensive support for storing, processing and querying of spatial data through its PostGIS extension [151]. Hence, it often is also chosen as a platform for other spatial data warehouse or spatial OLAP systems [49, 53]. The PostGIS extension for PostgreSQL uses, just like LISA, the GEOS library for geometry calculations and the internal representation of spatial objects. There exist other freely available database or OLAP systems with some support for spatial data, such as MySQL or MonetDB. Although they support the storage of spatial data, their support for functions and operators on this data is very limited, and we were not able to successfully execute our example queries using these systems. PostgreSQL has no native support for typical OLAP operations and does not provide a specific storage model that facilitates the execution of those. Using PostgreSQL's internal procedural language PL/pgSQL, we

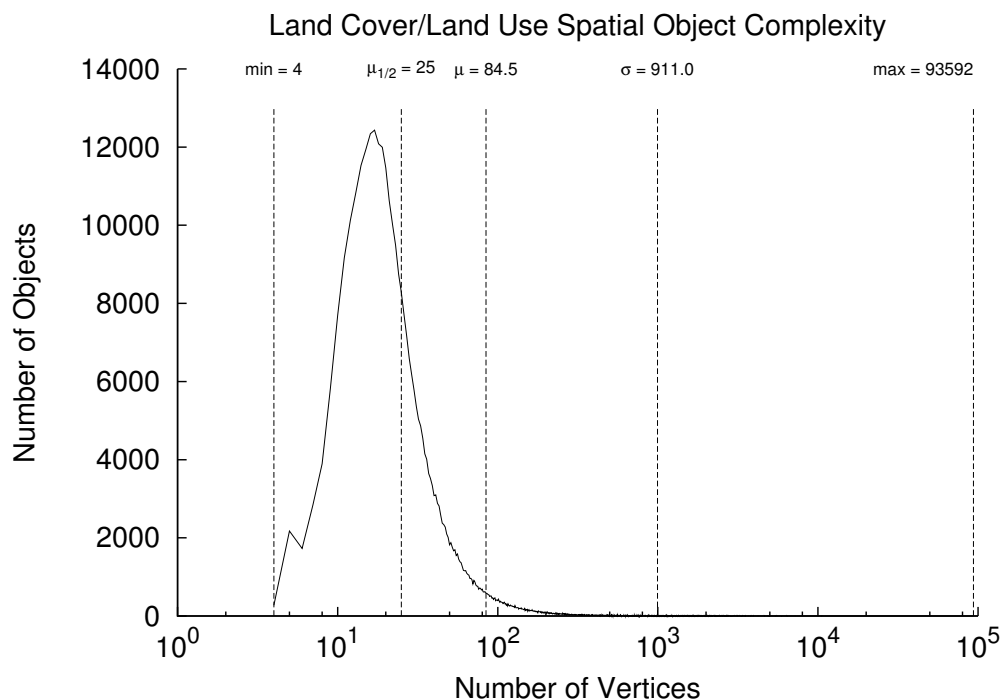


Figure 5.13: Distribution of complexity (i.e. number of vertices) among lulc layer objects.

were able to emulate the functionality of roll-up queries. These PL/pgSQL implementations, shown in Appendix B.6 and B.7, are hand-optimized and significantly more efficient than their standard SQL counterparts. The processing of a single query within PostgreSQL is inherently not parallel. However, PostgreSQL makes use of separate I/O processes and extensive caching.

The datasets used to execute our example queries with PostgreSQL were identical to those used for the evaluation of LISA. In addition, we created appropriate indices on the corresponding tables in PostgreSQL to facilitate the execution of queries, and used suitable configuration values for PostgreSQL’s system and cache configuration. The hardware platform of the PostgreSQL server was identical to that of the compute nodes used for the evaluation of LISA.

The performance of PostgreSQL’s query evaluation was measured in “wall time”, including the time for any network transfers between the PostgreSQL client and the server. The amount of data exchanged between client and server over the local 1Gbit/s Ethernet network was minimal and had no significant impact on the overall query

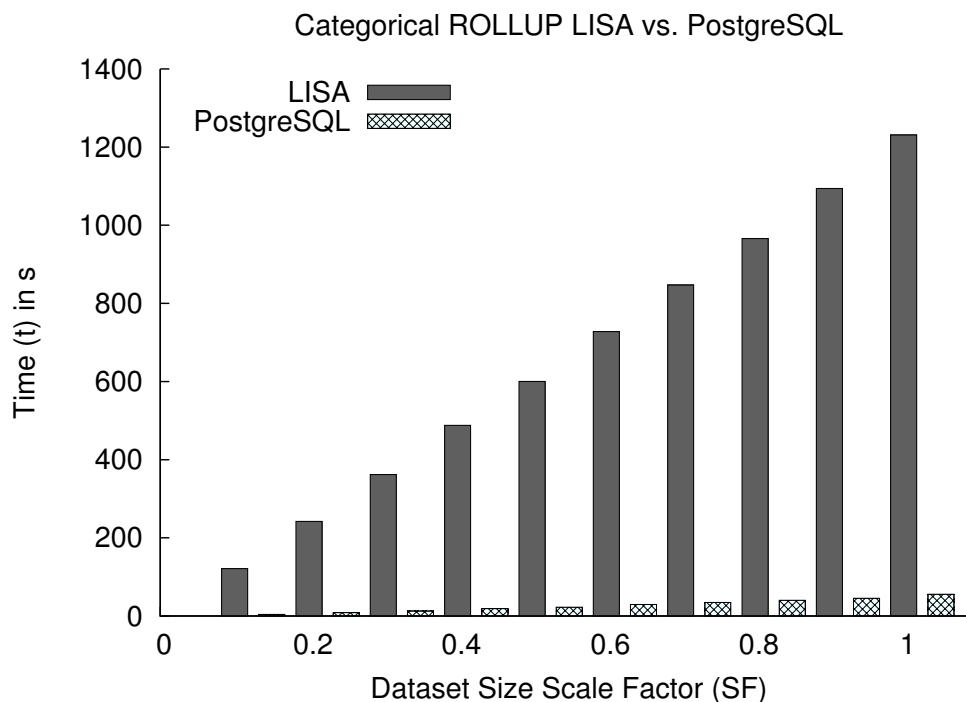


Figure 5.14: Comparison of LISA’s and PostgreSQL’s performance on traditional OLAP queries for various dataset sizes.

running time.

### Traditional OLAP queries

The traditional OLAP query evaluated for LISA in Section 5.2.1 was translated into a PostgreSQL PL/pgSQL function as shown in Appendix B.6.

The graph in Figure 5.14 compares the query running time of LISA and PostgreSQL for various data sizes. We can observe that PostgreSQL outperformed LISA by a factor of 22 for the full-sized dataset. This observation is not surprising, as PostgreSQL is highly optimized to deal with categorical data types and the operations involved in this query. In addition, we discussed in Section 5.2.1 that LISA’s performance on this query was limited by a high fixed cost for disk I/O, which can be attributed to the storage models and access methods that LISA uses to retrieve the data. PostgreSQL, on the other hand, employs a number of mechanisms, such as separate disk I/O processes, shared page caches, and pre-fetching, which allow it to handle this type of data and the operations executed on very efficiently. For LISA to

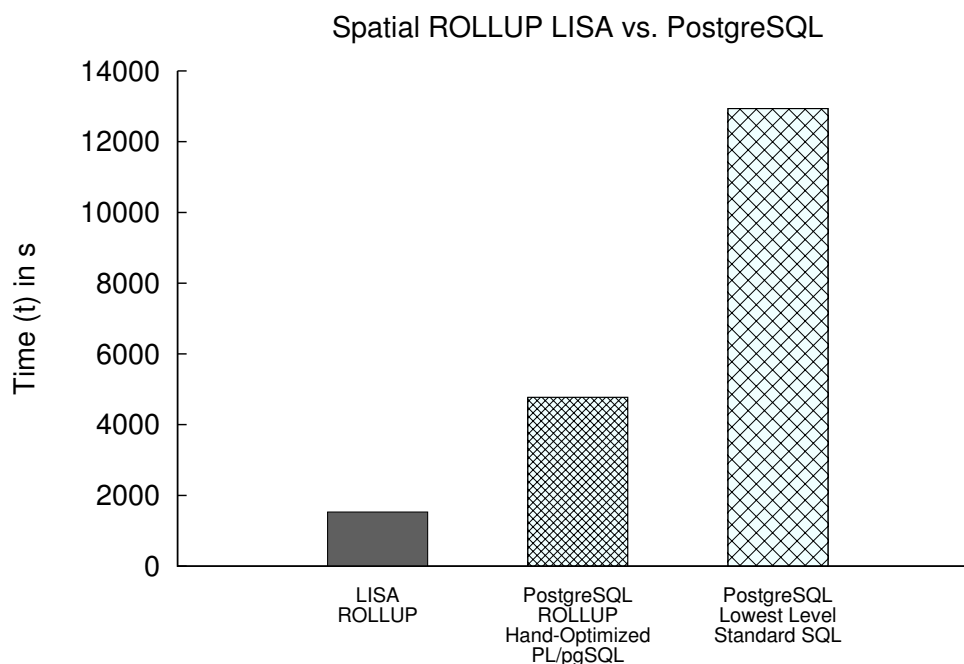


Figure 5.15: Comparison of LISA’s and PostgreSQL’s performance on spatial OLAP queries.

achieve a similar performance, a reengineering of its storage model and access methods for categorical data would be required. Also, much of LISA’s implementation was in Python, which does not perform as efficiently as PostgreSQL’s C implementation.

### Spatial OLAP queries

For the comparison of LISA and PostgreSQL with respect to spatial OLAP queries, we implemented the query used in Section 5.2.2 in PostgreSQL/PostGIS. Similar to the query used in the previous section, we employed hand-optimized PL/pgSQL to implement the query in PostgreSQL. The PL/pgSQL function implementing the query is shown in Appendix B.7. To illustrate the impact of the hand-optimized PL/pgSQL code, we also implemented the query that calculates the values for the lowest roll-up level (bottom level) in standard SQL without the help of PL/pgSQL (see Appendix B.8).

The graph in Figure 5.15 shows the running time of the spatial roll-up query

against the full-size dataset for LISA and PostgreSQL. In addition, it shows the running time of the standard SQL query that can be used to compute to lowest roll-up level. We can observe that the increased requirement in computation involved in this query can greatly benefit from LISA's architecture and utilize processor resources more effectively. This results in a factor of 3.1 speed-up for LISA compared to PostgreSQL's hand-optimized PL/pgSQL implementation.

### 5.3 Summary

In this chapter we introduced LISA, which realizes the pipeline-based query evaluation model for spatial OLAP queries discussed in Chapter 4. We showed that LISA is rich in features and allows the representation and execution of both OLAP and spatial OLAP queries on a full range of data and hierarchy types. Through our experimental evaluation, we also validated the main objective of the pipeline-based evaluation model: the better utilization of modern multi-processor and multi-core hardware platforms. LISA outperformed PostgreSQL on complex spatial queries, even though LISA is a largely unoptimized prototype rather than a production-quality database system. Many optimization techniques could be applied within the LISA framework. These are described in some detail in Appendix A.

## Chapter 6

### The geoCUBE Index<sup>1</sup>

A fundamental difference between a traditional OLAP system and a SOLAP system is that in OLAP all feature dimensions are categorical, i.e., have fixed cardinalities, such as  $\text{Color} \in \{\text{red, blue, green}\}$ , while in SOLAP feature dimensions may be *either* categorical or spatial. Spatial dimensions are typically geometric objects represented by either single points or ordered sets of points in continuous space, for example  $\text{latitude} = 44.854444^\circ$ ,  $\text{longitude} = -63.19916713^\circ$ . The categorical nature of feature dimensions in OLAP is typically exploited in several important ways. Firstly, categorical data stored in a star schema (see Figure 6.1) can be normalized in each dimension, which leads to the imposition of a fixed-cardinality grid on the categorical dimensions. Secondly, dimension hierarchies can easily be defined as ranges of values within these grids to support roll-up and drill-down operations. For continuous dimensions, a different approach is necessary.

In adding non-categorical dimensions to OLAP, two key challenges must be addressed: (1) how to define and perform aggregation on spatial/continuous values, and (2) how to represent, index, update, and efficiently query mixed categorical and continuous data. In this chapter we focus on these challenges for spatial objects without extent. Techniques for the aggregation of continuous spatial and temporal values have been well studied. For a broad survey of these results, see [112]. The representation, indexing, and querying of spatial data has also been well studied [73, 152]. Data in a spatial data warehouse is not exclusively spatial: it is a mix of spatial and categorical attributes. The representation and indexing of such mixed categorical and continuous data in an OLAP setting has received less attention (see Section 6.1 for a discussion of related work). While efficient multi-dimensional indexing mechanisms for continuous data exist [76], they are not specifically designed and optimized to meet the requirements of typical OLAP applications, such as aggregate queries and

---

<sup>1</sup>A preliminary version of this work appeared in JDIM 2007 [13].

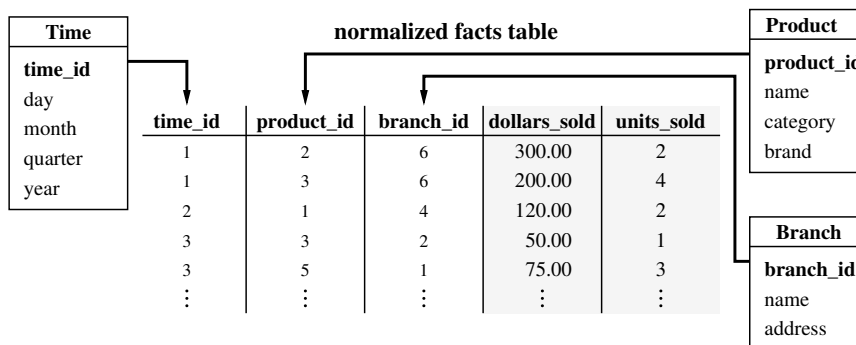


Figure 6.1: Star schema of a typical data warehouse with facts table, three feature dimensions and two measures. The feature dimensions have been normalized within the facts table in that they are represented as keys into dimension tables.

hierarchy operations.

An obvious approach to the representation and indexing of mixed categorical and continuous data is to treat the continuous values as if they were values of a categorical space by simply enumerating them. This discretizes the original continuous dimension, allowing the use of established mechanisms for view indexing and query evaluation. It also skews the original continuous dimension in a data-dependent manner, and once continuous dimensions have been transformed in this way, view updates that preserve the relative order of records with respect to their original space are not possible without recomputation of the complete view.

In this chapter we address the representation and indexing of mixed categorical and continuous data in a Relational OLAP (ROLAP) setting. We first describe a general approach that has been shown to be effective for representing and indexing relational tables in a variety of settings (e.g., sequential [96, 127], parallel [45], and P2P [158]) and then show how this method, based on the use of space-filling curves, can be extended to the mixed categorical/continuous setting. Our proposed method is specifically intended for the indexing of views in a ROLAP setting as the resulting data structures only represent those sections of the multidimensional view that have data records associated with them.

We implemented the proposed storage and indexing methods and evaluated their build, update, and query times using both synthetic and real datasets. Our experiments show that the proposed methods based on Hilbert curves of dynamic resolution



offers significant performance advantages. In the build phase we observed a speed improvement by a factor of approximately 20-25% over the standard pre-discretization approach. For updates, which can be performed without reordering the entire dataset when using the dynamic-resolution approach, we observed performance improvements of between a factor of 23 for an update batch whose size is 2% of the current view and a factor of 6 for an update batch whose size is 20% of the current view. As updates to data warehouses are commonly very small (less than 1 or 2%), the performance benefits of our dynamic adaptation approach over pre-discretization is quite significant.

We also compared the query performance of our Hilbert-curve-based index with a standard R-tree from the Spatial Index Library [77]. Again we observed a significant speed improvement. While the query time on the Hilbert-curve-based index increased only marginally with an increasing number of records and query results were reported in less than 0.2 seconds even for 3M records, the query time of the R-tree index quickly deteriorated for larger numbers of records. Note that some caution is required when interpreting this result. Since we are comparing two independent codebases, it is harder to determine how much of this improvement should be attributed to the improved I/O and cache efficiency of our algorithms and how much is simply due to better coding practices.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview of work related to this chapter. Section 6.2 introduces our basic approach, while Section 6.3 describes our method for dynamically adapting the resolution of the Hilbert curve and proposes new algorithms for indexing views with both categorical and continuous space dimensions. The construction of the index and its representation in memory is described in Section 6.4. In Section 6.4.2 we discuss updates to such views and introduce a merge algorithm that is flexible with respect to dimension cardinality. In Section 6.5 we describe an initial implementation of our algorithms, whose performance we analyze in Section 6.5.1. The chapter is concluded in Section 6.6 with a summary of our findings.

## 6.1 Related Work

The problem of indexing multi-dimensional points is not new. A common and often effective approach to this problem is to employ a multi-dimensional R-tree or one of its many variants. The R-tree [76] is a spatial indexing structure that allows the indexing and efficient querying of points and polygonal shapes. The queries that are supported by the R-tree are rectilinear range queries. Its original design was motivated by the B-tree, and it can be efficiently used as an external-memory data structure. For the application in spatial and spatio-temporal OLAP, a number of extensions [136,137,172] of the R-tree have been proposed to facilitate efficient evaluation of queries typical for these domains. Most approaches based on R-trees, however, focus only on continuous space dimensions and do not specifically address the properties of categorical dimensions. To support categorical dimensions, these dimensions are typically transformed into an equivalent representation in continuous space, but this creates problems: The efficient aggregation in OLAP queries over categorical dimensions often crucially relies on these dimensions being integer-valued, and the use of space-filling curves as a locality-preserving mapping from higher-dimensional space into one dimension, discussed next, relies on the existence of at least an implicit integer grid.

Space-filling curves have been used to support the indexing of purely categorical OLAP data [45]. This approach organizes multi-dimensional categorical OLAP views by using the Hilbert space-filling curve to generate a linear ordering of the records in multi-dimensional space and then indexing this linear ordering with a data structure similar to a B-tree. This method exploits that the Hilbert curve strongly preserves spatial locality [127].

Previously, Hilbert curves were used by Kamel and Faloutsos [98] as a mechanism to enhance the performance of R-trees when used for indexing multi-dimensional data. They used the Hilbert curve to obtain an ordering of the records within each node of the R-tree. This allows the exercise of new strategies for the distribution of records when nodes split and merge during insertion and deletion of records. Using these strategies, the query performance of the R-tree does not deteriorate as drastically as when arbitrarily distributing records between nodes during node splitting and merging.

Lawder and King proposed a multi-dimensional index that uses a tree structure derived from the construction rules of the Hilbert curve [106]. Each level of this tree corresponds to a level of resolution, or order, of the Hilbert curve and partitions the covered subspace into quadrants. At the root of the tree the entire space is covered. Records are stored at the leaf level of the tree in the subspace quadrants that correspond to the Hilbert values derived from their original coordinates.

None of the previous approaches to indexing multi-dimensional data with the help of Hilbert curves addressed the issue of continuous dimensions. All proposed techniques operate on multi-dimensional grids whose resolutions are known in advance, and records are mapped into cells of these grids. Hence, they require that the cardinality of each dimension is fixed and known in advance. This, however, is not practical in a spatial OLAP environment where multiple continuous dimensions may exist and their attribute values are not fixed and may change over any number of updates.

A more application-oriented approach to the integration of spatial and categorical data based on building composite systems that integrate existing OLAP and GIS systems has been pursued by both academic [56,69,83,153,165] and industrial [40,49,87,100,161] research groups. Systems such as SOVAT [161] or Kheops Technology's JMap [100,154] address the integration at a higher level, closer to the end-user, and internal data representation and storage mechanisms manage categorical and spatial data independently by means of traditional categorical data warehouses and spatial databases. While this approach speeds software development, it does not lead to systems that scale well in the face of massive datasets.

## 6.2 Our Approach

The central problem we address in this chapter is the representation and indexing of mixed categorical and continuous data in a Relational OLAP setting. We first describe a general approach that has been shown to be effective for representing and indexing relational tables in a variety of settings (e.g., sequential [96], parallel [45], and P2P [158]) and then show how this method, based on the use of space-filling curves, can be extended to the mixed categorical/continuous setting.

The approach has three basic steps.

1. Map the multi-dimensional data into a linear ordering using a locality-preserving

space-filling curve, such as a Hilbert curve [88].

2. Use the linear order to distribute the data over the available storage. In the sequential setting this may be one or more disks [96], in the parallel setting this may be disks across a set of processors [45], and in the P2P setting it may be disks across a collection of peers [158].
3. Build an index structure on top of the ordered data for efficient query processing. Typically, this is some variant of the B-tree [14] with query properties similar to those of the R-tree [76].

The resulting indexing structure is basically an R-tree; however, the use of space-filling curves to fold the multi-dimensional space into a one-dimensional space is particularly well-suited for the even distribution of records over multiple disks, processors, or peers, and it facilitates batch updates of views by merging record updates into the original view with a single linear scan.

In an OLAP setting, this indexing structure presents a number of advantages:

- The disk layout favours sequential block access and thus reduces the amount of disk I/O and seeks.
- Multi-dimensional data is indexed in such a way that no dimension is favoured over another and the disk layout preserves the locality of the data.
- Extensions to the indexing structure can provide additional support for OLAP-specific operations, such as range aggregate or roll-up queries [136].

In this chapter, we use the Hilbert curve as the space-filling curve that underlies our method. The Hilbert curve is defined on a  $d$ -dimensional grid with a side length of  $2^k$ , where  $d$  denotes the number of dimensions of the view and  $k$  is the *resolution* or *order* of the Hilbert curve. This grid covers the entire space of the view and the resolution  $k$  must be chosen in such a way that each grid cell contains at most one record (see Figure 6.2a). Given a view with  $d$  categorical dimensions and each dimension  $D_i$  containing  $|D_i|$  distinct values (dimension cardinality), then a grid with a resolution of  $k = \lceil \log_2 \max_i^d |D_i| \rceil$  ensures each record is located in a distinct grid cell.

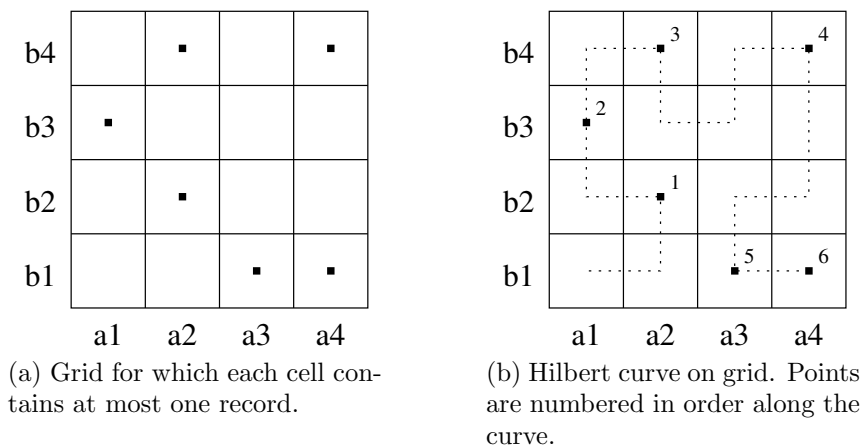


Figure 6.2: Hilbert curve.

The Hilbert curve on this grid passes through each record in the view exactly once and thereby generates the Hilbert order of the records (Figure 6.2b). The important property of the Hilbert curve is that it provides a mapping from  $d$ -dimensional space to 1-dimensional space that preserves much of the locality of the higher-dimensional space [127].

In the following section we show how to index continuous space dimensions without pre-discretization by proposing a technique that dynamically adapts the resolution of the Hilbert curve and determines the order of records locally. Unlike previous approaches that exploit the clustering properties of the Hilbert curve [45, 96, 98, 127], this new technique is flexible with respect to the resolution of the grid, as it attempts to optimally utilize the grid space for continuous dimensions; it achieves a significant reduction of the grid resolution required to map records with continuous dimensions into distinct grid cells when compared to a pre-discretization approach; and it also allows for the introduction of new records with previously unknown attribute values without the need to recompute the entire view. Introducing a batch of new records into an existing view has a cost proportional to that of sorting the records to be inserted in Hilbert order and linearly scanning the existing view to merge the new records into it.

### 6.3 Dynamic Resolution of Hilbert Curves

In this section we describe a method for ordering multi-dimensional records in Hilbert order when some of the dimensions are continuous. The main strength of this approach is that it is dynamic, allowing on-the-fly calculation of the resolution of the Hilbert space for continuous dimensions.

Generating a Hilbert ordering requires a discrete space in which each record is mapped to an associated coordinate and distinct records map to distinct coordinates. The naïve approach to generating a Hilbert ordering over a space that contains continuous dimensions would discretize those dimensions in advance and then generate the Hilbert ordering of the resulting entirely discrete space. This approach, however, exhibits a number of disadvantages that impact its effectiveness:

- The discretization of the continuous space requires multiple expensive sort and scan operations on the original dataset.
- Discretizing each continuous dimension individually causes the resulting space to be skewed, as only the relative order of records, but not the magnitude of their attributes, is preserved.
- The space resulting from the discretization is sparse and not well utilized, as each continuous value is assigned a distinct index, independent of whether or not this is in fact necessary to place each multi-dimensional record into its own grid cell.
- The entire space needs to be discretized when an update introduces a new attribute value.

A second approach is to map the continuous space attributes onto a multi-dimensional grid that defines the discrete space. Predetermining the resolution of this grid requires the computation of pairwise Euclidean distances between records, which is not practical when the number of records is large.

The method proposed here takes a different approach. Based on the recursive definition and the self-similarity of the Hilbert curve, the Hilbert ordering of records can be efficiently computed in continuous space by using an adaptive resolution for

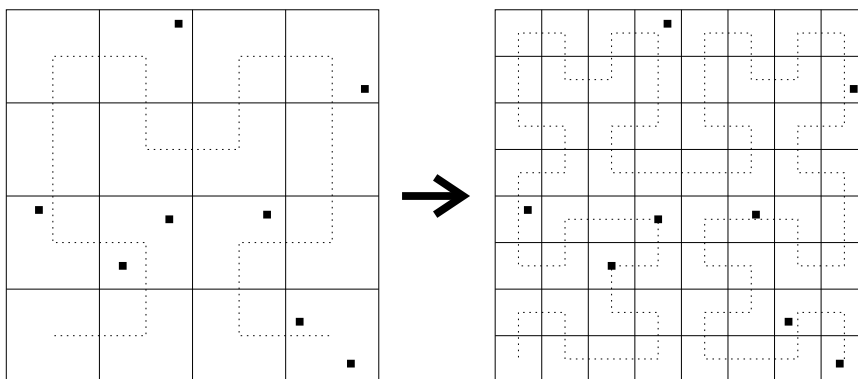


Figure 6.3: Increasing the resolution of the Hilbert curve preserves the order of records.

the underlying discretization grid. We observe that a Hilbert ordering of records has the property that the relative order of records is preserved when the resolution of the Hilbert curve is increased (see Figure 6.3). Increasing the resolution of the Hilbert curve, in turn, increases the resolution of the multi-dimensional grid, on which the curve is defined. This property can be used to dynamically adjust the resolution of the grid onto which the continuous space attributes are mapped during the generation of the Hilbert ordering of the records, in order to guarantee that distinct records map to distinct grid cells and thus become comparable in Hilbert order.

The idea behind this approach is to resolve any conflict where two records that are compared map to the same grid cell while sorting the records in Hilbert order. This is done by increasing the resolution of the underlying grid until both records map to distinct grid cells. Due to the self-similarity of the Hilbert curve, changing the resolution of the Hilbert curve does not have an impact on records that have been sorted already. Thus, the resolution can be increased dynamically during the comparison of pairs of records. Furthermore, this technique achieves a better space utilization than a pre-discretization approach because a grid cell is split into smaller cells only if there is in fact a conflict between two records in that cell. Finally, since the grid is dynamically adjusted to the actual multi-dimensional data, the space is not skewed and the relative differences between the continuous values of records are preserved. Figure 6.3 illustrates how the increase of resolution resolves conflicts between records that are mapped to the same grid cell.

Algorithm 6.1 outlines a method to produce a Hilbert ordering of records that

are defined in both continuous and discrete space. It uses a standard comparison-based sorting algorithm (e.g., Quicksort or Merge Sort) with a comparison function (Algorithm 6.2) that determines the relative order of two records on a Hilbert curve at a particular resolution. The relative order is determined by finding the minimum resolution of the Hilbert curve such that both records have a distinct rank with respect to the curve.

---

**Algorithm 6.1** Algorithm to sort records in Hilbert order using dynamic resolution adaptation.

---

**Procedure:** hilbert-sort

**Input:** set  $R$  of records in no particular order,

set of categorical dimensions  $D$ , with  $|D_i|$  being the number of distinct values in dimension  $D_i$

**Output:** set  $R$  of records in Hilbert order

▷ compute initial resolution

1:  $k \leftarrow \lceil \log_2 \max\{|D_i| : \forall D_i \in D\} \rceil$

▷ call sorting algorithm with **hilbert-compare** as comparison function

2: **sort**( $R$ , **hilbert-compare** <sub>$k$</sub> )

---



---

**Algorithm 6.2** Algorithm to determine the order of two records with respect to the Hilbert curve by dynamically adapting the resolution of the Hilbert curve if necessary.

---

**Procedure:** hilbert-compare

**Input:** pair of records  $(r_1, r_2)$ , initial resolution  $k$

**Output:**  $-1$  if  $\text{rank}_k(r_1) < \text{rank}_k(r_2)$ ,  $0$  if  $r_1 = r_2$  or  $1$  if  $\text{rank}_k(r_1) > \text{rank}_k(r_2)$

1: **if**  $r_1 = r_2$  **then**

2:     **return**  $0$

3: **else**

4:     **while**  $\text{rank}_k(r_1) = \text{rank}_k(r_2)$  **do**

5:          $k \leftarrow k + 1$  ▷ determine resolution suitable for comparison

6:     **end while**

7:     **if**  $\text{rank}_k(r_1) < \text{rank}_k(r_2)$  **then**

8:         **return**  $-1$

9:     **else**

10:         **return**  $1$

11:     **end if**

12: **end if**

---

The resolution that is found during a comparison is used as the initial resolution for subsequent comparisons and may be increased. That way, the resolution of the grid is either maintained or increased for each comparison. After the sorting process,



all records are in Hilbert order, and the resolution determined by the last comparison during the sorting process is the minimal resolution for a grid such that every record of the dataset maps to a distinct cell in that grid.

