Frontmatter

Editors

A Message from the Editor
S. Shin

SAC 2012 Report
H. Haddad

Selected Research Articles

A Spatiotemporal Extension for Dealing with Moving Objects with Extent in Oracle 11g
L. Matos, J. Moreira, and A. Carvalho

Transparency Increasing RMI Fault Tolerance
D. Reis and H. Miranda

One-Scan Rule Extraction to Explain Significant Vehicle Interactions with Guaranteed Error Value
O. Florez and C. Dyreson

Supporting Semantic Conflict Prevention in Real-Time Collaborative Programming Environments
H. Fan and C. Sun

A Unified Framework for Document Clustering with Dual Supervision
Y. Hu, E. Milios, and J. Blustein

Querying RDF Dictionaries in Compressed Space
M. Martínez-Prieto, J. Fernández, and R. Cánovas
Applied Computing Review

Editor in Chief
Sung Y. Shin

Designer/Technical Editor
John Kim

Associate Editors
Richard Chbeir
Hisham Haddad
Lorie M. Liebrock
W. Eric Wong

Editorial Board Members

Wood Alan
Davide Ancona
Papadopoulos
Apostolos
Alessio Bechini
Giampaolo Bella
Umesh Bellur
Ateet Bhalla
Stefano Bistarelli
Olivier Boissier
Gloria Bordogna
Auernheimer Brent
Barrett Bryant
Alex Buckley
Artur Caetano
Luís Carriço
Rogério Carvalho
Andre Carvalho
Matteo Casadei
Jan Cederquist
Alvin Chan
Richard Chbeir
Jian-Jia Chen
Claramunt Christophe
Yvonne Coady
Luca Compagna
Arthur Wm. Conklin
Massimo Cossentino
Federico Divina
Carlos Duarte
Mario Freire
Lorenz Froihoffer
João Gama
Xiao-Shan Gao
M. Karl GoeSchka
Claudio Guidi
Svein Hallsteinsen
Jason Hallstrom
Hyoil Han
A. Ramzi Haraty
Jean Hennebert
Jiman Hong
Andreas Humm
L. Robert Hutchinson
Maria-Eugenia Iacob
Francois Jacquenet
Hasan Jamil
Robert Joan-Arinyo
Andy Killens
Tei-Wei Kuo
Opt’ Martin Land
Ivan Lanese
Paola Lecca
Maria LencaStre
Va Hong Leong
Ki-Joune Li
Lorie Liebrock
Giuseppe Lipari
Shih-Hsi Liu
Rui Lopes
Mamei Marco
Eduardo Marques
Paulo Martins
Stan Matwin
Manuel Mazzara
Ronaldo Menezes
Marjan Mernik
Schumacher Michael
Fabien Michel
Dominique Michelucci
Ambra Molesini
Eric Monfroy
Barry O’Sullivan
Rui Oliveira
Andrea Omicini
Fernando Osorio
Edmundo Monteiro
V. Ethan Munson
Peter Otto
Brajendra Panda
Gabriella Pasi
Manuela Pereira
G. Maria Pimentel
Antonio Cosimo Prete
Kanagasabai Rajaraman
Rajiv Ramnath
Chandan Reddy
Eliot Rich
Pedro Rodrigues
Giráldez Raúl Rojo
Alexander Romanovsky
Agostinho Rosa
Davide Rossi
Corrado Santoro
Rodrigo Santos
Guido Schryen
Kumar Madhu SD
Jean-Marc Seigneur
Alessandro Sorniotti
Nigamanth Sridhar
Lindsay Yan Sun
Junping Sun
Chang Oan Sung
Emiliano Tramontana
Dan Tulpan
Seidita Valeria
Teresa Vasão
Mirko Viroli
Fabio Vitali
Giuseppe Vizzari
Denis Wolf
W. Eric Wong
Kokou Yetongnon
R. Osmar Zaiane

APPLIED COMPUTING REVIEW   VOL. 12,   NO. 2

3
A Message from the Editor

Welcome to Applied Computing Review! I am happy to tell you that ACR has been published on a quarterly basis since last winter. This issue includes selected papers presented at the 2012 ACM Symposium on Applied Computing (SAC), held in Riva del Garda, Italy. They have been expanded, revised, and reviewed for inclusion in ACR, and we are proud that all of them are high quality papers. I would like to take this opportunity to thank all the committee members and attendees who made the conference a great success.

ACR provides you with a platform for sharing novel ideas among practitioners and professionals in various fields of applied computing. In addition, we have provided excellent service to various technical communities and to the scientific computing society in a productive manner. We are working with the ACM SIG Governing Board to further expand SIGAPP by increasing membership. Also, we are working hard so that ACR can appear in Science Citation Index (SCI) in the near future. Your support and cooperation would be highly appreciated.

I would like to thank the authors for contributing the state-of-the-art methods in their research area. I am grateful to the highly qualified peer reviewers who coordinated an outstanding lineup of technical paper reviews. I especially wish to thank Dr. Hisham Haddad for his dedicated work and continuous support. This issue of ACR couldn’t have been published without significant efforts made by everyone, and I want to express my sincere gratitude to all of them.

Sincerely,

Sung Shin
Editor in Chief & Chair of ACM SIGAPP

Next Issue

The planned release for the next issue of ACR is September 2012.
SAC 2012 Report

The 27th Annual edition of the ACM Symposium on Applied Computing (SAC) was held in Riva del Garda, Trenton, Italy, March 2012. The conference was hosted by The Microsoft Research – University of Trento Centre for Computational and Systems Biology (COSBI). This year, SAC has overwhelming success with over 91% attendance rate. This is due to the exceptional dedication of the local organizing committee and the generous support the conference received from local sponsors, including COSBI, Provincia Autonoma Di Trento, Riva Del Garda Congressi, Comune Di Riva Del Garda, Istituto d’Istruzione G. Floriani, Open Viaggi Vacanze, and Banca di Trento e Bolzano/Bank fur Trent und Bozen. The Steering Committee extends its thanks and gratitude to the local sponsors for their generous contributions.

The Call for Track Proposals resulted in accepting 34 tracks. The selections were made based on the success of those Tracks in the previous editions of SAC as well as targeting new and emerging areas in applied computing. The Tracks were organized into five different themes: AI & Agents, Distributed Systems, Information Systems, Software Development, and System Software & Security. The Symposium Proceedings and the technical presentations were focused around these themes to form a series of related track sessions.

The Call for Papers attracted 1056 paper submissions from over 60 countries. All submitted papers underwent the blind review process and 270 papers were finally accepted as full papers for inclusion in the Conference Proceedings and presentation during the Symposium. The final acceptance rate for SAC 2012 is 25.6% among all tracks. In addition, 76 papers that received high review scores were invited as short papers for presentation during the Poster Program. The posters were presented over two sessions on Thursday. Monday Tutorials Program offered 5 tutorials covering variety of topics and attracting over 130 attendees. The four-day Technical Program consisted of keynote sessions and research presentations from all 34 tracks covering a wide range of topics on applied computing and emerging technologies. For more details about the technical program, please visit http://www.acm.org/conferences/sac/sac2012/.

The success of SAC 2012 was made possible through the hard work of many people from the scientific community who had volunteered and committed many hours to make it a successful event. Much credit goes to all the Track Chairs and their Program Committees. On behalf of SAC 2012 Organizing Committee and the Steering Committee, we congratulate all of the authors for having their papers accepted in their respective Tracks. We also wish to thank all of those who made this year's technical program a successful one, including the speakers, track chairs, reviewers, program committee members, session chairs, presenters, and attendees. Special thanks go to the High School student volunteers from Istituto d’Istruzione G. Floriani, and to their Principle and Teachers. Their help was a key to SAC success.

The social program was organized and made possible by the local organizing committee. Credit goes to the local sponsors and members of the local committee for their hard work to plan and execute all aspects of the conference. The social program included daily on-site lunches and coffee breaks, allowing attendees to gather, converse, and network. On Tuesday, SIGAPP Chair hosted a reception for all attendees, and on Wednesday the city Mayor hosted a Welcome Cocktail at La Rocca (the city museum). The main social event is the Banquet that was held at the CAFFÈ CASINÒ Cita di Arco (a historic palace in the area). The Banquet event included live entertainment and the awards ceremony during which best papers and poster awards were presented. Awards recognizing the local arrangement committee and the local sponsors were also presented. Please check SAC 2012 website for more information about the conference events, best paper/poster awards, and some pictures of the event.
The preparation for SAC 2013 is underway. The conference will be held in Coimbra, Portugal. It is hosted by the Institute of Engineering of the Polytechnic Institute of Coimbra (ISEC-IPC). The local organizing committee is led by Dr. Nuno Ferreira, from the Polytechnic Institute of Coimbra. A member of SAC Steering Committee, Dr. Sung Shin, will serve as the Conference Co-Chair alongside Dr. José Carlos Maldonado, from University of São Paulo, São Paulo, Brazil. The conference dates are set for March 18 – 22, 2013. We hope you consider SAC 2013 for your next submission and hope to see you there next year. A listing of the organizing committee is posted at http://www.acm.org/conferences/sac/sac2013/.

The Steering Committee is currently soliciting proposals for hosting SAC 2014 and beyond. Please contact any member of the steering committee for “SAC Hosting Guidelines” if you are interested in hosting SAC in the near future.

Best Regards to all,

Hisham M. Haddad
Member of SAC Organizing and Steering Committees

Notice to Contributing Authors

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library and in any Digital Library related services
- to allow users to make a personal copy of the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will refer requests for republication directly to you.
A spatiotemporal extension for dealing with moving objects with extent in Oracle 11g

Luís Matos
IEETA – University of Aveiro
Campus Universitário de Santiago
3810–193 Aveiro, Portugal
luismatos@ua.pt

José Moreira
IEETA/DETI – University of Aveiro
Campus Universitário de Santiago
3810–193 Aveiro, Portugal
jose.moreira@ua.pt

Alexandre Carvalho
INESC/DEI – University of Porto
R. Dr. Roberto Frias, 378
4200–465 Porto, Portugal
alexandre.carvalho@inescporto.pt

ABSTRACT

This paper deals with the design and implementation of a data model and operations for dealing with continuously changing spatial data in Oracle 11g object-relational DBMS. The data model relies on abstract data types but we introduce modifications to the internal structure of the spatiotemporal data representations proposed in the literature, to reduce storage requirements and to enable the reuse of data during the execution of queries. We show how to implement spatiotemporal operations relying on the spatial functions released by the underlying DBMS and how to use the alternative data representations to reduce the volume of temporary data created in the evaluation of spatiotemporal operations. We also demonstrate how to use the proposed data types and operations for storage and manipulation of moving objects using SQL. Finally, we discuss on the advantages and disadvantages of the proposed solutions.¹

Categories and Subject Descriptors

H.2.8 [Database management]: Database applications—Spatial databases and GIS; H.2.3.3 [Database management]: Languages—query languages

General Terms

Spatiotemporal data management

Keywords

data models and query languages, object-relational databases, spatiotemporal data, moving objects

1. INTRODUCTION

Research on spatiotemporal databases focuses on the efficient representation, management and querying of spatial information that changes over time. The properties describing spatial information may change discretely or continuously over time. The former assumes that the values of spatial properties change instantaneously at certain time instants, hence the values of those properties between consecutive changes are constant. The latter describes the values of spatial properties as continuous functions of time. Some authors claim that discrete changes can be modeled as a specialization of continuous changes by using trivial constant functions [5, 16].

The most well-known data models and query languages to deal with spatiotemporal information are based on constraint databases [6] and Abstract Data Types (ADT) [7, 5]. The constraint databases data model [6] is very specialized and its transposition into relational or object-relational database management systems (ORDBMS), the current commercial database standards, would be barely feasible [3]. On the other hand, the representation of ADT can be smoothly integrated into ORDBMS. This approach has been used to develop a spatiotemporal extension implemented on the top of Secondo, an academic and research database database prototype [12, 4], but it has not been fully implemented on ORDBMS.

There have been many attempts in recent years to develop domain-specific spatiotemporal extensions on the top of ORDBMS such as Oracle, Postgres or DB2, for a diversity of fields of application. The implementation of such extensions is an arduous task and the spatiotemporal support that is provided by most proposals is limited and oriented towards the requirements of the applications they are intended for. These observations indicate that the management of spatiotemporal information in standard DBMS is an important issue and that the release of generic spatiotemporal extensions would be an important advance in many domains of knowledge.

This work focuses on the design and implementation of spatiotemporal data types and algorithms for dealing with continuously changing spatial data on ORDBMS. The data model is based on ADT, but we also introduce modifications to the representation of moving regions proposed in [7, 5] to reduce memory requirements in storage and in the size of the intermediate data structures created during the evaluation of spatiotemporal operations. We present several algorithms to implement the spatiotemporal operations (projection, predicates and clipping) relying on the spatial features of the underlying DBMS that are not covered in previous spatiotemporal extensions for ORDBMS. We also discuss the merits and the weaknesses of data representations and the algorithms that we propose, and we identify open issues for future research.

The remainder of this paper is organized as follows: section 2 presents a brief research overview on spatiotemporal databases, focusing on data models and languages suitable for implementation on ORDBMS; section 3 defines our data model for the representation of spatiotemporal data; section 4 presents a selected set of algorithms for the implementation of the most representative spa-

¹This work is based on an earlier work: SAC ’12 Proceedings of the 2012 ACM Symposium on Applied Computing. Copyright 2012 ACM 978-1-4503-0857-1/12/03. http://doi.acm.org/10.1145/2245276.2245280.
tiotemporal operations in databases; section 5 demonstrates how to store and query moving objects’ data using our Oracle 11g spatiotemporal extension; section 6 discusses the solutions proposed in this paper and compares them with related works; section 7 concludes and presents guidelines for future research.

2. SPATIOTEMPORAL DATABASES

We will consider that a spatiotemporal object, also referred to as moving object, is a spatial object that may change its position or shape continuously over time [7].

2.1 Spatiotemporal database architectures

There are three main architectural approaches for representation of time and space in databases: layered, monolithic and extensible approaches [1].

Layered architectures implement advanced features in a layer that resides on top of an off-the-shelf DBMS. Applications which require these advanced features communicate with the DBMS through the layer interface. The main advantage of this approach is to provide very fast development of new features but, due to a clear separation of the layer from the DBMS core, transaction management and query optimization are only guaranteed for the DBMS level [2].

Monolithic architectures overcome these problems, since every specific feature and service is provided directly by the DBMS kernel. Yet, this architecture carries a daunting task which is the implementation process [11].

Extensible architectures try to combine the best from previous architectures: the support for application-specific or domain-specific features is placed within the DBMS but outside of its core. Generally, this is only possible if the DBMS supports the development of extensions. Extensible architectures can benefit from the support for transaction management, indexing, and query optimization. Moreover, the development of an extension is less complex than the development of new functions inside the DBMS kernel.