## 6.4 Exploiting Hilbert Order for I/O-Efficient Indexing

Once the Hilbert ordering of the records has been determined, the records are sequentially written to disk in a block-wise manner. Each block on disk stores a constant number,  $B$ , of records that are consecutive in Hilbert order. Then an indexing structure is built on top of the ordered records that provides features comparable to those of a combination of a B-tree and an R-tree [45]. Figure 6.4 illustrates the construction of such a tree structure with a specific example. While each intermediate node of the tree is very similar to a node in a conventional B-tree, it is also annotated with the minimum bounding box of the records in its subtree, similar to nodes in an R-tree. This allows for an efficient evaluation of multi-dimensional range queries while exhibiting most of the I/O-efficient properties of B-trees. Due to the self-similarity of the Hilbert curve and its property to not favour any specific dimensions, the bounding boxes of the records in the subtrees at each level of the indexing structure are usually fat and usually overlap only very little, if at all. This is crucial for increasing the performance of queries. Figure 6.4 illustrates the evaluation of a two-dimensional range query on this indexing structure. Note that to answer a query, the tree is traversed in a breadth-first manner, thus limiting the disk accesses to a combination of sequential reads and forward-seek operations and reducing the amount of random access [50].

### 6.4.1 Answering Range Aggregate Queries

Range aggregate queries are a very common query type in OLAP applications. In particular, when spatial information is processed, range-aggregate queries over millions of points may be typical. The indexing structure described above can be used to answer range aggregate queries. Without extensions it requires retrieval of each individual record within the query range in order to compute a desired aggregate value. To retrieve the required records, the tree is traversed by starting from the root and determining, for each child node of a non-leaf node, if its bounding box intersects with the query region. If so, all or some records in its subtree may contribute to

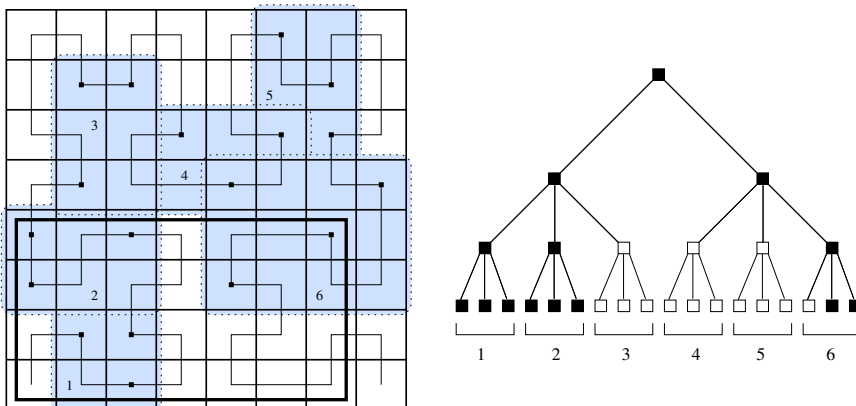


Figure 6.4: Records in Hilbert order are stored in consecutive blocks on disk and form multi-dimensional regions in the original space. The evaluation of a range query in breadth-first fashion only traverses part of the tree and results in a combination of sequential read and forward-seek operations at the leaf level.

the final query result, so the traversal continues in the subtree of this node. This traversal is performed in a breadth-first manner, so that records at the leaves of the tree can be reported by a combination of sequential read and seek-forward operations. The retrieval of all records contained in a range to compute a single aggregate value is, however, computationally expensive and is often overkill because the records only contribute to the aggregate value and are discarded after this computation.

As proposed in [136], a more efficient approach to answering aggregate range queries is the use of pre-aggregated values. The idea is to annotate non-leaf nodes in the tree with aggregate information derived from the points that are stored in the node's subtree. This supports an improved evaluation of aggregate queries, since a full traversal of the tree down to its leaf level may not be necessary to answer an aggregation query. When the bounding box of a non-leaf node is completely contained in the query region, all records in its subtree contribute to the final query result, and we can use the aggregate stored at the node instead of inspecting all records in its subtree. Thus, the answer to a query can potentially be assembled from a small set of partial results instead of inspecting all points in the query region.

To help answer aggregate OLAP queries efficiently, the aggregate information to be stored at each node is pre-computed while iterating over the children of the node during the bottom-up construction of the index tree. Note that this pre-computation

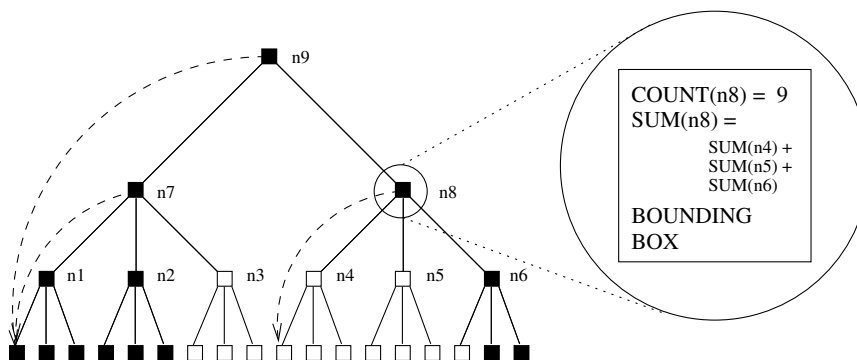


Figure 6.5: Annotation of non-leaf nodes with aggregate information and references to records at the leaf level.

of aggregate information works well for distributive (e.g., COUNT, SUM) and algebraic (e.g., AVERAGE) aggregation functions. Holistic aggregation functions (e.g., MEDIAN), on the other hand, require the retrieval of the actual records enclosed in the query region in order to compute the aggregate value. This can be supported by additionally annotating each non-leaf node with a direct reference to the sequence of leaves in its subtree. When evaluating a query, the records in a node's subtree can be reported immediately, once the node has been reached, without traversing the remaining subtree below this node (see Figure 6.5).

#### 6.4.2 Updating OLAP Views

A key advantage of using Hilbert curves of dynamic resolution over pre-discretization is that an existing, possibly very large view can be efficiently updated in a batched fashion in time proportional to the cost of sorting the records in the update batch and linearly scanning the existing view. To do so, all records of the update view are sorted, using the same comparison function as in Algorithm 6.2. The minimum resolution that has been determined while sorting the original records is used as the initial resolution when sorting the update records. Once the update records are in Hilbert order, they are merged with the original dataset in a single scan over both datasets. As update datasets in an OLAP environment are often only a fraction of the size of the entire data warehouse (typically 1%), the cost of sorting the update view is relatively small compared to rebuilding the entire view.

Algorithm 6.3 shows in detail how both sequences are merged. It iterates through

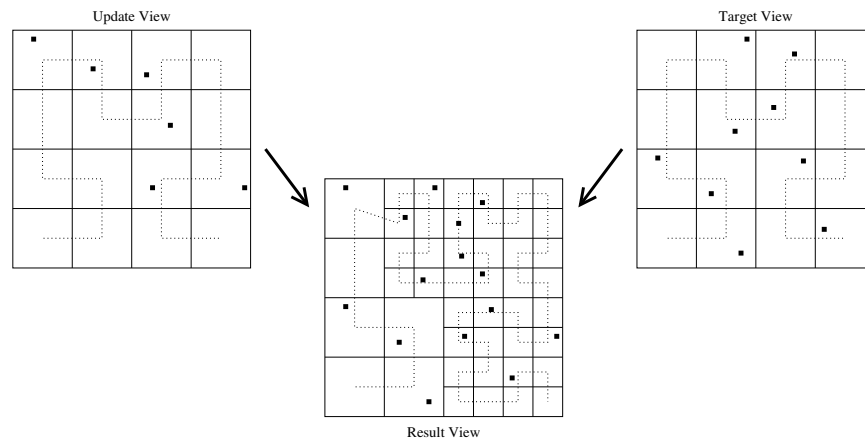


Figure 6.6: Merging the target view with the update view may result in a dynamic adaptation of the Hilbert curve resolution.

the records in both datasets in Hilbert order and repeatedly compares the two current records from both datasets. If both records share the same attribute values, the record from the update dataset is considered an update of the existing record and therefore moved into the output dataset, while the original record is discarded. If the attribute values of both records differ in at least one dimension, the Hilbert ranks of both records at the initial resolution are determined and compared. If one record has a lower Hilbert rank than the other, it is moved to the output dataset and a new current record is fetched from the respective input dataset. In the case when the same Hilbert rank is computed for both records, the resolution has to be increased until the Hilbert ranks differ. Since the locality of records mapped into Hilbert space is preserved when increasing the resolution of the Hilbert curve, records that have been merged already are not affected by the increase of resolution (see Figure 6.6). The merging process continues until all records from both datasets have been processed. The output dataset is a set of records in Hilbert order and will contain all records from the update dataset as well as those from the original dataset that have not been updated. Following the merging process, the indexing data structure is rebuilt over the output dataset.

## 6.5 Implementation and Experiments

Based on the proposed approach, a prototype, called “geoCUBE”, was implemented and extensively evaluated. The focus of the implementation was on determining the

---

**Algorithm 6.3** Merge algorithm to incorporate update dataset into target dataset.

---

**Procedure:** hilbert-merge

**Input:** stream  $U$  of update records in Hilbert order, stream  $T$  of existing target records in Hilbert order, resolution  $k$  of the Hilbert curve used to order the existing records

**Output:** merged stream  $O$  of records in Hilbert order

```

1: let  $u$  be the first element in  $U$  or empty if no such element exists
2: let  $t$  be the first element in  $T$  or empty if no such element exists
3: while  $u$  and  $t$  are not empty do
4:   if  $\text{hilbert-compare}(u, t, k) = 0$  then
5:     write  $u$  to  $O$ 
6:     let  $t$  be the next element in  $T$  or empty if no such element exists
7:     let  $u$  be the next element in  $U$  or empty if no such element exists
8:   else if  $\text{hilbert-compare}(u, t, k) < 0$  then
9:     write  $u$  to  $O$ 
10:    let  $u$  be the next element in  $U$  or empty if no such element exists
11:   else
12:     write  $t$  to  $O$ 
13:     let  $t$  be the next element in  $T$  or empty if no such element exists
14:   end if
15: end while
16: if  $u$  is not empty then
17:   write the remainder of  $U$  to  $O$ 
18: else if  $t$  is not empty then
19:   write the remainder of  $T$  to  $O$ 
20: end if

```

---

Hilbert order for records with attribute dimensions defined in continuous *and* discrete space and on evaluating query performance and the construction and batched update time of the proposed indexing structure. The software was written in C and includes implementations of Algorithms 6.1, 6.2 and 6.3 as well as variations of them to explore performance trade-offs.

Computing a point’s mapping from  $n$ -dimensional discrete space to one-dimensional Hilbert space was implemented based on a modified version of Doug Moore’s Hilbert mapping library [128]. The data types that are supported by the implementation are 64-bit integers and 32-bit floating-point numbers. The implementation presented here assumes that the attribute values of each continuous space dimension are normalized to the interval  $[0.0; 1.0)$  and that the attribute values of each discrete space dimension are natural numbers in the interval  $[0, c - 1]$ , with  $c$  being the cardinality of the dimension.

### 6.5.1 Performance Evaluation

The experimental platform was a workstation with one Intel Xeon 1.8GHz processor and 1.5 GB of RAM, running FreeBSD 6.2. The compiler used to translate the C programs was part of the GNU Compiler Collection version 3.4.6.

The experiments we conducted evaluate the cost of sorting records into Hilbert order, building the index, evaluating queries, and performing view updates. The running times were measured as wall-clock times in seconds. Both synthetic and “real-world” datasets were used in the evaluation. Synthetic datasets were generated with a uniform distribution so that we could better understand the effects of various dataset parameters on performance. The categorical and continuous dimensions of the synthetic datasets were generated with cardinalities of 64 and 1000, respectively. The real-world datasets were drawn from the HYDRO1k dataset published by the U.S. Geological Survey and, where necessary, were reduced in dimensionality and size through uniform random sampling. The main test dataset was a 6-dimensional dataset composed of two categorical dimensions, with 57 and 51 distinct categorical values, and four continuous dimensions with 956, 1000, 802 and 288 distinct continuous values. It was used in each of the following experiments unless explicitly stated otherwise.

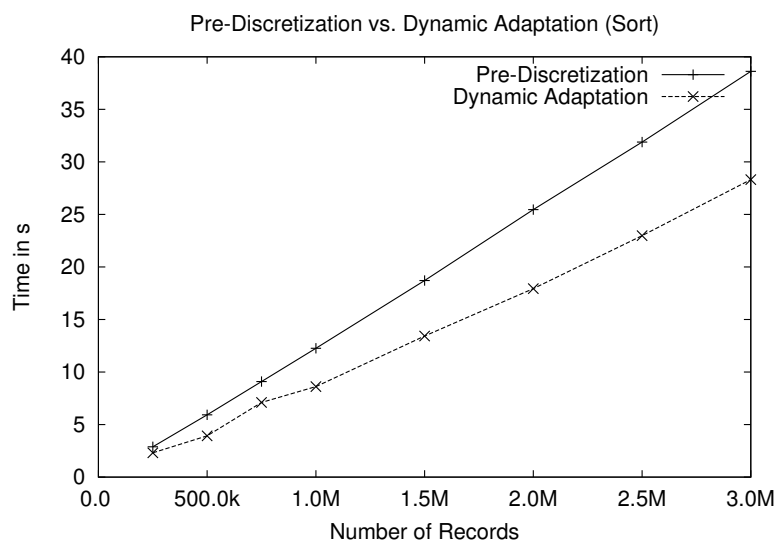


Figure 6.7: Overhead of pre-discretization compared to dynamic resolution adaptation.

Throughout the experiments, the impact of the sizes of the datasets on the running times of the various algorithms was investigated. Additionally the benefit of proposed optimizations was determined, and the traditional method of pre-discretizing continuous space into discrete space [45,96,127] was compared to the new dynamic approach. Unless stated otherwise, all experiments were performed with the most beneficial optimization enabled. In particular, each record was annotated with its last computed Hilbert rank.

For completeness, the evaluation also contains a performance comparison of the geoCUBE prototype to a conventional R-tree implementation from the Spatial Index Library [77].

### Dynamic Adaptation versus Pre-Discretization

Figure 6.7 shows a direct comparison of the dynamic adaptation of the Hilbert curve resolution versus the traditional method of pre-discretization. The top curve represents the time required to pre-discretize and sort the synthetic dataset into Hilbert order. The pre-discretization involves the sorting of each continuous space dimension and the subsequent enumeration of their unique values, resulting in only categorical

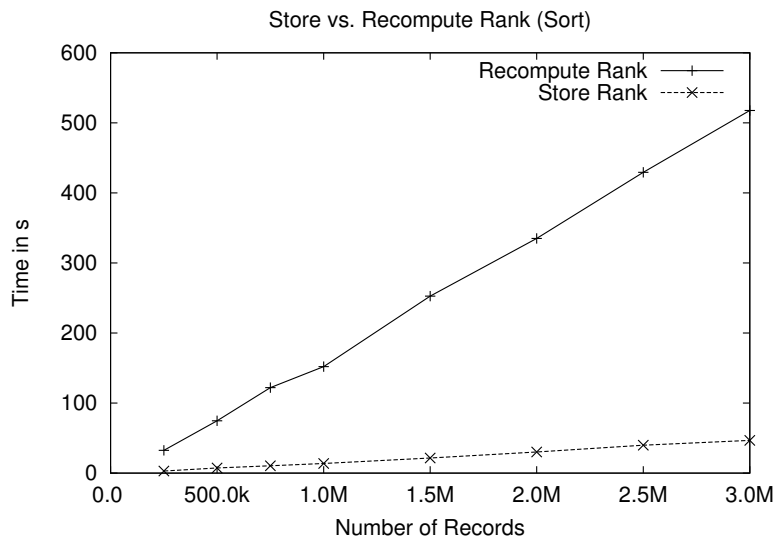


Figure 6.8: Recomputing each record’s Hilbert rank for each comparison versus storing the last computed Hilbert rank of each record.

dimensions. The dataset is then sorted in Hilbert order with a resolution derived from the cardinalities of the dimensions. The bottom curve represents the time required for sorting the same synthetic dataset into Hilbert order using our dynamic resolution adaptation approach. As one can see, dynamic adaptation of the Hilbert curve performs better than the pre-discretization approach. In particular for 3 million records, the pre-discretization approach requires approximately 38 seconds to pre-discretize and sort the dataset, while the dynamic approach takes only 28 seconds. This corresponds to a speed improvement of about 26%, which we also observed for most other datasets we tested. We believe that this better performance is the result of avoiding the expensive process of identifying distinct attribute values.

### Optimization 1: Annotating Records With Their Last Computed Hilbert Ranks

Before computing the Hilbert order of records by using an adaptive increment of the Hilbert curve resolution, the base resolution,  $k_{base}$ , that is required to map all discrete space dimensions into Hilbert space must be computed from their cardinalities. In the following, this base resolution is initialized to  $k_{base} = \lceil \log_2 c_{max} \rceil$  with  $c_{max} = \max |D_i|$



if there exists at least one discrete dimension, or  $k_{base} = 0$  if only continuous space dimensions are defined.

To compare two records based on their Hilbert ranks, the ranks of both records have to be computed at the same resolution. With  $\mathcal{O}(n \log n)$  comparisons to sort  $n$  records, recomputing the Hilbert rank of a record for each comparison it is involved in is computationally very expensive. To limit the amount of recomputation, each record can be annotated with its last computed rank and the resolution for which this rank is valid. Even though the resolution may increase, causing the Hilbert rank of the record to be recomputed, this will affect only records that are not in their final position yet and those for which the rank has not been computed yet. Also, instead of globally increasing the resolution for all subsequent comparisons, the resolution is only increased locally for each comparison if necessary. This avoids the computation of Hilbert ranks at an unnecessarily high resolution for other records. In practice, storing the last computed Hilbert rank reduces the number of rank calculations considerably and improves the overall sort time. The main drawback of storing the computed rank along with each record is a substantial increase in the space requirements. In the best case, the overhead, in bits, of storing the Hilbert rank for  $n$  records at a given resolution  $k$  for  $d$  dimensions is  $n \sum_{i=1}^d \min\{k, \lceil \log_2 |D_i| \rceil\}$ , where  $|D_i|$  is the cardinality of dimension  $D_i$ . However, this is true only when continuous dimensions are pre-discretized. For the dynamic mapping of continuous dimension values and the preservation of their spatial relationships,  $k$  may become significantly larger than  $\lceil \log_2 |D| \rceil$ , as the minimum Euclidean distance between two normalized records is the decimal precision  $p$  of the continuous dimensions (e.g.  $p = 10^{-3}$ ). In this case the overhead can be quantified as  $n \sum_{i=1}^z \min\{k, \lceil \log_2 |Z_i| \rceil\} + n \sum_{i=1}^r k$ , where  $z$  is the number of categorical dimensions,  $r$  the number continuous dimensions and  $k \leq \lceil \log_2 1/p \rceil$ . Thus, in the worst case the overhead in number of bits for  $z$  categorical dimensions and  $r$  continuous dimensions at resolution  $p$  is  $n(z+r) \cdot \max(\max_i^z(\lceil \log_2 |Z_i| \rceil), \lceil \log_2 1/p \rceil)$ . However, in the general case the minimum Euclidean distance between two normalized records is often greater than the decimal precision of the continuous dimensions, such that the worst case is rarely encountered. Also, the overhead for storing the Hilbert rank per record is not related to the number of records in the dataset, but depends on how these records are distributed within the space.

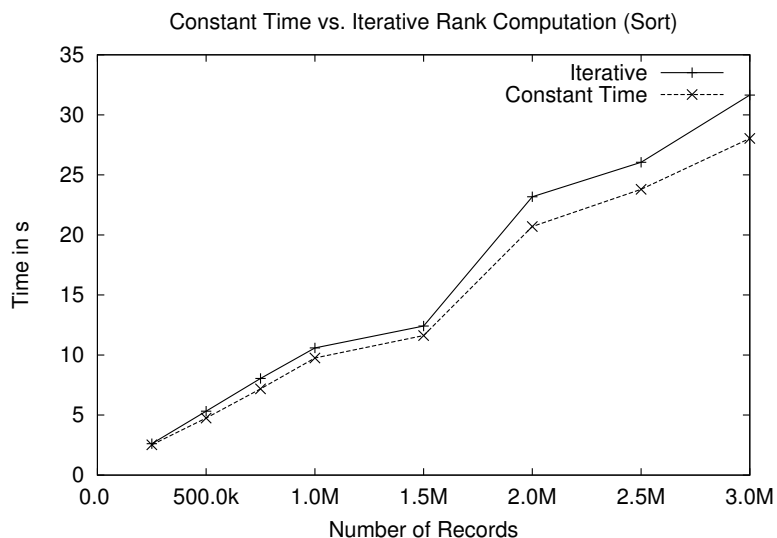


Figure 6.9: Iterative versus constant-time resolution determination.

Figure 6.8 shows the impact of annotating each record with its last computed Hilbert rank compared to recomputing the record’s Hilbert rank during each comparison. The dataset used for this experiment was the HYDRO1k dataset. As can be clearly seen, the optimization has a very significant effect on the overall running time and results in an order of magnitude performance gain. Specifically, for 3 million records, the annotation approach takes 46 seconds, while the recomputation approach takes 517 seconds; this represents a performance improvement by a factor of 11.2.

### Optimization 2: Iterative versus Constant Time Resolution Determination

Another possible bottleneck in the proposed comparison function is the iterative determination of the grid resolution that guarantees that no two records share the same rank. In the worst case, this requires many iterations, and consequently recomputations of Hilbert ranks for the records that are compared, if the two records are sufficiently close to each other in continuous space. It would be desirable to determine the necessary minimal resolution with a constant amount of computation for any two records. The approach presented here takes constant time to determine a

resolution at which both records do not share the same rank. This is achieved by computing the Euclidean distance between both records and determining the resolution at which the diagonal of a grid cell is shorter than the distance between the records. This guarantees that both records are mapped to distinct cells, but the determined resolution may not be minimal, as there may be a lower resolution that already maps both records into different cells. The disadvantage of obtaining a resolution that is not minimal is an additional space requirement and a slight increase in computation required to determine the Hilbert rank at this resolution.

Figure 6.9 illustrates the effects that the method of resolution determination has on the overall running time of the algorithm. For datasets with only continuous space dimensions, the on-the-fly determination of the resolution is essential, since no initial resolution can be computed. The curves presented in Figure 6.9 were obtained using experiments with the HYDRO1k dataset. The top curve represents the performance of the iterative method. Note that this method always finds the minimal resolution necessary to compute distinct Hilbert ranks for all records. The bottom curve shows the performance of the constant-time method. This method performs approximately 10% better than the iterative method. In some cases, however, the determined resolution of the Hilbert curve may not be minimal, causing the computation of each Hilbert rank to become more expensive. Also, as records may be arbitrarily close to each other in the original space, the constant-time approach may determine a resolution that is *significantly* higher than the minimum resolution determined by the iterative approach. Consequently, there is a trade-off between the performance gain of the constant-time approach and the likelihood that it will exceed the available storage to represent Hilbert ranks.

## Update Time

The implementation of the update mechanism follows the description in Algorithm 6.3. First the update dataset is sorted into Hilbert order using, as a starting resolution, the minimum resolution that was determined for the original dataset. The next step then merges both, the original and the update dataset, by sequentially scanning through them and repeatedly comparing the next records from both datasets. If both records are equivalent, only the update record is written to the output dataset; otherwise,

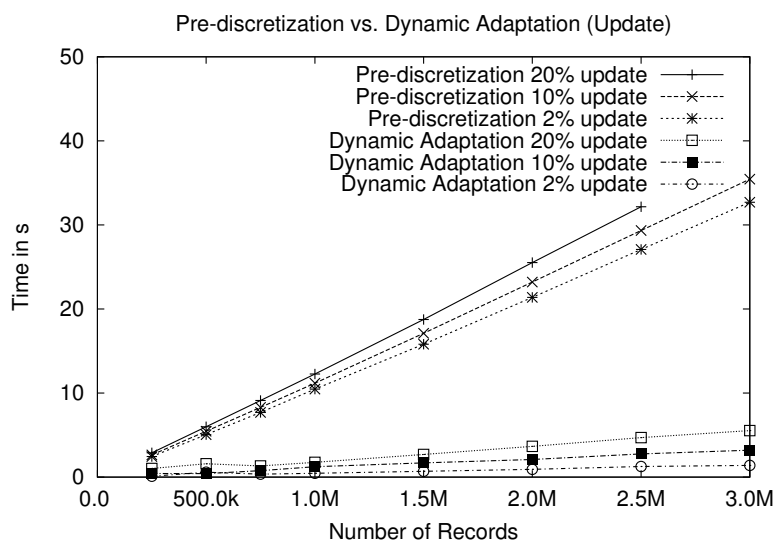


Figure 6.10: Batch update time for different update sizes.

the comparison function used for sorting both datasets is used to determine the order in which the records should be written to the output dataset. As in the sorting step, if the order of two distinct records cannot be determined at a given resolution, the resolution of the Hilbert curve is dynamically increased until both records map to distinct ranks on the Hilbert curve.

The main advantage of this approach is that only the updates need to be sorted, while a linear scan of the updated dataset is sufficient. In contrast, when applying updates that introduce new attribute values, the pre-discretization approach requires to newly discretize the *entire* merged dataset.

Figure 6.10 shows the times required to obtain the Hilbert order of the updated dataset for updates with 2%, 10% and 20% of the sizes of the original dataset. The graph compares the update performance of both the pre-discretization approach and our dynamic adaptation approach. Since the dynamic adaptation approach does not require the resorting of the entire updated dataset, its total update time is composed of the time required to sort the update into Hilbert order and the time required to sequentially merge the original dataset with the update dataset. Hence, our dynamic adaptation approach performs updates significantly faster in comparison with the pre-discretization approach, which requires the sorting of the entire updated dataset. In

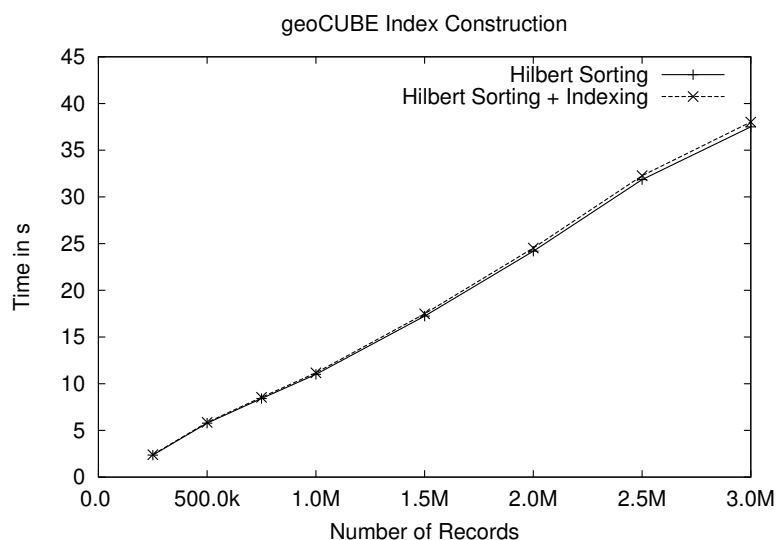


Figure 6.11: Time to construct the index including sorting of the dataset for the geoCUBE index.

particular, we observe an improvement between a factor of 23 for the 2% update and a factor of 6 for the 20% update. As updates to data warehouses are commonly very small (less than 1%), the performance benefits of our dynamic adaptation approach over pre-discretization, especially for small updates, is very significant.

## Index Construction

The construction of the indexing structure is implemented as a bottom-up approach starting with the sorted data at the leaf level of the tree. At each non-leaf level, nodes at this level have a fan-out of  $f$  children, so that nodes from the level immediately below are combined into groups of size  $f$ . For each of these groups, the corresponding parent node in the tree is annotated with the group's minimum bounding box. The nodes of each level of the tree are stored as consecutive segments in memory to minimize the effect of fragmentation. This method to construct the index turned out to be extremely efficient, accounting for about 1% of the total time required to build the geoCUBE for a given dataset, as shown in Figure 6.11.

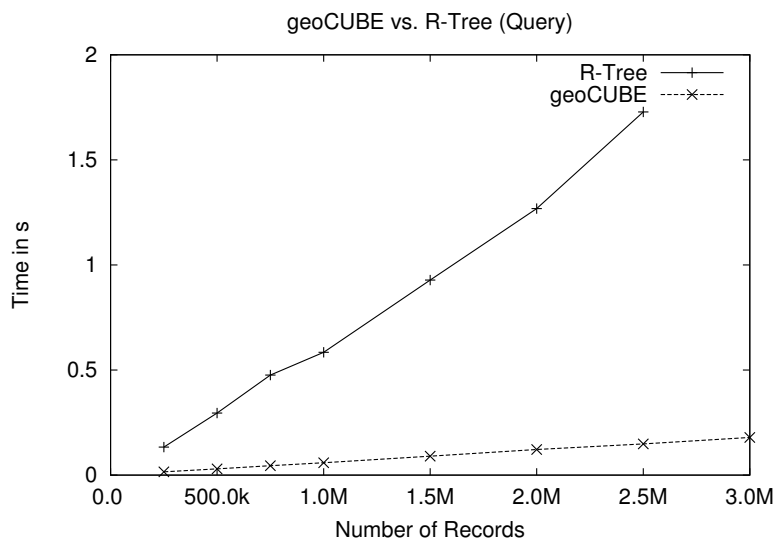


Figure 6.12: Query performance of geoCUBE vs. R-tree for real world data.

### Range Query Performance

Figure 6.12 shows the average range query time on the HYDRO1k dataset over 1000 experiments using our geoCUBE index and an R-tree index from the Spatial Index Library [77]. The queries were constructed as hypercuboids from pairs of records, randomly selected from the dataset and involving all dimensions of the dataset. This graph clearly shows a superior query performance of the geoCUBE index compared to this R-tree implementation. The geoCUBE query time increases only marginally with an increasing number of records, and query results are reported in less than 0.2 seconds even for 3M records. The query time of the R-tree index, on the other hand, quickly deteriorates for larger numbers of records, resulting in query times of more than 1.5 seconds for a single query. Note that some caution is required when interpreting these results. Since we are comparing two independent codebases, it is harder to determine how much of this improvement should be attributed to the improved I/O and cache efficiency of our algorithms and how much is due to better coding practices.

## 6.6 Summary

In this chapter we have proposed a new technique for representing and indexing relational OLAP views with mixed categorical and continuous dimensions. Our approach is flexible with respect to dimension cardinality and thus allows for the indexing of continuous space dimensions, while building on top of established mechanisms for index construction and querying. Our contribution is significant, as it integrates the representation of mixed categorical/continuous data at the storage level, thereby forming the basis for efficient Spatial OLAP systems that can handle massive amounts of data. Such systems are gaining in importance with the increasing amount of spatial data that is collected. Our experimental evaluation shows the practical benefits of the proposed approach. Although our method is specifically intended for the indexing of views in a relational OLAP (ROLAP) setting, some ideas used by our methodology, specifically the use of the Hilbert space-filling curve, may also be applicable to the multidimensional storage of views (MOLAP).

## Chapter 7

### OLAP for Moving Object Data<sup>1</sup>

#### 7.1 Introduction

The analysis of moving object databases is a field of research that has received significant attention in recent years [18,61,62,67,75,103,107,169,192]. Typical applications of this discipline are location-based services [155], traffic control [136], transport logistics [48], wildlife tracking [104] and epidemiology [168]. With the large adoption of Global Positioning Systems (GPS), Radio Frequency Identification (RFID), and mobile devices in everyday life, an increasing amount of data is being collected by such applications, and there is a growing need for the analysis of aggregated information about moving objects.

In traditional data warehouses a key instrument for the analysis of aggregated information is Online Analytical Processing (OLAP) [31]. OLAP enables the efficient analysis of multidimensional data by allowing the user to interactively explore the multidimensional space. As has been previously described in this thesis, this is achieved through a number of operations. The selection of views of the multidimensional space is realized by projecting multidimensional points (facts) into a lower-dimensional space defined by only a relevant subset of dimensions. This approach projects a number of points onto the same point in lower-dimensional space for which the measures of the projected points are aggregated using a predefined aggregation function. Additionally, there may be hierarchies defined for each dimension, which provide a hierarchical grouping of dimension values into categories resulting in a decrease of cardinality for the dimension at each hierarchy level. During a *roll-up* operation the categorical grouping of dimension values at each level of the hierarchy is used to aggregate the measures of facts that are mapped into the same group, similar to the aggregation of measures of multidimensional facts when they are projected into lower-dimensional space. Other important operations provided by OLAP

---

<sup>1</sup>A preliminary version of this work appeared in DEXA 2008 [12].



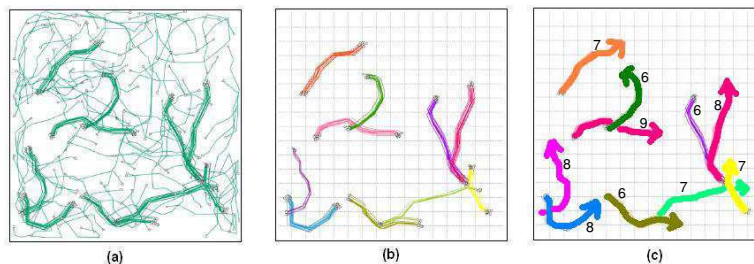


Figure 7.1: *OLAP For Trajectories* Example. (a) Input data. (b) Groups with support above the required minimum support. (c) Aggregate results reported (aggregate trajectories and counts).

are *slice* and *dice*, which can select sub-spaces of the multidimensional space and are comparable to range queries.

To enable a similar interactive analysis of moving object data in an OLAP manner, mechanisms need to be provided that support basic OLAP operations, such as conceptual aggregation of facts with respect to selected dimensions and at varying levels of granularity. When analyzing moving object data, the objects’ trajectories can be considered the facts of the analysis, and their relationship to each other a variable dimension.

In this chapter we propose algorithms that facilitate OLAP-like analysis of moving object data. Our algorithms focus on the identification of aggregate groups among trajectories at varying levels of resolution. However, the underlying conceptual ideas may be applicable to other analysis scenarios as well.

Consider a set of moving objects stored as a relational table *objects* where each record has an attribute *trajectory* =  $[(x_1, y_1, t_1), (x_2, y_2, t_2), \dots, (x_m, y_m, t_m)]$  representing the movement of the respective object as a sequence of positions at times  $t_1, t_2, \dots, t_m$ . Within the scope of this chapter we investigate the problem of evaluating OLAP-typical **GROUP BY** queries with respect to the *trajectory* dimension as subject of the **GROUP BY** operator. The problem is illustrated using the example shown in Figure 7.1. In Figure 7.1a, we observe a number of individual objects that move on random paths, plus ten groups of objects that move together on similar paths. Each group consists of more than five objects moving on similar paths which, taken together, appear to the human eye as “bold” paths.

Furthermore, consider the following SQL query where **trajectory** is both a *feature*

*dimension* that is the subject of the `GROUP BY` operator and a *measure dimension* that is subject to the aggregation function `AGGREGATE`:

```
SELECT AGGREGATE(trajectory) AS trajectory
      COUNT(trajectory) AS count
FROM objects
GROUP BY GROUP_TRAJECTORIES(trajectory, resolution)
HAVING COUNT(*) >= 5
```

In this example, the aim of the `GROUP BY` operation with respect to the *feature dimension trajectory* is to group similar trajectories and eliminate groups with less than a given minimum support (less than 5 similar trajectories). The resulting set of groups is shown in Figure 7.1b. Once the groups of trajectories have been determined, aggregate trajectories summarizing the trajectories in each group are reported. In this example, an aggregate trajectory is the average trajectory computed by calculating for each time  $t_i$ , the average of the locations  $(x_i, y_i)$  of the trajectories contained in the corresponding group. The result is shown in Figure 7.1c, where each qualifying group is represented by its aggregate trajectory and support (count).

In this chapter we propose a new class of `GROUP BY` operators specifically targeted to the OLAP analysis of trajectories and to answering aggregate queries with respect to the spatio-temporal movement of a set of objects. The main problem studied is how to identify aggregation groups with respect to a feature dimension representing trajectories. It is very unlikely that any two trajectories are exactly the same. Hence, standard aggregation of records based on groups with equivalent trajectory values is not very useful in most cases. Instead, we propose to partition the given trajectories into groups of trajectories using a new `GROUP BY` operator, which we term `GROUP_TRAJECTORIES`. This operator returns a group identifier for each trajectory, and then OLAP can proceed with standard aggregation according to the group identifiers instead of the trajectories themselves. The ideal case for forming these groups is to identify disjoint groups. However, when grouping moving objects, this may not always be possible or desired, due to the change of association over time. The goal of the new `GROUP_TRAJECTORIES` operator is to identify groups of objects that have sufficiently similar behavior and to deal with the variances in moving object data, which often renders traditional `GROUP BY` operators unusable.

There exist a number of previous approaches for identifying groups of moving

objects, which we will discuss in detail in Section 7.2. However, many proposed solutions have been developed for a specific domain and do not easily adapt to an application in a general-purpose OLAP framework.

Frequent pattern mining is the only method that has been largely adopted by the data warehousing and data mining community. Many frequent pattern mining approaches have shown good results when identifying patterns that are shared between individual objects. However, our experience is that they often do not produce results suitable for interactive analysis by the users. The most significant reason for this is the amount of redundant information generated by many of the algorithms. For example, with only a small amount of noise present in the original data, many frequent patterns are detected that in fact do not provide any valuable information to the user, as they are based on noisy data. Additionally, frequent pattern mining algorithms consider each pattern as an independent piece of information and do not take into account any relationships that may exist between the detected patterns. In many scenarios, however, interesting patterns may be related to each other by the trajectories they share. These relationships are of utmost importance in applications such as disease tracking, as they may characterize groups of people who potentially have communicated a disease virus through transitive relationships.

The solution proposed in this chapter extends the frequent pattern mining methods already available in today's data warehousing solutions with new algorithms that are more appropriate for identifying groups of moving objects. It is specifically designed to capture relationships between movement patterns and thus allows to identify groups of objects that exhibit a complex behavior, as in the example of a spreading virus above. At the same time it integrates well with existing OLAP models by using established data representations and query languages, as well as allowing the user to interactively browse the results at varying levels of resolution and aggregated information.

The main problem addressed in this chapter is how to define and compute the `GROUP_TRAJECTORIES` operator that defines groups that allow for a meaningful analysis of object movements via OLAP. We propose two versions of the `GROUP_TRAJECTORIES` operator to compute groups of trajectories that are appropriate for OLAP analysis in different circumstances and applications:



Figure 7.2: Illustration of two different version of operator `GROUP_TRAJECTORIES` (a) Group by Intersection, (b) Group by Overlap.

1. *Group by Overlap*
2. *Group by Intersection.*

Section 7.3 shows in detail how the two versions of the `GROUP_TRAJECTORIES` operator are defined and computed. The following outlines the intuition and motivation behind these operators.

The *Group by Intersection* method identifies subsets of trajectories that correspond to movements along a similar path. Figure 7.1 shows an example where movements that follow similar trajectories are aggregated. *Group by Intersection* also identifies groups with parallel movements, such as “marching band” style parallel trajectories. A schematic illustration is shown in Figure 7.2a. The trajectories shown could for example represent a group of four people walking together, and the aggregate would be a simplified representation of that movement. Our *Group by Overlap* method aggregates subsets of trajectories that correspond to sequences of movements with sufficient overlap between subsequent trajectories. A schematic illustration is shown in Figure 7.2b. The trajectories shown could for example represent movements of people who pass on a disease virus, and the aggregate would then represent the total movement of the virus.

The definitions and algorithms for the two versions of the `GROUP_TRAJECTORIES` operator presented in Section 7.3 depend on various parameters, including spatial and time resolution. This allows for analyzing trajectories at various levels of detail/resolution and provides another opportunity for an OLAP-style analysis by enabling drill-down and roll-up on this resolution dimension. An experimental evaluation of the proposed algorithms is presented in Section 7.5. The main goal of

the experiments is to determine how well the two versions of the `GROUP_TRAJECTORIES` operator allow meaningful analysis of object movements via OLAP. We have used various generated and real-life moving object data sets and tested whether the `GROUP_TRAJECTORIES` operator is appropriate for the OLAP analysis of trajectories in the context of different application scenarios. Finally, in Section 7.4, we briefly discuss an interactive OLAP environment for the analysis of trajectories that allows resolution drill-down and roll-up as well as parameter browsing, and is based on the algorithms discussed in this chapter.

## 7.2 Related Work

Research focusing on the storage and processing of moving object data is a relatively new discipline and has evolved from topics such as time series databases, spatio-temporal databases and databases for location-dependent information [74, 169, 192]. A number of data structures and access methods have been proposed to efficiently store, retrieve and update information about moving objects [4, 32, 78, 101, 144, 145, 148, 155, 174, 188, 200]. An overview and classification of most of these data structures can be found in [126]. In many applications, however, the amount of collected data makes it infeasible to analyze the raw information of every individual object. Instead, data is being processed and analyzed in an aggregated manner to extract trends, rules, and typical behavior, as well as exceptions from this typical behavior.

There are a multitude of criteria and information embedded in moving object data that can be analyzed in an aggregated manner. However, much of the previous work focused on the aggregation of only numerical facts that are without correlation to the spatio-temporal properties of the objects' trajectories. Typical choices for aggregate dimensions are often time-only dimensions to provide aggregation "by day" or "by year," or space-only dimensions for aggregation based on topological relationships, such as "by location square" or "within 10 km of" (e.g., [117]). A comprehensive survey of these methods can be found in [112].

A different kind of aggregated analysis of moving object trajectories that focuses on the combined spatio-temporal properties of the trajectories has its roots in the area of Visual Reasoning and Artificial Intelligence [196]. In these disciplines, it often is not sufficient to aggregate the numerical properties of trajectories with respect to

auxiliary dimensions. Instead, the spatio-temporal properties of the trajectories are aggregated with respect to groups of trajectories that exhibit a similar behavior.

Next we discuss the previous work in more detail, focussing first on identifying groups of moving objects and then on aggregating their associated information.

### 7.2.1 Identifying Group of Moving Objects

There exist a number of approaches to identify groups of moving objects, and the methods they employ vary greatly over a wide range of common techniques. For this section we will use the following, hopefully meaningful classification of common techniques:

- Clustering,
- Computational geometry,
- Edit distance and variants thereof, or
- Frequent pattern or association rule mining.

#### Clustering

An intuitive approach to identifying groups of moving objects is clustering. Traditional static geometry-based clustering techniques, such as  $k$ -means [86] or [84], however, are not always sufficient for moving objects, as objects may leave and enter clusters over time and thus impact the set of clusters that are formed. Li et al. addressed this problem by applying a micro-clustering strategy [108], which allows for efficient updating of clusters as time passes and the original set of clusters deteriorates. The underlying concept of this strategy is that objects move continuously and micro-clusters are formed among a small number of close objects with a similar short-term movement (their bounding box does not exceed a given threshold). As the motion within each micro-cluster is roughly the same, each micro-cluster can be considered as an individually moving unit, and updates over time only need to consider interactions among micro-clusters and locally within each cluster. This separation of global and local interactions leads to a significantly reduced amount of computation

required to maintain a reasonable clustering over time, when compared to recomputing clusters for the entire dataset at each time step. To report groups of similarly moving objects, however, the sequence of clusterings that was found through this approach has to be post-processed, an issue not addressed in [108].

In [102], Kriegel and Pfeifle revisited static clustering for moving object data and proposed an approach using what they call Medoid Clustering. This approach takes into account the uncertainty that is associated with the position of an object for a particular observation. By sampling object locations from a spatial density function that models the location uncertainty, it is possible to compute a set of static clusterings. These static clusterings are then compared and ranked according to their distance to each other. The clustering with the smallest rank, i.e., the smallest average distance to other clusterings, is then considered the medoid clustering—the average and most stable clustering of the original dataset.

To address the shortcomings of traditional clustering methods, such as  $k$ -means, with respect to clustering of moving object trajectories, Nanni and Pedreschi [129] proposed the use of density-based clustering [55]. Using a naïve distance function for trajectories, they showed that density-based clustering is more suitable for clustering trajectories, as the produced clusters can have an arbitrary non-spherical shape and are more robust with respect to noise. However, Nanni and Pedreschi noted that the naïve distance function considers trajectories as atomic entities, which is not suitable in many application. To address this issue, they proposed a technique called Temporal Focusing, which examines the neighborhood of each trajectory to identify time intervals that potentially produce a more meaningful and interesting clustering. Their results show that temporal focusing greatly improves the clustering quality of the naïve distance function.

Clustering is a technique that would be intuitive to use for the identification of groups of moving objects. However, the spatio-temporal nature of trajectories makes it difficult to apply common clustering approaches. While the solutions discussed above are suitable for a number of special-purpose applications, they do not fit into a general-purpose data warehousing and OLAP context. Our research focuses on methods that can be applied universally and build on top and integrate with existing database, data warehousing, and data mining technologies.

## Computational Geometry

One of the first approaches to use computational geometry to identify patterns in large moving object datasets was proposed by Laube et al. [105] as an extension to an analysis approach called REMO [104]. REMO is based on a coarse-grained analysis of motion parameters that have been stripped of their absolute positions in Euclidean space and instead use parameters such as orientation, speed and acceleration. These parameters, once mapped to discrete values, and obtained at constant time intervals for each object can be represented as a 2-dimensional bitmap. The two dimensions of the bitmap represent the set of objects and time respectively and the value for each element of the bitmap is a numerical value representing a particular motion parameter. Interesting patterns in the dataset are then identified by determining continuous regions of identical value within the bitmap. The patterns that can be detected using this approach are *constance* (an object maintains constant motion parameter values over a consecutive set of time intervals), *concurrency* (a number of objects have the same motion parameters during the same time interval), and *trend-setter* (a *constance* pattern followed by a *concurrency* pattern). Laube et al. extended this method by including absolute position information for each recorded observation and employ computational geometry algorithms to identify movement patterns in addition to those above. The general approach is to compute the spatial region, e.g., a circular region, for which a set of constraints is satisfied. These constraints can be for example the size of the region or the number of observations for different objects in the region. The additional patterns Laube et al. proposed to detect using the computational geometry approach were *track* (a *constance* pattern additionally constrained by a maximum Euclidean distance between two consecutive observations of the same object), *flock* (a *concurrency* pattern constrained by a maximum distance of the objects to each other), *leadership* (a *trend-setter* pattern constrained by a maximum distance between the objects when concurrent movement commences), *convergence* (a minimum number of objects pass through a region of fixed size independent of time), and *encounter* (a *convergence* pattern with the constraint that the objects are within the identified region at the same time). Gudmundsson et al. improved the complexity of the exact *encounter* algorithm from [105] and provided approximation algorithms for all grouping patterns discussed in [105] to estimate the size of a region



and the minimum number of objects for which interesting patterns can be detected in a given dataset [72]. More recently, Benkert et al. [19] provided computational geometry approximation algorithms with improved asymptotic bounds for the *flock* pattern and Andersson et al. [10] discussed an algorithm to detect *leadership* without an a priori knowledge of the time interval of interest.

The described computational geometry approaches to identifying groups of moving objects appear very powerful and able to identify well-defined patterns. These approaches, however, are not designed to integrate with technologies such as data warehouses or OLAP, and rather represent stand-alone solutions without the support of established data processing frameworks.

### **Edit Distance**

The edit distance approach, and variations of it, has also been the subject of many studies regarding similarity measures for trajectories that can be represented as sequences of motion parameters (e.g., location, orientation, speed, etc.). The most common method that is related to the edit distance and used to measure the similarity between two or more trajectories is finding the longest common subsequence (LCSS). LCSS has originally been proposed for finding similarity in time-series databases [195], which is a problem area closely related to moving object databases. Sclaroff et al. extended the LCSS approach to trajectories and proposed three new similarity functions [159]: (1) trajectories are similar when they are close to each other and, within a given tolerance, represent the same path; (2) trajectories are similar if, independent of their extent and location, their change in orientation and movement is similar; and (3) trajectories are similar if they follow a similar path but are translated from one another. By relaxing the requirement for an exact match and allowing limited deviation in space and time, Sclaroff et al. were able to provide better complexity bounds for their versions of LCSS compared to exact match approaches. For their third similarity function, Sclaroff et al. additionally proposed an approximation algorithm to estimate the amount of translation at which two trajectories match within a given error.

Vlachos et al. built on top of this approach and provided a more extensive analysis [187]. Shim and Chang extended the basic LCSS approach with a  $k$ -warping

technique [166], allowing up to  $k$  replications of motion segments in the query trajectory in order to match similar trajectories. Further, Zeinalipour-Yazti et al. addressed the interesting problem of querying similar trajectories from a bulk of trajectory fragments that are distributed across a network of nodes [198]. Their approach performs localized top- $k$  queries using LCSS to find the  $k$  most similar trajectory fragments at each node and then combine only fragments that are associated with the same trajectory across all nodes.

Edit distance techniques are powerful approaches for determining similarity between two sequences of discrete items. However, for large datasets they become computationally very expensive. In contrast, approaches applying sequential pattern mining, as described in the next subsection, are significantly more efficient. They are also better suited for data processing frameworks, such as data warehousing and OLAP, as they are very similar to other pattern mining approaches already available in such systems.

### **Frequent Pattern Mining**

Another, recently very common, approach to identifying groups of trajectories is pattern mining. Pattern mining is a method that has its origins in the mining of association rules from large sets of transactional data [6]. It is often described in the context of mining frequent sets of items from shopping baskets to identify items or products that are frequently bought together [81]. It has also been shown that the pattern mining approach is applicable to identifying patterns in sequence databases [7]. The analysis of sequential data is a requirement for a wide area of applications including for example the analysis of DNA or protein sequences, data streams in telecommunication or tracking of diseases. A number of approaches that focus on the analysis of sequence data have been proposed and show an improved performance in identifying patterns when compared to transactional approaches. Most recent representatives of these approaches are PrefixSpan introduced by Pei et al. [140,141] and CloSpan introduced by Yan et al. [193], as well as an algorithm suitable for noisy data streams by Yang et al. [194]. A special case of identifying patterns in sequence data is the identification of patterns that occur periodically in constant time intervals. This special case of sequential pattern mining has been addressed by [80] and [113].

One of the first approaches that utilizes frequent pattern mining for the analysis of spatio-temporal databases was introduced by Tsoukatos and Gunopulos [177]. Their algorithm is based on the SPADE algorithm [197], which was originally proposed for mining frequent subsequences from sequence databases. Both algorithms represent the search space for frequent subsequences as a lattice and then traverse the lattice to identify frequent subsequences that occur in the dataset. The two algorithms differ by the method that is used to search the lattice and by the results they produce. The SPADE algorithm performs a lattice-decomposition to obtain sublattices on which it can perform localized in-memory breadth-first search to find all frequent subsequences. Tsoukatos and Gunopulos’s algorithm, on the other hand, uses a depth-first search on the entire search space lattice and finds only maximal frequent subsequences, i.e., frequent subsequences for which no supersequences exist that are also frequent. Many authors have since focused on developing improved algorithms for the mining of patterns from spatio-temporal databases and in particular moving object databases. Other than in classical subsequence mining, the data stored in moving object databases is inherently noisy and exact subsequences are rarely found. Wang et al. [191] and Hwang et al. [91] introduced algorithms specifically for the detection of grouping behavior among moving objects. In both cases the algorithms are intended to find patterns which resemble the movement of multiple objects in unison along similar trajectories. In [191] the trajectories are represented as sequences of observation points, and Wang et al. proposed algorithms which identify similar trajectories by permitting a certain level of spatial and temporal mismatch when comparing observation points of two trajectories. The two algorithms Wang, et al. proposed have been derived from the well-known frequent pattern mining algorithms Apriori [6] and FP-growth [82] and extended to allow for the desired “fuzziness” when comparing observation points. The Apriori algorithm exploits the property of frequent itemsets that every subset of a frequent itemset is also a frequent itemset, to efficiently prune the search space for frequent itemsets. The FP-growth algorithm, on the other hand, first encodes the data set in a data structure called FP-tree and then extracts frequent itemsets from this data structure directly.

Hwang et al. extended the algorithms proposed in [191] by representing trajectories as sequences of line segments rather than observation points, so that for each

point along a trajectory, the distance to another trajectory can be determined. This approach improves the quality of results compared to those obtained in [191] and is in particular superior when observations are sparsely distributed along trajectories. Similarly, Cao et al. described an approach allowing trajectory data to be noisy and imprecise by employing a line simplification method which locally removes segment points from a trajectory if they are within a certain distance to the resulting trajectory. Frequent patterns are then determined based on the simplified trajectories [27].

With a focus on mining periodic spatio-temporal patterns, Mamoulis et al. proposed two algorithms: STPMine1 and STPMine2 [116]. Both algorithms consider trajectories as sequences of locations that have been sampled in uniform time intervals. To determine frequent periodic patterns within these trajectories, both algorithms first employ a clustering of locations that have been sampled at time steps that are multiples of a fixed period length apart from each other. The clustering determines dense regions of locations, and regions with a number of locations below the minimum support are discarded. The remaining regions are then considered frequent 1-itemsets from which the STPMine1 (Apriori-based) and STPMine2 (FP-growth-based) algorithms can generate all remaining frequent itemsets. While the generation of the remaining frequent itemsets is efficient, especially when using the STPMine2 algorithm, the generation of the initial frequent 1-itemsets is costly for both algorithms, as it requires the use of expensive clustering methods.

Two interesting application scenarios for the mining of patterns from moving object databases are described by Peng and Chen in [143] and Gidófalvi and Pedersen in [62]. Peng and Chen addressed the problem of data allocation in mobile communication systems and how the identification of frequent patterns in the movement of the system's users can help to optimize the allocation of system resource and improve the system's performance [143].

In [62], Gidófalvi and Pedersen focused on the problem of identifying ride-share opportunities for commuters. Their dataset consisted of long-term route information for a number of cars that were equipped with GPS recording devices. Gidófalvi and Pedersen's approach employs a frequent pattern mining algorithm that finds closed frequent itemsets and is based on a database projection method allowing the algorithm to be entirely implemented in SQL and executed on the database system storing the

actual trajectory data.

### 7.2.2 Extending OLAP Query Languages

The easiest way to provide a universally accessible OLAP framework for moving object data is by extending the already existing query languages offered by data warehouses with syntax that enables the desired OLAP functionality. In this chapter we introduce the `GROUP_TRAJECTORIES` operator and adopt the idea of extending the SQL syntax to simplify OLAP queries from Gray et al., who first proposed a similar extension with the `CUBE` operator in [70]. Gray et al. suggested to extend SQL's `GROUP BY` syntax to allow the inclusion of grouping functions in the `GROUP BY` list. This approach is illustrated with an example application that uses the grouping functions *Day* to map timestamps to individual days, and *Country* to map longitude/latitude pairs to distinct country names:

```
SELECT day, nation, MAX(Temp) FROM Weather
GROUP BY Day(Time) AS day,
         Country(Latitude, Longitude) AS nation;
```

### 7.3 OLAP for Trajectories

Consider  $N$  moving objects, each identified by a unique tag number  $i$ . Object movements are recorded through a set of readings  $(i, t, (x, y))$  indicating that object  $i$  was located at position  $(x, y)$  at time  $t$ . The  $N$  moving objects are represented by a relational table *objects* with  $N$  records. Each record contains attributes such as *tag*, *name*, *size*, *color*, etc. describing an object in a traditional relational manner that can be represented as a Star schema [31]. Among these attributes is an attribute *trajectory* representing the movement of the respective object as a sequence  $[(x_1, y_1, t_1), (x_2, y_2, t_2), \dots, (x_m, y_m, t_m)]$  of positions at times  $t_1, t_2, \dots, t_m$ . Our goal is to group objects with respect to attribute dimension *trajectory*. For this purpose, we define a new operator `GROUP_TRAJECTORIES`, which returns for each trajectory a group identifier, and then proceeds with standard OLAP aggregation according to the group identifiers instead of the trajectories themselves.

In this section we present two different implementations of the operator `GROUP_TRAJECTORIES`, which compute groups of trajectories that are appropriate for OLAP

analysis in different application scenarios: *Group by Overlap* and *Group by Intersection*.

### 7.3.1 General Framework and Preprocessing

We begin with an outline of the high-level framework and preprocessing steps for the two implementations of the `GROUP_TRAJECTORIES` operator. The details of these implementations are then discussed in Sections 7.3.2 and 7.3.3.

The analysis of trajectories begins with a preprocessing step. The goal of this preprocessing step is to prepare the set of input trajectories  $T$  for the further processing by one of the implementations of our `GROUP_TRAJECTORIES` operator. The preprocessing stage is composed of three parts: (1) the mapping of trajectories to sequences of distinct items, (2) the extraction of frequent patterns from the set of mapped trajectories, and (3) the reverse association of frequent itemsets with sets of trajectories that contain them.

As discussed earlier, the readings of real-world trajectories are often inherently noisy and, even though two trajectories follow approximately the same path, the attribute values of their readings may differ. This issue is addressed in the first part of the preprocessing stage by transforming the attribute values of each reading into corresponding discretized dimensions whose resolution is selected by the user. The motivation behind this transformation is two-fold: (1) noise and minor variances within trajectories are being compensated by mapping fine-grained coordinates to a coarser-grained grid, and (2) the user has control over the granularity of the mapping in an interactive manner.

This transformation of attribute values is also applied to the time dimensions, allowing trajectories to be analyzed at different level of time resolution. For example, the time granularity “day” may be sufficient for a high-level analysis of GPS data for the movement of a fleet of ships. An analysis of the set of paths taken by a group of ships entering a port, on the other hand, may require a time granularity “minute”.

Following the mapping of each trajectory’s time/position pairs into a discrete space, individual readings from different trajectories may now be mapped onto the same discrete attribute values and consequently can be considered equivalent. Based on this property, we now employ frequent itemset mining using an external tool

set [110] to identify sets of distinct time/position pairs that are shared among a minimum number of trajectories. We specifically mine for *closed frequent itemsets*, which are frequent itemsets for which any superset does not have the same support [138]. We refer to each of these sets as a frequent itemset and call the number of trajectories which share the frequent itemset the “support” of the frequent itemset. The minimum support an itemset must have to be considered frequent is user-specified and allows the user to control the size of initial groups of trajectories that should be considered for further analysis.

Additionally we apply a threshold on the size of each frequent itemset, which is also controlled by the user. It allows the user to limit the frequent itemsets to those which contain a minimum number of items. The motivation for this threshold is that frequent itemsets with a larger number of distinct items corresponds to longer shared paths, while frequent itemsets with fewer items correspond to shorter shared paths. The rationale for this is that we assume longer shared paths to be more interesting than shorter paths. This method can be analogously applied to prefer shorter paths.

The last step of the preprocessing is dedicated to the mapping of frequent itemsets back to the sets of trajectories that contain them. For each frequent itemset  $f$ , we determine the set  $C$  of trajectories such that each trajectory contains all of the time/position pairs in  $f$ . We let  $\mathcal{C}$  be the set of all pairs  $(f, C)$ , where  $f$  is a frequent itemset and  $C$  is the corresponding set of trajectories. After the completion of the preprocessing phase, the group merging phase follows, which employs our new `GROUP_TRAJECTORIES` operator. The details of our two versions of the `GROUP_TRAJECTORIES` operator Group by Overlap and Group by Intersection are discussed in the following subsections.

### 7.3.2 Group by Overlap

The *Group by Overlap* method, shown in Algorithm 7.1, introduces a parameter, called the *overlap ratio threshold* (ORT), that controls the strength of the grouping process. The interactive OLAP environment discussed in Section 7.4 allows for an interactive modification of this parameter. The method uses a relationship graph  $\Gamma$  whose vertices correspond to the trajectories. For each frequent itemset  $f$  and corresponding set  $C$  of trajectories, we consider all pairs of trajectories  $(t_i, t_j)$  with

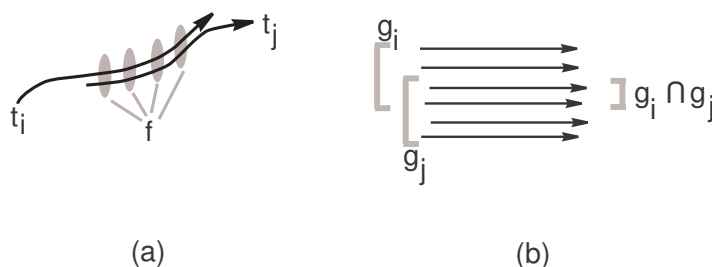


Figure 7.3: Illustration of (a) Overlap Ratio and (b) Intersection Ratio.

$t_i \in C$  and  $t_j \in C$  and add an edge  $(t_i, t_j)$  to  $\Gamma$  if  $\frac{2 \cdot |f|}{|t_i| + |t_j|} \geq ORT$ . We call  $\frac{2 \cdot |f|}{|t_i| + |t_j|}$  the overlap ratio of the trajectories. The intuition is to quantify the amount of overlap between two trajectories relative to their sizes. Figure 7.3a illustrates the relationship between trajectories and their overlap that is characterized by a frequent itemset. The resulting graph  $\Gamma$  then contains edges between those trajectories that have an overlap ratio of at least  $ORT$ . We then compute the connected components of graph  $\Gamma$  and report each connected component as a group of trajectories. The nature of the obtained groups of trajectories is characterized by two factors: (1) the *overlap ratio threshold* determines how much overlap two trajectories must have to be considered to belong to the same group, and (2) the construction of connected components captures transitivity among trajectories and, thus, cascaded “relay”-type movements, as illustrated in Figure 7.2b. Depending on the chosen *overlap ratio threshold*, objects will have to move in unison for more or less of their own individual movements to form cascading trajectories.