2.2 Data models and languages

A moving object \( m \) can be defined as a triple \( (S, I, f) \) where \( S \subset \mathbb{R}^2 \) denotes the geometry of a spatial object, \( I \subset \mathbb{R} \) is a time interval and \( f : \mathbb{R}^2 \times \mathbb{R} \rightarrow \mathbb{R}^2 \) is a continuous function defining a transformation, i.e., the changes in the position or geometry of \( S \) during \( I \). The semantics of an atomic moving object is \( \text{Object} = \{(x, y, t) \in \mathbb{R}^2 \times \mathbb{R} | ((x', y')) \in S \land t \in I \land (x, y) = f(x', y', t) \} \) [3]. The most common classes of geometries for spatial objects are point, line and region, but there is no commonly accepted definition or terminology in the literature. The continuous function \( f \) enables the representation of translations, scaling and transformations in the geometry (deformation) of the moving objects. The representation of complex transformations, such as, splitting or merging moving objects, may also be an important feature in some applications. For a systematic approach on handling changes in spatiotemporal phenomena, refer to [8].

The most well-known approach for the representation of spatiotemporal data in databases uses Abstract Data Types (ADT) [7]. The paper proposes an abstract data model, where complex data types are constructed from simpler data types. This is a particularly interesting approach for implementation on extensible database architectures. It defines Base data types (int, real, string and bool), Spatial data types (point, points, line and region) and a temporal data type (instant). The type constructor range(\( a \)) produces types whose values are finite sets of pairwise disjoint intervals over the domain of \( a \in \text{Base} \cup \text{Spatial} \). The most important are moving(point) and moving(region).

The discrete data model [5] proposes a sliced representation, where moving values are represented as an ordered collection of slices (units). Each unit represents the evolution of a spatial object during a time interval by a linear function. Moving point and moving segment are fixed size units, while moving region is a variable size unit. A moving segment unit is defined by a pair of moving points units and a moving region unit is defined as a set of moving segments units. The signatures, the semantics and the algorithms for the implementation of a large set of spatiotemporal operations using this data model are presented in [12].

In [17] it is proposed an SQL extension for the management of spatiotemporal data where time and spatial data are defined in terms of spatial quanta. The representation of the quanta is isomorphic to natural numbers. The formalization of the data model and operations is elegant and compact but this framework was designed to deal only with discrete changes in time and space.

2.3 Spatiotemporal ORDBMS

Current ORDBMS are extensible architectures that enable developing extensions for dealing with complex data types, which can be smoothly attached to the database and manipulated via SQL. Notable examples are the spatial extensions for dealing with geographical information, such as, Oracle Spatial, DB2 Spatial extender or PostGIS.

Hermes is an extension developed on Oracle to deal with discretely or continuously changing spatial data [16]. The spatial data types and operations arise from Oracle Spatial and the temporal data types are implemented using the TAU Temporal Literal library [13]. This extension provides a large variety of moving data types, including moving points, moving lines and moving regions. The moving segments that connect the vertices of moving lines or moving regions can be straight lines or arcs. It also includes data types for moving regions with holes and moving collections to model collections of objects of any moving data type. The movement of the objects can be modeled using linear, polynomial of first and second degree, square root of polynomial of second degree or constant functions. It includes a set of spatiotemporal operations to deal with moving points in location-based services applications [16, 14]. However, it only presents a limited number of algorithms to implement spatiotemporal operations for the other moving types. For details on the implementation of these works we recommend reading the technical report [15].

The SpatioTemporal Object Cartridge (STOC) is also an extension developed on Oracle for the management of discretely or continuously changing spatial data. This extension delivers data types to represent continuously changing moving points [9]. Moving rectangles and discretely changing moving lines are covered in [18]. The so-called spatial object-relational operations on moving data types return a moving string, denoting the spatial relationship between the objects, or a moving number for distances. In [10], the
authors present a similar extension for dealing with objects moving in a three-dimensional space (e.g., aircrafts) that is oriented towards the implementation of applications in defense simulation.

There are several works on dealing only with discretely changing spatiotemporal data. In opposition with previous works, classified as extensible architectures, these proposals usually follow a layered approach and in many cases are special purpose developments oriented towards the requirements of specific domains of applications, such as, environment applications (e.g., bio-diversity or land usage applications), crime analysis or travel behavior. However, the management of discretely changing spatial data is out of the scope of this paper.

3. SPATIOTEMPORAL DATA REPRESENTATION

This work focuses on the representation, manipulation and querying of moving objects, i.e., spatial objects that may change position or shape continuously over time, on ORDBMS. The aim is to provide a spatiotemporal extension for developing applications dealing with current and past information about moving objects, via SQL. The extension is based on the ADT approach proposed in [7, 5] and was developed and tested on Oracle 11g ORDBMS and Oracle11g Spatial.

We have chosen to preserve the same representation of Oracle Spatial sdo geometry type and use variable-size arrays to define the vertices of moving regions. Nevertheless, we have decided to use nested tables to represent the collections of units in moving objects’ representations, because these structures are easier to update and the number of elements is not predetermined. They can be both stored in database tables and manipulated using SQL. In the following definitions we will use [⋯] to denote variable-size arrays and ⟨⋯⟩ to denote nested tables.

3.1 Data definitions

We map the base and the spatial data types directly into Oracle 11g native and Oracle11g Spatial data types.

Definition 1. A real and an int are mapped into the Oracle’s number data type. An instant is mapped into the Oracle’s date data type.

Definition 2. A time interval denotes a period of time between two time instants:

\[ \text{interval} \equiv \{ (\text{begin}, \text{end}) \mid \text{begin} \in \text{instant} \land \text{begin} \leq \text{end} \}\]

The type sdo geometry in Oracle 11g Spatial is a container for storing primitive geometries such as points, lines and polygons, and complex geometries composed of collections of primitive ones. The edges of lines and polygons can be straight lines as circular arcs. The elements in a collection of a complex geometry may be of different types.

Definition 3. The spatial types point, line, and region are mapped into Oracle’s Spatial sdo geometry type.

We also need to store and manipulate multidimensional values, such as, the position or the speed vector of a moving point. We have just recognized that sdo geometry is adequate to handle multidimensional values, but it has a complex structure: the representation of a simple value such as a point, requires storing its coordinates, as well as the kind of geometry and layer’s information. Hence, for practical reasons, we have included in our framework a data type for dealing with multidimensional values.

Definition 4. An nValue is two-dimensional value

\[ nValue \equiv \{(x, y) \mid x, y \in \text{real}\} \]

Definition 5. A set of time intervals is an ordered collection of pairwise disjoint time intervals:

\[ \text{intervalSet} \equiv \{ \text{interval} \mid \text{interval} \in \text{interval} \land (\text{interval}_i \ land \leq \text{interval}_{i+1}. \text{begin}) \}\]

Definition 6. A moving point is an ordered collection of units composed by a time interval \( \text{t} \), the position \( \nu \) of the moving point at \( \text{t }. \text{begin} \) and a variability function \( \nu v \) describing the movement of the point during \( \text{t} \).

\[ \text{mPoint} \equiv \{ (\text{t}, \nu, \nu v) \mid \text{t} \in \text{interval} \land \nu, \nu v \in \text{nValue} \land \]

\[ (\text{interval}_i \land \leq \text{interval}_{i+1}. \text{begin}) \land \]

\[ (\text{interval}_{i+1}. \text{begin} - \text{interval}_i. \text{end} \leq \xi_i) \land \]

\[ \nu v o f i . x - (\nu v o f i . x + \nu v o f i . y \cdot (\text{interval}_i. \text{end} - \text{interval}_i. \text{begin})) \leq \xi i \land \]

\[ \nu v o f i . y - (\nu v o f i . y + \nu v o f i . y \cdot (\text{interval}_i. \text{end} - \text{interval}_i. \text{begin})) \leq \xi i \}\]

Condition a) ensures that the time intervals are pairwise disjoint and so, a moving point cannot have two different positions at the same time. \( \nu v \) is a linear function of time that returns the position of a moving point for any \( \text{t} \in \text{T} \).

\[ (x', y') = (x + \nu v . x \cdot t, y + \nu v . y \cdot t) \]

Condition b) validates the position of a moving point in consecutive time intervals. \( \xi_i \) and \( \xi_i \) are constants denoting a temporal and a spatial precision. This means that when two times intervals \( \text{T}_1 \) and \( \text{T}_2 \) are close (or touch) in time, the position of a moving point at the end of the first one (\( \text{T}_1 \)) must be close (or equal) to its position at the beginning of the second one (\( \text{T}_2 \)).

The internal structure of our moving region data type introduces several modifications relatively to the data representations proposed in [7, 5, 14]. These are also discussed in section 6.

To start with, we need the following auxiliary data types:

\[ \text{mPointId} \equiv \{ (\text{id}, \text{mp}) \mid \text{id} \in \text{int} \land \text{mp} \in \text{mPoint} \}, \]

denoting a uniquely identified moving point and

\[ \text{mPointIdSet} \equiv \{ (\text{pid}) \mid \text{pid} \in \text{mPointId} \}, \]

denoting a collection of moving points identifiers such that \( i \neq j \Rightarrow \text{pid} i \neq \text{pid} j \). Let us also consider that for any \( \text{mp} \in \text{mPoint} \) and \( s \in \text{mPointSetId} \),
denote the collection of all time intervals in a moving point’s representation and the collection of all moving points identifiers in a collection of moving points, respectively.

Definition 7. A moving region is an ordered collection of units composed by a time interval \( \bar{i} \) and a set of the moving points identifiers defining its geometry.

\[
mRegion \equiv \{ \langle \bar{i}, s \rangle \mid \bar{i} \in \text{interval} \land s \in mPointSet \land (t_{s, \text{end}} \leq t_{i+1, \text{begin}}) \land |s| \geq 3 \land \forall s_j \in s \{ s_j, \text{pid} \}} \cap \bar{i} \equiv \text{this}\bar{i} \}
\]

The time intervals must not overlap and the set of moving points must hold three points, at least. The last condition ensures that a moving region’s unit cannot hold points whose position is undefined during any instant within its temporal domain. It is assumed that the region is always closed, i.e., the last moving point is implicitly connected to the first one.

It is important to point out that the units in an \( mRegion \) and the units of the \( mPoints \) defining its vertices are independent. Thus, it is possible to update a \( mPoint \) and keep the corresponding \( mRegion \) unchanged. Indeed, it is only necessary to insert a new unit in a \( mRegion \) when the number of moving points defining its shape changes. This allows minimizing the costs of updates, reducing the size of the data structures and improving the performance of the operations dealing with moving regions.

We also use a single-typed table to store the moving points defining the geometry of all moving regions in a database table. Hence, these moving points may be shared by several moving regions.

We will consider that a \( mValue \) is a specialization of \( mPoint \) and represents one or two-dimensional moving values, \( \text{time} \) is a generalization of \( \text{instant} \), \( \text{interval} \) and \( \text{intervalSet} \), and a \( mObject \) is a generalization of \( mPoint \) and \( mRegion \). These definitions are only used to support the representation of the signatures of the operations in more compacted forms and they do not have any impact on the expressiveness of the data model.

4. SPATIOTEMPORAL ALGORITHMS

This section presents the algorithms for the implementation of the main spatiotemporal operations, namely, projections, predicates and clipping. Due to space restrictions, we only present the algorithms that illustrate better the main issues raised by the implementation of these operations.

As the current version of the data model does not include a data type for the representation of multiple moving regions, the spatial clipping algorithms can only operate with convex shapes. This ensures that the intersection of two moving regions returns always a single moving region.

The implementation of operations dealing with temporal or numerical features of moving objects is, in most cases, straightforward. In the following, we will just introduce their signatures and present a brief description of them. However, the operations dealing with the spatial features of moving objects are more complex and we will present the main steps of the algorithms to implement them. The general strategy that we use to implement these operations has two (for projections and predicates) or three main steps (for clipping):

1. Transform a moving object into spatial (\( \text{sd}o_{\text{geometry}} \)) objects
2. Perform spatial operations using Oracle11g Spatial functions
3. Assemble the results into a moving object

In the following algorithms these steps will be tagged using the notation \([\#x]\).

4.1 Projections

Projections are operations with the signature \( mObject \rightarrow a \), where \( a \) stands for \( \text{intervalSet} \), \( mValue \) or \( \text{spatial} \).

The values of \( a \) may be standard projections, e.g., the temporal units (\( \rightarrow \text{intervalSet} \)) or the speed (\( \rightarrow mValue \)) in a moving object’s representation.

A spatial projection gives the footprint of a moving object. Hence, the spatial projection of a moving point \( mp \) is a line obtained as follows:

\[
\begin{align*}
\text{initialize} \quad r &:= \text{null} \quad | \quad r \in \text{line}; \\
& \text{for each} \quad u \in mp; \\
& \text{get} \quad p_u := u_0.v; \\
& \text{compute} \quad p_x := u_0.v + u_0.vv \times u_n.\text{end}; \\
& \text{add} \quad p_x, p_u \text{ to } r; \\
& \text{return} \quad r;
\end{align*}
\]

This algorithm performs a simple scan of the units \( u \in mp \), computes, for each iteration, the corresponding starting and the ending positions \((1,2)\), and adds the line segment formed by these two points into the result \( r \) \((3)\).

The following algorithm computes the spatial projection of a moving region \( mr \) during a single unit \( u_n \in mr \). The result is a region.

\[
\begin{align*}
\text{initialize} \quad r &:= \text{null} \quad | \quad r \in \text{region}; \\
& \text{for each pair} \quad s_1, s_{i+1} \subset u_n; \quad \{s\}; \\
& \text{mp}_{s_1} := s_1.pid.mp; \quad \text{mp}_{s_{i+1}} := s_{i+1}.pid.mp; \\
& \text{compute} \quad x_a := \text{projection} \text{ mp}_{s_1} \text{ during } u_n.\bar{i}; \\
& \text{compute} \quad x_b := \text{projection} \text{ mp}_{s_{i+1}} \text{ during } u_n.\bar{i}; \\
& \text{create region } rr \text{ from } x_a, x_b, \text{mp}_{s_1}, \text{mp}_{s_{i+1}}; \\
& \text{compute} \quad r := r \cup rr; \\
& \text{loop;}
\end{align*}
\]

The first iteration in the loop is depicted in Figure 1 (left side). The dashed lines are the projections of the moving points 1 and 2 obtained in \((2,3)\). These statements are a temporal clipping operation (see section 4.3), to obtain a moving point representation for the interval of interest, followed by a spatial projection. The region in gray, created in \((4,5)\), gives the space covered by the first segment during \( u_n.\bar{i} \). Notice that this segment, defined by moving points 1
and 2, starts as a ‘point’. The region in the middle gives the space covered by the second segment (moving points 2 and 3), and so on. The result is the union of the regions computed during the loop iterations (5) with the projections of \( w_n \) at \( w_n.t.begin \) (initial shape) and at \( w_n.t.end \) (final shape) of the moving region (6, 7, 8).

The algorithm below shows how to compute an intersection predicate between a moving object and a static spatial object, but the predicates having a second argument of a non-moving data type are implemented through a projection followed by a numerical, temporal or spatial operation provided by the underlying database system.