### 7.3.3 Group by Intersection

The *Group by Intersection* method is shown in Algorithm 7.2. It introduces a parameter called the *intersection ratio threshold* (IRT), which is used to control how aggressively groups are formed. In an interactive OLAP environment, this parameter can be modified interactively as part of the explorative analysis process. The *Group by Intersection* algorithm first creates an initial set  $\mathcal{G}$  of groups of trajectories, where each group  $C$  is the set of trajectories associated with a frequent itemset  $f$  as determined in the preprocessing step discussed in 7.3.1. Each group  $C$  is assigned a group strength



---

**Algorithm 7.1** Group by Overlap
 

---

**Input:**

1. Set  $T$  of trajectories,
2. Set  $\mathcal{C}$  of mappings from frequent itemsets identified using [110] to sets of trajectories as determined in Section 7.3.1,
3. Overlap ratio threshold  $ORT$ .

**Output:** Set of groups  $\mathcal{G}$ ***Build relationship graph***  $\Gamma = (V_\Gamma, E_\Gamma)$ 

- 1: Initialize set of vertices  $V_\Gamma \leftarrow T$
- 2: Initialize set of labeled edges  $E_\Gamma \leftarrow \emptyset$
- 3: **for all**  $(f, C) \in \mathcal{C}$  **do**
- 4:   **for all** pairs  $(t_i, t_j)$  with  $t_i \in C$  and  $t_j \in C$  **do**
- 5:     Add an edge  $(t_i, t_j)$  to  $E_\Gamma$  if  $\frac{2 \cdot |f|}{|t_i| + |t_j|} \geq ORT$
- 6:   **end for**
- 7: **end for**

***Determine overlap groups in***  $\Gamma$ 

- 8: Compute connected components  $\mathcal{G}$  of graph  $\Gamma$
  - 9: Remove singletons from  $\mathcal{G}$
  - 10: **return**  $\mathcal{G}$
- 

$GS(C)$ , which is initially set to the size of the corresponding frequent itemset. Using the unweighted size of the frequent itemset allows us later to identify groups that are characterized by long frequent itemsets. The remainder of our method merges groups in  $\mathcal{G}$  by iterating the following loop (Lines 6-20 in Algorithm 7.2): for each pair  $g_i, g_j \in \mathcal{G}$ , compute the *intersection ratio*  $IR(g_i, g_j) = \min\left(\frac{|g_i \cap g_j|}{|g_i|}, \frac{|g_i \cap g_j|}{|g_j|}\right)$ , which represents the number of trajectories that occur in both  $g_i$  and  $g_j$ , relative to the sizes of  $g_i$  and  $g_j$ . The intuition behind this definition is that pairs of groups that share a large percentage of their trajectories are more strongly related to each other than pairs that do not share as many trajectories relative to their group sizes. Figure 7.3b illustrates a pair of trajectory groups and the trajectories they share. After computing the intersection ratio, we consider only those pairs  $(g_i, g_j)$  as candidates for merging whose intersection ratio is larger than the intersection ratio threshold. The parameter allows control of how strongly related two trajectory groups are required to be to qualify for merging. For each qualifying pair  $(g_i, g_j)$ , we compute its *merge strength*, which is the average of their group strength values:  $MS(g_i, g_j) = \frac{GS(g_i) + GS(g_j)}{2}$ . The merge strength represents the average support in terms of shared locations the new

---

**Algorithm 7.2** Group by Intersection
 

---

**Input:**

1. set  $\mathcal{C}$  determined as in Section 7.3.1 using [110]
2. Intersection Ratio Threshold  $IRT$ .

**Output:** set of groups  $\mathcal{G}$ *Create initial set of intersection groups*

- 1:  $\mathcal{G} \leftarrow \emptyset$
- 2: **for all**  $(f, C) \in \mathcal{C}$  **do**
- 3:    $\mathcal{G} \leftarrow \mathcal{G} \cup \{C\}$
- 4:   set initial Group Strength  $GS(C) = |f|$
- 5: **end for**

*Merge intersection groups*

- 6: **repeat**
  - 7:   **for all**  $g_i, g_j \in \mathcal{G}, g_i \neq g_j$  **do**
  - 8:     set Intersection Ratio
  - 9:      $IR(g_i, g_j) = \min\left(\frac{|g_i \cap g_j|}{|g_i|}, \frac{|g_i \cap g_j|}{|g_j|}\right)$
  - 10:     **if**  $IR(g_i, g_j) > IRT$  **then**
  - 11:       set Merge Strength
  - 12:        $MS(g_i, g_j) = \frac{GS(g_i) + GS(g_j)}{2}$
  - 13:     **else**
  - 14:       set Merge Strength  $MS(g_i, g_j) = 0$
  - 15:     **end if**
  - 16:   **end for**
  - 17:   find  $(g_{i'}, g_{j'})$  for which  $MS(g_{i'}, g_{j'})$  is maximal
  - 18:   **if**  $MS(g_{i'}, g_{j'}) \neq 0$  **then**
  - 19:      $\mathcal{G} \leftarrow (\mathcal{G} \setminus \{g_{i'}, g_{j'}\}) \cup \{g_{i'} \cup g_{j'}\}$
  - 20:     set Group Strength
  - 21:      $GS(g_{i'} \cup g_{j'}) = MS(g_{i'}, g_{j'})$
  - 22:   **end if**
  - 23: **until**  $MS(g_{i'} \cup g_{j'}) = 0$
  - 24: **return**  $\mathcal{G}$
-

group receives from each contributing group independent of the actual number of trajectories in the group. We chose to not weight the merge strength by the size of the groups that contribute to the merged group as the goal of the merging is to identify groups that are supported by long frequent itemsets. The influence of the number of trajectories in each group is only intended to impact the *intersection ratio* and otherwise have no influence on the grouping process. Hence, the number of trajectories in each group is not considered when computing the *merge strength*.

All candidate pairs are then ranked by their merge strength and a pair  $(g_{i'}, g_{j'})$  with maximum merge strength is merged to produce a new group. The intuition for using a pair with maximum merge strength is that the average support in terms of shared locations in the resulting group is highest. This implies that trajectories within the group are on average supported by more shared locations than in other groups and are consequently characterized by longer frequent itemsets. The group strength  $GS(g_{i'} \cup g_{j'})$  of the new merged group is the merge strength  $MS(g_{i'}, g_{j'})$  of the pair  $(g_{i'}, g_{j'})$ . This process is repeated until there are no more pairs of groups with intersection ratio larger than the intersection ratio threshold and non-zero merge strength.

The *Group by Intersection* method aggregates subsets of trajectories that constitute a movement of individual objects in parallel similar to that of a marching band or a flock of birds. The nature of the obtained groups of trajectories is characterized by: (1) the intersection ratio threshold, which determines how many shared trajectories between two groups are “sufficient” for them to be merged, and (2) the merge strength, which favors the merging of groups that are supported by long frequent itemsets. Thus, groups that are merged do not only share trajectories, but share trajectories that are supported by long and interesting frequent itemsets. We first merge those candidate groups whose shared trajectory segments are longest and then proceed top-down, merging groups with shorter shared trajectory segments. Unlike the *Group by Overlap* method, which combines sequences of trajectories, the *Group by Intersection* method combines parallel trajectories.



Figure 7.4: Screenshot of the interactive environment for *OLAP For Trajectories*.

## 7.4 Interactive OLAP for Trajectories

The algorithms for the two different versions of the `GROUP_TRAJECTORIES` operator presented in Section 7.3 are guided by the following parameters:

- Space resolution,
- Time resolution,
- Minimum support of a frequent itemset,
- Intersection ratio threshold, and
- Overlap ratio threshold.

This provides an opportunity for an interactive OLAP analysis of groups of trajectories at various levels of resolution or “connectedness.” For example, consider the analysis of tracking data from GPS-enabled trucks. A high-level analysis of trajectories belonging to trucks that travel long distances may be sufficient at a time-dimension granularity of “day.” However, an analysis of trajectories that belong to trucks within an urban area may require a time-dimension granularity of “minute” to capture more detailed movement patterns. This approach suggests to provide the user with the

ability to browse the parameter space interactively and allow the parameters to be adjusted as necessary for a given analysis objective. As an example for browsing a parameter like the overlap ratio threshold, consider a set of trajectories representing movements of people who pass on a disease virus. The grouping determined by the *Group by Overlap* method could be used to analyze the total movement of the virus. In this example, the overlap ratio threshold would represent the amount of interaction between individuals required to pass on the virus. Changing the threshold value allows to evaluate how far the virus will spread based on different assumptions about its transmission. A prototype of an interactive environment for the OLAP analysis of trajectories is shown in Figure 7.4. The interactive environment allows for resolution drill-down and roll-up, as well as parameter browsing. The system visualizes an implementation of the two `GROUP_TRAJECTORIES` operators presented in this chapter. It does not yet include other OLAP functionality, such as spatio-temporal aggregation or aggregation with respect to other feature dimensions. The system allows to explore the results of the `GROUP_TRAJECTORIES` operators depending on different resolution and threshold values for several synthetic and real-life data sets.

## 7.5 Experimental Evaluation

The experimental evaluation of the `GROUP_TRAJECTORIES` operators presented in this chapter is divided into four parts. Section 7.5.1 provides a detailed analysis of each algorithm using appropriate examples. In Section 7.5.2 the robustness of the `GROUP_TRAJECTORIES` operators against background noise is evaluated. Section 7.5.3 then investigates the influence of the algorithms' parameters on the results produced. Finally, Section 7.5.4 concludes the experimental evaluation with results obtained using real-world data and compares them to results obtained using frequent pattern mining only.

### 7.5.1 Detailed Analysis

In this section we evaluate our algorithms with respect to a variety of synthetic scenarios that are carefully designed to demonstrate the strengths and weaknesses of the grouping algorithms. Each scenario addresses a specific property of the dataset and evaluates the influence of changing that property on the results obtained by both

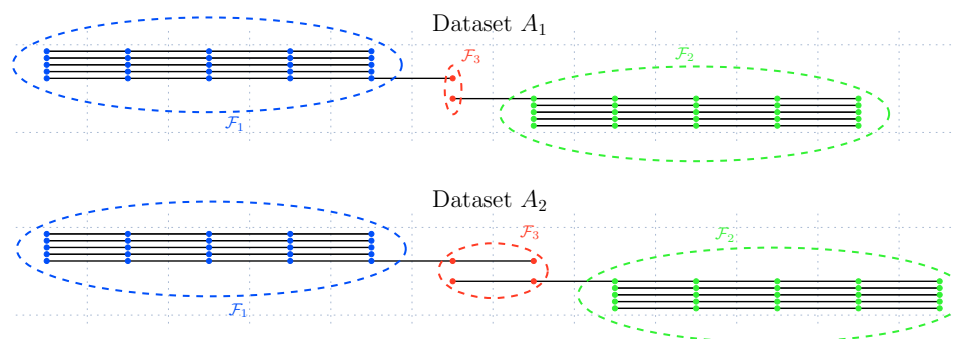


Figure 7.5: Two datasets each containing three closed frequent itemsets but with different associated overlap.

of our algorithms.

## Overlap

The overlap property represents the number of shared frequent items (grid cells) between at least two trajectories compared to their total lengths. This property is used by the *Group by Overlap* algorithm to identify trajectories that should be grouped together.

Figure 7.5 shows two datasets, each containing three frequent itemsets with a minimum support of 2. The frequent itemsets are *closed*, that is, there are no frequent itemsets with the same or greater support that are supersets of the frequent itemsets shown. Both datasets are identical except for the change in the length of the overlapping trajectories that appear to connect the “left” and the “right” side of each dataset: the number of shared locations in frequent itemset  $\mathcal{F}_3$  changes from  $|\mathcal{F}_3| = 1$  to  $|\mathcal{F}_3| = 2$ .

Figure 7.6 shows the resulting groups for each of our algorithms when applied to dataset  $A_1$ . For the *Group by Overlap* algorithm, we chose an overlap ratio threshold of  $ORT = 0.25$ ; for the *Group by Intersection* algorithm, we chose an intersection ratio threshold of  $IRT = 0.25$ . None of our algorithms combines the groups on either side of the overlap, as the overlap ratio  $OR = 1/6$  does not satisfy the overlap ratio threshold  $ORT = 0.25$ , and the intersection ratio of  $IR = 1/5$  for the association of the overlapping trajectories with the shorter trajectories does not satisfy the threshold  $IRT = 0.25$  required for grouping by intersection. Hence, for the *Group by*

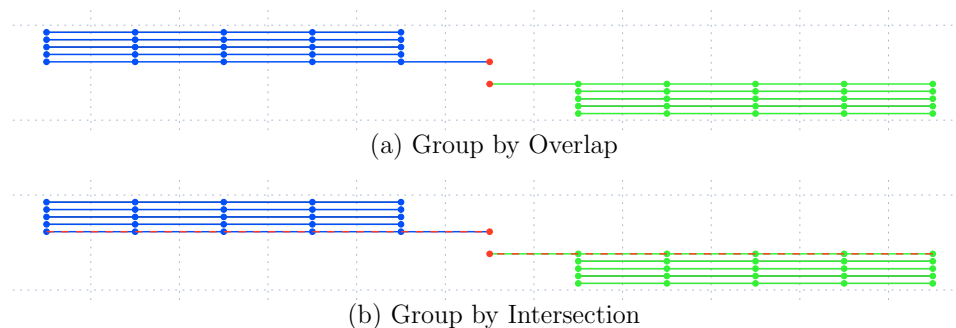


Figure 7.6: Grouping results for dataset  $A_1$  with  $ORT$  and  $IRT$  set to 0.25 respectively.

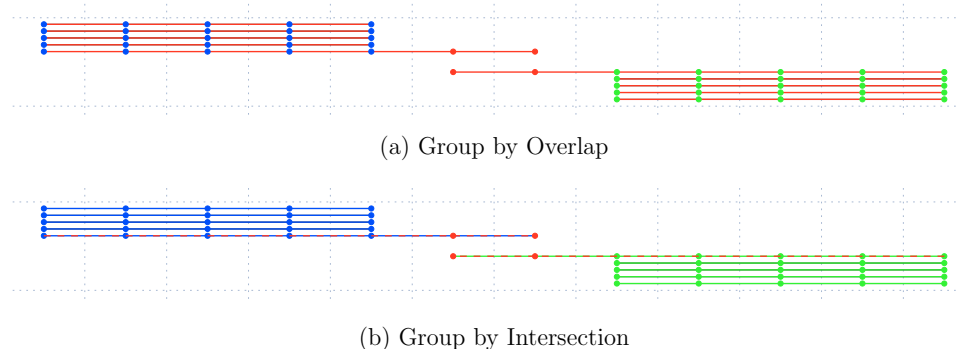


Figure 7.7: Grouping results for dataset  $A_2$  with  $ORT$  and  $IRT$  set to 0.25 respectively. Note, the change in overlap results in a different grouping for *Group by Overlap* when compared to dataset  $A_1$ .

*Intersection* algorithm, each of the overlapping trajectories is present in two groups (represented as dashed lines in Figure 7.6), one group for each closed frequent itemset it shares.

When increasing the length of the overlap between the “left” and the “right” side from  $|\mathcal{F}_3| = 1$  to  $|\mathcal{F}_3| = 2$  shared frequent items in dataset  $A_2$ , the overlap ratio for the overlapping segment changes from  $1/6$  to  $2/7$  and consequently satisfies the overlap ratio threshold of  $ORT = 0.25$ . This results in the merging of the “left” and “right” side into a single group by the *Group by Overlap* algorithm, as shown in Figure 7.7. The result of the *Group by Intersection* algorithm does not change, as the length of the overlap does not affect the intersection ratio.

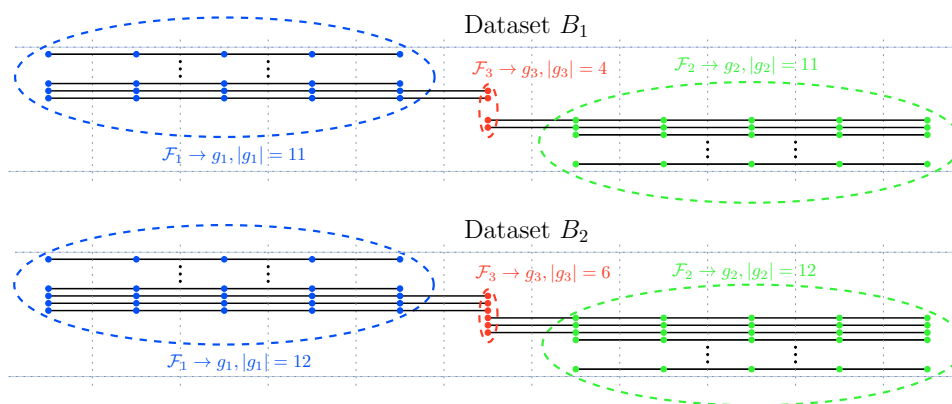


Figure 7.8: Changing the *Intersection Ratio* by changing the number of parallel trajectories.

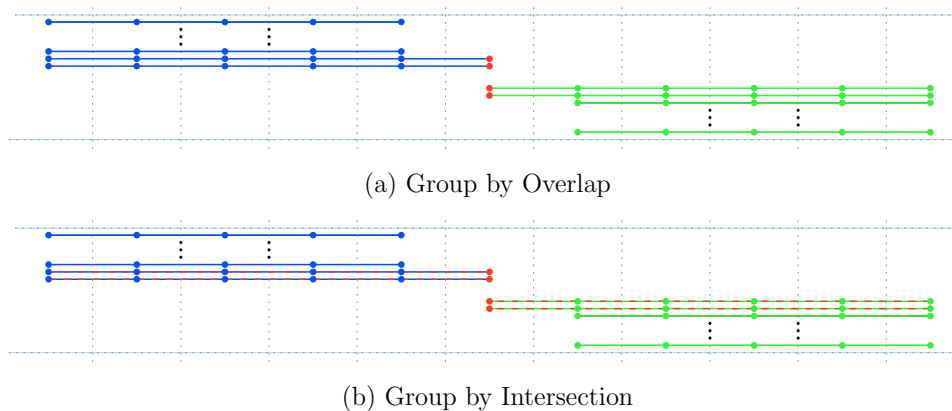


Figure 7.9: Grouping results for dataset  $B_1$  with  $ORT$  and  $IRT$  set to 0.2, respectively.

## Intersection

The *Intersection Ratio* between two groups is the number of trajectories shared between the groups relative to the size of the larger of the two groups. This property is used by the *Group by Intersection* algorithm to recursively identify groups that can be merged.

Figure 7.8 illustrates two datasets, each containing three closed frequent itemsets with a minimum support of 4. Both datasets are identical except for two additional trajectories added to dataset  $B_2$ . The added trajectories have an influence on the intersection ratio between the initial groups that are formed from the three frequent itemsets.



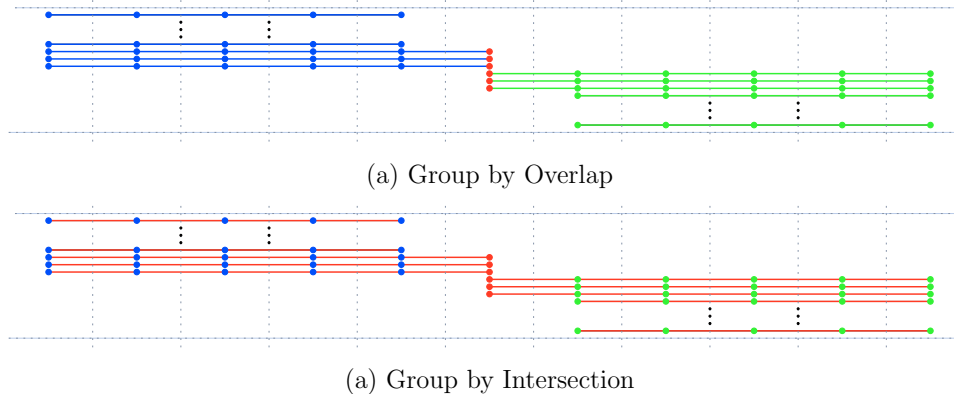


Figure 7.10: Grouping results for dataset  $B_2$  with  $ORT$  and  $IRT$  set to 0.2, respectively. Note, the increase of intersection size results in a different grouping for *Group by Intersection* when compared to dataset  $B_1$ .

Figure 7.9 shows the results for each of our grouping algorithms when there are only two trajectories on every side overlapping with the trajectories of the other side (dataset  $B_1$ ). The *Overlap Ratio Threshold*  $ORT$  and *Intersection Ratio Threshold*  $IRT$  were both chosen to be 0.2. For the *Group by Intersection* algorithm, the initial groups  $g_1$ ,  $g_2$  and  $g_3$  are created from the frequent itemsets  $\mathcal{F}_1$ ,  $\mathcal{F}_2$  and  $\mathcal{F}_3$ , respectively. Since the size of the intersection between groups  $g_1$  and  $g_3$ , or  $g_2$  and  $g_3$  is only of size 2, and  $g_1$  and  $g_2$  each have 11 trajectories, the intersection ratio of  $IR = 2/11$  does not satisfy the requirement of  $IRT = 0.2$ . Hence, the *Group by Intersection* algorithm does not combine the groups into a single group. Similarly, the overlap of  $|\mathcal{F}_3| = 1$  is not sufficient to match the required overlap ratio threshold  $ORT = 0.2$  for the *Group by Overlap* algorithm.

However, the *Group by Intersection* algorithm is able to combine groups if the size of the intersection is increased so that the requirement of  $IRT = 0.2$  is satisfied for the candidate groups. For the first merge iteration, the example contains two candidate groups  $g_1 \cup g_3$  and  $g_2 \cup g_3$ , both with an intersection ratio of  $3/12 = 0.25$ . Since both candidate groups have the same merge strength, either group can be chosen to be created first. Assume group  $g_1 \cup g_3$  is chosen to be created first. The next merge iteration then contains a single candidate group  $(g_1 \cup g_3) \cup g_2$  with an intersection ratio of  $3/15 = 0.2$ . The intersection ratio 0.2 satisfies the requirement  $IRT = 0.2$ , and the group is created, resulting in a single group containing all trajectories in dataset  $B_2$ .

as shown in Figure 7.10. In this scenario the *Group by Overlap* algorithm is not able to combine the groups, as the size of the intersection does not have an influence on the overlap ratio between two trajectories.

The experiments discussed in this section demonstrate the detailed behaviour of our algorithms for carefully designed synthetic datasets and explore various boundary cases of our algorithms. The experiments show the different properties of the input dataset for which either our Group by Overlap algorithm or our Group by Intersection algorithm is better suited to identify groups. For the Group by Overlap algorithm, the experiments show that the algorithm favours the combination of groups whose trajectories have a sufficient amount of locations in common. It can therefore be used to identify groups of trajectories that are formed by sequentially connected trajectories, where each “connection” must be of a certain length in comparison to the length of each trajectory in the group. The experiments for the Group by Intersection algorithm, on the other hand, show that the algorithm is better suited for identifying groups that are composed of trajectories that run in parallel. It favors the merging of groups that have entire trajectories in common, i.e., trajectories that run in parallel, and the number of trajectories in common is at least a given fraction of the size of each group. Thus, if the goal is to identify groups that are formed by the partial overlap of trajectories, the Group by Overlap algorithm should be applied. If groups should be identified that have many parallel trajectories, then the Group by Intersection algorithm is a more adequate choice.

### 7.5.2 Robustness Against Noise

We tested the robustness of the `GROUP_TRAJECTORIES` implementations against background noise. For that, we created a synthetic dataset of 10 groups with 10 trajectories each, and then added random trajectories as background noise. Figure 7.11a shows four input datasets, all containing the same groups but each with a different amount of background noise. For this example we have chosen datasets with 0% (no noise), 50%, 75% and 95% noise. The percentage of noise is specified with respect to the total number of trajectories in the dataset. The subject of the evaluation is: What level of noise can be present in the input data while maintaining a correct result? Here, correctness means that `GROUP_TRAJECTORIES` reports the original groups

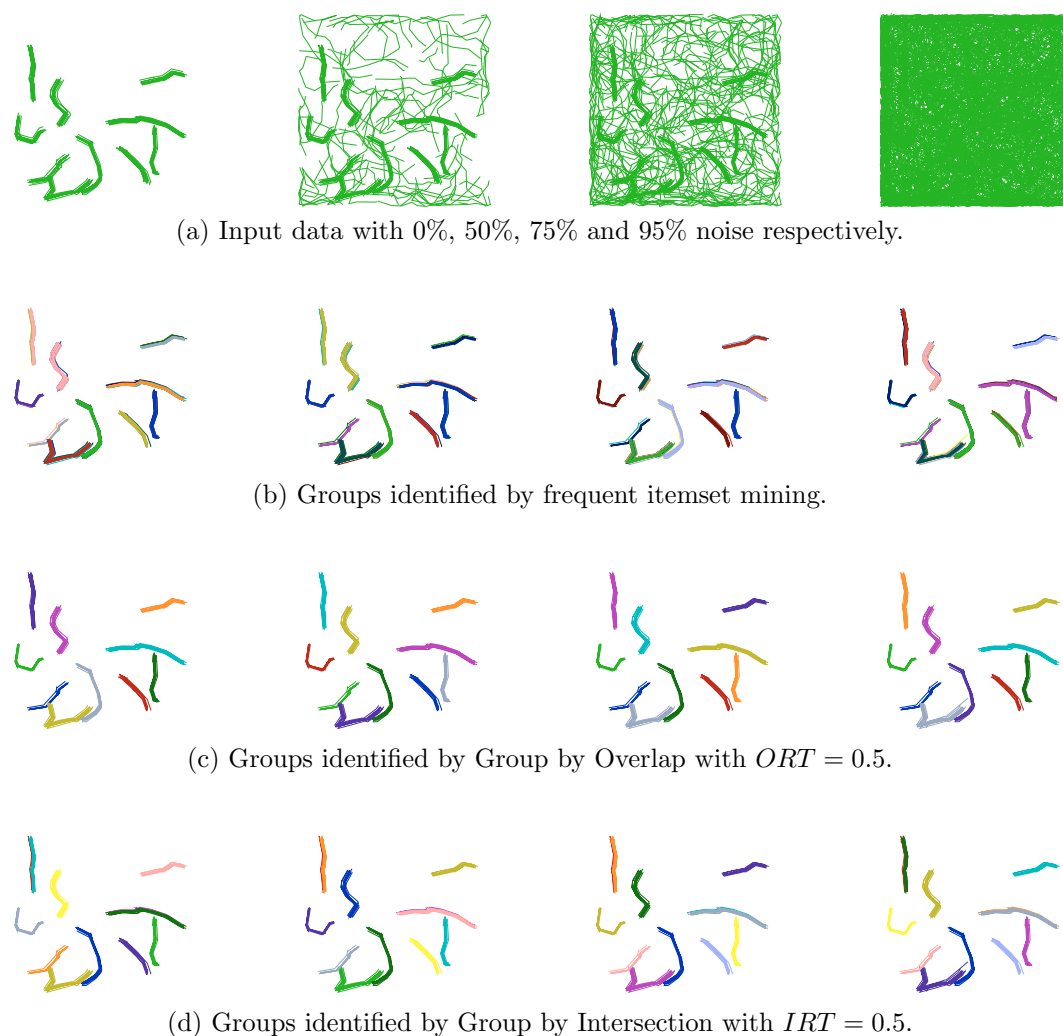


Figure 7.11: Groups identified by each of our algorithms from a dataset with varying levels of noise. The input parameters were set to a space resolution of 5, a minimum support of 4 and a minimum frequent itemset length of 4.

and discards the randomly added trajectories. Figure 7.11b shows the initial groups obtained by the frequent itemset mining algorithm before applying the *Group by Overlap* or *Group by Intersection* algorithm. The frequent itemset mining algorithm identifies 51 groups for noise levels of 0%, 50% and 75% and 52 groups for a noise level of 95%. While it eliminates most of the noise from the input data, it does not identify a suitable grouping of the remaining trajectories, as it does not capture the relationships between trajectories characterized by different frequent itemsets. The *Group by Overlap* algorithm on the other hand identifies for each dataset the correct set of 10 distinct groups (see Figure 7.11c). It improves on the grouping obtained

from frequent itemset mining, and the 10 groups it identifies sufficiently represent the original groups. Only 16% of trajectories that belong to groups in the input dataset were not captured by the algorithm. These *false negatives*, i.e. trajectories that should be detected, but are not, can be attributed to the choice of resolution used for the discretization of the trajectories as their discretized representation does not generate any frequent itemsets. Those trajectories are not characterized by any frequent itemset and, thus, could not have been included in the final grouping. For this example, at 95% noise, the frequent itemset mining also classifies only a single trajectory (1.18%) as a *false positive*, i.e. a trajectory that should not be detected, but is detected regardless, which subsequently is reported as part of a final result group determined by the *Group by Overlap* algorithm.

Figure 7.11d shows the final groups identified by the *Group by Intersection* algorithm. It improves on the grouping determined by frequent itemset mining. However, as it shares the same preprocessing steps with the *Group by Overlap* algorithm it is also missing the 16% of trajectories that are not found by the frequent itemset mining. For this scenario, the *Group by Intersection* algorithm does not produce the same quality of groups as the *Group by Overlap* algorithm. For the 0% and 50% noise levels, it produces 12 final groups; and for 75% and 95% noise levels, it produces 13 final groups, resulting in some trajectories not being grouped together in the same group, though still being reported as a separate group. Depending on the application scenario, this deficiency is acceptable, as the overall result is still superior when compared to results obtained by frequent itemset mining only. Note, at a noise level of 95%, it becomes difficult for the human eye to visually detect the original groups; however, both `GROUP_TRAJECTORIES` methods still report very good results.

### 7.5.3 Input Parameters

In this section we examine the influence of the algorithms' input parameters on the results produced. As input data, we use a synthetic dataset that consists of a mix of groups of trajectories. Some groups are of the type that is best for *Group by Overlap*, and some groups are of the type that is best for *Group by Intersection*. The dataset is shown in Figure 7.12. It consists of three spirals with parallel paths in each spiral for a total of 24 groups. While a spiral-like movement is not a pattern



Figure 7.12: Synthetic dataset with 24 groups each consisting of 10 trajectories and a partial overlap of approximately 25%.

commonly occurring in real world data, it represents a challenging pattern for our methods. Note the subdivision of the spiral into several color-coded segments. Each such segment represents a group of trajectories that are moving in unison. To achieve a more realistic movement, a small random variance is added to the movement of each trajectory. Furthermore, each segment overlaps with the previous and the following segment.

The parameters whose influence on the results obtained by our algorithms we examine are the resolution at which the frequent pattern mining is performed, the overlap ratio threshold (*ORT*) parameter of the *Group by Overlap* algorithm, and the intersection ratio threshold (*IRT*) parameter of the *Group by Intersection* algorithm.

## Resolution

The resolution parameter determines the discretization of the space that is performed before the mining of the frequent itemsets from the trajectories. It is used to generalize the trajectories' locations so that locations that are mapped to the same grid cell are regarded as identical locations. This approach reduces noise and local variances in trajectories and allows for more meaningful frequent itemsets to be found. It has a strong influence on the quality of the final groups found by our algorithms. Figure 7.13 shows the results of both of our algorithms for the dataset described above, a set of fixed parameters  $min\_support = 4$ ,  $min\_length = 4$ ,  $ORT = 0.2$ ,  $IRT = 0.2$ , and a variable resolution between 2 and 8 bits across the extent of each dimension. We observe that the *Group by Overlap* algorithm indeed favors the overlapping spiral segments and eventually identifies one group for each spiral for a total of 3 groups at

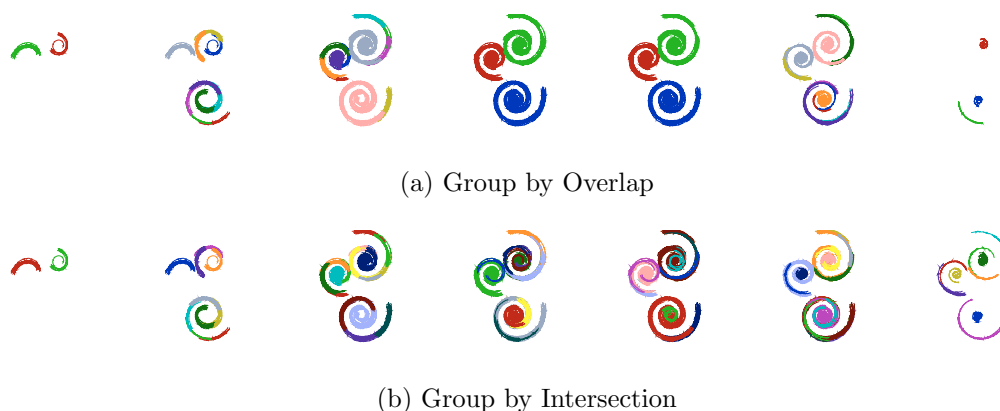


Figure 7.13: Groups identified by each of our algorithms at levels of resolution between 2 and 8 (left to right). Fixed parameters are  $min\_support = 4$ ,  $min\_length = 4$ ,  $ORT = 0.2$ ,  $IRT = 0.2$ .

resolution levels of 5 and 6 bits. The *Group by Intersection* algorithm, on the other hand, does not make use of the overlap between the segments and cannot clearly identify individual spirals. Even at higher resolutions, when the generalization of the trajectories becomes less effective, our *Group by Intersection* algorithm can still be used to identify individual segments of the spirals that constitute parallel movements. Although, it detects 21 groups at resolutions 6 and 7, the quality of these groups does not appear to be as good as the quality of the 18 groups detect at resolution 4. We can also observe, that due to the choice of resolution some details about trajectories are lost. In particular groups at the centers of the spirals are affected and are either not detected or grouped together into larger groups. Note, that at lower and higher resolutions the sizes of the detected groups significantly decrease compared to medium resolutions. This can be attributed to the fact that at lower resolutions most frequent itemsets do not satisfy the minimum length of at least 4 items, and at higher resolutions frequent itemsets do not satisfy the minimum support of 4 trajectories due to the variance in the trajectories.

Figure 7.14 illustrates the relationship between the level of resolution and the number of groups identified by each algorithm. It can be observed that the *Group by Intersection* algorithm tends to identify many, but smaller groups, while *Group by Overlap* favors fewer but larger groups.

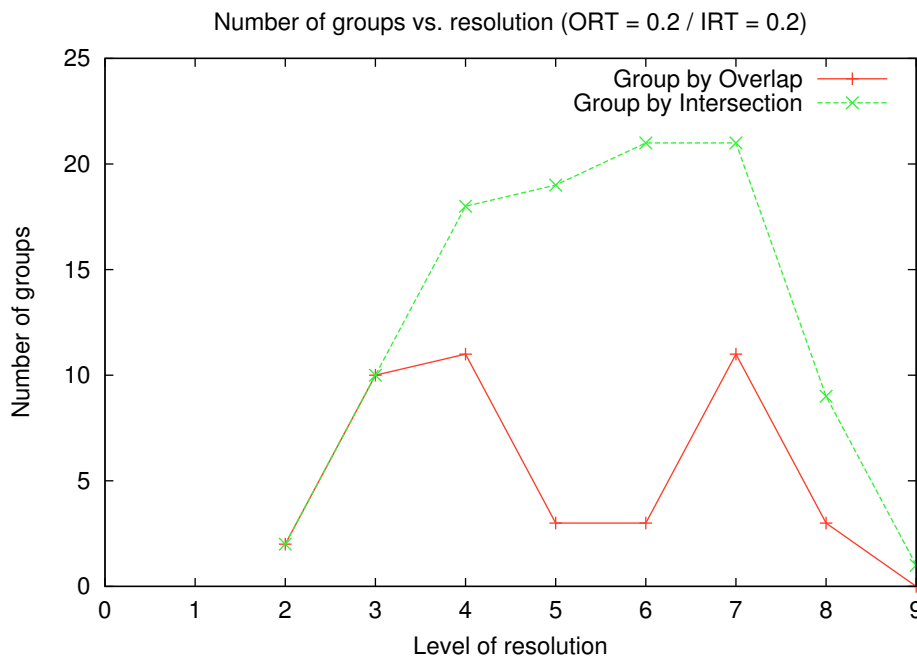


Figure 7.14: Number of groups each algorithm identifies for the given input dataset depending on the resolution. Note, *Group by Overlap* tends to identify fewer but larger groups, *Group by Intersection*, on the other hand, identifies more but smaller groups.

### Overlap Ratio Threshold and Intersection Ratio Threshold

The overlap ratio threshold ( $ORT$ ) and intersection ratio threshold ( $IRT$ ) input parameters for our `GROUP_TRAJECTORIES` implementations influence their sensitivity towards identifying groups.

Using the dataset in Figure 7.12, we tested our `GROUP_TRAJECTORIES` implementations for various values of  $ORT$  and  $IRT$ . The results are shown in Figure 7.15.

We observe that, for lower values of  $ORT$  and  $IRT$ , the identified groups become larger in size, but fewer groups are identified. This behavior is expected, as a lower threshold provides less constraints on the formation of groups. For larger values of  $ORT$  and  $IRT$ , on the other hand, the algorithms become more restrictive in terms of identifying and merging groups, and other, more subtle patterns are detected.

In the example shown in Figure 7.15, the *Group by Overlap* method identifies larger groups for low values of  $ORT$ , and reports the entire spirals with a total of 3 groups

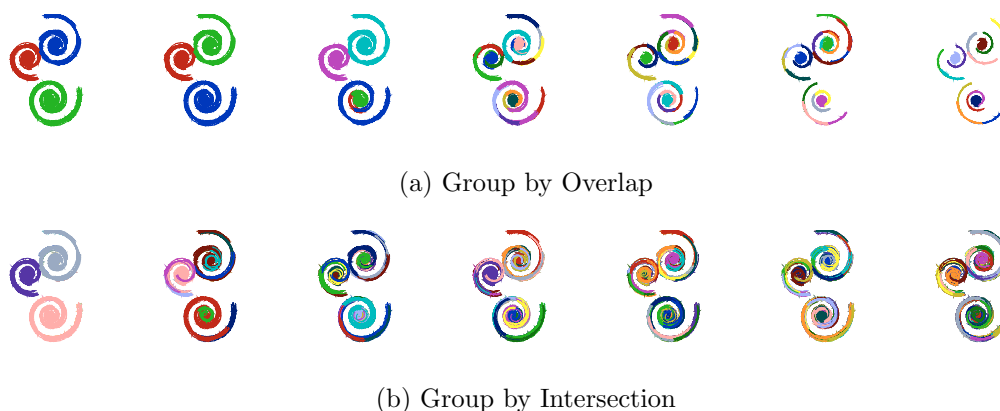


Figure 7.15: Groups (identified by color) computed by both of our methods for  $ORT = IRT = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7$  ( $min\_support = 4, min\_length = 4$ ).

for  $ORT = 0.1$  and  $ORT = 0.2$ . The identified groups become closer to the initial set of groups of trajectories in the spirals as the value for  $ORT$  increases. For large values of  $ORT$ , there is insufficient overlap between parallel paths and the reported groups become smaller to a point where individual trajectories are not grouped any more and are being discarded as singletons. For the *Group by Intersection* method, we observe that the number of groups reported increases from 11 for  $IRT = 0.1$  to 1594 for  $IRT = 0.9$  with increasing values for  $IRT$  as the algorithm becomes more and more discriminating between the initial groups formed by frequent itemset mining, eventually reporting almost each individual frequent itemset as a separate group. A summary of the number of groups reported as a function of  $ORT = IRT$  is given in Figure 7.16.

#### 7.5.4 Real World Data

For the evaluation of our methods on real-world data, we have chosen the school buses dataset that can be freely obtained from [175]. The dataset contains 145 trajectories of buses that are moving in and around an urban area.

Figure 7.17a shows the majority of the data set around the urban center (a few routes going to far out places were removed to better display the data). Figure 7.17b represents the 76 groups that are identified by applying only frequent pattern mining (as e.g. in [27, 62, 116]) (plus a minimum length cutoff as used in our methods). The large number of groups reported by frequent-pattern-mining-based methods is often a



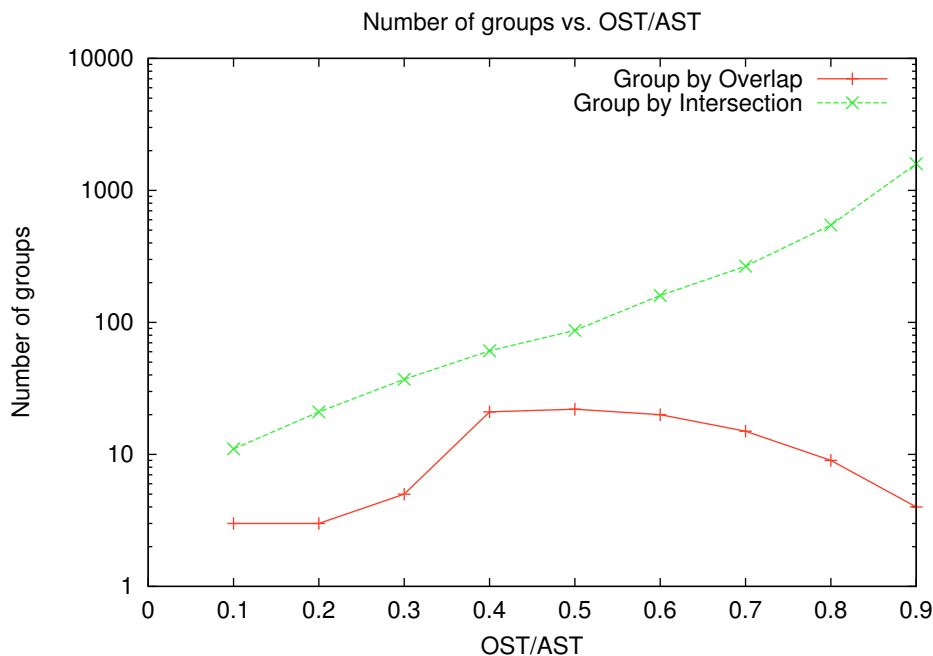


Figure 7.16: Relationship between the number of identified groups and values for *Overlap Ratio Threshold*  $ORT$  and *Intersection Ratio Threshold*  $IRT$ .

disadvantage because it may lead to very little aggregation in an OLAP setting. Figure 7.17c shows the groups reported by the *Group by Overlap* method for  $ORT$  values 0.4, 0.5, 0.6, and 0.7. We observe that the parameter  $ORT$  in our *Group by Overlap* method allows for a much finer control over the grouping of trajectories reported and that this method reports a considerably smaller number of groups when compared to the original number of frequent patterns. Additionally, the groups found by the *Group by Overlap* algorithm are significantly more distinct than the original frequent itemsets and one can distinguish sets of trajectories that appear to be grouped due to their spatial proximity and the locations they share. For example, the orange group in Figure 7.17c for  $ORT = 0.5$  primarily includes trajectories in the southern region of the sample dataset. For the larger  $ORT$  values of 0.6 and 0.7, however, this group is no longer identified, as the overlap among trajectories in this group does not satisfy the  $ORT$  constraint anymore. The groups that remain for  $ORT$  values of 0.6 and 0.7 are spatially smaller and denser groups where many trajectories share locations within a smaller spatial region. This selectivity of groups depending on the values of  $ORT$  makes the *Group by Overlap* algorithm a good choice for the analysis of the

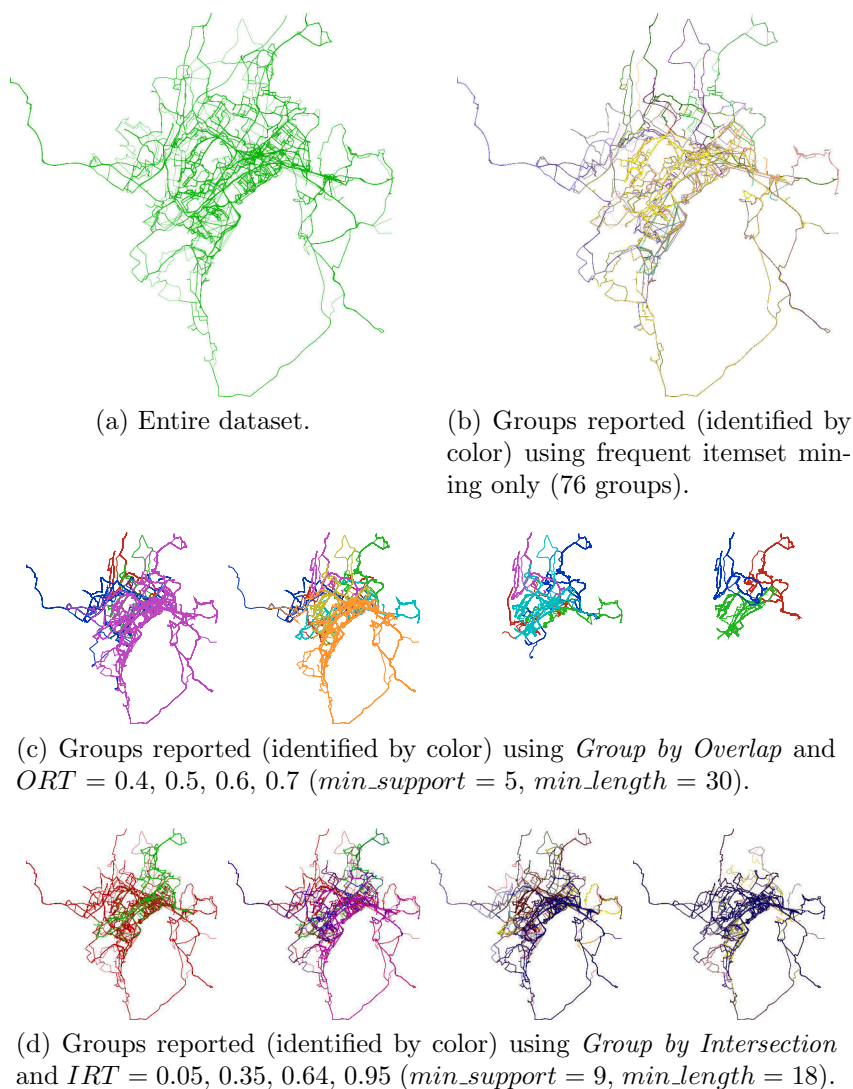


Figure 7.17: Results obtained for the School Buses Dataset.

given dataset. It furthermore enables an interactive exploration of moving object data and helps discovering hidden relationships among the moving objects.

Figure 7.17d shows groups within the real-world dataset that have been identified using our *Group by Intersection* algorithm. At a first glance the groups that are identified by the *Group by Intersection* algorithm are not as distinct as those identified by the *Group by Overlap* algorithm. However, the *Group by Intersection* algorithm does exhibit a very subtle selection of groups with properties that satisfy the intersection ratio threshold (IRT). Assume the example dataset represents the trajectories of roaming individuals and the subject of the analysis is the spreading of

an infectious disease, where a transfer of the disease is only likely when a critical mass of individuals in the same location is met. Given an  $IRT$  value of 0.05, i.e. a critical mass of 5% of the population in a particular location, we can see in the left-most plot in Figure 7.17d that there are two distinct groups (red and green) which are able to carry the disease. If the critical mass increases to 65%, i.e.  $IRT = 0.65$  (third plot from the left), we observe that the *Group by Intersection* algorithm identifies more distinct groups that can carry the disease; however, the population in each group is much smaller. Assuming the origin of a disease infection is known, the use of this algorithm can help to quickly identify populations that may be at risk.

## 7.6 Summary

In this chapter we have introduced a novel approach for the evaluation of *group-by* queries over trajectories to facilitate an OLAP-like analysis of moving object data. This approach leads naturally to a number of possible extensions (see Appendix C). We introduced the concept of the `GROUP_TRAJECTORIES group-by` operator and provided two implementations of this operator. Our approach builds on top of the established frequent pattern mining method, which is readily available in many data warehousing systems, while improving its results by making them more suitable for interactive analysis. Each of our two algorithms is designed to group trajectories that exhibit a particular type of movement, and we support this claim with a detailed experimental evaluation using synthetic and real-world datasets.

## Chapter 8

### Conclusion

In this thesis we have investigated computational methods for the design and implementation of spatial OLAP systems. The main challenges of spatial OLAP systems, when compared with traditional OLAP systems, is the integration of new spatial data types with all aspects of a typical OLAP system in a uniform manner. As such a spatial OLAP system must support the evaluation of OLAP queries over data with dimension hierarchies represented by spatial and non-spatial attribute values as well as spatial and non-spatial measures. Additionally, it must allow for the efficient storage and retrieval of spatial and non-spatial data and provide functions and operators to facilitate OLAP analysis.

In particular, our focus has been on

- the analysis of spatial dimension hierarchies;
- a pipeline-based query evaluation model for spatial OLAP;
- the indexing of records with spatial and non-spatial attribute values in an efficient and uniform manner;
- operators for the identification of groups in moving object data to support aggregate analysis.

We elaborate on each of these contributions in Section 8.1 and provide directions for future work in Section 8.2.

#### 8.1 Contributions

##### Analysis of Spatial Dimension Hierarchies

Using a simple application example, we have explored the influence of asymmetric, multiple-alternative, generalized and non-strict spatial hierarchies on the evaluation

of *roll-up* and *cube* queries and how the query results can be expressed. Our findings identify that generalized and non-strict dimension hierarchies are common in spatial data but are not readily supported by traditional OLAP models. New spatial OLAP models must account for these particular types of spatial dimension hierarchies.

### **A Pipeline-Based Query Evaluation Model for Spatial OLAP**

We introduced a pipeline-based model for the evaluation of spatial OLAP queries, which employs a uniform representation of spatial and non-spatial data while supporting all major spatial dimension hierarchy types. Our model is designed to satisfy critical requirements of spatial OLAP systems, such as performance, flexibility, and extensibility. In addition, it provides a high level of abstraction and expressiveness. A reference implementation of our pipeline-based query evaluation model called “LISA” is provided and validates the main objectives for the development of the model. LISA supports complex spatial and non-spatial OLAP queries and demonstrates high performance and scalability, particularly on modern multi-processor and multi-core hardware platforms.

### **Indexing of Records With Spatial and Non-Spatial Attribute Values**

We introduce the “geoCUBE” index—a new data structure for the indexing of records in a spatial OLAP system. It addresses the challenge of indexing attribute values defined in continuous space, while preserving locality of attribute values and records by exploiting the recursive nature of the Hilbert space filling curve. In addition, our geoCUBE index permits efficient batch updates of existing views, and we demonstrate its effectiveness and superior performance with an experimental evaluation.

### **Operators for the Identification of Groups in Moving Object Data**

We present a new class of `GROUP BY` operators that facilitate the aggregate analysis of moving object data while seamlessly fitting into established OLAP query models. Our new `GROUP_TRAJECTORIES` operators exploit data mining methodologies and transitive relationships between trajectories to identify groups of moving objects. We present two variants of the `GROUP_TRAJECTORIES` operator and show our operators can be

used to reliably distinguish groups of related trajectories when applied to synthetic and real-world moving object data.

## 8.2 Future Work

This thesis has addressed a number of fundamental problems in the design and implementation of spatial OLAP systems. It provides a concrete framework to realize a spatial OLAP system that can form the basis for future work. In the following we provide some directions for prospective research opportunities.

### Query Languages

While SQL is a powerful and flexible query language, the specific requirements of traditional OLAP have given rise to alternative query languages that specifically facilitate OLAP-type analysis, such as MDX, and XML for Analysis [124,125]. These query languages, however, have been developed for traditional OLAP analysis and do not provide the feature set required to express queries for meaningful spatial analysis. Hence, extensions to existing OLAP query languages or new query languages must be explored to allow data analysts to express complex spatial analysis objectives [22,68].

### Spatial Aggregates

Spatial OLAP systems not only support spatial attribute values, but also spatial measures. Therefore, it is important to define the aggregation of these spatial measures in order to permit aggregate analysis. Aggregate functions for spatial data types, such as *union* and *intersection*, require further research. Also, many more specialized application-specific spatial aggregate functions can be considered [39,65,112,139].

### Pre-Materialization

To address the high demands on query response time for interactive analysis, many traditional OLAP systems employ pre-materialization of all or a subset of the views in the data cube to pre-compute aggregate values and answer queries faster using these pre-materialized views. Similar techniques can likely provide notable performance savings for spatial OLAP queries, as the operations involved in spatial analysis are

often significantly more complex than for data types involved in traditional OLAP queries. There exists some initial research on this subject [139,147], but the proposed methodologies have not yet been integrated into a fully functional spatial OLAP system that supports spatial and non-spatial data.

### **Scalable Storage and Computation Platforms**

With the increasing amount of spatial and non-spatial data being collected from location-based services and mobile applications, traditional architectures for data warehouses and OLAP systems are quickly reaching their limitations. There already exists a large amount of academic [35, 109, 182] and commercial [123] work to utilize distributed and parallel architectures for traditional OLAP systems. These approaches can serve as a basis for the extension to spatial OLAP systems. Additionally, new query evaluation models, such as introduced in Chapter 4, can be designed that naturally adapt to parallel and distributed architectures.

### **User Interfaces**

One of the greatest outstanding challenges of spatial OLAP is the availability of appropriate user interfaces to support the multi-dimensional analysis of both spatial and non-spatial data. In traditional OLAP, a tabular representation of dimension attributes and numerical measures is typical, and the pivot table has become the de facto standard for navigating analysis results. Spatial data, on the other hand, presents itself naturally in the form of maps that the analyst is familiar with navigating. While some research suggests that thematic maps, which overlay non-spatial data on top of spatial data, may be a suitable method for the representation of spatial OLAP results [154, 161], it is not clear whether such an interface is sufficiently universal to support a broad variety of spatial OLAP analysis objectives.

## Appendix A

### Optimization Opportunities for LISA

This appendix provides, in some detail, directions for optimization opportunities that exists in LISA's current implementation.

#### A.1 Query Planner and Query Optimization

Although not subject of the investigation in Chapter 5, one of the current major fundamental drawbacks of our LISA implementation is the lack of an automatic query planner. A query planner uses a symbolic representation of a query, for example SQL instructions, and transforms it into a query execution plan. In addition, a query planner typically also employs a number of algorithms and heuristics in an attempt to identify an evaluation plan that satisfies specific cost constraints that might be imposed upon the query.

At this point, our implementation of LISA requires the manual construction of query plans and their translation into programs that are executable within the LISA framework. A fully automatic query planner specifically designed for LISA would have to take into account in which order the different layers involved in the query should be queried and arrange mini-engines accordingly. In addition, it should consider the foreseeable load on each mini-engine and identify appropriate points in the data flow where the use of multiple tracks could be potentially beneficial in satisfying cost constraints.

Query planners are an important subject of study and a vast amount of research in this area exists. Although the majority of this research focusses on relational database systems [79], some research exists that targets other special-purpose models [3,47,167].



## A.2 Implementation Language and Process Model

Another shortcoming of our current LISA implementation is its implementation language and the process model it employs. Although not a functional weakness, we have observed these factors to have a notable impact on LISA's performance.

LISA's current primary implementation language is Python, and Python programs are executed through interpreted byte-code. The primary objective of this language, however, is not performance but instead to provide a high-level, flexible, and feature-rich environment for rapid application development using a very clean and structured syntax. While it has suited our requirements of providing a reference implementation of LISA well, future development efforts should explore more efficient low-level implementation languages that can be compiled into native machine code, such as C or C++. Also, as the pipeline-model is closely related to functional computing models, the use of functional programming languages, such as Haskell, Erlang, or Scala may remove some complexity from the implementation.

The process model currently used by LISA does not scale well for queries involving a large number of mini-engines or that make use of a large number of processing tracks. At this point, each mini-engine is represented as an independent system process to allow the operating system to allocate different system resources to different mini-engines. However, for each process, additional overhead is incurred associated with scheduling and context switching performed by the operating system. Furthermore, inter-process communication methods, which are required for mini-engines to exchange information, contribute additional overhead due to their use of mechanisms for dealing with concurrency and signaling. Both of these issues can be addressed without undermining the fundamental idea of independent and well encapsulated mini-engines by employing a hybrid process model. Such a hybrid model can combine the use of a small number of system processes or threads, which are subject to common preemptive scheduling, and lightweight cooperatively scheduled "fibres" of execution that are not subject to the operating system scheduler. In this model, an instance of LISA can create as many system processes as there are physical processing cores, to allow the operating system to allocate one processing core to each independent system process. Within each of these "native" processes, a cooperative scheduler then executes individual mini-engines as needed. The cooperative scheduling approach is

particularly suitable for LISA, as multiple mini-engines depend on one another to produce and respectively consume records to perform their tasks. That is, a producing mini-engine can, after producing a record, voluntarily yield to other mini-engines that can consume this record. This cooperative scheduling model is very effective, as it does not require the intervention of the operating system, and it does not require expensive context switches between system processes. In addition, mini-engines that are executed within the same system process do not require inter-process communication to exchange data, as data can be passed directly through the process's own memory space. The hybrid process model still requires some inter-process communication between mini-engines that are executed within different system processes, but an intelligent and dynamic allocation of mini-engines to system processes can greatly reduce the amount of communication required and dynamically adapt to changing requirements throughout the execution of the query.

While most operating systems provide extensive support for multi-processing, multi-threading, and inter-process communication based on a preemptive scheduling model, the implementation of cooperative multitasking is commonly done at the application level. Some programming languages have native support for cooperative scheduling, typically implemented as “continuations” or “coroutines” (e.g., Ruby, Stackless Python, Python, Scala, and Erlang). For other languages that do not support this cooperative scheduling model natively, it can often be added using extension libraries, such as “Boost.Coroutine” for C++ [46], or GNU Portable Threads for C [52].

### **A.3 I/O Latency and Bandwidth**

In Section 5.2.1 we have identified that I/O latency and bandwidth have a significant impact on the query evaluation performance. Although these issues are largely inherent to the underlying hardware platform, some of them can be addressed through carefully engineered software solutions. These software approaches typically include the use of separate I/O processes, which run independently of the actual computation processes and are solely responsible for the execution of I/O operations. The benefit from using dedicated I/O processes results from the operating system not having to manage a large number of processes that compete for access to the same I/O resources.

In addition, I/O processes can implement application-specific algorithms or heuristics that optimize the sequence in which the I/O operations are executed to achieve a higher throughput from the I/O subsystem. Furthermore, the software system can be extended with shared I/O caches. These caches can be combined with dedicated I/O processes or with individual mini-engines. The advantage of using shared caches is that in situations where different mini-engines need to access the same data multiple times or the same data is accessed by multiple mini-engines, the presence of a shared cache results in data being retrieved only once and then being held in memory for subsequent accesses. This caching approach does not only reduce the latency for subsequent accesses to the same data but also reduces the load on the I/O subsystem, which can instead be used to retrieve data that has not yet been retrieved. Many database and data warehousing systems already employ dedicated I/O processes and caching mechanisms extensively, and there exists a vast selection of literature which discusses I/O optimizations for database software systems [28, 60, 99].

Other approaches for addressing the issues around I/O latency and bandwidth involve the optimization of the underlying hardware platform. The use of high-performance disk arrays that stripe data across multiple disks can greatly increase the throughput between the disk subsystem and the application processes. The use of disks that are specifically designed for low latency access, such as solid state drives (SSD), can reduce the latency on disk operations. Also, common for large installations where cost factors play an important role, are multi-tier non-volatile caches that consist of hierarchies of mass-storage devices where each hierarchy level provides different guarantees on bandwidth and latency, and data migrates through the cache hierarchies based on access frequency and other criteria.

Alternatively, other approaches to address the limitations of I/O subsystems is to distribute I/O accesses and, if possible, computations across multiple systems. The pipeline-based query evaluations is well suited for a distributed computation model in which each system contributes only a part of the overall result and multiple systems work in parallel sharing only a small amount of resources (shared-nothing model) [29, 119, 171].

#### A.4 Storage Models, Access Methods, and Indexing

Our current implementation of LISA uses a mixture of different storage models and access methods depending on the type of data being stored. Non-spatial data is currently stored in SQLite databases and indexed using SQLite's B-tree. SQLite's architecture manages data effectively and employs caching mechanisms appropriate for its target domain [89]. However, SQLite's process model is inherently based on a single process, and multiple processes that share access to a single SQLite database do not share any caches or information that can aid them in collectively optimizing access to the data. In addition, the access to data stored in SQLite through the SQL query language imposes an additional overhead, as each query must be parsed and evaluated by SQLite's query evaluator, even though LISA only uses a subset of SQLite's SQL capabilities. A more effective way for LISA to store and access non-spatial data may be a storage model natively implemented as part of LISA, in the spirit of the one currently used for spatial data. Such a storage model can store records of one entity in an ordered fashion in a flat binary data file and in addition construct B-tree indices over individual attributes to quickly locate records in the data file. This approach removes the overhead of parsing each query on an entity using a fully featured SQL parser. In addition, access to the data file can be block-based instead of record-based, which could result in the aggregation of multiple record requests into a single I/O operation. Furthermore, LISA can use dedicated I/O processes, as discussed in Section A.3, to provide additional optimizations of I/O operations and facilities for shared block caches. At this point, LISA is solely intended as a read-only system, which can greatly simplify the implementation of alternative storage models with a much greater performance.