The algorithm below shows how to compute an intersection predicate between a moving \( m \) and a static spatial object \( g \).

```
compute x := spatial projection of m;
if (x is null) return false; else return true;
```

Figure 2 illustrates the intersection between a moving point (left side) or a moving region (right side) with a static spatial object, but the algorithms to compute other topological, directional or distance based predicates, are similar.

### 4.2 Predicates

Predicates are operations that return the values \( \text{true} \) or \( \text{false} \). The signature is \( mObject \times \alpha \rightarrow \text{boolean} \), where \( \alpha \) stands for \( \text{time}, \text{number}, \text{nValue}, \text{spatial} \) or \( mObject \).

The predicates having a second argument of a non-moving data type, are implemented through a projection followed by a numerical, temporal or spatial operation provided by the underlying database system.

The following algorithm implements the clipping of a moving point \( mp \) by a static region \( g \) using an intersection relationship, as depicted in (Figure 3, top).

```
initialize r := null \mid r \in mPoint;
for each \( u_n \in mp \):
compute x := projection of \( u_n \) during \( t \);
if \( s \subseteq g \) add \( u_n \) to \( r \);
if \( x \) is null return false;
return r;
```

Notice that the value of \( r \) defined in the loop is isomorphic to real numbers and so, its implementation is not straightforward. To deal with this issue, we make a discretization of the temporal dimension using a temporal quantum (\( \Delta t \)). The quantum gives the increment of \( t \) between iterations. The statements inside the loop test whether the spatial projections (1,2) of the two moving regions at time \( \Delta t \) intersect (3). When so, the algorithm finishes and the predicate yields \( \text{true} \). If the projections of the two regions never intersect, then the algorithm yields \( \text{false} \).

### 4.3 Clipping

Clipping operations enable filtering spatiotemporal values according to a given criteria. The result is a subset of the initial spatiotemporal value for which the criteria holds. The signatures are \( \text{moving} \times \text{instant} \rightarrow \text{spatial} \) and \( \text{moving} \times \alpha \rightarrow \text{moving} \), where \( \alpha \) stands for \( \text{interval}Set, \text{number}, \text{spatial} \) or \( mObject \).

The clipping of a moving object at a time instant is a special case because the result is a static object: a point or a region. The algorithms are easy to implement and we skip their presentation in this paper.

It is important to notice that if \( mr \) is a moving region, then \( mr.s[1] \) holds the identification of moving points rather than their internal representations. Hence, copying or splitting an unit \( u_n \in mr \) requires inserting new records into the nested table of moving regions, but the shared nested table of moving points keeps unchanged, saving space.

The following algorithm implements the clipping of a moving point \( mp \) by a static region \( g \) using an intersection relationship, as depicted in (Figure 3, top).

```
initialize r := null \mid r \in mPoint;
for each \( u_n \in mp \):
compute x := projection of \( u_n \) during \( t \);
if \( x \subseteq g \) add \( u_n \) to \( r \);
if \( x \) is null return false;
return r;
```

This algorithm performs a spatial projection for each unit \( u_n \in mp \) (1) and checks for intersections with \( g \) (2). If the projection (a line) is inside \( g \) (3) then \( u_n \) is added to the result; if it partially overlaps (intersects) \( g \) (4) then the algorithm computes the point(s) where they intersect, recalculates the boundaries of \( u_n.t \), updates \( u_n.v \), if required, and adds the clipped unit to the result; otherwise, \( u_n \) is discarded.
The procedure to convert the intersections in two consecutive snap-input geometries. The cloned points are explained below.

A Steiner point is an extra vertex that is not a member of the input geometries. The siblings of the moving points in both geometries are obvious (see the solid lines connecting \( m_{i1} \) or \( m_{i2} \) or Steiner points (white dots) or cloned points (gray dots). A Steiner point is an extra vertex that is not a member of the input geometries. The cloned points are explained below.

The procedure to convert the intersections in two consecutive snapshots into a moving object requires finding the siblings in the two geometries (Figure 5).

1) assigns an index value to each position in the arrays of \( A \) and \( B \) and 2) decides the pairs of siblings using a linear transformation: 

\[
\text{sibling}(id_x) = \text{round}(\frac{|y|}{x} \times id_x),
\]

where \(|y|\) stands for the length of the array with fewer elements (three elements in \( A \)) and \( |x| \) the length of the other array (five elements in \( B \)). Hence, \( s_2 \) and \( m_{p1} \) have two siblings each and must be cloned. The same procedure holds for the points in the arrays \( m_{p5} \) through \( m_{p6} \). The moving points \( m_{p3} \) and \( m_{p4} \) just vanish during the elapsed time \( \Delta t \) and do not need any special treatment.

Notice that: 1) the arrays are circular (the last index connects with the first one); 2) if an intersection has only Steiner points, it is necessary to choose artificially a reference point to start with the finding siblings step. In the latter case, we use a simple heuristic rule: the siblings are the upper (or upper left, if ties occur) points in the two geometries.

The steps after finding the siblings are:

- Create new moving points for the clones discovered in previous step.
- Calculate the variability functions for the pairs holding Steiner points and clones. This is a simple formula, as the algorithm holds the position of two points (pair of siblings) and \( \Delta t \).
- Compare the array of moving points identifications in the last unit of \( r \) with the arrays of moving points identifications just created using the following criteria:
  1. The arrays have the same length.
  2. The moving points with a direct match (the solid lines) have the same indexes in both arrays.

When these two conditions hold, it is enough to update the upper bound of the time interval in the last unit of \( r \). This is a coalescing operation. Otherwise, it is necessary to insert a new unit into \( r \).

- Compare the variability functions of the pairs holding Steiner points with the corresponding moving points (by array positioning) in the last unit of \( r \), and insert new units into the moving points where the variability functions do not match.

Notice that, as our data model allows sharing the moving points that define the vertices of moving regions, it is only necessary to create new moving points for the clones. The other moving points are shared with those in the input moving regions.

For simplicity, exceptions such as the initializations or the processing of the first iteration in the loop are omitted.

The algorithm to implement a clipping operation using a moving point and a static region is similar. The main differences are: the second projection (2, 4) is not required and the points in the vertices of the static region must be cloned and converted into moving points where all parameters in the variability function are zero.

5. QUERY LANGUAGE

This paper presents an extension for dealing with moving objects that may change position or location continuously over time (e.g., icebergs) in an Oracle 11g database. We are currently working on tools for semi-automated extraction of spatial objects with complex
shapes from a sequence of satellite or aerial images, but in this paper, for demonstration purposes, we will use some simple generated moving objects.

### 5.1 Creating the data model

Figure 6 depicts the spatial transformations of a moving object during a time interval. The numbers in the vertices of the polygons represent the identifiers of each moving point.

**Figure 6: A simple moving region.**

This figure puts in evidence that the number of vertices delimiting the moving object's shape between two consecutive snapshots (satellite or aerial images) may differ. Indeed, they probably will differ when dealing with real-world objects. In those cases, for instance, to represent the transition from the pentagon in the first snapshot to the hexagon in the second one, it is required to add an extra moving point into the initial shape. In this example we chose to clone the moving point in vertex one. As each moving point has its own movement analytical formula, each one may move towards a different vertex at the ending shape. Notice that the choice of the initial location for the extra moving points has a major impact on the intermediate shapes of the moving objects. This issue is related with the quality of data representations and is out of the scope of this paper. Similarly, the transition from the hexagon in the second snapshot to the rectangle in the third snapshot has two pairs of moving points that converge to the same vertices and so one of them in each pair should be deleted from the moving region's topology at the end of the transition.

In the following, we demonstrate how to create such moving region using the spatiotemporal extension proposed in this paper. First, it is required to create a table to store the moving objects' data (Figure 7). For simplicity, the table has only two attributes: a numeric object identifier and a nested table to store the objects' movement. The nested table is defined by a time interval, the coordinates of the moving object at the beginning of the time interval and a variability function describing the movement of the point during that time interval. This procedure must be repeated for all moving points needed to represent the topology of the moving object.

**Figure 7: SQL command to create the moving objects table.**

```
CREATE TABLE MovingObjects
    (movingObjectID NUMBER PRIMARY KEY,
     objectMovement MRegionUnitTable)

SELECT objectMovement STORE AS objectRoute
    ( PRIMARY KEY (NESTED_TABLE_ID, ti.startDate)
      ORGANIZATION INDEX );
```

It is also necessary to create a shared table to store the moving points defining the vertices of all objects in the moving regions table. As the SQL statements to create these tables are similar to previous ones, they are omitted.

Figure 8 shows how to insert a moving point into the shared table.

**Figure 8: SQL command to insert a moving point into the moving points table.**

```
INSERT INTO MPointTable VALUES
    ( MPoint (1, PointArray (Point(6.0, 19.0), Point(15.5, 14.33), Point(30.0, 8.0)),
      DateArray (TO_DATE( '01-01-2012 15:04:00', 'DD-MM-YYYY HH24:MI:SS'),
                 TO_DATE( '01-01-2012 15:02:00', 'DD-MM-YYYY HH24:MI:SS'),
                 TO_DATE( '01-01-2012 15:00:00', 'DD-MM-YYYY HH24:MI:SS')));
```

**Figure 9: SQL command to insert a moving object into the database.**

```
CREATE TABLE MRegionsUnits
    ( PRIMARY KEY (NESTED_TABLE_ID, ti.startDate)
      ORGANIZATION INDEX );

INSERT INTO MRegionsUnits VALUES
    ( 15, MRegionUnitTable (MRegionUnit TimeInterval( TO_DATE( '01-01-2012 15:00:00', 'DD-MM-YYYY HH24:MI:SS'),
                                                    TO_DATE( '01-01-2012 15:02:00', 'DD-MM-YYYY HH24:MI:SS'),
                                                    TO_DATE( '01-01-2012 15:04:00', 'DD-MM-YYYY HH24:MI:SS')),
       NumberArray(1, 3, 4, 5, 6),
       MRegionUnit TimeInterval( TO_DATE( '01-01-2012 15:02:00', 'DD-MM-YYYY HH24:MI:SS'),
                                TO_DATE( '01-01-2012 15:04:00', 'DD-MM-YYYY HH24:MI:SS'),
                                TO_DATE( '01-01-2012 15:06:00', 'DD-MM-YYYY HH24:MI:SS')),
       NumberArray(1, 2, 3, 4, 5, 6),
       MRegionUnit TimeInterval( TO_DATE( '01-01-2012 15:04:00', 'DD-MM-YYYY HH24:MI:SS'),
                                TO_DATE( '01-01-2012 15:06:00', 'DD-MM-YYYY HH24:MI:SS'),
                                TO_DATE( '01-01-2012 15:08:00', 'DD-MM-YYYY HH24:MI:SS')),
       NumberArray(1, 2, 3, 4, 5, 6) );
```

The moving objects data is stored as a sequence of MRegionsUnits defined by a time interval and a set of moving point identifiers that were previously inserted into the shared table (Figure 9).
The whole representation of the moving region is shown in Figure 10.

Figure 10: DB representation of a moving object.

The moving object with identification 15 has only three units (see [1]). The first is a moving pentagon, the second is a moving hexagon and the last one is an open unit that must be completed when a new snapshot with the most recent location and shape of the moving object will be available. All moving points defining the vertices of the moving region (mRegion) are stored into a single table (see [2]) and so, they may be shared. Each moving point has its own internal representation (see [3]) and the number of units of the moving points defining the moving region is independent. Hence, it is possible to add new units into the representation of the moving point with id = 1 (nested table [3]) and keep the representation in nested table [1] unchanged. In fact, it is only necessary to insert a new unit into the nested table [1] when the number of the vertices (moving points) changes (e.g., the transformation of the pentagon into hexagon).

5.2 Query operations

Having described the data model supporting the moving objects representation, this section presents three queries to demonstrate the main spatiotemporal operations developed in this work.

The query in (Figure 11) retrieves the spatial projection of the moving region defined in previous examples.

```
DECLARE
  mrut MRegionUnitTable;
  sdo SDO_GEOMETRY;
BEGIN
  SELECT objectMovement INTO mrut
  FROM MovingObjects
  WHERE movingObjectId = 15;
  sdo := MOV_SPT.movRegion2SDO(mrut);
END;
```

Figure 11: A spatial projection.

of the moving objects that intersect a rectangle with coordinates [(23,1), (23,5), (27,5), (27,1)], as depicted also in Figure 12.

```
SELECT movingObjectId
FROM MovingObjects
WHERE MOV_SPT.intersectSdoGeom (
  objectMovement,
  MOV_SPT.pointArray2SDOPolygon (
    PointArray (Point(23,1), Point(23,5),
               Point(27,5), Point(27,1))
  )
);
```

Figure 13: SQL command to select the moving regions that intersect a static region.

The `intersectSdoGeom` is a predicate (see 4.2) that receives a moving region (`objectMovement`) and a static region (the rectangle given by `PointArray(...)`) as arguments and returns `true` if they intersect at any time instant. The `pointArray2SDOPolygon` function transforms a set of points into a `SDO_GEOMETRY` value. As the moving objects table in this example has only one moving region and the movement of that region intersects the rectangle, then the result of this query is 15.

The last example (Figure 14) is a clipping operation (see 4.3) of a moving point and a static region.

```
DECLARE
  mrut MRegionUnitTable;
  sdo SDO_GEOMETRY;
BEGIN
  SELECT objectMovement INTO mrut
  FROM MovingObjects
  WHERE movingObjectId = 15;
  sdo := MOV_SPT.movRegion2SDO(mrut);
END;
```

Figure 12: The projection of the moving region in Figure 6 and a static rectangle.

3`SDO_GEOMETRY` is the Oracle 11g Spatial Data Option data type for representing generic polygons (regions)

The operation `clippingSDOGeom` receives a moving point and a static region, and returns another moving point composed by the parts of the initial moving point that intersect the static region. In this query it is assumed that the moving point with the identifier
SELECT MOV_SPT.clippingSDOGeom ( 
    MPoint(mpId, mov), 
    MOV_SPT.pointArray2SDOPolygon ( 
        PointArray (Point(4,2), Point(4,6), Point(10,6), Point(12,8), Point(12,2) 
    ) 
) 
FROM MPointTable 
WHERE mpId = 7;

Figure 14: SQL command showing a clipping operation.

7 was previously inserted into the MPointTable (see 5.1 for an example on how to insert values into a moving points table). This operation is depicted in figure 15.

Figure 15: The result of the spatial clipping operation.

6. DISCUSSION

The spatiotemporal extension presented in this paper is based on ADT. The representation of a moving point is similar to the original proposal [7, 5], which also has been followed in subsequent works [14, 16, 4, 10, 9, 18]. The representation of moving regions introduces the following modifications to previous works:

- The shape of a moving region is defined by an ordered collection of moving points, rather than moving segments. This means that in previous works, each vertex defining the geometry of a moving region is represented twice. Therefore, it is necessary to assure that they are always connected, i.e., the position of the ending moving point in a moving segment is the same as the position of the starting moving point in the next moving segment, along time; otherwise, the geometry of the moving region is invalid. This representation enables using circular arcs in [16]. In [5] it is claimed that it helps on the implementation of the plane sweep technique in geometric operations.

- The units representing the spatiotemporal behavior of a moving region are loosely coupled with the units of the moving points that define its geometry. This design choice allows reducing the number of units in a moving region representation, improving storage and memory requirements and decreasing the costs of update operations.

- The moving points that define the geometry of moving regions have unique identifiers and may be used in multiple moving regions. This means that the regions that move together may share the moving points in common, avoiding data duplication, and, more important, the moving regions resulting from spatiotemporal operations, e.g., clipping, may share the moving points that they have in common with the input moving regions.

We use two main approaches to develop the algorithms that implement the spatiotemporal operations:

- The general approach considers that any spatiotemporal operation can be implemented as a sequence of spatial operations over snapshots (projections) of the moving objects. This approach is simple to implement and it is orthogonal to the kinds of variability functions (linear, quadratic, etc.) that we use. The performance and the precision of the results are obviously dependent on the temporal quantum used. The main difficulty is to assemble the results of the spatial operations for each snapshot, back into a moving data type (just for clipping operations with moving regions). We have introduced a simple method to perform a sort of morphing between pairs of regions in consecutive snapshots, but the mapping of the sibling points is not perfect: 1) side effects may occur during small periods of time in the neighbourhood of the transitions, i.e., when adding or removing vertices in the resulting moving region; 2) the paths of the moving points may cross resulting in non-realistic shapes; and 3) the number of units created is not optimal due to those side effects. This issue requires further investigation.

- In some spatiotemporal operations, we use directly movement formulas to compute the geometry of the regions covered by a moving object and then execute spatial operations. In opposition to the previous one, this approach can only be used to implement some operations.

These issues have not been covered in other spatiotemporal extensions implemented on ORDBMS. The Hermes implementation [14] has focused on the operations for dealing with moving points for location-based services. The technical report [15] presents algorithms to compute the spatial projection of moving regions at a given time instant and the projection of a moving line made of straight-line segments whose moving points move linearly, during a time interval. The algorithms to compute predicates and clipping operations are not covered. The solutions presented in [18] focus on moving points and moving rectangles.

In [12], the authors present a comprehensive list of algorithms for the implementation of spatiotemporal operations, including predicates, projections, set operators and clipping, on the Secondo database prototype. These special-purpose solutions are full implementations and so they can be optimized as a whole, while our solutions can only be optimized in parts. However, these algorithms are harder to implement and they can only deal with movements described by linear functions. The utilization of other kinds of variability functions implies a significant reformulation of the algorithms.
7. CONCLUSION

Early works on spatiotemporal databases have focused on the definition of data models and query languages for dealing with discretely and continuously changing spatial data. Later, with the advent of mobile devices, research in this area has been largely oriented towards location-based services and the management of the so-called networked-moving objects. These objects are modeled as points (the size and shape are irrelevant) that move along networks (graphs), such as roads or rivers.

The data model presented in this paper follows the basic principles used in Secondo and Hermes, but we also introduce alternative data representations that allow reducing storage requirements and the size of the temporary data structures used in the evaluation of spatiotemporal operations.

We also present algorithms to implement the main spatiotemporal operations in moving objects databases, relying on the spatial functions made available by the underlying database, that are new in spatiotemporal extensions for ORDBMS. This framework was implemented on the top of Oracle11g DBMS and Oracle11g Spatial, but the solutions presented here may also be applied to other ORDBMS.

The data model presented in this paper does not include data types for the representation of moving lines, moving regions with holes or collections of moving regions. The latter would also enable extending the spatiotemporal clipping operations to deal with non-convex regions.

We are developing a tool to recognize and extract moving objects from satellite and aerial images, and to load them into a spatiotemporal database. Presently, this tool is able to delimit objects in an image, using segmentation techniques such as Mixture Modeling Thresholding, Otsu Thresholding and Snake, but the mapping of the vertices between two consecutive snapshots is performed manually. We are searching for algorithms and techniques in images processing literature, to develop automated or semi-automated mapping data acquisitions processes. This will help in obtaining larger data sets for the evaluation and comparison of the performance of the solutions proposed in this paper. It may also give useful insights to improve the solution that we have proposed to convert a sequence of static regions into a moving region.

8. REFERENCES


ABOUT THE AUTHORS:

Luís Miguel de Oliveira Matos was born in Caracas, Venezuela, in January 1985. He received his M.S. degree in computer and telematics engineering from University of Aveiro, Portugal, in July 2009. He is enrolled in the MAP-i Doctoral Programme. Currently, he is working at the Signal Processing Laboratory of Institute of Electronics and Telematics Engineering of Aveiro (IEETA) which is associated with the Department of Electronics, Telecommunications and Informatics (DETI) of University of Aveiro. His main research interests are biomedical and data compression and image/video coding. He has also worked in spatiotemporal databases management systems. Luís Matos can be contacted at luismatos@ua.pt.

José Moreira received his Ph.D. in Computer Science and Networks from the École Nationale Supérieure des Télécommunications de Paris (France) and the Faculdade de Engenharia da Universidade do Porto (Portugal) in 2001. He is currently an Assistant Professor at the Department of Electronics, Telecommunications and Informatics of the Universidade de Aveiro and a researcher at IEETA, a non-profit R&D institute affiliated to the same university. His main research interests cover spatiotemporal database systems and Geographical Information Systems. He is also interested on Semantic web and spatiotemporal data mining.

Alexandre Valle de Carvalho is an auxiliar professor at the Computer Engineering Department of Faculdade de Engenharia, University of Porto. He is also a senior researcher at INESC Porto. Currently his main research interests are on spatio-temporal data management and retrieval and spatio-temporal information visualization.
Abstract

Traditional Remote Procedure Call systems (RPCs) have a single point of failure at the server side. To address this issue, a number of approaches extend RPC syntax and semantic to provide fault tolerance. However, these solutions are neither transparent for the application programmer nor for the client. This paper describes FTRMI, a middleware platform that enhances the Java Remote Method Invocation (JRMI) with strong replica consistency, increasing fault tolerance. FTRMI is completely transparent for the client and does not require recoding on the server side.1

Categories and Subject Descriptors

C.2.4 [Computer-communication networks]: Distributed Systems

General Terms

Reliability, Algorithms

Keywords

JRMI, Distributed Systems, Replication, Fault Tolerance.

1. INTRODUCTION

Remote Procedure Calls (RPCs) aim at simplifying the development of distributed applications by hiding the complexity involved in the communication between two hosts, so that its syntax becomes similar to the local procedure invocation. Although RPCs are not new (see for example [5]), they experienced a significant boost with the advent of object-oriented languages, when a number of OO RPC systems like CORBA [27], DCOM [24] and RMI [8] emerged.

RPCs are one possible instance of the client/server model, where the client remotely executes procedures on the server. In general, each call is internally converted by the RPC middleware at the client in a device independent message, which includes an id of the object and of the method and the method's parameters. Upon receiving the message, the RPC middleware at the server translates the message to its internal format, passes the arguments to the method being invoked and forwards the reply to the client. This model lacks the scalability required to cope with a large number of simultaneous clients and the fault tolerance required to cope with failures at the network or the server. Scalability and fault tolerance can be addressed by replicating the service in multiple hosts, so that each runs a copy of the service.

Service replication can be challenging, especially when the service keeps state that must be consistently distributed to every replica to ensure correct responses. In practice, many RPC implementations delegate the problem of state consistency to a third-party, namely a database, which becomes in charge of providing a consistent view to every replica. This can be overwhelming for some services, which would otherwise dismiss the use of a database, provided that the value of some variables was kept consistent across the replicas. Unfortunately, ensuring state consistency in this scenario is hard, as one must cope with a number of distinct problems like faults and message ordering. As an example, note that to be consistent, every update to the state must be propagated to every replica in the same order, what may not happen for example if: i) a server crashes after notifying an update of the state to some (but not all) of the replicas; or ii) if two concurrent updates are triggered by distinct replicas.

A number of proposals have been made to address the state replication problem in Object Oriented RPCs (e.g. [9],[11],[12],[15],[18],[20],[21],[23]), releasing the burden of enforcing state consistency from the application programmer. However, most of them either require changes to the API (thus requiring a change on the interface of the client and server side) or to the messages exchanged between clients and servers. Both cases force clients to use non-standard middleware versions.

This paper describes the Fault-tolerant Transparent Remote Method Invocation (FTRMI) framework. The framework extends JRMI with an additional communication layer that multicasts every request to all server's replicas, simplifying the development of fault-tolerant services. FTRMI is completely transparent for both the client and server sides of distributed applications using JRMI what makes it appealing for legacy applications and for scenarios where the client code cannot be easily changed, for example when it is distributed over multiple management domains. The paper describes the framework, and compares its performance with another approach that is not transparent to regular JRMI applications.

The paper is organized as follows. Section 2 presents previous work related with our approach. The section addresses in separate Object Oriented RPC frameworks, group communication services and frameworks providing fault tolerance and load balancing for OO RPCs. FTRMI is described in Section 3. A comparison of its performance with one of the most similar approaches is presented in Section 4. Conclusions and future work are the focus of Section 5.
2. RELATED WORK

The implementation of FTRMI is based on the combination of two components. The Java RMI Distributed Object Framework (DOF) provides the standard remote method invocation interface for the clients (who invoke the remote objects) and the server (who implements the objects). A Group Communication Framework extends the reliable point-to-point network service provided by TCP/IP with rich communication capabilities for multiple participants. This section addresses previous work performed in both components as well as some of the past experiences in combining them to achieve fault tolerance in DOFs.

2.1 Distributed Object Frameworks

The role of Distributed Object Frameworks (DOF) is to simplify the development of distributed applications using object oriented languages like Java or C++. These frameworks hide from the application programmer both the network and the heterogeneity of the participants. In return, they provide an interface for remote method invocation that is similar to the one used by the application programmer for calling methods on objects sharing the address space.

In spite of their efforts, it is not possible to completely hide the network from the application programmer. Difficulties arise from the distributed nature of the system, where some kinds of problems (e.g. network outages) lie outside the scope of the DOF. In addition, distribution raises the need to locate the address space where the remote object is executing. To address this issue, DOFs typically provide a naming service. This service, coined for different DOFs with different names, maps the unique object ID of the object on a server network address. The location of the name server must be known in advance by both the client and the server. CORBA [27] and Java RMI [8] are both examples of DOFs sharing many aspects of the general description above. Below, the description focus on JRMI, which is the main focus of the work presented in this paper.

The Java Remote Method Invocation (JRMI) [8] specification complies with the Distributed Java Object Model (DJOM) [26], which maintains all semantics of the Java Object Model (JOM). In the scope of this paper, the relevant aspects of the JRMI architecture can be arranged in the 3 layers depicted in Figure 1.

The role of the stub and skeleton components is respectively to provide an API that mirrors the one made available by the remote object and to invoke the methods at the remote object. In addition, stub and skeletons are responsible for marshaling and unmarshaling method's arguments and return values. The remote reference layer defines and supports invocations semantics providing a RemoteRef object that represents the location of the remote object that implements the service. Finally, transport layer connects Java Virtual Machines (JVMs) using the Java Remote Method Protocol (JRMP) over TCP/IP.

JRMI references identify an object running on a specific host. Figure 2 depicts the flow of a JRMI distributed application. Server objects initially generate their references and publish them on a registry server. The registry is an auxiliary application running on a well-known location that provides a name service, mapping object IDs on references. Server’s publication of remote objects and client’s queries are remote invocations obeying the JRMI protocol. In particular, the registry presents itself like any other remote object, although with a well-known interface, whose stub is part of the standard JRMI distribution.

2.2 Group Communication Systems

A possible approach to support state replication is to force all replicas to converge to a consistent state by processing the same set of deterministic requests by the same order. This model is usually referred as a Distributed State Machine. Unfortunately, the distributed state machine is not trivial to implement in realistic scenarios, for example in the presence of unbounded communication delays (as they make impossible to distinguish a slow process from a faulty one) and when distinct requests are received at multiple hosts, given that all replicas must process the requests by the same order.
A number of frameworks (e.g. [10],[16],[17] to name a few) providing powerful enough abstractions to support distributed state machine implementations have been developed. These frameworks, hereafter named Group Communication Systems (GCSs), combine Virtual Synchrony [4] with Atomic Broadcast. Virtual synchrony creates the concept of a group, to which processes can join and leave (either voluntarily or by failing). Each change in the group membership results in a view change, notified to every member. In addition, virtual synchrony ensures that every correct process delivers exactly the same set of messages in each view and that those messages are delivered in the view they were sent. Atomic broadcast in turn combines reliable broadcast, which ensures the delivery of a message either to every correct process or to none, with total order, that guarantees that messages are delivered by the same order to every correct process (including the sender).

Uniform Total Order algorithms extend the properties provided by (regular) Total Order algorithms by ensuring that failed processes either do not deliver a message or deliver it in the same order of correct processes. Enforcing uniformity typically increases the cost of total order algorithms. However, in the scope of this work, uniformity plays a fundamental role as it prevents inconsistencies resulting from the client receiving replies from failing processes. The interested reader is referred to [7] for an interesting survey on the algorithms implementing these properties.

In a distributed state machine, a group of processes runs the same deterministic code so that they progress independently toward the same state. Atomic broadcast can be used to implement the distributed state machine model because it imposes a deterministic order to message delivery. However, messages are delivered exclusively to correct processes. Once a process fails, it can never recover, as its state could never converge because the recovering process did not receive some messages. Virtual synchrony simplifies process recovery by: i) notifying the group of the entry and failure of other members; and ii) creating synchronization points, which can be used for state transfer, once a process recovers.

GCSs hide the complexity of the implementation of these properties from the application programmer. These properties are abstracted under a simple interface, with a small number of primitives for managing group membership and to multicast and receive messages from the group.

Work described in this paper makes use of the Appia [16] GCS to implement the distributed state machine approach. Appia was selected mainly because it was developed in house, is implemented in Java and has a library of protocols that permits to experiment the performance of FTRMI with different total ordering protocols. The interested user is referred to [2],[25] for interesting comparisons among GCSs.

2.3 RPC Replication Frameworks

A number of frameworks providing replication and fault-tolerance support for RPC middleware have been proposed (see [9],[11],[12],[18],[21] for examples of frameworks addressing the problem for CORBA). Unfortunately, most of the approaches are not transparent either for the client (which is required to run specialized versions of the libraries) or for the server. This section describes some proposals that are more related to our approach.

To implement a fault-tolerant DOF, [13] introduces a preprocessing stage at Java byte code compile time. The preprocessing defines wrappers for the remote method invocation and makes a preliminary inspection of the code executed at the methods, by observing exclusively direct attributions. Inspection tags each method as read-only or read-write.