For spatial data, LISA already uses its own storage model consisting of a data file that stores the records and an R-tree index to efficiently locate records in the data file. The R-tree implementation is provided through the Spatial Index library [77] and uses page-based access and caching methods. However, accesses to the data file are not explicitly cached other than through the operating system. In Section 5.2.2 we have identified that spatial objects can greatly vary in their complexity and that the performance of some operations on spatial objects is significantly affected by the objects' complexity. This provides motivation to investigate alternative storage models

and indexing structures for spatial data, specifically targeted to operations performed within LISA. For example, to calculate the intersection of two spatial objects that only partially overlap, not all vertices of the objects need to be retrieved, but only those that form the overlapping region. One method that potentially improves the performance of such operations is to store the vertices of the spatial objects in a hierarchical data structure, e.g., quadtree [57], and only retrieve data from those paths of the hierarchy for which the objects overlap. The retrieved vertices then form fragments of the corresponding spatial objects that are sufficient to calculate their intersection. There exists a large amount of literature discussing data structures for spatial applications, which can serve as a starting point for future work in this area [59]. As discussed for non-spatial data storage models, spatial data storage models can similarly benefit from block-based I/O access and shared caches.

### **A.5 Integration with Existing Data Sources**

Last but not least, one shortcoming of our current LISA implementation that constrains its use in real-world applications is its lack of integration with existing and external data sources. Often times, the data on which OLAP or spatial OLAP queries are to be executed is already present in existing databases or data warehouses. LISA should be able to directly interface with such data sources or provide mechanisms to convert such data into LISA's native data storage models. To support such functionality, LISA can be extended with adaptors to standardized interfaces, such as the Open Database Connectivity standard [94], and use SQL for the retrieval of data. In addition, a generic adaptor API can be provided which allows users of LISA to develop their own adaptors to legacy systems.

## Appendix B

# Implementation of Example Queries in LISA

## B.1 Query 1

```
1 #####
2 #
3 # Query 1
4 #
5 # SELECT species.id, MAX(plants.height)
6 # FROM species
7 # LEFT JOIN plants ON plants.species_id = species.id
8 # WHERE plants.age >= 10 AND plants.age <= 50
9 # GROUP BY species.id;
10 #
11 #####
12
13 # Schema definition of the query stream: an interval across all species
14 # IDs.
15 query_schema = Schema()
16 query_schema.append(Attribute('species.id', IntInterval))
17
18 # Schema definition of the species record stream.
19 species_schema = Schema()
20 species_schema.append(Attribute('species.id', int))
21
22 # Schema definition of the plant record stream.
23 plants_schema = Schema()
24 plants_schema.append(Attribute('plants.id', int))
25 plants_schema.append(Attribute('plants.height', int))
26 plants_schema.append(Attribute('plants.age', int))
27 plants_schema.append(Attribute('plants.species_id', int, True))
28
29 # Filter plants to only include those 10 years or older and 50 years or
30 # younger.
31 class FilterAge(object):
32     def __init__(self, input_schema):
33         self._input_schema = input_schema
34         self._p = [
35             (
36                 input_schema.index('plants.age'),
37                 lambda x: x >= 10 and x <= 50
38             ),
39         ]
40
41     def accepts(self, other_schema):
42         return self._input_schema == other_schema
43
44     def __call__(self, r):
45         for p in self._p:
46             if not p[1](r[p[0]]):
47                 return False
48         return True
49
50 # Aggregation function for max height.
```

```

51 class MaxHeightAggregator(object):
52     def __init__(self, input_schema):
53         self._input_schema = input_schema
54         self._af = []
55         for a in self._input_schema:
56             if a.name() == 'plants.height':
57                 # Only keep the maximum
58                 self._af.append((
59                     0,
60                     lambda x, v: x >= v and x or v,
61                 ))
62             else:
63                 # Everything else keep as is
64                 self._af.append((
65                     None,
66                     lambda x, v: v,
67                 ))
68
69     def accepts(self, other):
70         return self._input_schema == other
71
72     def init(self):
73         '''
74         Initializes and resets the aggregation value.
75         '''
76         self._c = list(af[0] for af in self._af)
77         self._calls = 0
78
79     def record(self):
80         '''
81         Returns the record that represents the current aggregation value.
82         '''
83         return tuple(self._c)
84
85     def count(self):
86         return self._calls
87
88     def __call__(self, r):
89         '''
90         Adds the specified record to the aggregate value.
91         '''
92         self._calls += 1
93         for i, c in enumerate(self._c):
94             self._c[i] = self._af[i][1](c, r[i])
95
96     engines = []
97     # The query stream contains only a single query.
98     query_streamer = ArrayStreamer(query_schema, [
99         (IntInterval(0, int(1E10)),),
100     ])
101     engines.append(query_streamer)
102
103     # Create a species data source: a table in the input database.
104     species_source = DBTable(input_file, 'species', species_schema)
105     # Data accessor for the species data source.
106     species_accessor = DataAccessor(
107         query_streamer.output(),
108         species_source,
109         FindRange
110     )
111     engines.append(species_accessor)
112
113     demux = Demux(species_accessor.output())
114     engines.append(demux)
115
116     mux_streams = []
117     for i in range(tracks):

```

```

118     channel = demux.channel()
119
120     # Select only the species ID for querying plants.
121     species_id_select = Select(
122         channel,
123         UniversalSelect(
124             species_accessor.output().schema(),
125             {
126                 'plants.species_id': {
127                     'type': int,
128                     'args': ['species.id'],
129                     'function': lambda v: v
130                 }
131             }
132         )
133     )
134     engines.append(species_id_select)
135     # Data source for the plants.
136     plants_source = DBTable(input_file, 'plants', plants_schema)
137     # Data accessor for the plants data source.
138     plants_accessor = DataAccessor(
139         species_id_select.output(),
140         plants_source,
141         FindIdentities
142     )
143     engines.append(plants_accessor)
144
145     plants_filter = Filter(
146         plants_accessor.output(),
147         FilterAge(plants_accessor.output().schema())
148     )
149     engines.append(plants_filter)
150
151     # Select only the species ID for querying plants.
152     plants_height_select = Select(
153         plants_filter.output(),
154         UniversalSelect(
155             plants_filter.output().schema(),
156             {
157                 'plants.height': {
158                     'type': int,
159                     'args': ['plants.height'],
160                     'function': lambda v: v
161                 }
162             }
163         )
164     )
165     engines.append(plants_height_select)
166
167     plants_height_aggregate = Aggregate(
168         plants_height_select.output(),
169         MaxHeightAggregator(plants_height_select.output().schema())
170     )
171     engines.append(plants_height_aggregate)
172
173     species_id_grouper = Group(
174         channel,
175         {'species.id': lambda a, b: a == b}
176     )
177     engines.append(species_id_grouper)
178
179     joiner = Join(species_id_grouper.output(), plants_height_aggregate.output())
180     engines.append(joiner)
181     mux_streams.append(joiner.output())
182
183     mux = Mux(*mux_streams)
184     engines.append(mux)

```



```

185
186 result_file = ResultFile(
187     'query1-results.txt',
188     mux.output(),
189 )
190 engines.append(result_file)

```

## B.2 Query 2

```

1 #####
2 #
3 # Query 2
4 #
5 # SELECT nation.id, customer.id, orders.id, MAX(lineitem.price)
6 # FROM nation
7 # LEFT JOIN customer ON customer.nation_id = nation.id
8 # LEFT JOIN orders ON orders.customer_id = customer.id
9 # LEFT JOIN lineitem ON lineitem.order_id = orders.id
10 # WHERE lineitem.quantity >= 10 AND lineitem.quantity <= 15
11 # GROUP BY ROLLUP(nation.id, customer.id, orders.id)
12 #
13 #####
14
15 # Schema definition of the query stream: an interval across all families.
16 query_schema = Schema()
17 query_schema.append(Attribute('nation.id', IntInterval))
18
19 # Schema definition of the nation record stream.
20 nation_schema = Schema()
21 nation_schema.append(Attribute('nation.id', int))
22
23 # Schema definitions of the customer record stream.
24 customer_schema = Schema()
25 customer_schema.append(Attribute('customer.id', int))
26 customer_schema.append(Attribute('customer.nation_id', int, True))
27
28 # Schema definitions of the orders record stream.
29 orders_schema = Schema()
30 orders_schema.append(Attribute('orders.id', int))
31 orders_schema.append(Attribute('orders.customer_id', int, True))
32
33 # Schema definition of the lineitem record stream.
34 lineitem_schema = Schema()
35 # lineitem_schema.append(Attribute('lineitem.id', int))
36 lineitem_schema.append(Attribute('lineitem.quantity', int))
37 lineitem_schema.append(Attribute('lineitem.price', float))
38 lineitem_schema.append(Attribute('lineitem.order_id', int, True))
39
40 # Filter lineitem to only include those 10 years or older and 50 years or
41 # younger.
42 class FilterQuantity(object):
43     def __init__(self, input_schema):
44         self._input_schema = input_schema
45         self._p = [
46             (
47                 input_schema.index('lineitem.quantity'),
48                 lambda x: x >= 10 and x <= 15
49             ),
50         ]
51
52     def accepts(self, other_schema):
53         return self._input_schema == other_schema
54

```

```

55     def __call__(self, r):
56         for p in self._p:
57             if not p[1](r[p[0]]):
58                 return False
59         return True
60
61     # Aggregation function for max price.
62     class MaxPriceAggregator(object):
63         def __init__(self, input_schema):
64             self._input_schema = input_schema
65             self._af = []
66             for a in self._input_schema:
67                 if a.name() == 'lineitem.price':
68                     # Only keep the maximum
69                     self._af.append((
70                         0,
71                         lambda x, v: x >= v and x or v,
72                     ))
73                 else:
74                     # Everything else keep as is
75                     self._af.append((
76                         None,
77                         lambda x, v: v,
78                     ))
79
80         def accepts(self, other):
81             return self._input_schema == other
82
83         def init(self):
84             '''
85             Initializes and resets the aggregation value.
86             '''
87             self._c = list(af[0] for af in self._af)
88             self._calls = 0
89
90         def record(self):
91             '''
92             Returns the record that represents the current aggregation value.
93             '''
94             return tuple(self._c)
95
96         def count(self):
97             return self._calls
98
99         def __call__(self, r):
100             '''
101             Adds the specified record to the aggregate value.
102             '''
103             self._calls += 1
104             for i, c in enumerate(self._c):
105                 self._c[i] = self._af[i][1](c, r[i])
106
107     engines = []
108
109     # The query stream contains only a single query.
110     query_streamer = ArrayStreamer(query_schema, [
111         (IntInterval(0, int(1E10)),),
112     ])
113     engines.append(query_streamer)
114
115     # Create a nation data source: a table in the input database.
116     nation_source = DBTable(input_file, 'nation', nation_schema)
117     # Data accessor for the orders data source.
118     nation_accessor = DataAccessor(
119         query_streamer.output(),
120         nation_source,
121         FindRange

```

```

122 )
123 engines.append(nation_accessor)
124
125 # A group mini-engine to split the nation IDs into groups.
126 nation_id_grouper = Group(
127     nation_accessor.output(),
128     {'nation.id': lambda a, b: a == b}
129 )
130 engines.append(nation_id_grouper)
131
132 # Select only the nation ID for querying genera.
133 nation_id_select = Select(
134     nation_accessor.output(),
135     UniversalSelect(
136         nation_accessor.output().schema(),
137         {
138             'customer.nation_id': {
139                 'type': int,
140                 'args': ['nation.id'],
141                 'function': lambda v: v
142             }
143         }
144     )
145 )
146 engines.append(nation_id_select)
147
148
149 # Data source for the genera.
150 customer_source = DBTable(input_file, 'customer', customer_schema)
151
152
153 # Data accessor for the genera data source.
154 customer_accessor = DataAccessor(
155     nation_id_select.output(),
156     customer_source,
157     FindIdentities
158 )
159 engines.append(customer_accessor)
160
161
162 # A join mini-engine to associate families with genera.
163 nation_customer_joiner = Join(
164     nation_id_grouper.output(),
165     customer_accessor.output(),
166 )
167 engines.append(nation_customer_joiner)
168
169
170 # A group mini-engine to split the (nation, customer) IDs into groups.
171 nation_customer_id_grouper = Group(
172     nation_customer_joiner.output(),
173     {
174         'nation.id': lambda a, b: a == b,
175         'customer.id': lambda a, b: a == b
176     },
177 )
178 engines.append(nation_customer_id_grouper)
179
180
181 # Select only the customer ID for querying orders.
182 customer_id_select = Select(
183     nation_customer_joiner.output(),
184     UniversalSelect(
185         nation_customer_joiner.output().schema(),
186         {
187             'orders.customer_id': {
188                 'type': int,

```

```

189         'args': ['customer.id'],
190         'function': lambda v: v
191     }
192 }
193 )
194 )
195 engines.append(customer_id_select)
196
197
198 # Data source for the orders.
199 orders_source = DBTable(input_file, 'orders', orders_schema)
200
201
202 # Data accessor for the orders data source.
203 orders_accessor = DataAccessor(
204     customer_id_select.output(),
205     orders_source,
206     FindIdentities
207 )
208 engines.append(orders_accessor)
209
210
211 # A join mini-engine to associate families, genera and orders.
212 nation_customer_orders_joiner = Join(
213     nation_customer_id_grouper.output(),
214     orders_accessor.output(),
215 )
216 engines.append(nation_customer_orders_joiner)
217
218 demux = Demux(nation_customer_orders_joiner.output())
219 engines.append(demux)
220
221 mux_streams = []
222 for i in range(tracks):
223     channel = demux.channel()
224
225     # Select only the orders ID for querying lineitem.
226     orders_id_select = Select(
227         channel,
228         UniversalSelect(
229             channel.schema(),
230             {
231                 'lineitem.order_id': {
232                     'type': int,
233                     'args': ['orders.id'],
234                     'function': lambda v: v
235                 }
236             }
237         )
238     )
239     engines.append(orders_id_select)
240
241     # Data source for the lineitem.
242     lineitem_source = DBTable(input_file, 'lineitem', lineitem_schema)
243     # Data accessor for the lineitem data source.
244     lineitem_accessor = DataAccessor(
245         orders_id_select.output(),
246         lineitem_source,
247         FindIdentities
248     )
249     engines.append(lineitem_accessor)
250
251     lineitem_filter = Filter(
252         lineitem_accessor.output(),
253         FilterQuantity(lineitem_accessor.output().schema())
254     )
255     engines.append(lineitem_filter)

```

```

256
257 # Select only the orders ID for querying lineitem.
258 lineitem_price_select = Select(
259     lineitem_filter.output(),
260     UniversalSelect(
261         lineitem_filter.output().schema(),
262         {
263             'lineitem.price': {
264                 'type': int,
265                 'args': ['lineitem.price'],
266                 'function': lambda v: v
267             }
268         }
269     )
270 )
271 engines.append(lineitem_price_select)
272
273 lineitem_price_aggregate = Aggregate(
274     lineitem_price_select.output(),
275     MaxPriceAggregator(lineitem_price_select.output().schema())
276 )
277 engines.append(lineitem_price_aggregate)
278
279 nation_customer_orders_id_grouper = Group(
280     channel,
281     {
282         'nation.id': lambda a, b: a == b,
283         'customer.id': lambda a, b: a == b,
284         'orders.id': lambda a, b: a == b
285     }
286 )
287 engines.append(nation_customer_orders_id_grouper)
288 # mux_streams.append(nation_customer_orders_id_grouper.output())
289
290 orders_lineitem_joiner = Join(
291     nation_customer_orders_id_grouper.output(),
292     lineitem_price_aggregate.output()
293 )
294 engines.append(orders_lineitem_joiner)
295 mux_streams.append(orders_lineitem_joiner.output())
296
297 mux = Mux(*mux_streams)
298 engines.append(mux)
299
300
301 # First aggregation level output selection
302 nation_customer_orders_select = Select(
303     mux.output(),
304     UniversalSelect(
305         mux.output().schema(),
306         [
307             ('nation.id', {
308                 'type': int,
309                 'args': ['nation.id'],
310                 'function': lambda v: v
311             }),
312             ('customer.id', {
313                 'type': int,
314                 'args': ['customer.id'],
315                 'function': lambda v: v
316             }),
317             ('orders.id', {
318                 'type': int,
319                 'args': ['orders.id'],
320                 'function': lambda v: v
321             }),
322             ('lineitem.price', {

```

```

323         'type': int,
324         'args': ['lineitem.price'],
325         'function': lambda v: v
326     }),
327 ]
328 )
329 )
330 engines.append(nation_customer_orders_select)
331
332 # Second aggregation level output selection
333 nation_customer_select = Select(
334     nation_customer_orders_select.output(),
335     UniversalSelect(
336         nation_customer_orders_select.output().schema(),
337         [
338             ('nation.id', {
339                 'type': int,
340                 'args': ['nation.id'],
341                 'function': lambda v: v
342             }),
343             ('customer.id', {
344                 'type': int,
345                 'args': ['customer.id'],
346                 'function': lambda v: v
347             }),
348             ('lineitem.price', {
349                 'type': int,
350                 'args': ['lineitem.price'],
351                 'function': lambda v: v
352             }),
353         ]
354     )
355 )
356 engines.append(nation_customer_select)
357
358 nation_customer_sorter = Sort(
359     nation_customer_select.output(),
360     [
361         ('nation.id', lambda a, b: cmp(a, b)),
362         ('customer.id', lambda a, b: cmp(a, b))
363     ],
364     True # sort all input, not only the current partition
365 )
366 engines.append(nation_customer_sorter)
367
368 # Generate aggregation groups for second aggregation level.
369 nation_customer_grouper = Group(
370     nation_customer_sorter.output(),
371     {
372         'nation.id': lambda a, b: a == b,
373         'customer.id': lambda a, b: a == b
374     },
375 )
376 engines.append(nation_customer_grouper)
377
378 # Aggregate second level
379 nation_customer_aggregate = Aggregate(
380     nation_customer_grouper.output(),
381     MaxPriceAggregator(nation_customer_grouper.output().schema())
382 )
383 engines.append(nation_customer_aggregate)
384
385 # Third aggregation level output selection
386 nation_select = Select(
387     nation_customer_aggregate.output(),
388     UniversalSelect(
389         nation_customer_aggregate.output().schema(),

```

```

390     [
391         ('nation.id', {
392             'type': int,
393             'args': ['nation.id'],
394             'function': lambda v: v
395         }),
396         ('lineitem.price', {
397             'type': int,
398             'args': ['lineitem.price'],
399             'function': lambda v: v
400         }),
401     ]
402 )
403 )
404 engines.append(nation_select)
405
406 # Generate aggregation groups for third aggregation level.
407 nation_grouper = Group(
408     nation_select.output(),
409     {
410         'nation.id': lambda a, b: a == b,
411     },
412 )
413 engines.append(nation_grouper)
414
415 # Aggregate third level
416 nation_aggregate = Aggregate(
417     nation_grouper.output(),
418     MaxPriceAggregator(nation_grouper.output().schema())
419 )
420 engines.append(nation_aggregate)
421
422 # Fourth aggregation level output selection
423 all_select = Select(
424     nation_aggregate.output(),
425     UniversalSelect(
426         nation_aggregate.output().schema(),
427         {
428             'lineitem.price': {
429                 'type': float,
430                 'args': ['lineitem.price'],
431                 'function': lambda v: v
432             },
433         }
434     )
435 )
436 engines.append(all_select)
437
438 # Generate aggregation groups for fourth aggregation level.
439 all_grouper = Group(
440     all_select.output(),
441     {
442     },
443 )
444 engines.append(all_grouper)
445
446 # Aggregate fourth level
447 all_aggregate = Aggregate(
448     all_grouper.output(),
449     MaxPriceAggregator(all_grouper.output().schema())
450 )
451 engines.append(all_aggregate)
452
453 result_file = ResultFile(
454     'query6-results.txt',
455     nation_customer_orders_select.output(),
456     nation_customer_aggregate.output(),

```

```

457     nation_aggregate.output(),
458     all_aggregate.output(),
459 )
460 engines.append(result_file)

```

### B.3 Query 3

```

1  #####
2  #
3  # Query 3
4  #
5  # SELECT counties.id, COUNT(geonames.*) FROM counties
6  # LEFT JOIN geonames ON CONTAINS(counties.geom, geonames.location)
7  # WHERE
8  #   CONTAINS(
9  #     MakeBox2D(
10 #       MakePoint(-93.88, 49.81),
11 #       MakePoint(-65.39, 24.22)
12 #     ),
13 #     geonames.location
14 #   )
15 # GROUP BY counties.id;
16 #
17 #####
18
19 # Schema definition of the query stream: an interval across all counties.
20 query_schema = Schema()
21 query_schema.append(Attribute('counties.geom', Geometry))
22
23 # Aggregation function for max height.
24 class SumAggregator(object):
25     def __init__(self, input_schema, f):
26         self._input_schema = input_schema
27         self._af = []
28         for a in self._input_schema:
29             if a.name() == f:
30                 # Only keep the maximum
31                 self._af.append((
32                     0,
33                     lambda x, v: x + v,
34                 ))
35             else:
36                 # Everything else keep as is
37                 self._af.append((
38                     None,
39                     lambda x, v: v,
40                 ))
41
42     def accepts(self, other):
43         return self._input_schema == other
44
45     def init(self):
46         '''
47         Initializes and resets the aggregation value.
48         '''
49         self._c = list(af[0] for af in self._af)
50         self._calls = 0
51
52     def record(self):
53         '''
54         Returns the record that represents the current aggregation value.
55         '''
56         return tuple(self._c)

```



```

57
58     def count(self):
59         return self._calls
60
61     def __call__(self, r):
62         '''
63         Adds the specified record to the aggregate value.
64         '''
65         self._calls += 1
66         for i, c in enumerate(self._c):
67             self._c[i] = self._af[i][1](c, r[i])
68
69     engines = []
70
71     # The query stream contains only a single query box.
72     query_streamer = ArrayStreamer(query_schema, [
73         (query,),
74     ])
75     engines.append(query_streamer)
76
77     counties_source = Rtree(counties_file, 'counties.geom')
78     counties_accessor = DataAccessor(
79         query_streamer.output(),
80         counties_source,
81         FindRange,
82     )
83     engines.append(counties_accessor)
84
85     demux = Demux(counties_accessor.output())
86     engines.append(demux)
87
88     def intersection(a, b):
89         g1 = a.geom()
90         g2 = b.geom()
91         try:
92             if g1.is_valid and g2.is_valid:
93                 i = g1.intersection(g2)
94                 return Geometry(i)
95             else:
96                 return None
97         except:
98             return None
99
100     mux_streams = []
101     for i in range(tracks):
102         channel = demux.channel()
103
104         # To query the locations in the geonames layer, trim the counties to
105         # the query.
106         counties_select = Select(
107             channel,
108             UniversalSelect(
109                 channel.schema(),
110                 {
111                     'geonames.location': {
112                         'type': Geometry,
113                         'args': ['counties.geom'],
114                         'function': lambda v: intersection(v, query),
115                     }
116                 }
117             )
118         )
119         engines.append(counties_select)
120
121     geonames_source = Rtree(geonames_file, 'geonames.location')
122     # Data accessor for the geonames.
123     geonames_accessor = DataAccessor(

```

```

124     counties_select.output(),
125     geonames_source,
126     FindRange
127 )
128 engines.append(geonames_accessor)
129
130 # XXX At this point no additional filter for the constraining the
131 # geonames to the query region is required.
132
133 # Send '1' for each retrieved geoname location.
134 geonames_select = Select(
135     geonames_accessor.output(),
136     UniversalSelect(
137         geonames_accessor.output().schema(),
138         {
139             'count': {
140                 'type': int,
141                 'args': ['geonames.location'],
142                 'function': lambda v: 1
143             }
144         }
145     )
146 )
147 engines.append(geonames_select)
148
149 geonames_aggregate = Aggregate(
150     geonames_select.output(),
151     SumAggregator(geonames_select.output().schema(), 'count')
152 )
153 engines.append(geonames_aggregate)
154
155 select = Select(
156     channel,
157     UniversalSelect(
158         channel.schema(),
159         {
160             'oid': {
161                 'type': int,
162                 'args': ['oid'],
163                 'function': lambda v: v
164             }
165         }
166     )
167 )
168 engines.append(select)
169
170 counties_grouper = Group(
171     select.output(),
172     {'oid': lambda a, b: a == b}
173 )
174 engines.append(counties_grouper)
175
176 joiner = Join(counties_grouper.output(), geonames_aggregate.output())
177 engines.append(joiner)
178 mux_streams.append(joiner.output())
179
180 mux = Mux(*mux_streams)
181 engines.append(mux)
182
183 result_file = ResultFile(
184     'query3-results.txt',
185     mux.output(),
186 )
187 engines.append(result_file)

```

## B.4 Query 4

```

1 #####
2 #
3 # Query 4
4 #
5 # SELECT states.id, counties.id, COUNT(geonames.*) FROM states
6 # LEFT JOIN counties
7 #   ON CONTAINS(states.geom, counties.geom)
8 # LEFT JOIN geonames ON CONTAINS(counties.geom, geonames.location)
9 # WHERE
10 #   CONTAINS(
11 #     MakeBox2D(
12 #       MakePoint(-93.88, 49.81),
13 #       MakePoint(-65.39, 24.22)
14 #     ),
15 #     geonames.location
16 #   )
17 # GROUP BY ROLLUP(states.id, counties.id);
18 #
19 #####
20
21 # Schema definition of the query stream: an interval across all states.
22 query_schema = Schema()
23 query_schema.append(Attribute('states.geom', Geometry))
24
25 # Aggregation function for max height.
26 class SumAggregator(object):
27     def __init__(self, input_schema, f):
28         self._input_schema = input_schema
29         self._af = []
30         for a in self._input_schema:
31             if a.name() == f:
32                 # Only keep the maximum
33                 self._af.append((
34                     0,
35                     lambda x, v: x + v,
36                 ))
37             else:
38                 # Everything else keep as is
39                 self._af.append((
40                     None,
41                     lambda x, v: v,
42                 ))
43
44     def accepts(self, other):
45         return self._input_schema == other
46
47     def init(self):
48         '''
49         Initializes and resets the aggregation value.
50         '''
51         self._c = list(af[0] for af in self._af)
52         self._calls = 0
53
54     def record(self):
55         '''
56         Returns the record that represents the current aggregation value.
57         '''
58         return tuple(self._c)
59
60     def count(self):
61         return self._calls
62

```

```

63     def __call__(self, r):
64         '''
65         Adds the specified record to the aggregate value.
66         '''
67         self._calls += 1
68         for i, c in enumerate(self._c):
69             self._c[i] = self._af[i][1](c, r[i])
70
71     # Helper function to compute the intersection between two geometries.
72     def intersection(a, b):
73         g1 = a.geom()
74         g2 = b.geom()
75         try:
76             if g1.is_valid and g2.is_valid:
77                 i = g1.intersection(g2)
78                 return Geometry(i)
79             else:
80                 return None
81         except:
82             return None
83
84     engines = []
85     counters = []
86
87     # The query stream contains only a single query box.
88     query_streamer = ArrayStreamer(query_schema, [
89         (query,),
90     ])
91     engines.append(query_streamer)
92
93     # Query the states from the data source.
94     states_source = Rtree(states_file, 'states.geom')
95     states_accessor = DataAccessor(
96         query_streamer.output(),
97         states_source,
98         FindRange
99     )
100    engines.append(states_accessor)
101
102    # Trim the states to the query region.
103    states_select = Select(
104        states_accessor.output(),
105        UniversalSelect(
106            states_accessor.output().schema(),
107            {
108                # trim geometry
109                'states.geom': {
110                    'type': Geometry,
111                    'args': ['states.geom'],
112                    'function': lambda v: intersection(v, query),
113                },
114                # keep OID
115                'states.oid': {
116                    'type': int,
117                    'args': ['oid'],
118                    'function': lambda v: v,
119                }
120            }
121        )
122    )
123    engines.append(states_select)
124
125    # Only keep the geometry for querying
126    states_query = Select(
127        states_select.output(),
128        UniversalSelect(
129            states_select.output().schema(),

```

```

130         {
131             'counties.geom': {
132                 'type': Geometry,
133                 'args': ['states.geom'],
134                 'function': lambda v: v,
135             },
136         }
137     )
138 )
139 engines.append(states_query)
140
141 # Finally query the counties
142 counties_source = Rtree(counties_file, 'counties.geom')
143 counties_accessor = DataAccessor(
144     states_query.output(),
145     counties_source,
146     FindRange,
147 )
148 engines.append(counties_accessor)
149
150 # Rename the OID attribute of the counties
151 counties_oid_select = Select(
152     counties_accessor.output(),
153     UniversalSelect(
154         counties_accessor.output().schema(),
155         {
156             'counties.oid': {
157                 'type': int,
158                 'args': ['oid'],
159                 'function': lambda v: v,
160             },
161             'counties.geom': {
162                 'type': Geometry,
163                 'args': ['counties.geom'],
164                 'function': lambda v: v,
165             },
166         }
167     )
168 )
169 engines.append(counties_oid_select)
170
171 # Group states by OID
172 states_group = Group(
173     states_select.output(),
174     {'states.oid': lambda a, b: a == b}
175 )
176 engines.append(states_group)
177
178 # Join counties and states
179 states_counties_join = Join(
180     states_group.output(),
181     counties_oid_select.output(),
182 )
183 engines.append(states_counties_join)
184
185 # De-multiplex the joined stream across multiple tracks for better CPU core
186 # utilization.
187 demux = Demux(states_counties_join.output())
188 engines.append(demux)
189
190 mux_streams = []
191 for i in range(tracks):
192     channel = demux.channel()
193
194     # To query the locations in the geonames layer, trim the counties to
195     # the state and query boundary.
196     counties_select = Select(