Wrappers permit to confirm preliminary inspection results in runtime. By default, all methods are executed in read-only mode and therefore are exclusively executed at one of the object replicas, thus fostering load balancing. Write operations trigger exceptions which abort the method execution and puts it in read-write mode. Read-write methods are delivered in total order to all the replicas using the Panda [1] framework. To improve performance, the framework implements its own object serialization mechanism. A limitation of the framework is its dependency on the compiler defined in the scope of this work and on the use of a proprietary protocol for client/server communication, which invalidates any communication with either clients or servers using JRMI.

The Object Group System (Jgroup) [17] is an integration of Group Communication Systems (GCS) technology with distributed objects implemented in Java. Jgroup platform supports a programming paradigm called object groups that can enable the development of fault tolerance services based on replication. The Autonomous Replication Management (ARM) [14] framework is built on top of Jgroup technology to simplify the implementation of applications that need specific replication policies. This framework provides mechanisms for distributing replicas to hosts and recovering from replica failures. These mechanisms are important for satisfying application goals such as maintaining a fixed redundancy level.

In Jgroup/ARM [15] services, a daemon is required to be running on both the clients and the servers. The daemon implements basic mechanisms for GCS such as failure detection, group membership, multicast and replica creation. The management of the number of replicas as well as the monitoring of their availability is the responsibility of the Replica Management (RM) component of the system.

The Jgroup/ARM framework provides a distributed object framework where the invocations at the remote objects are totally ordered. Although the model of this framework is similar to the original JRMI, APIs differ to better address the problems raised by replication. As a result, this platform is not transparent to a normal JRMI application. This is a problem shared by Filterfresh [3], who equally requires the use of classes developed in the scope of the project at both the client and the server.

In contrast, the Aroma framework [19] concentrates its replication efforts externally to the client and server software by capturing all the traffic exchanged between these components. This is an interesting approach as it does not require any change to the software. Unfortunately, traffic must be equally captured on the client side, what may raise privacy issues for some classes of applications.

3. FTRMI

The main goal of the Fault-Tolerant Remote Method Invocation (FTRMI) framework is to supply a transparent object replication service with fault tolerance. This application leverages on the ubiquity of JRMI and extends it with new libraries that enhance
JRMI with fault tolerance and recovery capabilities. An interesting aspect of FTRMI is its full transparency to clients, given that the extended libraries are only required on the server side, where it replaces some standard JRMI libraries with versions developed in the scope of the project. Because the new set of libraries respects the original API, the substitution is exclusively performed through changes of the parameters of the class loader leaving the code at the server untouched. As a result, many distributed applications using JRMI can be made fault tolerant without any code change.

Figure 3 depicts the architecture of a distributed application using FTRMI. FTRMI is implemented on a layer at the server side, placed between the standard libraries and the skeleton. In the figure, arrows represent the execution of a remote call using FTRMI. On the client side, a regular JRMI call is prepared and sent to the network. Once it arrives at the FTRMI layer on the server side the request is multiplexed to all servers using an Appia channel [16]. The Appia channel is configured to provide virtual synchrony and atomic broadcast, thus supporting the implementation of a distributed state machine. Upon delivery of the call by the Appia layer of every node, the request is passed by FTRMI to the skeleton. Notice that from the skeleton and application layer perspective, the call being received is a regular FTRMI to the skeleton. Notice that from the skeleton and the call by the Appia layer of every node, the request is passed by FTRMI. FTRMI is implemented on a layer at the server side, where it replaces some standard JRMI libraries with versions developed in the scope of the project. Because the new set of libraries respects the original API, the substitution is exclusively performed through changes of the parameters of the class loader leaving the code at the server untouched. As a result, many distributed applications using JRMI can be made fault tolerant without any code change.

Figure 3. FTRMI architecture

3.1 Transparent Handling of Multiple Replies
To cope transparently with server failures that occur while the requests are being handled, all replicas send their reply to the client. This multitude of replies is hidden from clients by having all replies but the first to be discarded at the TCP level of the client. To achieve this goal, the server receiving the request uses Linux's libcap library to gather IP and TCP protocol level information (sequence number, ACK number, source and destination IP addresses) from the segments carrying the request. This data is forwarded together with the JRMI request to every replica using the Appia channel. After receiving the reply from the server, each replica uses the low level connection information to prepare and forward a datagram with IP and TCP headers that match client TCP connection expectations. At the application level on the client, a reply is delivered at most once, given that TCP discards all but the first, given that they are assumed to be duplicates.

3.2 Fault Recovery
FTRMI assumes that when a server recovers from a failure the state of all its objects is empty. However, failure detectors used by GCSs can easily confuse failures with network disconnections, in which case the server would reconnect with an outdated version of the state. FTRMI handles equally both cases, by using Appia's majority partition facility which installs new views on servers of at most one network partition. Upon recovery (either of a failed node or of a partition), one of the most up-to-date replicas imposes its state to those that have recovered. Virtual synchrony features provided by Appia play an important role in the process by notifying group members about processes that have recently joined. Therefore, the state required by an outdated process and the processes that own the most up-to-date version of the state can be easily determined. FTRMI uses standard Java object serialization features to transfer the most recent state version to replicas joining the group.

3.3 Determinism
Atomic broadcast ensures that requests are delivered by the same order to the Appia layer on every server. However, thread scheduling policies can subvert total order if multiple requests are being concurrently handled at some server. To avoid race conditions among threads in different servers, a single executor waiting queue is associated to each object. The queue serializes the requests by the order decided by the atomic broadcast and uses standard thread synchronization features to prevent concurrent handling of requests.

3.4 Object Location
FTRMI improves service availability by creating multiple replicas of the server objects. However, availability would be reduced if only one replica could be located by clients, as it is assumed that all replicas have an equal probability of failing. An ideal solution for this problem would be to create a distributed registry, to make the registry fault tolerant, and to change the JRMI Object IDs model, replacing server location by some reference allowing locating any of the available replicas of the service. While the former could be easily achieved using FTRMI, the later is challenging as it would require a change in the JRMI standard libraries, thus invalidating the original requirements of client transparency.

The compromise solution adopted by FTRMI is depicted in Figure 4. For each object server, a dedicated JRMI registry server is created, possibly hosted in the same physical host of the server. Load distribution is achieved by relying on the standard Domain Name Service (DNS) record rotation mechanism [6]. That is, all registry servers are identified by the same DNS URI although associated to distinct IP addresses.

A limitation of this approach is that DNS will continue to point to registries owning references of both failed and correct servers. From the clients' perspective, this will appear as if the service is having transient connectivity problems. As future work, the authors plan to apply the fault tolerance capabilities of FTRMI to the registry, so that a number of replicas of the registry become available. The implementation is not straightforward because, to maintain client transparency, the problems associated with the ObjID references must be solved at the FTRMI layer level.
4. EVALUATION
An implementation of FTRMI was developed and made publicly available at the Appia's web site. Both its performance and its resilience to failures have been experimented.

4.1 Performance
To evaluate the impact of distribution and transparency in the performance of remote method invocation, FTRMI and Jgroup/ARM [15] have been experimented. Jgroup/ARM was selected for comparison because it shares both the implementation language and the approach to achieve replication with FTRMI. However, these frameworks have opted for distinct Group Communication Systems, respectively Appia and Jgroup as well as for distinct approaches concerning transparency. Experiments using standard Sun JDK JRMI are used as control tests.

Performance is compared using two metrics. Latency evaluates the average time between the moment the request is performed at the client and the reply is delivered at the client’s application layer. Traffic accounts with the total number of bytes transferred over the network for satisfying each request. Traffic was estimated using the information made available by the operating system for the network interfaces. Although these results may not be accurate, it should be noted that the error should be similar for all the frameworks experimented.

Tests were performed in a lightly loaded 100Mb/s Ethernet network. During the tests, four servers were used exclusively as replicas. These servers have between 2Gb and 8Gb memory and processors AMD Opteron™ 248, Intel® Xeon® CPUs 3060 and X3370. Requests were issued in fast sequence by clients, installed on 4 Intel® Core™ 2 Duo CPUs E7500 and E8300 desktops with 2Gb RAM. All computers were running a Linux 2.6 kernel.

Tests were performed with calls to a server object with a single method. The method receives a string as its argument; increments an integer counter and return its value. The method is intentionally simple to reduce to a minimum the influence of the code on the results. Experiments were performed with a zero length string (thus using the smallest possible message) and with a random string with 2000 bytes, which forces message fragmentation.

Three total order algorithms were experimented with FTRMI. Results depicted as FTRMI-1 use a coordinator-based total order algorithm, while results depicted as FTRMI-2 use causal order based total order algorithm. Results depicted as FTRMI-U use a uniform total order algorithm. All are off-the-shelf protocols, made available with the standard Appia distribution. Jgroup/ARM implements total order using a distributed agreement protocol.

Registry deployment reflects the design option of each framework. In FTRMI, a registry server coexists with each server application. Load balancing is assured by instructing the same number of clients to connect to each registry server. In the Jgroup/ARM framework the single dependable registry server was placed together with the server with the most powerful hardware.

Tests did not consider processes failures. The tests are initiated after a warm-up phase, required to allow the Java Virtual Machine to load all the classes. To reduce the impact of external factors, results presented below are the average of 500,000 remote invocations performed by each client.

Figure 5 and Figure 6 show that latency increases with the number of clients. This is expected as a bigger load on the servers increases their response time. A comparison of two scenarios with a similar number of clients (i.e. \(s=2, c=4\) and \(s=4, c=4\) however, shows that the latency also suffers a non-negligible increase with the number of servers. This is attributed to the complexity of the algorithms required to provide atomic broadcast, which are known to not scale well. Message size equally plays a non-negligible role in latency. It is interesting to notice that the monotonically increasing pattern exhibited by every framework in Figure 5 cannot be found in Figure 6 what suggests an increased weight of the message size and a reduced weight of the server load. In the comparison between frameworks however, the pattern is regular, with the excellent performance of the non-replicated JRMI confirming that replication has a negative impact on performance that surpasses the gains of the load balancing. As expected, the additional delivery guarantees provided by the uniform total order protocols are achieved at expenses of an increasing effort in the number of messages and latency.

The results observed in Figure 7 and Figure 8 may contribute to justify the worst latency of Jgroup/ARM, as the traffic required to complete an invocation is, in comparison with the remaining, considerably higher. The increment due to the transmission of the multiple replies to the client in FTRMI is not visible in the figures because of their small size.

The figures show that handling one invocation consumes 5 to 10 times more traffic between servers than between the client and the server, due to the need to distribute and coordinate the replicas. Interestingly, the results do not linearly mirror what was observed for latency. Apparently, FTRMI-2 always consumes less traffic than FTRMI-1, although its delay is higher. The other remarkable aspect is the traffic decrease exhibited by Jgroup/ARM with an increased load and which is orthogonal to the number of servers.

---

2 http://appia.di.fc.ul.pt
Figure 5. Latency of FTRMI, Jgroup/ARM and JRMI for empty arguments

Figure 6. Latency of FTRMI, Jgroup/ARM and JRMI for arguments with 2000 bytes

Figure 7. Client-Server traffic with empty arguments

Figure 8. Server-Server traffic with empty arguments

The traffic results for remote invocations with 2000 bytes arguments are depicted in Figure 9 and Figure 10. These results mirror in general those that have been observed for empty arguments although with a scale that reflects the increased message size. In Client-Server communication, traffic consistently increases for every combination of servers and clients by a factor of 20 in JRMI and on every total order protocol experimented with FTRMI. Jgroup/ARM exhibits a significantly lower impact, by multiplying traffic by a factor of 8. Server-Server traffic increases by a factor of 3 for Jgroup/ARM and by a factor of 6 for all experiments using FTRMI.

Figure 9. Client-Server traffic with 2000 bytes arguments

4.2 Fault-Tolerance

To assert FTRMI resilience to failures, a 24h test was performed. In this test, 4 replicas running on distinct servers were permanently receiving requests from 4 clients, for a grand total of approximately $10 \times 10^6$ requests completed successfully.
For these tests, the servers’ code was instrumented to abruptly terminate their execution at 5 distinct test points of the request processing flow. The location of the test points in relation with the request flow is depicted in Figure 11 and detailed below:

- **TP1**: before the request is delivered to Appia;
- **TP2**: immediately before the request is delivered to the skeleton;
- **TP3**: immediately after the reply is received from the skeleton;
- **TP4** and **TP5**: in different places of the code before the server forwards the reply to the client.

For each request, each of these test points crashes the server with a probability of 0.5%. In addition, a shell script ensures that immediately after a crash, a new server is launched. It should be noted that not all the test points allow for a successful execution of the request. In particular, requests producing failures in TP1 had not been delivered to other replicas what prevents their successful completion. Results confirmed our expectations with all the servers presenting a consistent state at the end of the 24h experimentation.

5. CONCLUSION AND FUTURE WORK

This paper described FTRMI, a fully transparent active replication framework for JRMI that increases server dependability. FTRMI extends the JRMI client/server communication paradigm with one additional communication layer, in charge of intercepting remote invocations, distribute them to the replicas and recover failed nodes or partitions. FTRMI allows any regular JRMI application to be fully replicated without any recompilation. Application programmers are only required to rerun server-side code with the libraries developed in the scope of the project. This facilitates the construction of replicated systems based on the object-oriented paradigm given that no change to the client code (which can be dispersed over a multitude of platforms and management authorities) is required. Evaluation results show that FTRMI is competitive with alternative approaches, although presenting a noticeable performance penalty over the non-fault tolerant standard JRMI.

As future work, authors plan to address the limitations presented by the current implementation of the registry. Research directions include the implementation of a dynamic registry that coordinates with the group communication service to return references to correct replicas.

6. ACKNOWLEDGMENTS

Work described in this paper was partially supported by the Portuguese Fundação para a Ciência e Tecnologia (FCT) through project PTDC/EIA-EIA/103751/2008 - PATI - IoT-Aware business Processes.

7. REFERENCES


ABOUT THE AUTHORS:

Diogo Reis has a degree in computer science from the University of Lisbon. He is concluding his Master degree on computer science by the same University. His research activities are performed in the scope of the Large Scale Computer Systems Laboratory (LaSIGE), a research unity of the University of Lisbon. His current research interests are on applications of protocol composition frameworks to simplify the development of fault-tolerant applications.

Hugo Miranda has a PhD in Informatics from the University of Lisbon. He is an assistant professor at the Informatics Department of the University of Lisbon and a senior researcher of the Large Scale Computer Systems Laboratory (LaSIGE). He is a member of the ACM and IEEE. He co-authored more than 40 papers in journals and conferences on the fields of distributed systems and mobile networks and co-edited a book on Middleware for Network Eccentric and Mobile Applications. His research interests include middleware protocols for wired and wireless networks, protocol composition frameworks and network level protocols for mobile ad hoc networks.
One-Scan Rule Extraction to Explain Significant Vehicle Interactions with Guaranteed Error Value

Omar U. Florez and Curtis Dyreson
Utah State University
Logan, Utah, USA
{Omar.Florez, Curtis.Dyreson}@usu.edu

ABSTRACT

Counting frequent itemsets allows us to compute the importance of items over a stream of data. Translating this concept to video streams imposes the need of representing activities as a sequence of activities over a video stream. In this paper, we present a model to find approximate co-occurring associations between activities from video stream data considering an unsupervised clustering of activities. We show that a hierarchical Topic Model of two stochastic processes is needed to jointly learn both an unknown number of activities in the video and the visual features that positively correlate for each activity. Unlike most of previous works, we decouple the analysis of associations between multiple moving objects from the discovery of activities. While the discovery of activities is an off-line process in which event distributions are grouped, the discovery of rules is an on-line process that approximates the importance of each rule with guaranteed error value. Our method reduces space complexity by adapting the algorithm to the amount of memory available before any process to update frequency values for itemsets is incrementally performed. The most visible aspect of this effort is the incremental generation of rules that discover the interaction of frequent activities for current scenes. Our experimental results show that our approach efficiently and automatically discovers sets of activities in a video stream coming from surveillance videos containing complex traffic scenes governed by multiple semaphores, while evaluating their frequent occurrence and co-occurring relationships.3

Categories and Subject Descriptors
H.3 [Information Systems]: Information storage and retrieval; I.5 [Pattern Recognition]: Clustering

General Terms
Algorithms

Keywords
Scene discovery, Statistical learning, Video streams

1. INTRODUCTION

In many real-world applications, data takes the form of an ordered sequence of items that arrive continuously. Over time, a huge amount of data can accumulate and the distribution of data within a stream can vary. Traditional examples include Sensor Networks [5] and Internet Packet Streams [8]. These scenarios provide explicit representation of atomic elements and the frequency of their combinations must be computed in a single scan because of the continuous arrival of data.

Video can also be modeled as a data stream. Video is widely used in real-time monitoring applications, e.g., of an oil spill, a store entrance, or an airport. In a video data stream we are interested in discovering and monitoring the hidden rules that govern the behavior of multiple objects occurring in the same scene. Discovering these associations over streams of video data raises three new issues, which extend traditional techniques.

1. Common behaviors describe activities - The similar actions of different moving objects discovered in the stream (e.g., a car moving from right to left) need to be categorized under a common behavior called an activity.

2. No a priori knowledge of activities - The activities are not known in advance, rather they depend on the moving objects present in each video. Some method or model is needed to automatically infer activities from a video.

3. Users expect real-time processing and query answering - Answers to user queries about activities must be timely. Since there is a vast amount of data in a video stream, and the data arrives continuously, a method is needed to incrementally discover activities in a single pass over the data stream. Users will be satisfied to have an approximate answer with guaranteed low error value.

These challenging issues motivate our design of a framework for the real-time analysis of streaming video data. Our visual surveillance system is designed to automatically answer questions such as: “Which is the most frequent scene seen so far?”, “How important is that scene in the stream?”, and “Is there a rule that explains the interactions between activities that are also seen in similar scenes?”. To do this every scene is modeled as a combination of zero or more activities made by individual moving objects. That set of activities describes the interactions between activities found in an scene and its frequency represents its importance over the stream. In this paper, we tackle the problem of finding the rules that govern the co-occurrence of activities in a continuous video data stream with no prior knowledge of the number of activities. As we shall show, the discovery of activities is an off-line process in which event distributions are grouped. The discovery of rules is an on-line process that approximates the importance of each rule with guaranteed error value.
1.1 Contributions

This paper makes the following contributions.

- We propose an unsupervised framework that efficiently addresses the complete process of scene understanding over video data streams. Previous research (see Section 5) proposes either time consuming algorithms that are hard to scale to a stream or assumes a fixed number of activities in a video.

- We propose an algorithm to extract rules in a single pass over a video stream approximating the frequency of the activities found in a video with lower/upper bounds and providing guaranteed error values.

- We provide evidence that shows that hierarchical Bayesian models, including the hierarchical Dirichlet Process (HDP), are non-parametric Statistical models that suit the problem of activity recognition and visual surveillance. Most previous research has considered the generation of topic models under those models for large text datasets. We extend those findings to a more noisy domain such as video databases and demonstrate that a hierarchy of two processes is needed to automate the discovery of activities.

1.2 Paper organization

The rest of this paper is organized as follows. Section 2 describes a method to discover activities from video data and provides an intuitive example to explain the need of a hierarchy for this problem. Section 3 discusses the frequency analysis of activities incrementally discovered from streams of video data. Section 4 experimentally demonstrates the usefulness of the proposed approach for real traffic video datasets. Section 5 discusses related work. Section 6 indicates limitations and possible extenstions to this work. Finally, Section 7 concludes the paper.

2. DISCOVERY OF ACTIVITIES

The problem of discovering activities in a video involves three kinds of information: events, actions, and activities. An event is an low-level interest point that represents a pixel with high variance in its spatio-temporal neighborhood. For a moving object, events occur in a bounding box forming a particular spatial arrangement of points that characterizes the action being performed. While a set of events characterizes an action (e.g., a car moving from right to left or a person walking in certain direction), activities are clusters of actions with similar event representation. This terminology and hierarchical relationship between events, actions, and activities have been adopted by the Computer Vision community, so we too use these common definitions. Given an input video, we take two consecutive frames and use a threshold to remove pixels with low intensity, as shown in Figure 1 (a). Then, we extract their events (gradient points) using a technique by Laptev et al. [4]. We evaluate connected components in Figure 1 (a) (represented as bounding boxes) to find moving objects in the scene as shown in Figure 1 (b). Finally, we place grids on those boxes to discretize the location of existing events into \( n \times n \) small regions, as shown in Figure 1 (c). Note that we want that every connected component often corresponds to a single moving object, so we obtained better results by only considering rectangular bounding boxes, enclosing components, with width-to-length ratio in the interval of \((0.7, 1.3)\). When we divide the number of events found in every small region by the total number of events in a motion grid, we estimate the probability of finding an event in that region. For objects performing the same activity \((G_1)\) in Figure 1 (c), we can see how the grids also show a similar spatial arrangement of events.

Our goal at this stage is to model how events are organized into activities. Thus, in this section we use a hierarchical model of two levels to generate activities in video data as multimodal probability distributions over events. The following example intuitively explains how this hierarchy works.

\textbf{Example 1.} Let \( G_0 \) be a collection of possible traffic activities \( G_0 = \{ \text{TurnRight}, \text{TurnLeft}, \text{GoSouth} \} \) present in a video. We model a video as a sequence of combinations of activities with temporal order. Thus, instances of activities in \( G_0 \) are sparse over video frames forming the next sequence of actions: \( G = \{ [\text{TurnLeft}_{11}, \text{TurnLeft}_{12}, \text{TurnRight}_{21}, \text{TurnLeft}_{13}, \text{GoSouth}_{23}], \ldots \} \), where \([\_\_\_]\) delimits a scene as shown in Figure 2. Each action in the scene is described by an \( n \times n \) bounding box, whose row-wise traversal defines a histogram \( y = (y_1, y_2, \ldots, y_n \times n) \) that represents the multimodal probability distribution \( \theta_j \) associating the activity \( j \) to the \( y_i \): the probability of finding an event in the \( i-th \)
Figure 2. A concrete example to show the hierarchical process to discover activities in video data.

position of its motion grid. In this example, the turn-left activity $G_1$ is formed by three similar (but not identical) turn-left actions \{TurnLeft$_1$, TurnLeft$_2$, TurnLeft$_3$\}, which have similar event distributions (represented as histograms in the lowest level of the hierarchy). Hence, the locations of events within a bounding box effectively characterizes the above actions into the activity $G_1$. Activities $G_2$ and $G_3$ are formed in a similar way by considering groups with similar event distributions.

The above example gives us an intuition about the hierarchical dependency between events, actions, and activities involved in this problem and modeled with two levels in Figure 2. The first level in the lower part of the hierarchy generates a mixture of events $y_i$ that uniquely define an action with multimodal distribution $\theta_{ji}$. The second level generates a list of activities $G_0$ distributed as the mixture model $G_J$ over several multimodal distributions $\theta_{ji}$. These two groups of information come from different, but related mixture models. The hierarchical way of forming activities seems to indicate that both groups share some mixture parameters. However, note that we do not know the number of mixture component in $G_0$ needed to represent the clustering process involved. In our case, it is difficult to specify a priori the number of event observations (regions in a grid) and activities needed to correctly interpret interactions in a traffic video. Our approach is to set the number of event observations as an external parameter dependent on the resolution of a particular video, but infer the number of activities by using a Dirichlet process in each group of actions. The use of a Dirichlet Process is justified by its property of providing a non-parametric estimation of the number of mixture components for groups of observations.

We first define the Dirichlet Process and then present a hierarchy of two Dirichlet Processes that can discover a number of activities in video data.

### 2.0.1 Dirichlet Process

Each event observation can be generated independently by a mixture component $\theta_{ji}$. Let $\theta$ be a mixture component (cluster) associated to the event observation $y_{ji}$.

**Definition 1 (Dirichlet Process).** A Dirichlet Process (DP) is a stochastic process that generates a distribution $G$ in the form of an infinite mixture of components $\theta = \{\theta_1, \theta_2, \ldots\}$, a base distribution $G_0$, and a positive scaling parameter $\alpha$.

The construction of the Dirichlet Process can be formulated with sequences of independent random variables $(\pi_i)_{i=1}^{\infty}$ and $(\theta_i)_{i=1}^{\infty}$, as originally stated in [7]:

$$\pi_i \mid \alpha, G_0 \sim \text{Beta}(1, \alpha)$$

$$\theta_i \mid \alpha, G_0 \sim G_0$$

such that the random distribution $G$ is then defined as:

$$G = \sum_{i=1}^{\infty} \pi_i \delta_{\theta_i}$$

where $\delta_{\theta_i}$ is an atomic distribution centered on $\theta_i$. For convenience, we shall abbreviate the construction of $\pi$ as $\pi \sim \text{GEM}(\alpha)$. Note that $\theta_i$ is a multinomial probability distribution over event observations $y_i$. In other words, the random variable $\theta_i$ has a probability of being associated to the set of event $y = \{y_1, y_2, \ldots, y_{n\times n}\}$. Hence, the distribution base $G_0$ also needs to be distributed as a multinomial distribution. This property of having a family of multivariate probability distributions is especially found in the Dirichlet distribution\(^2\), so we model $G_0$ as being distributed as that distribution, $G_0 \sim \text{Dirichlet}(D_0)$.

The Dirichlet Process generates a list of clusters of events $\theta = \{\theta_1, \theta_2, \ldots\}$ from the mixture model $G$ that characterizes an activity based on the event observations $y_i$. Although this setting can represent appropriately one activity, it cannot represent several activities, which is needed for activity recognition in video data. The modeling of activities is defined as a hierarchy of two DPs that relates the generation and activities jointly.

### 2.1 The Hierarchical Model

We employ the Hierarchical Dirichlet Process (HDP) introduced by Teh et al. [9] to mutually learn both actions and activities by considering a second DP which models groups of actions $\theta_{ji}$ into activities $G_j$. The result is a hierarchical process which can be understood as the two level DP represented in Figure 3.

The lower level of the hierarchy generates an unbounded number of HMMs (Hidden Markov Models) that learn activities with an unknown number of states, considering event probabilities from a motion grid as observable variables. The upper level combines similar actions (learned in the HMM) into activities.

\(^2\)This is the reason why a Dirichlet distribution is commonly denominated as a distribution of distributions.
Lower Level.
The first level in the hierarchy constructs a variant of the
Lower Level. The first level in the hierarchy constructs a variant of the
Hidden Markov Model with state transitions distributed as
$G_1$. The HMM is a doubly stochastic Markov chain in which
a sequence of state variables $x = \{x_1, x_2, \ldots, x_T\}$ is hid-
en, but the sequence of observations $y = \{y_1, y_2, \ldots, y_T\}$ is
observable. Changes between states are modeled with
state transition probabilities and every state $x_i$ is a multi-
modal variable that emits a discrete set of observations with
some probability distribution. Traditionally, HMM assumes
a Gaussian distribution for this property. However, it could
represent even more complex observation behaviors when
the output of the states is represented as the mixture of two
or more Gaussians.

Every HMM is defined by the probability of each state to
transition to other states and the probability of each state to emit an
observation. In our model, both groups of information
are assumed to be distributed as probability mixture
models $G_j$ for states and $\theta_{ji}$ for observations. A Dirichlet
Process is used to approximate each mixture model with an
unknown number of mixture components. Since we do not
assume an arbitrary number of states, the transition to an
infinite number of states is modeled using a DP following the
construction procedure presented in Definition 1.

$$G_0 \mid \gamma, H \sim GEM(\gamma)$$

$$G_j \mid \alpha, G_0 \sim DP(\alpha, G_0)$$

$$\theta_{ji} \mid \alpha, G \sim DP(\alpha_j, G_j)$$

for each $j = 1, 2, \ldots$, the probability $\theta_{ji}$ related to the activity
$j$ are learned with a HMM of states $x$ and observations $y$, which
have the following distributions.

$$x_t \mid x_{t-1}, (G_{ji})_{j=1}^{\infty} \sim G_{x_{t-1}}$$

for states

$$y_t \mid x_t, (\theta_{ji})_{j=1}^{\infty} \sim F(\theta_{x_t})$$

for observations

where, $G_j$ is the distribution for the squared matrix that
represents the transitions between states for the activity $j$. Diff-
ferent activities will be learned by HMMs with different
distributions $G_j$.

Upper Level.
While the lower level generates a list of HMMs that recog-
nizes individual activities, the upper level in the hierarchy
selects the optimal HMMs associated to the activity $j$. The
result is a list of activities $G = \{G_1, G_2, \ldots\}$ distributed as
a mixture model $G_0$ with base distribution $H$, and a positive
scaling parameter $\gamma$.

$$G_0 \mid \gamma, H \sim GEM(\gamma)$$

In other words, the base distribution $G_0$ generates the dis-
tributions $G_j$ by grouping similar HMMs that learn similar
event distributions $\theta_{ji}$. Teh et al. [9] also use Gibbs sampling
schemes to do inference under the HDP model. To detect
the activity associated to a bounding box with a sequence of
events observations $\{y_1, y_2, \ldots, y_{n<\infty}\}$, first the trained HMM
with highest log-likelihood score is selected. Second, the ac-
activity of the corresponding $G_j$ associated to the item $j$ is
chosen.

3. DISCOVERY OF INTERACTIONS

The output of the hierarchical model present in Section 2
is able to discover the activities that positively correlate in
frequent scenes. The discovery of important interactions be-
tween those activities can take at any time over the video
stream as scenes continuously arrive. Thus, analysis of this
data must be incremental as multiple scans to learn the un-
derlying interaction model of the video stream would be too
expensive. The algorithm proposed in this section extends
the well-known Apriori algorithm [1] by generating approxi-
rates over a video data stream in a single pass and with
guaranteed error value. The output of the algorithm is a
subset of dynamic activity interactions with a low impor-
tance value over the video stream at time $t$.