```

```

197     channel,
198     UniversalSelect(
199         channel.schema(),
200         {
201             'geonames.location': {
202                 'type': Geometry,
203                 'args': ['states.geom', 'counties.geom'],
204                 'function': lambda s, c: intersection(s, c),
205             }
206         }
207     )
208 )
209 engines.append(counties_select)
210
211 # Data source for geonames
212 geonames_source = Rtree(geonames_file, 'geonames.location')
213 # Data accessor for the geonames.
214 geonames_accessor = DataAccessor(
215     counties_select.output(),
216     geonames_source,
217     FindRange
218 )
219 engines.append(geonames_accessor)
220
221 # XXX At this point no additional filter for the constraining the
222 # geonames to the query region is required.
223
224 # Send '1' for each retrieved geoname location.
225 geonames_select = Select(
226     geonames_accessor.output(),
227     UniversalSelect(
228         geonames_accessor.output().schema(),
229         {
230             'count': {
231                 'type': int,
232                 'args': ['geonames.location'],
233                 'function': lambda v: 1
234             }
235         }
236     )
237 )
238 engines.append(geonames_select)
239
240 # Aggregate the geonames
241 geonames_aggregate = Aggregate(
242     geonames_select.output(),
243     SumAggregator(geonames_select.output().schema(), 'count')
244 )
245 engines.append(geonames_aggregate)
246
247 # Select only the OIDs from each of the hierarchy levels.
248 select = Select(
249     channel,
250     UniversalSelect(
251         channel.schema(),
252         {
253             'states.oid': {
254                 'type': int,
255                 'args': ['states.oid'],
256                 'function': lambda v: v
257             },
258             'counties.oid': {
259                 'type': int,
260                 'args': ['counties.oid'],
261                 'function': lambda v: v
262             }
263         }

```

```

264     )
265   )
266   engines.append(select)
267
268   # Generate appropriate groups
269   states_counties_grouper = Group(
270     select.output(),
271     {
272       'states.oid': lambda a, b: a == b,
273       'counties.oid': lambda a, b: a == b
274     }
275   )
276   engines.append(states_counties_grouper)
277
278   joiner = Join(
279     states_counties_grouper.output(),
280     geonames_aggregate.output()
281   )
282   engines.append(joiner)
283   mux_streams.append(joiner.output())
284
285   mux = Mux(*mux_streams)
286   engines.append(mux)
287
288   states_level_select = Select(
289     mux.output(),
290     UniversalSelect(
291       mux.output().schema(),
292       {
293         'states.oid': {
294           'type': int,
295           'args': ['states.oid'],
296           'function': lambda v: v,
297         },
298         'count': {
299           'type': int,
300           'args': ['count'],
301           'function': lambda v: v,
302         }
303       }
304     )
305   )
306   engines.append(states_level_select)
307
308   states_ungroup = Group(
309     states_level_select.output(),
310     {
311     }
312   )
313   engines.append(states_ungroup)
314
315   states_sort = Sort(
316     states_ungroup.output(),
317     [
318       ('states.oid', None)
319     ]
320   )
321   engines.append(states_sort)
322
323   states_level_group = Group(
324     states_sort.output(),
325     {
326       'states.oid': lambda a, b: a == b,
327     }
328   )
329   engines.append(states_level_group)
330

```

```

331 # Aggregate second level
332 states_level_aggregate = Aggregate(
333     states_level_group.output(),
334     SumAggregator(states_level_group.output().schema(), 'count')
335 )
336 engines.append(states_level_aggregate)
337
338 all_level_select = Select(
339     states_level_aggregate.output(),
340     UniversalSelect(
341         states_level_aggregate.output().schema(),
342         {
343             'count': {
344                 'type': int,
345                 'args': ['count'],
346                 'function': lambda v: v,
347             }
348         }
349     )
350 )
351 engines.append(all_level_select)
352
353 all_group = Group(
354     all_level_select.output(),
355     {
356     }
357 )
358 engines.append(all_group)
359
360 # Aggregate third level
361 all_level_aggregate = Aggregate(
362     all_group.output(),
363     SumAggregator(all_level_select.output().schema(), 'count')
364 )
365 engines.append(all_level_aggregate)
366
367
368 output_level1_attr = Select(
369     mux.output(),
370     UniversalSelect(
371         mux.output().schema(),
372         [
373             ('states.oid', {
374                 'type': int,
375                 'args': ['states.oid'],
376                 'function': lambda v: v,
377             }),
378             ('counties.oid', {
379                 'type': int,
380                 'args': ['counties.oid'],
381                 'function': lambda v: v,
382             }),
383             ('count', {
384                 'type': int,
385                 'args': ['count'],
386                 'function': lambda v: v,
387             }),
388         ]
389     )
390 )
391 engines.append(output_level1_attr)
392
393 output_level2_attr = Select(
394     states_level_aggregate.output(),
395     UniversalSelect(
396         states_level_aggregate.output().schema(),
397     ]

```



```

398         ('states.oid', {
399             'type': int,
400             'args': ['states.oid'],
401             'function': lambda v: v,
402         }),
403         ('count', {
404             'type': int,
405             'args': ['count'],
406             'function': lambda v: v,
407         }),
408     ]
409 )
410 )
411 engines.append(output_level2_attr)
412
413 output_level3_attr = Select(
414     all_level_aggregate.output(),
415     UniversalSelect(
416         all_level_aggregate.output().schema(),
417         [
418             ('count', {
419                 'type': int,
420                 'args': ['count'],
421                 'function': lambda v: v,
422             }),
423         ]
424     )
425 )
426 engines.append(output_level3_attr)
427
428 # Output
429 result_file = ResultFile(
430     'query4-results.txt',
431     output_level1_attr.output(),
432     output_level2_attr.output(),
433     output_level3_attr.output(),
434 )
435 engines.append(result_file)

```

## B.5 Query 5

```

1 #####
2 #
3 # Query 5
4 #
5 #####
6
7 # Schema definition of the query stream.
8 query_schema = Schema()
9 query_schema.append(Attribute('queries.geom', Geometry))
10
11 # Aggregation function for max height.
12 class SumAggregator(object):
13     def __init__(self, input_schema, f):
14         self._input_schema = input_schema
15         self._af = []
16         for a in self._input_schema:
17             if a.name() == f:
18                 # Only keep the maximum
19                 self._af.append((
20                     0,
21                     lambda x, v: x + v,
22                 ))

```

```

23         else:
24             # Everything else keep as is
25             self._af.append((
26                 None,
27                 lambda x, v: v,
28             ))
29
30     def accepts(self, other):
31         return self._input_schema == other
32
33     def init(self):
34         '''
35         Initializes and resets the aggregation value.
36         '''
37         self._c = list(af[0] for af in self._af)
38         self._calls = 0
39
40     def record(self):
41         '''
42         Returns the record that represents the current aggregation value.
43         '''
44         return tuple(self._c)
45
46     def count(self):
47         return self._calls
48
49     def __call__(self, r):
50         '''
51         Adds the specified record to the aggregate value.
52         '''
53         self._calls += 1
54         for i, c in enumerate(self._c):
55             self._c[i] = self._af[i][1](c, r[i])
56
57     def intersection(a, b):
58         g1 = a.geom()
59         g2 = b.geom()
60         try:
61             if g1.is_valid and g2.is_valid:
62                 i = g1.intersection(g2)
63                 return Geometry(i)
64             else:
65                 return None
66         except:
67             return None
68
69     class UniversalFilter(object):
70         def __init__(self, input_schema, filters):
71             self._input_schema = input_schema
72             self._p = []
73             for k in filters:
74                 self._p.append((input_schema.index(k), filters[k]))
75
76         def accepts(self, other_schema):
77             return self._input_schema == other_schema
78
79         def __call__(self, r):
80             for p in self._p:
81                 # print '--> %s : %s' % (r[p[0]], p[1](r[p[0]]))
82                 if not p[1](r[p[0]]):
83                     return False
84             return True
85
86     #####
87     #
88     # Query
89     #

```

```

90 #####
91
92 engines = []
93
94 # The query stream contains only a single query box.
95 query_streamer = ArrayStreamer(query_schema, [
96     (query,),
97     StopWord(),
98 ])
99 engines.append(query_streamer)
100
101 #####
102 #
103 # States
104 #
105 #####
106
107 states_query = Select(
108     query_streamer.output(),
109     UniversalSelect(
110         query_streamer.output().schema(),
111         {
112             'states.geom': {
113                 'type': Geometry,
114                 'args': ['queries.geom'],
115                 'function': lambda v: v,
116             },
117         }
118     )
119 )
120 engines.append(states_query)
121
122 #query_schema = Schema()
123 #query_schema.append(Attribute('states.geom', Geometry))
124 # The query stream contains only a single query box.
125 #query_streamer = ArrayStreamer(query_schema, [
126     # (query,),
127     # StopWord(),
128     #])
129 #engines.append(query_streamer)
130
131 states_source = Rtree(states_file, 'states.geom')
132 states_accessor = DataAccessor(
133     states_query.output(),
134     states_source,
135     FindRange
136 )
137 engines.append(states_accessor)
138
139 states_select = Select(
140     states_accessor.output(),
141     UniversalSelect(
142         states_accessor.output().schema(),
143         {
144             'states.oid': {
145                 'type': int,
146                 'args': ['oid'],
147                 'function': lambda v: v,
148             },
149             'states.geom': {
150                 'type': Geometry,
151                 'args': ['states.geom'],
152                 'function': lambda v: v,
153             }
154         }
155     )
156 )

```

```

157 engines.append(states_select)
158
159 # join input query stream with the selected states
160 states_join = Join(
161     query_streamer.output(),
162     states_select.output()
163 )
164 engines.append(states_join)
165
166 # trim the states to the boundary of the query
167 states_trim = Select(
168     states_join.output(),
169     UniversalSelect(
170         states_join.output().schema(),
171         {
172             'states.oid': {
173                 'type': int,
174                 'args': ['states.oid'],
175                 'function': lambda v: v,
176             },
177             'states.geom': {
178                 'type': Geometry,
179                 'args': ['queries.geom', 'states.geom'],
180                 'function': lambda a, b: intersection(a, b),
181             }
182         }
183     )
184 )
185 engines.append(states_trim)
186
187 # group states by state ID
188 states_group = Group(
189     states_trim.output(),
190     {
191         'states.oid': lambda a, b: a == b,
192     }
193 )
194 engines.append(states_group)
195
196 #####
197 #
198 # Counties
199 #
200 #####
201
202 counties_query = Select(
203     states_trim.output(),
204     UniversalSelect(
205         states_trim.output().schema(),
206         {
207             'counties.geom': {
208                 'type': Geometry,
209                 'args': ['states.geom'],
210                 'function': lambda v: v,
211             }
212         }
213     )
214 )
215 engines.append(counties_query)
216
217 counties_source = Rtree(counties_file, 'counties.geom')
218 counties_accessor = DataAccessor(
219     counties_query.output(),
220     counties_source,
221     FindRange,
222 )
223 engines.append(counties_accessor)

```

```

224
225 counties_select = Select(
226     counties_accessor.output(),
227     UniversalSelect(
228         counties_accessor.output().schema(),
229         {
230             'counties.oid': {
231                 'type': int,
232                 'args': ['oid'],
233                 'function': lambda v: v,
234             },
235             'counties.geom': {
236                 'type': Geometry,
237                 'args': ['counties.geom'],
238                 'function': lambda v: v,
239             }
240         }
241     )
242 )
243 engines.append(counties_select)
244
245 counties_join = Join(
246     states_group.output(),
247     counties_select.output()
248 )
249 engines.append(counties_join)
250
251 counties_trim = Select(
252     counties_join.output(),
253     UniversalSelect(
254         counties_join.output().schema(),
255         {
256             'states.oid': {
257                 'type': int,
258                 'args': ['states.oid'],
259                 'function': lambda v: v,
260             },
261             'states.geom': {
262                 'type': Geometry,
263                 'args': ['states.geom'],
264                 'function': lambda v: v,
265             },
266             'counties.oid': {
267                 'type': int,
268                 'args': ['counties.oid'],
269                 'function': lambda v: v,
270             },
271             'counties.geom': {
272                 'type': Geometry,
273                 'args': ['states.geom', 'counties.geom'],
274                 'function': lambda a, b: intersection(a, b),
275             }
276         }
277     )
278 )
279 engines.append(counties_trim)
280
281 counties_filter = Filter(
282     counties_trim.output(),
283     UniversalFilter(
284         counties_trim.output().schema(),
285         {
286             'counties.geom': lambda g: g and g.geom().is_valid and g.geom().area
287         }
288     )
289 )
290 engines.append(counties_filter)

```

```

291 counties_group = Group(
292     counties_filter.output(),
293     {
294         'states.oid': lambda a, b: a == b,
295         'counties.oid': lambda a, b: a == b,
296     }
297 )
298 engines.append(counties_group)
299
300 #####
301 #
302 # Zip
303 #
304 #####
305
306 zip_query = Select(
307     counties_filter.output(),
308     UniversalSelect(
309         counties_filter.output().schema(),
310         {
311             'zip.geom': {
312                 'type': Geometry,
313                 'args': ['counties.geom'],
314                 'function': lambda v: v,
315             }
316         }
317     )
318 )
319 engines.append(zip_query)
320
321 zip_source = Rtree(zip_file, 'zip.geom')
322 zip_accessor = DataAccessor(
323     zip_query.output(),
324     zip_source,
325     FindRange,
326 )
327 engines.append(zip_accessor)
328
329 zip_select = Select(
330     zip_accessor.output(),
331     UniversalSelect(
332         zip_accessor.output().schema(),
333         {
334             'zip.oid': {
335                 'type': int,
336                 'args': ['oid'],
337                 'function': lambda v: v,
338             },
339             'zip.geom': {
340                 'type': Geometry,
341                 'args': ['zip.geom'],
342                 'function': lambda v: v,
343             }
344         }
345     )
346 )
347 engines.append(zip_select)
348
349 zip_join = Join(
350     counties_group.output(),
351     zip_select.output(),
352 )
353 engines.append(zip_join)
354
355 zip_trim = Select(
356     zip_join.output(),
357     UniversalSelect(

```

```

358     zip_join.output().schema(),
359     {
360         'states.oid': {
361             'type': int,
362             'args': ['states.oid'],
363             'function': lambda v: v,
364         },
365         'states.geom': {
366             'type': Geometry,
367             'args': ['states.geom'],
368             'function': lambda v: v,
369         },
370         'counties.oid': {
371             'type': int,
372             'args': ['counties.oid'],
373             'function': lambda v: v,
374         },
375         'counties.geom': {
376             'type': Geometry,
377             'args': ['counties.geom'],
378             'function': lambda v: v,
379         },
380         'zip.oid': {
381             'type': int,
382             'args': ['zip.oid'],
383             'function': lambda v: v,
384         },
385         'zip.geom': {
386             'type': Geometry,
387             'args': ['counties.geom', 'zip.geom'],
388             'function': lambda a, b: intersection(a, b),
389         }
390     }
391 )
392 )
393 engines.append(zip_trim)
394
395 zip_filter = Filter(
396     zip_trim.output(),
397     UniversalFilter(
398         zip_trim.output().schema(),
399         {
400             'zip.geom': lambda g: g and g.geom().is_valid and g.geom().area
401         }
402     )
403 )
404 engines.append(zip_filter)
405
406 demux = Demux(zip_filter.output())
407 engines.append(demux)
408 mux_streams = []
409 for i in range(tracks):
410     channel = demux.channel()
411
412     zip_group = Group(
413         channel,
414         {
415             'states.oid': lambda a, b: a == b,
416             'counties.oid': lambda a, b: a == b,
417             'zip.oid': lambda a, b: a == b,
418         }
419     )
420     engines.append(zip_group)
421
422 cover_query = Select(
423     channel,
424     UniversalSelect(
425         channel.schema(),

```

```

425         {
426             'cover.geom': {
427                 'type': Geometry,
428                 'args': ['zip.geom'],
429                 'function': lambda v: v,
430             }
431         }
432     )
433 )
434 engines.append(cover_query)
435
436 cover_source = Rtree(cover_file, 'cover.geom')
437 cover_accessor = DataAccessor(
438     cover_query.output(),
439     cover_source,
440     FindRange
441 )
442 engines.append(cover_accessor)
443
444 cover_select = Select(
445     cover_accessor.output(),
446     UniversalSelect(
447         cover_accessor.output().schema(),
448         {
449             'cover.geom': {
450                 'type': Geometry,
451                 'args': ['cover.geom'],
452                 'function': lambda v: v,
453             }
454         }
455     )
456 )
457 engines.append(cover_select)
458
459 cover_join = Join(
460     zip_group.output(),
461     cover_select.output(),
462 )
463 engines.append(cover_join)
464
465 cover_area = Select(
466     cover_join.output(),
467     UniversalSelect(
468         cover_join.output().schema(),
469         [
470             ('states.oid', {
471                 'type': int,
472                 'args': ['states.oid'],
473                 'function': lambda v: v,
474             }
475             ),
476             # 'states.geom': {
477             #     'type': Geometry,
478             #     'args': ['states.geom'],
479             #     'function': lambda v: v,
480             # },
481             ('counties.oid', {
482                 'type': int,
483                 'args': ['counties.oid'],
484                 'function': lambda v: v,
485             }
486             ),
487             # 'counties.geom': {
488             #     'type': Geometry,
489             #     'args': ['counties.geom'],
490             #     'function': lambda v: v,
491             # },
492             ('zip.oid', {
493                 'type': int,

```



```

492         'args': ['zip.oid'],
493         'function': lambda v: v,
494     }),
495     #'zip.geom': {
496     #     'type': Geometry,
497     #     'args': ['counties.geom', 'zip.geom'],
498     #     'function': lambda a, b: intersection(a, b),
499     # },
500     ('area', {
501     'type': float,
502     'args': ['zip.geom', 'cover.geom'],
503     'function':
504         lambda a, b:
505             intersection(a, b).geom().area / b.geom().area
506     })
507 ]
508 )
509 )
510 engines.append(cover_area)
511
512 #####
513 #
514 # 1st level aggregation
515 #
516 #####
517
518 cover_aggregate = Aggregate(
519     cover_area.output(),
520     SumAggregator(cover_area.output().schema(), 'area')
521 )
522 engines.append(cover_aggregate)
523 mux_streams.append(cover_aggregate.output())
524
525 mux = Mux(*mux_streams)
526 engines.append(mux)
527
528 #####
529 #
530 # 2nd level aggregation
531 #
532 #####
533
534 counties_level_select = Select(
535     mux.output(),
536     UniversalSelect(
537         mux.output().schema(),
538         [
539             ('states.oid', {
540             'type': int,
541             'args': ['states.oid'],
542             'function': lambda v: v,
543             }),
544             ('counties.oid', {
545             'type': int,
546             'args': ['counties.oid'],
547             'function': lambda v: v,
548             }),
549             ('area', {
550             'type': float,
551             'args': ['area'],
552             'function': lambda v: v,
553             }),
554         ]
555     )
556 )
557 engines.append(counties_level_select)
558

```

```

559 counties_level_ungroup = Group(
560     counties_level_select.output(),
561     {}
562 )
563 engines.append(counties_level_ungroup)
564
565 counties_level_sort = Sort(
566     counties_level_ungroup.output(),
567     [
568         ('states.oid', None),
569         ('counties.oid', None)
570     ]
571 )
572 engines.append(counties_level_sort)
573
574 counties_level_group = Group(
575     counties_level_sort.output(),
576     {
577         'states.oid': lambda a, b: a == b,
578         'counties.oid': lambda a, b: a == b,
579     }
580 )
581 engines.append(counties_level_group)
582
583 counties_level_aggregate = Aggregate(
584     counties_level_group.output(),
585     SumAggregator(counties_level_group.output().schema(), 'area')
586 )
587 engines.append(counties_level_aggregate)
588
589 #####
590 #
591 # 3rd level aggregation
592 #
593 #####
594
595 states_level_select = Select(
596     counties_level_aggregate.output(),
597     UniversalSelect(
598         counties_level_aggregate.output().schema(),
599         [
600             ('states.oid', {
601                 'type': int,
602                 'args': ['states.oid'],
603                 'function': lambda v: v,
604             }),
605             ('area', {
606                 'type': float,
607                 'args': ['area'],
608                 'function': lambda v: v,
609             }),
610         ]
611     )
612 )
613 engines.append(states_level_select)
614
615 states_level_ungroup = Group(
616     states_level_select.output(),
617     {}
618 )
619 engines.append(states_level_ungroup)
620
621 states_level_sort = Sort(
622     states_level_ungroup.output(),
623     [
624         ('states.oid', None)
625     ]

```

```

626 )
627 engines.append(states_level_sort)
628
629 states_level_group = Group(
630     states_level_sort.output(),
631     {
632         'states.oid': lambda a, b: a == b,
633     }
634 )
635 engines.append(states_level_group)
636
637 states_level_aggregate = Aggregate(
638     states_level_group.output(),
639     SumAggregator(states_level_group.output().schema(), 'area')
640 )
641 engines.append(states_level_aggregate)
642
643
644 #####
645 #
646 # 4th level aggregation
647 #
648 #####
649
650 all_level_select = Select(
651     counties_level_aggregate.output(),
652     UniversalSelect(
653         counties_level_aggregate.output().schema(),
654         {
655             'area': {
656                 'type': float,
657                 'args': ['area'],
658                 'function': lambda v: v,
659             },
660         }
661     )
662 )
663 engines.append(all_level_select)
664
665 all_level_group = Group(
666     all_level_select.output(),
667     {}
668 )
669 engines.append(all_level_group)
670
671 all_level_aggregate = Aggregate(
672     all_level_group.output(),
673     SumAggregator(all_level_group.output().schema(), 'area')
674 )
675 engines.append(all_level_aggregate)
676
677 #####
678 #
679 # Output
680 #
681 #####
682
683 result_file = ResultFile(
684     'query5-results.txt',
685     mux.output(),
686     counties_level_aggregate.output(),
687     states_level_aggregate.output(),
688     all_level_aggregate.output()
689 )
690 engines.append(result_file)

```

## B.6 Query 2 in optimized PostgreSQL PL/pgSQL

```

1  \begin{Verbatim}[commentchar=CREATE TYPE query6_result_type AS (
2      nation_id INTEGER,
3      customer_id INTEGER,
4      orders_id INTEGER,
5      price FLOAT
6  );
7  CREATE OR REPLACE FUNCTION query6_compute()
8      RETURNS SETOF query6_result_type AS
9  $BODY$
10 DECLARE
11     _nation RECORD;
12     _nation_max FLOAT;
13     _customer RECORD;
14     _customer_max FLOAT;
15     _orders RECORD;
16     _price FLOAT;
17     _result query6_result_type;
18 BEGIN
19     FOR _nation IN SELECT nation.id as id
20         FROM nation_50 AS nation LOOP
21         _nation_max := 0.0;
22         FOR _customer IN SELECT customer.id AS id
23             FROM customer_50 AS customer
24             WHERE customer.nation_id = _nation.id LOOP
25             _customer_max := 0.0;
26             FOR _orders IN SELECT orders.id AS id
27                 FROM orders_50 AS orders
28                 WHERE orders.customer_id = _customer.id LOOP
29                 SELECT MAX(lineitem.price) INTO _price
30                 FROM lineitem_50 AS lineitem
31                 WHERE
32                     lineitem.order_id = _orders.id
33                     AND lineitem.quantity >= 10
34                     AND lineitem.quantity <= 15;
35                 IF _price IS NOT NULL THEN
36                     _result.nation_id = _nation.id;
37                     _result.customer_id = _customer.id;
38                     _result.orders_id = _orders.id;
39                     _result.price = _price;
40                     _customer_max := greatest(_customer_max, _price);
41                     RETURN NEXT _result;
42                 END IF;
43             END LOOP;
44             _result.nation_id = _nation.id;
45             _result.customer_id = _customer.id;
46             _result.orders_id = NULL;
47             _result.price = _customer_max;
48             _nation_max := greatest(_nation_max, _customer_max);
49             RETURN NEXT _result;
50         END LOOP;
51         _result.nation_id = _nation.id;
52         _result.customer_id = NULL;
53         _result.orders_id = NULL;
54         _result.price = _nation_max;
55         RETURN NEXT _result;
56     END LOOP;
57     RETURN;
58 END
59 $BODY$
60 LANGUAGE 'plpgsql';

```

## B.7 Query 5 in optimized PostgreSQL PL/pgSQL

```

1 CREATE TYPE query5_result_type AS (
2     counties_id INTEGER,
3     states_id INTEGER,
4     zip5_id INTEGER,
5     area FLOAT
6 );
7 CREATE OR REPLACE FUNCTION query5()
8     RETURNS SETOF query5_result_type AS
9 $BODY$
10 DECLARE
11     query geometry;
12     state RECORD;
13     state_total FLOAT;
14     county RECORD;
15     county_total FLOAT;
16     zip RECORD;
17     lulc RECORD;
18     cover FLOAT;
19     result query5_result_type;
20 BEGIN
21     query := CAST(makebox2d(
22         makepoint(-93.88, 24.22),
23         makepoint(-65.39, 49.81)
24     ) AS geometry);
25     FOR county IN SELECT
26         counties.id AS id,
27         INTERSECTION(query, counties.geom) AS geom
28     FROM counties
29     WHERE
30         counties.geom && query
31         AND INTERSECTS(counties.geom, query)
32     LOOP
33         county_total := 0.0;
34         IF (GeometryType(county.geom) = 'MULTIPOLYGON'
35             OR GeometryType(county.geom) = 'POLYGON')
36             AND isValid(county.geom)
37             AND NOT isEmpty(county.geom)
38         THEN
39             FOR state IN SELECT
40                 states.id as id,
41                 INTERSECTION(county.geom, states.geom) as geom
42             FROM states
43             WHERE
44                 states.geom && county.geom
45                 AND INTERSECTS(states.geom, county.geom)
46             LOOP
47                 state_total := 0.0;
48                 IF (GeometryType(state.geom) = 'MULTIPOLYGON'
49                     OR GeometryType(state.geom) = 'POLYGON')
50                     AND isValid(state.geom)
51                     AND isSimple(state.geom)
52                     AND NOT isEmpty(state.geom)
53                 THEN
54                     FOR zip IN SELECT
55                         zip5.id AS id,
56                         INTERSECTION(state.geom, zip5.geom) AS geom
57                     FROM zip5
58                     WHERE
59                         zip5.geom && state.geom
60                         AND INTERSECTS(zip5.geom, state.geom)
61                     LOOP
62                         IF (GeometryType(zip.geom) = 'MULTIPOLYGON'

```

```

63         OR GeometryType(zip.geom) = 'POLYGON')
64         AND isValid(zip.geom)
65         AND NOT isEmpty(zip.geom)
66     THEN
67         cover := 0;
68         FOR lulc IN SELECT
69             lulc_large.id AS id,
70             (AREA(INTERSECTION(zip.geom, lulc_large.geom))
71              / AREA(lulc_large.geom)) as c
72         FROM lulc_large
73         WHERE
74             lulc_large.geom && zip.geom
75             AND INTERSECTS(lulc_large.geom, zip.geom)
76         LOOP
77             cover = cover + lulc.c;
78         END LOOP;
79         IF cover IS NOT NULL THEN
80             result.counties_id = county.id - 1;
81             result.states_id = state.id - 1;
82             result.zip5_id = zip.id - 1;
83             result.area = cover;
84             state_total := state_total + cover;
85             RETURN NEXT result;
86         END IF;
87     END IF;
88 END LOOP;
89 END IF;
90 IF state_total          result.counties_id = county.id - 1;
91 result.states_id = state.id - 1;
92 result.zip5_id = NULL;
93 result.area = state_total;
94 county_total := county_total + state_total;
95 RETURN NEXT result;
96 END IF;
97 END LOOP;
98 END IF;
99 IF county_total          result.counties_id = county.id - 1;
100 result.states_id = NULL;
101 result.zip5_id = NULL;
102 result.area = county_total;
103 RETURN NEXT result;
104 END IF;
105 END LOOP;
106 RETURN;
107 END
108 $BODY$
109 LANGUAGE 'plpgsql';

```

## B.8 Query 5 in PostgreSQL standard SQL

```

1  SELECT
2  counties_large.id,
3  states_large.id,
4  zip5_large.id,
5  SUM(
6  Area(
7  Intersection(
8  Intersection(
9  Intersection(
10     counties_large.geom,
11     states_large.geom
12     ),
13     zip5_large.geom

```

```

14         ),
15         lulc_large.geom
16     )
17     ) / AREA(lulc_large.geom)
18     )
19 FROM
20     counties_large
21 JOIN
22     states_large
23 ON
24     counties_large.geom && states_large.geom
25     AND INTERSECTS(counties_large.geom, states_large.geom)
26 JOIN
27     zip5_large
28 ON
29     counties_large.geom && zip5_large.geom
30     AND states_large.geom && zip5_large.geom
31     AND
32     INTERSECTS(
33         INTERSECTION(
34             counties_large.geom,
35             states_large.geom
36         ),
37         zip5_large.geom
38     )
39 JOIN
40     lulc_large
41 ON
42     counties_large.geom && lulc_large.geom
43     AND states_large.geom && lulc_large.geom
44     AND zip5_large.geom && lulc_large.geom
45     AND
46     INTERSECTS(
47         Intersection(
48             Intersection(
49                 counties_large.geom,
50                 states_large.geom
51             ),
52             zip5_large.geom
53         ),
54         lulc_large.geom
55     )
56 WHERE
57     lulc_large.geom && makebox2d(
58         makepoint(-93.88, 24.22),
59         makepoint(-65.39, 49.81)
60     )
61     AND
62     INTERSECTS(
63         lulc_large.geom,
64         makebox2d(
65             makepoint(-93.88, 24.22),
66             makepoint(-65.39, 49.81)
67         )
68     )
69 GROUP BY counties_large.id, states_large.id, zip5_large.id;

```

## Appendix C

### OLAP for Moving Object Data: Research Opportunities

The solutions proposed in Chapter 7 address only some of the aspects of OLAP on moving object data. There are some open problems that may provide opportunities for future work. Next we outline some of them.

#### C.1 Alternative Grouping Operators

The `GROUP_TRAJECTORIES` operators introduced in this chapter are based on a heuristic approach by using specific properties (overlap and intersection) of the relationships between groups of trajectories as criteria for merging sufficiently related groups into larger combined groups of trajectories. Alternative approaches that define other `GROUP_TRAJECTORIES` operators and use different criteria for combining groups should be discussed in future work. One such approach is probabilistic modeling of the groups of trajectories. For this approach, future work needs to identify appropriate mechanisms to assign probabilities to relationships of trajectories that are characterized by frequent itemsets, and formalize the criteria for combining trajectories into groups. For example, in such a model a frequent pattern that is shared by two or more trajectories may be used to assign a probability to the pairwise relationships between the trajectories based on the size of the frequent itemset relative to the length of the related trajectories. Once the initial probabilities of relationships between trajectories have been determined, we can use the joint probability of two relationships to determine the probability of a relationship between two trajectories that do not share a frequent itemset but have a common neighbour who shares a frequent itemset with both.



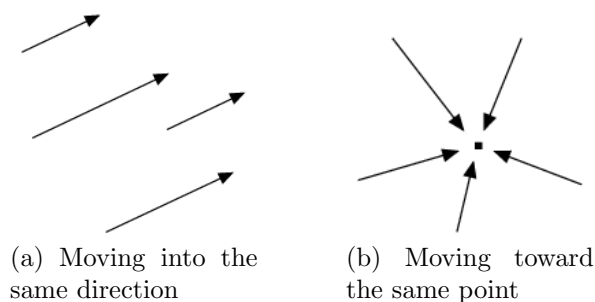


Figure C.1: Examples of different movement patterns.

## C.2 Alternative Movement Patterns

For the algorithms introduced in Chapter 7, it was assumed that frequent itemsets were determined on trajectories that are represented within their original space but potentially mapped to a lower resolution. This enables the algorithms to group together trajectories that are moving in unison for a certain number of shared locations along their individual paths. In many applications, however, it may be desirable to identify groups of the trajectories that show different movement patterns, such as those shown in Figure C.1.

Future work should investigate how preprocessing of trajectories can be used to facilitate the identification of alternative movement patterns without significant modifications to the existing grouping algorithms. For example a transformation of trajectories of the form  $[(x_1, y_1, t_1), \dots, (x_m, y_m, t_m)]$  to  $[(\alpha_1, t_1), \dots, (\alpha_m, t_m)]$ , where  $\alpha_i$  is the orientation of the object with respect to a reference axis, may allow to identify groups of trajectories that move in the same direction, as shown in Figure C.1a.

## C.3 Aggregation of Trajectories

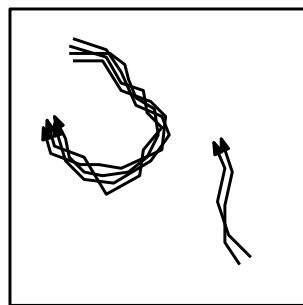
As shown in the previous section, there is a lot of work that focuses on the identification of groups of moving objects. However, only few authors address the issue of summarizing and aggregating the information that captures the characteristics of each group of moving objects.

Güting was one of the first authors who investigated the problem of spatial aggregation [73]. He suggested aggregation functions such as UNION (the union of all  $(x_i, y_i, t_i)$  tuples that define the trajectories in the aggregation group) and CONTOUR

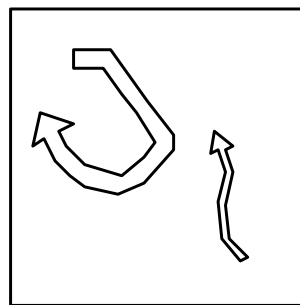
(the outline of a group of trajectories) to capture summarized properties of groups of spatial objects. In [139] Pedersen and Tryfona provided an in-depth analysis of spatial aggregation with respect to pre-aggregation and the derivation of aggregation results from already pre-aggregated data. Additionally, many current commercial database systems have adopted spatial aggregation functions in their spatial extensions and provide support for the implementation of custom, special-purpose aggregation functions that can be defined by the user [93, 122, 133]. To facilitate this, frameworks to support the implementation of aggregation functions have been proposed that specifically cater to tasks related to data mining [189]. Based on the aggregation of spatial properties, various approaches for the aggregation of spatio-temporal data have been derived. An extensive review of these approaches for spatio-temporal aggregation is provided in [112].

The identification of groups of trajectories is a necessary part of OLAP on moving object data. However, to facilitate an interactive analysis of the data, methods have to be provided that reduce the amount of information displayed to the user to the essential minimum. In OLAP this is achieved using aggregation. Depending on the application scenario, existing aggregation functions for categorical or spatial data may not be sufficient when applied to moving object data, and there is a need for powerful and expressive aggregation functions that support the aggregation of trajectories.

For example, when aggregating trajectories, an **AVERAGE** function may be suitable to determine the average path that is described by all trajectories that are within a group. Alternatively, a **GENERALIZE** function, that can be used to determine a visual generalization of the movements of a group of trajectories may be useful when a graphical representation of the aggregate is required. Figures C.2a and C.2b show such graphical representations and suggest that objects moving along a shared path may be represented as a single “arrow” whose width is proportional to the size of the group it represents.



(a) Non-generalized groups of trajectories



(b) Generalized groups of trajectories

Figure C.2: A visual generalization function for groups of trajectories.

## Bibliography

- [1] SQL Part 2: Foundation. Spec, International Organization for Standardization, 2003. ISO/IEC 9075-2:2003.
- [2] Improving emergency planning and response with geographic information systems. White paper, ESRI, 2005.
- [3] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of XQueries. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*, pages 615–626. ACM, 2009.
- [4] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66(1):207–243, 2003.
- [5] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 506–521. Morgan Kaufmann, 1996.
- [6] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.
- [7] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14. IEEE, 1995.
- [8] T. Ahmed. Spatial on-line analytical processing (SOLAP): Overview and current trends. In *Proceedings of the 2008 International Conference on Advanced Computer Theory and Engineering*, pages 1095–1099. IEEE, 2008.
- [9] J. H. Albrecht. Universal GIS operations for environmental modeling. In *Proceedings of the 3rd International Conference on Integrating GIS and Environmental Modeling*, 1996.
- [10] M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle. Reporting leadership patterns among trajectories. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 3–7. ACM, 2007.
- [11] T. Badard. Developing geospatial business intelligence solutions. *Geospatial Today*, Nov. 2010.
- [12] O. Baltzer, F. Dehne, S. Hambrusch, and A. Rau-Chaplin. OLAP for trajectories. In *Proceedings of the 19th International Conference on Database and Expert Systems Applications*, pages 340–347. Springer, 2008.

- [13] O. Baltzer, A. Rau-Chaplin, and N. Zeh. Storage and indexing of relational OLAP views with mixed categorical and continuous dimensions. *Journal of Digital Information Management*, 5(4):180, 2007.
- [14] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.
- [15] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.
- [16] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, 19(2):322–331, 1990.
- [17] Y. Bédard, T. Merrett, and J. Han. *Research Monographs in GIS*, chapter Fundamentals of Spatial Data Warehousing for Geographic Knowledge Discovery in Geographic Data Mining and Knowledge Discovery, pages 53–73. Taylor & Francis, 2001.
- [18] R. Benetis, C. S. Jensen, G. Karčiauskas, and S. Šaltenis. Nearest and reverse nearest neighbor queries for moving objects. *The International Journal on Very Large Data Bases*, 15(3):229–249, 2006.
- [19] M. Benkert, J. Gudmundsson, F. Huebner, and T. Wolle. Reporting flock patterns. In *Proceedings of the 14th European Symposium on Algorithms*, pages 660–671. Springer, 2006.
- [20] E. Bernier, P. Gosselin, T. Badard, and Y. Bédard. Easier surveillance of climate-related health vulnerabilities through a web-based spatial OLAP application. *International Journal of Health Geographics*, 8(1):18, 2009.
- [21] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and Iceberg CUBE. *ACM SIGMOD Record*, 28(2):359–370, 1999.
- [22] S. Bimonte, V. Fatto, L. Paolino, M. Sebillio, and G. Vitiello. A visual query language for spatial data warehouses. *Geospatial Thinking*, pages 43–60, 2010.
- [23] S. Bimonte, J. Gensel, and M. Bertolotto. Enriching spatial OLAP with map generalization: A conceptual multidimensional model. In *Proceedings of the 2008 IEEE International Conference on Data Mining*, pages 332–341. IEEE, 2008.
- [24] S. Bimonte, P. Wehrle, A. Tchounikine, and M. Miquel. GeWolap: A web based spatial olap proposal. In *Proceedings of the OnTheMove Workshops*, pages 1596–1605. Springer, 2006.
- [25] C. Böhm, S. Berchtold, H.-P. Kriegel, and U. Michel. Multidimensional index structures in relational databases. *Journal of Intelligent Information Systems*, 15(1):51–70, 2000.

- [26] H. Boral and D. J. DeWitt. Design considerations for data-flow database machines. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*, pages 94–104. ACM, 1980.
- [27] H. Cao, N. Mamoulis, and D. W. Cheung. Mining frequent spatio-temporal sequential patterns. In *Proceedings of the 5th International Conference on Data Mining*, pages 82–89. IEEE, 2005.
- [28] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.
- [29] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.
- [30] N.-B. Chang, H. Y. Lu, and Y. L. Wei. GIS technology for vehicle routing and scheduling in solid waste collection systems. *Journal of Environmental Engineering*, 123(9):901–910, 1997.
- [31] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1), 1997.
- [32] J. Chen and X. Meng. Update-efficient indexing of moving objects in road networks. *GeoInformatica*, 13(4):397–424, 2009.
- [33] P. P.-S. Chen. The entity-relationship model-toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [34] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. PnP: Parallel and external memory iceberg cubes. In *Proceedings of the 21st International Conference on Data Engineering*, pages 576–577. IEEE, 2005.
- [35] Y. Chen, A. Rau-Chaplin, F. Dehne, T. Eavis, D. Green, and E. Sithirasenan. cgmOLAP: Efficient parallel generation and querying of terabyte size ROLAP data cubes. In *Proceedings of the 22nd International Conference on Data Engineering*, page 164. IEEE, 2006.
- [36] E. Clementini and P. Di Felice. A comparison of methods for representing topological relationships. *Information Sciences Applications*, 3(3):149–178, 1995.
- [37] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP to user-analysts: An IT mandate. Technical report, E.F.Codd & Associates, 1993.
- [38] T. L. Cohen, J. R. Baitty, R. G. Plamer, K. T. Adams, and J. A. Weyl. Health resources and services administration geospatial data warehouse. In *Proceedings of the 2004 ESRI International Users Conference*. ESRI, 2004.

- [39] J. da Silva, V. Times, A. Salgado, C. Souza, R. Fidalgo, and A. de Oliveira. A set of aggregation functions for spatial measures. In *Proceeding of the 11th ACM International Workshop on Data Warehousing and OLAP*, pages 25–32. ACM, 2008.
- [40] J. da Silva, V. C. Times, R. N. Fidalgo, and R. S. M. Barros. *Web Technologies Research and Development*, chapter Providing Geographic-Multidimensional Decision Support over the Web, pages 477–488. Springer, 2005.
- [41] M. Davis, Y. Bychkov, et al. GEOS: Geometry Engine Open Source, Apr. 2010. <http://trac.osgeo.org/geos/>.
- [42] M. Davis et al. Java Topology Suite, 2002–2006. <http://www.vividsolutions.com/jts/jtshome.htm>.
- [43] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer-Verlag, 2 edition, 1998.
- [44] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 319–326. Springer, 2001.
- [45] F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel multi-dimensional ROLAP indexing. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, pages 86–96. IEEE, 2003.
- [46] G. P. Deretta. Boost.coroutine, 2006. <http://www.crystalclearsoftware.com/soc/coroutine/>.
- [47] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *Proceedings of the 24th International Conference on Data Engineering*, pages 754–763. IEEE, 2008.
- [48] Z. Ding and R. H. Guting. Managing moving objects on dynamic transportation networks. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 287–296. IEEE, 2004.
- [49] E. Dube and T. Badard. GeoMondrian, 2009. <http://www.spatialytics.org/projects/geomondrian/>.
- [50] T. Eavis. *Parallel relational OLAP*. PhD thesis, Dalhousie University, 2003.
- [51] M. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, pages 86–95, 1994.
- [52] R. S. Engelschall. Gnu portable threads, 1999–2006. <http://www.gnu.org/software/pth/>.

- [53] A. Escribano, L. Gomez, B. Kuijpers, and A. Vaisman. Piet: A GIS-OLAP implementation. In *Proceedings of the 10th ACM International Workshop on Data Warehousing and OLAP*, pages 73–80. ACM, 2007.
- [54] ESRI Inc. ArcGIS, 2010. <http://www.esri.com/software/arcgis/arcgis10/index.html>.
- [55] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.
- [56] R. N. Fidalgo, V. C. Times, J. Silva, and F. F. Souza. *Data Warehousing and Knowledge Discovery*, chapter GeoDWFrame: A Framework for Guiding the Design of Geographical Dimensional Schemas, pages 26–37. Springer, 2004.
- [57] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
- [58] C. Franklin. An introduction to Geographic Information Systems: Linking maps to databases. *Database*, 15(2):12–21, 1992.
- [59] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [60] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.
- [61] G. Gidófalvi and T. Pedersen. Mining long, sharable patterns in trajectories of moving objects. *GeoInformatica*, 13(1):27–55, 2009.
- [62] G. Gidófalvi and T. B. Pedersen. Mining long, sharable patterns in trajectories of moving objects. In *Proceedings of the 3rd Workshop on Spatio-Temporal Database Management*, 2006.
- [63] S. Gillies, H. Butler, and B. Pedersen. Python R-tree bindings, Apr. 2010. <http://trac.gispython.org/lab/wiki/Rtree>.
- [64] S. Gillies et al. Shapely – GEOS Python bindings, Jan. 2011. <http://trac.gispython.org/lab/wiki/Shapely>.
- [65] L. I. Gómez, S. Haesevoets, B. Kuijpers, and A. Vaisman. Spatial aggregation: Data model and implementation. *Information Systems*, 34(6):551–576, 2009.
- [66] L. I. Gómez, B. Kuijpers, B. Moelans, and A. Vaisman. A survey of spatio-temporal data warehousing. *International Journal of Data Warehousing and Mining*, 5(3):28–55, 2009.



- [67] L. I. Gómez, B. Kuijpers, and A. A. Vaisman. Aggregation languages for moving object and places of interest data. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 857–862. ACM, 2008.
- [68] L. I. Gómez, A. Vaisman, and S. Zich. Piet-QL: A query language for GIS-OLAP integration. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10. ACM, 2008.
- [69] L. I. Gómez, A. Vaisman, and E. Zimányi. Physical design and implementation of spatial data warehouses supporting continuous fields. *Data Warehousing and Knowledge Discovery*, pages 25–39, 2010.
- [70] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A relational aggregational operator for generalizing group-bys, cross-tabs, and sub-totals. Technical Report MSR-TR-95-22, 1995.
- [71] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceedings of the 12th International Conference on Data Engineering*, page 152, 1996.
- [72] J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *Proceedings of the 12th ACM International Workshop on Geographic Information Systems*, pages 250–257. ACM, 2004.
- [73] R. H. Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.
- [74] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *Transactions on Database Systems*, 25(1):1–42, 2000.
- [75] R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.
- [76] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Readings in database systems*, pages 599–609, 1988.
- [77] M. Hadjieleftheriou. Spatial index library, Jan. 2011. <http://trac.gispython.org/spatialindex/wiki>.
- [78] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proceedings of the 8th International Conference on Extending Database Technology*, pages 251–268. Springer, 2002.
- [79] A. Hameurlain and F. Morvan. Evolution of query optimization methods. *Transactions on Large-Scale Data-and Knowledge-Centered Systems*, pages 211–242, 2009.

- [80] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of the 15th International Conference on Data Engineering*, pages 106–115. IEEE, 1999.
- [81] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.
- [82] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.
- [83] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Proceedings of the 2nd Pacific-Asia Conference on Research and Development in Knowledge Discovery and Data Mining*, pages 144–158. Springer, 1998.
- [84] S. Har-Peled. Clustering motion. *Discrete and Computational Geometry*, 31(4):545–565, 2004.
- [85] G. Häring et al. SQLite Python bindings, Jan. 2011. <http://docs.python.org/library/sqlite3.html>.
- [86] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.
- [87] V. Hernandez, A. Voss, and W. Gohring. Sustainable decision support by the use of multi-level and multi-criteria spatial analysis on the nicaragua development gateway, from pharaohs to geoinformatics. In *Proceedings of the Fédération Internationale des Géomètres Working Week*, pages 16–21. Fédération Internationale des Géomètres, 2005.
- [88] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.
- [89] D. R. Hipp et al. The architecture of SQLite, Jan. 2011. <http://www.sqlite.org/arch.html>.
- [90] D. R. Hipp et al. SQLite, Jan. 2011. <http://www.sqlite.org/>.
- [91] S. Y. Hwang, Y. H. Liu, J. K. Chiu, and E. P. Lim. Mining mobile group patterns: A trajectory-based approach. In *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 713–718. Springer, 2005.
- [92] J. Hyde et al. Mondrian OLAP Server, 2010. <http://mondrian.pentaho.com/>.
- [93] IBM Corporation. *Informix Dynamic Server v11 Information Center*. <http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp>.

- [94] K. Idehen. Open database connectivity without compromise, 1993. <http://www.openlinksw.com/info/docs/odbcwhp/tableof.htm>.
- [95] Intelli3 Inc. Map4Decision, 2010. [http://www.intelli3.com/en/map4decision\\_en.php](http://www.intelli3.com/en/map4decision_en.php).
- [96] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 332–342. ACM, 1990.
- [97] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.
- [98] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 500–509. Morgan Kaufmann, 1994.
- [99] R. Karedla, J. Love, and B. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, 2002.
- [100] KHEOPS Technologies. JMap spatial OLAP – Innovative technology to support intuitive and interactive exploration and analysis of spatio-temporal multidimensional data. Technical report, 2005.
- [101] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th Symposium on Principles of Database Systems*, pages 261–272. ACM, 1999.
- [102] H.-P. Kriegel and M. Pfeifle. Clustering moving objects via medoid clusterings. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, pages 153–162. Lawrence Berkeley Laboratory, 2005.
- [103] B. Kuijpers and A. A. Vaisman. A data model for moving objects supporting aggregation. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 546–554. IEEE, 2007.
- [104] P. Laube and S. Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *Proceedings of the 2nd International Conference on Geographic Information Science*, pages 132–144. Springer, 2002.
- [105] P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO—detecting relative motion patterns in geospatial lifelines. In *Proceedings of the 11th International Symposium on Spatial Data Handling*, pages 201–214. Springer, 2004.
- [106] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the Hilbert space-filling curve. *ACM SIGMOD Record*, 30(1):19–24, 2001.

- [107] L. Leonardi, G. Marketos, E. Frentzos, N. Giatrakos, S. Orlando, N. Pelekis, A. Raffaetà, A. Roncato, C. Silvestri, and Y. Theodoridis. T-Warehouse: Visual OLAP analysis on trajectory data. In *Proceedings of the 26th International Conference on Data Engineering*, pages 1141–1144. IEEE, 2010.
- [108] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 617–622. ACM, 2004.
- [109] A. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel OLAP query processing in database clusters with data replication. *Distributed and Parallel Databases*, 25(1):97–123, 2009.
- [110] G. Liu, H. Lu, J. Yu, W. Wei, and X. Xiao. AFOPT: An efficient implementation of pattern growth approach. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*. CEUR-WS.org, 2003.
- [111] P. A. Longley and M. Batty, editors. *Spatial Analysis: Modelling in a GIS Environment*. John Wiley & Sons, 1996.
- [112] I. F. V. López, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.
- [113] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of the 17th International Conference on Data Engineering*, pages 205–214. IEEE, 2001.
- [114] E. Malinowski and E. Zimányi. Spatial hierarchies and topological relationships in the spatial MultiDimER model. In *Proceedings of the 22nd British National Conference on Databases*, pages 17–28. Springer, 2005.
- [115] E. Malinowski and E. Zimányi. Implementing spatial data warehouse hierarchies in object-relational DBMSs. In *Proceedings of the 9th International Conference on Enterprise Information Systems*, volume 7, pages 186–191, 2007.
- [116] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *Proceedings of the 2004 International Conference on Knowledge Discovery and Data Mining*, pages 236–245. ACM, 2004.
- [117] P. Marchand, A. Brisebois, Y. Bédard, and G. Edwards. Implementation and evaluation of a hypercube-based method for spatiotemporal exploration and analysis. *ISPRS Journal of Photogrammetry & Remote Sensing*, 59(1), 2004.
- [118] S. Martino, S. Bimonte, M. Bertolotto, and F. Ferrucci. Integrating Google Earth within OLAP tools for multidimensional exploration and analysis of spatial data. *Enterprise Information Systems*, pages 940–951, 2009.

- [119] M. Mehta and D. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.
- [120] Microsoft Corporation. U.s. government agency stores aerial imagery in new 25-terabyte warehouse growing warehouse. Case study, 2004.
- [121] Microsoft Corporation. SQL Server 2005 Analysis Services, 2007. <http://www.microsoft.com/sqlserver/2005/en/us/analysis-services.aspx>.
- [122] Microsoft Corporation. *SQL Server 2005 Programming Reference*, 2007.
- [123] Microsoft Corporation. SQL Server 2008 R2 Parallel Data Warehouse, 2008. <http://www.microsoft.com/sqlserver/2008/en/us/parallel-data-warehouse.aspx>.
- [124] Microsoft Corporation. Multidimensional Expressions (MDX) Reference, 2011. <http://msdn.microsoft.com/en-us/library/ms145506.aspx>.
- [125] Microsoft Corporation and Hyperion Solutions Corporation. XML for Analysis Specification, Apr. 2001. <http://msdn.microsoft.com/en-us/library/ms977626.aspx>.
- [126] M. Mokbel, T. Ghanem, and W. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.
- [127] B. Moon, H. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, January 2001.
- [128] D. Moore. C Hilbert mapping library, 1998. <http://computation.pa.msu.edu/NO/F90/SFC/hilbert.c>.
- [129] M. Nanni and D. Pedreschi. Time-focused clustering of trajectories of moving objects. *Journal of Intelligent Information Systems*, 27(3):267–289, 2006.
- [130] A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.
- [131] J. R. Nuckols, M. H. Ward, and L. Jarup. Using geographic information systems for exposure assessment in environmental epidemiology studies. *Environmental Health Perspectives*, 112(9), 2004.
- [132] Open GIS Consortium. OpenGIS simple feature specification for SQL, 1999.
- [133] Oracle Corporation. *Oracle Database Data Cartridge Developer’s Guide 10.2*, 2005.
- [134] Oracle Corporation. *Oracle Spatial User’s Guide and Reference*, 2005.

- [135] R. Osegueda, A. Garcia-Diaz, S. Ashur, O. Melchor, S.-H. Chang, C. Carrasco, and A. Kuyumcu. GIS-based network routing procedures for overweight and oversized vehicles. *Journal of Transportation and Engineering*, 125(4), 1999.
- [136] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 443–459. Springer, 2001.
- [137] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *Proceedings 18th International Conference on Data Engineering*, pages 166–175. IEEE, 2002.
- [138] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416. Springer, 1999.
- [139] T. B. Pedersen and N. Tryfona. Pre-aggregation in spatial data warehouses. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 460–480. Springer, 2001.
- [140] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern. In *Proceedings of the 17th International Conference on Data Engineering*. IEEE, 2001.
- [141] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, 2004.
- [142] N. Pendse. The OLAP report, May 2005. <http://www.olapreport.com/>.
- [143] W. C. Peng and M. S. Chen. Developing data allocation schemes by incremental mining of user moving patterns in a mobile computing system. *Transactions on Knowledge and Data Engineering*, 15(1):70–85, 2003.
- [144] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 395–406. Morgan Kaufmann, 2000.
- [145] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 59–78. Springer, 2001.
- [146] PostgreSQL Global Development Group. PostgreSQL, 1997–2011. <http://www.postgresql.org/>.

- [147] S. Prasher and X. Zhou. Multiresolution amalgamation: dynamic spatial data cube generation. In *Proceedings of the 15th Australasian Database Conference*, pages 103–111. Australian Computer Society, 2004.
- [148] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An efficient self-adjusting index for moving objects. In *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments*, pages 178–193. Springer, 2002.
- [149] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.
- [150] F. Rao, L. Zhang, X. L. Yu, Y. Li, and Y. Chen. Spatial hierarchy and OLAP-favored search in spatial data warehouse. In *Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP*, pages 48–55. ACM, 2003.
- [151] Refractions Research et al. PostGIS, 2000–2011. <http://postgis.refractions.net/>.
- [152] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann, 2002.
- [153] S. Rivest, Y. Bédard, and P. Marchand. Toward better support for spatial decision making: Defining the characteristics of spatial on-line analytical processing (SOLAP). *Geomatica*, 55(4):539 – 555, 2001.
- [154] S. Rivest, Y. Bédard, M.-J. Proulx, M. Nadeau, F. Hubert, and J. Pastor. SOLAP technology: Merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data. *ISPRS Journal of Photogrammetry & Remote Sensing*, 60:17–33, 2005.
- [155] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proceedings of the 18th International Conference on Data Engineering*, pages 463–472. IEEE, 2002.
- [156] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.
- [157] SAS Institute Inc. SAS Enterprise BI Server, 2010. <http://www.sas.com/technologies/bi/entbiserver/index.html>.
- [158] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 08(3):19–26, 2004.
- [159] S. Sclaroff, G. Kollios, and M. Betke. Motion mining: discovering spatio-temporal patterns in databases of human motion. In *Proceedings of the 2001 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*. ACM, 2001.

- [160] M. Scotch and B. Parmanto. Development of SOVAT: A numerical-spatial decision support system for community health assessment research. *International Journal of Medical Informatics*, 75(10–11):771–784, 2005.
- [161] M. Scotch and B. Parmanto. SOVAT: Spatial OLAP visualization and analysis tool. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 142–144. IEEE, 2005.
- [162] M. Scotch, B. Parmanto, and V. Monaco. Evaluation of SOVAT: An OLAP-GIS decision support system for community health assessment data analysis. *BMC Medical Informatics and Decision Making*, 8(1):22, 2008.
- [163] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.
- [164] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.
- [165] S. Shekhar, C. Lu, X. Tan, S. Chawla, and R. Vatsavai. Map Cube: A visualization tool for spatial data warehouses. In H. Miller and J. Han, editors, *Geographic Data Mining and Knowledge Discovery*, pages 74–109. Taylor & Francis London, 2001.
- [166] C. B. Shim and J. W. Chang. A new similar trajectory retrieval scheme using k-warping distance algorithm for moving objects. In *Proceedings of the 4th International Conference on Advances in Web-Age Information Management*, pages 433–444. Springer, 2003.
- [167] A. Shukla, P. Deshpande, J. Naughton, et al. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 488–499. Morgan Kaufmann, 1998.
- [168] G. Sinha and D. M. Mark. Measuring similarity between geospatial lifelines in studies of environmental health. *Journal of Geographical Systems*, 7(1):115–136, 2005.
- [169] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13th International Conference on Data Engineering*, pages 422–432. IEEE, 1997.
- [170] N. Stefanović. Design and implementation of on-line analytical processing (OLAP) of spatial data. Master’s thesis, Simon Fraser University, 1997.
- [171] M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.
- [172] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 431–440. Morgan Kaufmann, 2001.



- [173] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(12):1555–1570, 2004.
- [174] Y. Tao, D. Papadias, and J. Sun. The TPR\*-tree: An optimized spatio-temporal access method for predictive queries. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 790–801. Morgan Kaufmann, 2003.
- [175] Y. Theodoridis. The R-tree-portal, 2003. <http://www.rtreportal.org/>.
- [176] Transaction Processing Performance Council. TPC-H benchmark specification, Nov. 2010. <http://www.tpc.org/tpch/>.
- [177] I. Tsoukatos and D. Gunopulos. Efficient mining of spatiotemporal patterns. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 425–442. Springer, 2001.
- [178] U.S. Census Bureau. 5-digit zip boundaries, 2000. <http://www.census.gov/geo/www/cob/z52000.html>.
- [179] U.S. Census Bureau. County boundaries, 2000. <http://www.census.gov/geo/www/cob/co2000.html>.
- [180] U.S. Census Bureau. State boundaries, 2000. <http://www.census.gov/geo/www/cob/st2000.html>.
- [181] U.S. Geological Survey. Land cover and land usage, Apr. 2007. <http://water.usgs.gov/GIS/dsdl/ds240/>.
- [182] A. Vaisman, M. Espil, and M. Paradela. P2P OLAP: Data model, implementation and case study. *Information Systems*, 34(2):231–257, 2009.
- [183] A. Vaisman and E. Zimányi. A multidimensional model representing continuous fields in spatial data warehouses. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 168–177. ACM, 2009.
- [184] G. van Rossum. Python programming language, 2011. <http://www.python.org/>.
- [185] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. *ACM SIGMOD Record*, 28(4), 1999.
- [186] P. Vassiliadis, A. Simitis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In D. Theodoratos, editor, *Proceedings of the 5th International Workshop on Data Warehousing and OLAP*, pages 14–21. ACM, 2002.
- [187] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings of the 18th International Conference on Data Engineering*, pages 673–684. IEEE, 2002.

- [188] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *ACM SIGMOD Record*, 29(2):331–342, 2000.
- [189] H. Wang, C. Zaniolo, and C. R. Luo. ATLAS: A small but complete sql extension for data mining and data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1113–1116. Morgan Kaufmann, 2003.
- [190] X. Wang and S. Huang. Development of a flood warning system. In *Proceedings of the 19th ESRI International User Conference*. ESRI, 1999.
- [191] Y. Wang, E.-P. Lim, and S.-Y. Hwang. On mining group patterns of mobile users. In V. Marík, W. Retschitzegger, and O. Stepánková, editors, *Proceedings of the 14th International Conference on Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 287–296. Springer, 2003.
- [192] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 111–122. IEEE, 1998.
- [193] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of the 3rd International Conference on Data Mining*, pages 166–177. Society for Industrial and Applied Mathematics, 2003.
- [194] J. Yang, W. Wang, P. S. Yu, and J. Han. Mining long sequential patterns in a noisy environment. In *Proceedings of the 2002 International Conference on Management of Data*, pages 406–417. ACM, 2002.
- [195] N. Yazdani and Z. M. Ozsoyoglu. Sequence matching of images. In *Proceedings of the 8th International Conference on Scientific and Statistical Database Management*, pages 53–62. IEEE, 1996.
- [196] K. Yip and F. Zhao. Spatial aggregation: Theory and applications. *Journal of Artificial Intelligence Research*, 5:1–26, 1996.
- [197] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1):31–60, 2001.
- [198] D. Zeinalipour-Yazti, S. Lin, and D. Gunopulos. Distributed spatio-temporal similarity search. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 14–23. ACM, 2006.
- [199] D. Zhang and V. J. Tsotras. Improving min/max aggregation over spatial objects. In *Proceedings of the 9th ACM International Symposium on Advances in Geographic Information Systems*, pages 88–93. ACM, 2001.

- [200] R. Zhang, H. Jagadish, B. Dai, and K. Ramamohanarao. Optimized algorithms for predictive range and KNN queries on moving objects. *Information Systems*, 2010.
- [201] X. Zhou, D. Truffet, and J. Han. Efficient polygon amalgamation methods for spatial OLAP and spatial data mining. In *Proceedings of the 6th International Symposium on Advances in Spatial Databases*, pages 167–187. Springer, 1999.