The algorithm is based on the plausible idea that a video
stream is formed by a sequence of short scenes in which ac-
activities are simultaneous. Two or more consecutive frames
define a scene that contains subsets of the set of activities
learned in Section 2 with a hierarchy of two Dirichlet Pro-
cesses. Those subsets of activities are denominated activity
sets in this paper to represent a spatio-temporal environ-
ment for co-occurring activities. Note that complex activi-
ties do not occur solely in one scene, but they propagate their
behavior over consecutive ones. Hence, we define a transac-
tion as a larger time window that contains a sequence of
scenes. This is a range that specifies the maximum allowed
time difference between the earliest and latest occurrence of
activities.

Two important properties of activity sets are their support
($f$) and error value ($\epsilon$). While the support refers to the
approximate number of transactions that contain an activity
set, the error value is guaranteed that the approximate re-
results will not exceed that number. Frequent activity sets are
those with support value above some threshold. We want to
compute rules of the form $set_1 \rightarrow set_2$ such that $set_1 \cup set_2$
has high support.

To compute approximate support values in a single pass, we
need to store frequent counts of activity sets previously seen
in the stream into a data structure $D$. Note that $D$ can
be any data structure that efficiently implements the oper-
the current transaction. If so, we assign the lower bound set \( \bar{f} \) to the set \( f \) such that a low uncertainty value is obtained when a set \( f \) is seen at least once on the stream and \( \epsilon \) is the desired error value.

The algorithm starts by logically dividing the incoming stream into transactions of \( w = \frac{1}{2} \) scenes. The set of activities, let’s call it \( cand \), contained in an incoming scene represents recent information to be analyzed by its frequency count in the current transaction. To do that we first evaluate its power set \( P(cand) \), excluding the null set, to evaluate the possible combinations of activities that may be frequent. Note that the power set of a scene does not produce a huge number of activity sets \( C \) since a scene commonly contains only the interaction of a limited number of activities. The index of the current transaction being analyzed is denoted as \( b_{current} \) and its value is computed as \( \frac{N}{w} \).

For every activity set \( set \) in \( C \), the algorithm looks up \( set \) in \( D \) to know whether it exists or not. When \( set \) is seen for the first time, and therefore it is not in \( D \), we check whether \( set \) has a high support value in the \( w \) scenes of the current transaction. If so, we assign the lower bound on its frequency as \( \bar{f} = 1 \) and the extension of its confidence interval as \( \Delta_f = b_{current} = \frac{N}{w} \). This indicates that we have currently seen \( set \) at least once on the stream and at most once on every transaction, respectively. The lower bound \( \bar{f} \) will increase when new instances of \( set \) are found in later windows.

If \( set \) is currently stored in \( D \), its lower bound \( \bar{f} \) is increased by 1. The extension of its interval is updated by the difference between its current and former transaction indices such that a low uncertainty value is obtained when a set recently occurs often and a high uncertainty value is computed when there is a large gap between occurrences. An upper bound below \( \frac{N}{w} \) (i.e., the current number of time windows processed so far) is an indicator that the activity set may have been frequent in early transactions, but has not been updated recently. Hence, we delete those entries with \( \bar{f} + \Delta_f \leq \frac{N}{w} \) to only generate co-occurring rules based on recent and frequent activity sets.

Every activity set \( set \) is inserted in a priority queue \( Q \) of size \( K \). Subsequently, after all the scenes in the current transaction have been processed, \( Q \) will automatically retain the \( K \) most frequent activities sorted by their approximate support \( \bar{f} \). The process of estimating approximate activities is summarized in Algorithm 1.

The use of lower and upper bounds assure us that after reading \( N \) scenes in the stream, those activity sets whose frequency exceed \( \frac{N}{w} \) will be stored. Since \( w = \frac{1}{2} \), the threshold is \( \frac{N}{2w} \) and thus \( \epsilon N \) is a guaranteed worst-case approximation on the value of \( f \).

Algorithm 1 SetCounting(\( T, D, N, w \))

1: \( \bar{T} \) //Current transaction
2: for \( i = 0 \) to \( w - 1 \) do
3: \( \text{cand} \leftarrow T_i; \) //activity set \( \text{cand} \) at the \( i \)th scene
4: \( b_{current} \leftarrow \frac{N}{w} \);
5: \( \bar{T} \) //Power set on each scene
6: \( C \leftarrow P(\text{cand}) \) − \( \{\emptyset\} \);
7: for \( j = 0 \) to \( /C/ \) do
8: \( \text{set} \leftarrow C_j; \)
9: \( \bar{T} \leftarrow \text{count(set)} \)
10: if \( \exists D, \text{exist(set)} \) & \( \bar{T} \geq \frac{N}{w} \) then
11: \( D, \text{insert(set, } \bar{T}, b_{current} \)>
12: else
13: //Updating approximate support
14: \( \text{old} \leftarrow D, \text{get(set)} \)
15: \( \bar{T} \leftarrow \bar{T} + \text{set}; \)
16: \( \bar{T} \leftarrow \bar{T} + \Delta_f; \)
17: \( \text{interval} \leftarrow \text{b_{current} - (old + \Delta_f)} \);
18: \( D, \text{update(set, } \bar{T}, \text{interval} \)>
19: end if
20: //Pruning condition
21: if \( \bar{T} + (b_{current})_{temp} \leq (b_{current})_set \) then
22: \( D, \text{remove(set)} \)
23: end if
24: if \( D, \text{get(set)} \neq \{\emptyset\} \) then
25: \( Q, \text{push(set, } \bar{T}, \Delta_f \)>
26: end if
27: \( N \leftarrow N + 1; \)
28: end for
29: end for
30: return \( Q \);

Given the frequent activity set \( set = set_1 \cup set_2 \) seen so far in the stream, we want to represent its interactions in the same scene by \( set_1 \rightarrow set_2 \ (set_1 \cap set_2 = 0, set_2 = set - set_1) \). The importance of this abstraction is measured by its confidence.

\[
\text{confidence(set}_1 \rightarrow \text{set}_2) = \frac{f(set_1, set_2)}{f(set_1)}
\]

Note that this value is not exact since its evaluation relies on approximate frequency counts, so we consider a guaranteed worst-case approximation of \( f \). The process of incrementally producing rules for a current transaction is summarized in Algorithm 2. In our experiments we use the confidence and support of a rule to determine its importance. Note that while the Apriori algorithm uses the anti-mono-tone property to avoid computing all possible combinations of activities, we evaluate all possible subsets of activities in a scene in seek of approximate their counting over the video stream. However, we reduce the candidate space by keeping only those that are frequent and recent over the stream. We thus assume that if a subset of activities is frequent in a transaction, it could also be frequent during the incoming part of the stream. Those subsets that are frequent, but become outdated, are removed as candidates. The inserting of frequent candidate subsets in \( Q \) provides frequent interactions for the \( \text{the current transaction} \), where we can discover rules with high confidence values as the stream arrives.
Algorithm 2 ApproximateRuleExtraction(Stream S)

1: $D \leftarrow 0; // D$: data structure
2: $Q \leftarrow 0; // Q$: priority queue
3: $N \leftarrow 0; // N$: number of scenes so far
4: $w \leftarrow \frac{1}{2}; // w$: number of scenes in $T$
5: while true do
6:   $Q \leftarrow \text{SetCounting}(S, \text{getTransaction()}, D, N, w)$;
7:   while $Q\.\text{hasElements}().$ do
8:     < set, $\Delta_{\text{set}} > \leftarrow Q\.\text{pop}();$
9:     for every nonempty subset $s_{1}$ of set do
10:       $s_{2} = s - s_{1};$
11:       if $\text{confidence}(s_{1}, s_{2}) \geq \text{confidence}_{\text{min}}$ then
12:         print(rule : $s_{1} \rightarrow s_{2}$);
13:         print(support : $\Delta_{\text{set}} \pm \frac{\sqrt{N}}{2}$);
14:       end if
15:     end for
16:   end while
17: $N \leftarrow N + w$
18: end while

Note that $s_{1} \rightarrow s_{2}$ defines the co-occurring interactions between the antecedent and consequent part of that rule and not the estimate of the probability of finding $s_{2}$ under the condition that these transactions also contain $s_{1}$. This information could also be modeled as an additional layer in the Topic Model described in Section 2. However, the resulting model would not be as dynamic and incremental as the one proposed in this paper. Our algorithm, rather than solving the efficient storing of frequent itemsets, aims to make it’s working adaptable to the limited resources of a system while providing bounds and a guaranteed error for frequency values. We study this problem in detail in the next subsection.

3.1 Dynamic Memory Usage

The approximate counting of frequent itemsets helps us to understand the significance of scenes in a video stream. However, both the detection of items in the video (Section 2) and their posterior frequency evaluation (Section 3) have different memory requirements. While the detection of itemsets takes consecutive frames to recognize items based on events, the approximate counting of itemsets is much faster and provides a better approximations when more time windows are considered. To illustrate this concept consider Figure 4 as an example. A stream of video frames arrives continuously as input. We trained the Dirichlet Process introduced in Section 2 to recognize itemsets by assigning labels to activities. The output of this process is a time window of $w$ transactions.

We can increase the probability of finding itemsets in several transactions by reading $B > 1$ time windows. This also provides a more effective pruning of those itemsets with supports below than $(s - \epsilon)N$. Thus, we store in a queue as many time windows as possible, being $B = 1$ the lowest value. However, memory space is a limited resource that may hinder the evaluation of approximate frequency counts when few transactions are considered. This problem is especially important in our case because of the amount of memory already devoted to recognize activities in a continuous stream of video frames. This makes dynamic the amount of memory available in the system with values even less than required to store one time window.

Thus, there is a trade-off between storing as many windows as possible, to better approximate frequency values, and compute the frequency of itemsets within a window as soon as one time window is received. An intermediate solution is to dynamically adjust the parameter $B$ to control the number of time windows to be stored in the queue.

If the number of windows is less than $B$ and enough memory is available, then we keep inserting time windows in the queue. If the amount of memory is almost full and the number of windows in the queue is less than $B$, then we delete those windows to avoid storing not recent frequencies. If memory is not full and there is more than $B$ windows in the queue, we also keep inserting windows until some threshold is exceeded and then take them to improve the approximation on frequent counts.

Let $M_{B}$ be the space in memory allocated to store $B$ windows, $M_{B} = wB$. A window $w$ contains $1/I$ transactions with at most $I$ items (i.e., possible types of activities). Thus, the maximum amount of memory to allocate a window is $w = \frac{1}{I} + \frac{1}{\epsilon}$ transactions and therefore we store $M_{B} = \frac{B/I}{\epsilon}$ units in the queue. The amount of available memory at time $t$ is then defined as the difference between the total memory at time $t$ and the memory employed to store $B$ time windows at time $t$.

$$M^{t}_{\text{available}} = M^{t}_{\text{total}} - B^{t}/I$$

Assuming that the last state in the queue, in which we took $B$ time windows, is stable, we want to preserve the balance between windows stored in the queue and the corresponding amount of memory available:

$$\frac{\#\text{transactions}^{t-1}}{M^{t-1}_{\text{available}}} = \frac{\#\text{transactions}^{t}}{M^{t}_{\text{available}}}$$

once $M_{\text{available}}$ and $B$ are considered in the above equation, we obtain the amount of time windows needed to read in memory to assure the stable working of the system at time $t$.

$$B^{t} = \frac{M^{t}_{\text{total}}B^{t-1}}{M^{t-1}_{\text{total}}}$$

Note that although both $I$ and $\epsilon$ are fixed parameters, we can consider $B^{t}$ as a running parameter in our model. An inadequate control of $B$ may hinder the performance of the system since it affects the number of transactions that need to be created, updated, or removed from the data structure $D$. This is specially important when $D$ is implemented on hard disk, a common scenario when very large stream databases are considered.

By modeling the allocation of transactions in memory in terms of $B$, we make this process adaptable to that limited resource. We would like to always set $B$ to a high value to have more accurate approximations for the counts of itemsets, but this value needs to be decided according to the total amount of memory in the system at time $t - 1$, which is not constant.
3.2 Summary

In this section, we have computed frequent itemsets with approximate support and guaranteed error value while adjusting dynamically the number of transactions needed to effectively evaluate support values. Intuitively, the more transactions we take, the more precise frequency approximations we obtain. However, more available memory will be consumed as well. Our method reduces space complexity by adapting the algorithm to the amount of memory available before any process to update frequency values for itemsets is incrementally performed.

4. EXPERIMENTS

In this section, we test the performance of our technique with outdoor videos where moving objects describe traffic scenes governed by the state of multiple semaphores. The co-occurring interactions are modeled by frequent sets of activities with large confidence values over the video stream. We experiment on the following datasets: Street Intersection\(^3\) (normal quality, 25fps, 45 minutes, 5 semaphores), Karl-Wilhelm & Strabe Streets\(^4\) (normal definition, 25fps, 1 hour, 3 semaphores), and Roundabout Junction\(^5\) (normal quality, 25fps, 1 hour, 3 semaphores).

The experiments are run on a 3.6 GHz Pentium 4 with 2 GB RAM and all the above datasets are publicly available to facilitate later experimental comparisons.

4.1 Experiment 1: Discovering activities

Before studying the quality of the co-occurring relationships generated to describe scenes in videos, we analyze in this experiment the recognition of activities with the model presented in Section 2. We compare our instantiation of the hierarchical Dirichlet Process (HDP) to another probabilistic topic model, the Latent Dirichlet Allocation (LDA) \(^2\), recently used in \(^3\) to also segment visual features from video data into activities, and a supervised approach, the Support Vector Machine (SVM). Most existing research employs hierarchical Bayesian models or non-probabilistic classifiers to recognize activities in video, so by using both LDA and SVM we cover two representative techniques in the literature.

We used the first 20 minutes of each dataset for training in order to take the remaining data for testing by labeling and recognizing activities on it. Pairs of consecutive frames are processed to identify moving pixels, events, and connected components. The observations consist of bounding boxes around moving objects with resolutions of \(8 \times 8\) for the Karl-Wilhelm & Strabe dataset and \(4 \times 4\) for the Street Intersection and Roundabout Junction datasets. This is because the camera in the first dataset is placed on a far building, so we need grids with higher resolutions to describe small objects. This process provides a collection of unlabeled motion grids to the hierarchical model. We do not assume any prior knowledge in the number activities to be discovered. DP parameters were fixed at \(\{\alpha = 11, \gamma = 0.9\}\).

Table 2 shows the number of activities found and the time consumed by each dataset. Since LDA and SVM cannot find the number of activities from the video, we first use HDP to discover the number of activities and then employ K-means to generate a number of clusters in the training dataset and assign them a label. Finally, we train SVM with these already labeled groups and perform generalization in the remaining video to assign labels to moving objects, as also described by Yin and Meng \(^13\). By contrast, LDA only needs the number of activities provided by HDP to be trained and recognize activities. The finding of activities by these three techniques is measured by comparing their clustering “goodness” on the testing data considering the same number of activities and increasing database size.

Figure 5 shows the result of grouping activities in each dataset with the trained HDP, LDA, and SVM. Since there is not a direct way to compare the accuracy of two clustering algorithms, we estimate the average clustering error by averaging the distance between the activities that belong to

---

\(^1\)http://www.eecs.qmul.ac.uk/~jianli/Junction.html
\(^2\)http://i21www.ira.uka.de/image_sequences/
\(^3\)http://www.eecs.qmul.ac.uk/~jianli/Roundabout.html

---

Figure 4: A broad representation of the approach introduced in this paper. A continuous video stream is discretized in windows \(B\) that contain a variable number of transactions at time \(t\) to approximate the frequency counts of itemsets in the window.
each cluster by their total number of cluster elements. This number indicates how similar are the actions, represented by their event distributions, that belong to the same cluster. A large value will indicate poor clustering since the actions within a cluster will be heterogeneous indicating a large entropy in the cluster. Hence, an algorithm with a low average clustering error is preferred. We computed the error along the duration of each video and plotted the result in Figure 5. The technique used in this paper provides the lowest error value for every dataset, but this difference is smaller for datasets with fewer activities. For instance, while we can clearly see the advantage of using HDP to detect activities in the Street Intersection dataset (37 activities) and the Karl-Wilhelm & Strabe dataset (31 activities) in comparison to other methods, the discovery of activities in datasets with fewer activities, like the Roundabout Junction dataset (24 activities), produces comparable error values in each method. This behavior was also noticed by Wang and Mori [11] when comparing different Bayesian models for human activities recognition in video data. Additionally, LDA seems to have a better behavior than SVM with K-means in all the datasets considered in this paper.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Activities</th>
<th>Time to Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Street Intersection</td>
<td>37</td>
<td>4.38 min.</td>
</tr>
<tr>
<td>Karl-Wilhelm &amp; Strabe</td>
<td>31</td>
<td>3.51 min.</td>
</tr>
<tr>
<td>Roundabout Junction</td>
<td>24</td>
<td>2.57 min.</td>
</tr>
</tbody>
</table>

Table 1: Information on the training stage for each dataset.

4.2 Experiment 2: Discovering Interactions

In this experiment, we study the significance of the generated rules to understand the dependencies between the activities discovered in Experiment 1. Transactions of size \( |w| = 25 \) scenes is a value that works in all the datasets in order to find activities temporally correlated in the same window. Thus, every transaction is the input to the Algorithm 1 and since \( w = \frac{7}{2} \), we approximate the frequency counts of every activity set with an error value of \( \epsilon = 0.04 \).

The number of transactions, topics, and the processing time to discover association rules for every dataset are summarized in Table 2. The Street intersection dataset exhibits more topics than the Karl-Wilhelm & Strabe dataset since five traffic lights decomposes complex activities into a large number of well defined scenes. On the other hand, the Roundabout Junction dataset contains a few number of topics due to the limited types of activities performed and considerable amount of frames with no activities. The processing time to generate rules seems to be proportional to the number of topics discovered in each dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time windows</th>
<th>Topics</th>
<th>Rule generation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Street Intersection</td>
<td>(~ 15000)</td>
<td>37</td>
<td>8.74 sec.</td>
</tr>
<tr>
<td>Karl-Wilhelm &amp; Strabe</td>
<td>(~ 11000)</td>
<td>31</td>
<td>5.41 sec.</td>
</tr>
<tr>
<td>Roundabout Junction</td>
<td>(~ 7000)</td>
<td>24</td>
<td>3.28 sec.</td>
</tr>
</tbody>
</table>

Table 2: Information on the datasets preprocessed to discover association rules.

We consider a minimum support value of 4%, a minimum confidence value of 90%, and activity clusters with more than 10 elements in order to generate representative rules. We thus extract rules from the Street Intersection (37 topics and 16 rules), Karl-Wilhelm & Strabe (31 topics and 13 rules), and Roundabout Junction (24 topics and 10 rules) datasets. In these datasets as more constraints govern the activities (e.g., traffic lights, one-way roads, intersections, etc.), more topics are generated and more frequent rules are discovered. This evidence seems to indicate that every constraint imposes an underlying logic that fragments complex activities into a large number of small scenes, which are easy to represent with events and form well-defined activities, and therefore are likely to be frequent during the video. For the Street Intersection dataset, some of the rules uncovered with the algorithm proposed in this paper are depicted in Figure 6 (a) and detailed in Figure 6 (b). The first three rules are high-confidence associations that suggest a strong correlation between vehicles moving in parallel lanes \((G_2\) and \(G_3\)) or those moving from side to side \((G_7\) or \(G_{11}\)) while other vehicles move away from the center to the top left of the scene \((G_4\). Those activities are mutually exclusive since there are five traffic lights that pre-
This behavior, exemplified by rule $G_1 \rightarrow G_3$, the co-occurring dependency between cars turning right (G), and others that are moving from right to left (G). This behavior, exemplified by rule $G_3 \rightarrow G_5$, is justified since those vehicles use the same traffic light to move from the bottom right part of the scene to either the bottom left or the top left edge, as seen in scene 4 and 5 of Figure 6.

For the Karl-Wilhelm & Strabe dataset, three confident interactions are shown in Figure 7 (a) and expressed with rules in Figure 7 (b). We notice the regular presence of the activity $G_5$ in those scenes. This behavior is reasonable since the activity $G_5$ corresponds to vehicles going along Strabe avenue, a very busy road in the dataset. The first rule $G_2 \rightarrow G_3$ exemplifies the interaction of vehicles going in parallel lanes without restrictions. The second rule is similar, but additionally contains the activity of cars going from the center to the bottom left of the screen (G). Furthermore, the usual interaction of cars going straight in the avenue (G) and then turning right after that (G) is explained by rule $G_5 \rightarrow G_7$.

For the Roundabout Junction dataset, we show common interactions in Figure 8 (a) and detailed in Figure 8 (b). The roundabout in the video segments motion of vehicles into multiple activities. The first rules $G_1 \rightarrow G_3$ represents the sequence of car activities going straight (G) and then joining the roundabout (G). By contrast, the second rule $G_3 \rightarrow G_2$ explains the co-occurring relationship of vehicles taking lanes separated by the roundabout. Finally, while some vehicles circulate alongside the roundabout emerging as two activities (G and G), another set of cars take a different way by turning left from the center to the upper part of the scene (G). These sequences of co-occurring associations reflect the transitions between significant scenes in datasets governed by multiple lanes, traffic lights, and a roundabout.

### Table: Confidence and Support

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rule</th>
<th>Confidence</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$G_1 \rightarrow G_3$</td>
<td>97.2%</td>
<td>7.6%</td>
</tr>
<tr>
<td>2</td>
<td>$G_2 \rightarrow G_3$</td>
<td>97.3%</td>
<td>11.2%</td>
</tr>
<tr>
<td>3</td>
<td>$G_3 \rightarrow G_5$</td>
<td>97.4%</td>
<td>8.8%</td>
</tr>
<tr>
<td>4</td>
<td>$G_5 \rightarrow G_3, G_7$</td>
<td>95.5%</td>
<td>6.5%</td>
</tr>
<tr>
<td>5</td>
<td>$G_3, G_7 \rightarrow G_2$</td>
<td>95.5%</td>
<td>8.2%</td>
</tr>
</tbody>
</table>

**Figure 7** Experiment on the Karl-Wilhelm & Strabe dataset. (a) A selection of high confidence association rules. (b) Scenes of the Street Intersection dataset with high confidence values over time.
come infrequent over time. In other words, by decoupling frequent activities in a transaction and removes those that be-

able to learn global interactions disregarding temporal in-

Both techniques, modeling behavior correlations through a hierarchical probabilistic Latent Semantic Analysis (pLSA). Both techniques, LDA, and HDP Bayesian models and providing extended versions of integral probabilistic hierarchical Bayesian models (LDA, HDP, and Dual-HDP mixture models) to cluster moving pixels into atomic activities and interactions. Similarly, Li et al. [6] infer global behavior patterns through modeling behavior correlations through a hierarchical probabilistic Latent Semantic Analysis (pLSA). Both techniques, however, learn global interactions disregarding temporal information. By contrast, our on-line technique relates frequent activities in a transaction and removes those that become infrequent over time. In other words, by decoupling both the discovery of activities and interactions, we can incrementally learn interactions without assuming the same probability of co-occurring relationships over time, a reasonable scenario imposed by the processing of continuous video streams.

6. LIMITATIONS AND POSSIBLE EXTENSIONS OF THIS WORK

To understand the behavior of moving objects in a video stream, we adopt events based on moving pixels as low-level visual features that characterize their spatial dependencies. While we have showed their expressiveness to discover activities and co-occurring relationships, others features can also be adopted to consider different attributes such as shape, trajectory, and appearance to distinguish between vehicles and people, detect when vehicles stop because of a read light, or when people describe two trajectories by crossing at a crosswalk in both directions. A extension of this work is to use trajectory and appearance information to differentiate activities (e.g. a car going left and a person going left). Additionally, the assumption of having a static camera recording activities is reasonable for traffic surveillance, but we will need to stabilize regions containing moving information, as in [11], to extend our work to more dynamic scenes. As future research, we plan to study other similar domains where crowds of moving objects also have different behaviors at the same time. Also, we want to study the relationship between frequent activities and recognition of abnormal situations, often related to dangerous events for the domain of traffic roads. Additionally, video summarization can also be implemented by providing to the user a sequence of important scenes in terms of confidence and support values.

5. RELATED WORK

In this section we compare our approach with related efforts. For clarity, we keep our comparison focused in each stage of the video process (i.e., discovery of activities and discovery of interactions). The recognition of activities in video data is an open problem that has received much attention lately. Commonly, low-level visual features and actions have been modeled and classified to provide interpretation of activities. While the traditional way to categorize existing research is by motion representation such as local features (e.g., changes in velocity, changes in curvature of motion trajectories, and gradients) or global features (e.g., key frames), recent research has employed hierarchical Bayesian models such as LDA [2] and HDP [9] to cluster local motions into activities successfully, c.f., [3, 10, 13]. The above research has led to techniques that can discover atomic activities, but such techniques omit the complex interactions between activities commonly present in video data. Wang et al. [10] approach this problem by adding one more level to the hierarchy of the LDA and HDP Bayesian models and providing extended versions of integral probabilistic hierarchical Bayesian models (LDA, HDP, and Dual-HDP mixture models) to cluster moving pixels into atomic activities and interactions. Similarly, Li et al. [6] infer global behavior patterns through modeling behavior correlations through a hierarchical probabilistic Latent Semantic Analysis (pLSA). Both techniques, however, learn global interactions disregarding temporal information. By contrast, our on-line technique relates frequent activities in a transaction and removes those that become infrequent over time. In other words, by decoupling

Table 1: Scenarios of the Roundabout and Street Intersection

<table>
<thead>
<tr>
<th>Scene</th>
<th>Rule</th>
<th>Confidence</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$G_1 \rightarrow G_3$</td>
<td>94.5%</td>
<td>27.3%</td>
</tr>
<tr>
<td>2</td>
<td>$G_9 \rightarrow G_2$</td>
<td>94.4%</td>
<td>13.5%</td>
</tr>
<tr>
<td>3</td>
<td>$G_5, G_{14} \rightarrow G_6$</td>
<td>94.8%</td>
<td>19.8%</td>
</tr>
</tbody>
</table>

Figure 8: Experiment on the Roundabout Junction dataset. (a) A selection of high confidence association rules. (b) Scenes of the Street Intersection dataset with high confidence values over time.

be computed, in practice it is not common to find all the possible activities in the same scene. Thus, this number is small allowing us to obtain an approximation algorithm for this problem.

7. CONCLUSION

In this paper, we propose a framework to find approximate co-occurring associations from video stream data considering unsupervised clustering of events (low-level visual features) into activities. We define activities as actions described by similar event distributions. A hierarchy of two stochastic processes is used to avoid considering an arbitrary number of activities in the video. The most visible aspect of this effort is the incremental generation of rules that discover the interaction of frequent activities for current scenes. Our experimental results show that our approach efficiently and automatically discovers sets of activities in a video stream while evaluating their frequent occurrence and co-occurring relationships with guaranteed error value.

8. ACKNOWLEDGEMENTS

We thank IBM Research for funding part of the research with a Scalable Data Analytics 2010 Innovation Award.

9. REFERENCES


ABOUT THE AUTHORS:

Omar U. Florez is a Ph.D. Student in Computer Science under the supervision of Dr. Curtis Dyreson at Utah State University. He has published 20+ papers in Large Scale Data Mining and Pattern Recognition for Multimedia Databases. He did his undergrad studies at San Agustin University in Peru and is member of the Peruvian Computing Society. He has earned an Innovation Award on Scalable Data Analytics from IBM Research in 2010 and did three internships at IBM Research and Intel Labs. Omar expects to graduate soon and follow a career in Large Scale Analysis of Unstructured Data.

Curtis Dyreson is an assistant professor in the Department of Computer Science at Utah State University. He is the Information Director for ACM TODS and for ACM SIGMOD, serves on the SIGMOD Executive Committee, and served as the SIGMOD DiSC editor, and the SIGMOD Anthology editor. He has published 80+ papers in the areas of temporal databases, XML databases, data cubes, and providing support for metadata. He previously worked at Washington State University, James Cook University, Aalborg University, and Bond University. He obtained his Ph.D. from the University of Arizona under the supervision of Richard T. Snodgrass.
Supporting Semantic Conflict Prevention in Real-Time Collaborative Programming Environments

Hongfei Fan
School of Computer Engineering
Nanyang Technological University
Singapore
FANH0003@e.ntu.edu.sg

Chengzheng Sun
School of Computer Engineering
Nanyang Technological University
Singapore
CZSun@ntu.edu.sg

ABSTRACT
Real-time collaborative programming environments support multiple programmers to concurrently edit shared source code documents at the same time over communication networks. One of the key challenges in supporting real-time collaborative programming is semantic conflicts, which may occur when multiple programmers are performing concurrent and incompatible programming work in the same source code region or in different source code regions with dependency relationships. Semantic conflicts may result in programming errors that are difficult to detect and costly to resolve. This paper presents in-depth analyses of representative collaborative programming scenarios for understanding the nature and general conditions of semantic conflicts, and proposes a novel Dependency-based Automatic Locking (DAL) approach for supporting semantic conflict prevention in real-time collaborative programming environments. The novelty of the DAL approach lies in its capabilities of supporting automatic and fine-grained locking on selected source code regions with dependency relationships to balance conflict prevention, concurrent work, and programmer convenience in real-time collaborative programming. The DAL approach and technical solutions have been implemented in the CoEclipse prototype system as a proof-of-concept for this emerging technique, which is being continuously extended for further exploration and evaluation.¹

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques – computer-aided software engineering; D.2.6 [Software Engineering]: Programming Environments – interactive environments; H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces – computer-supported cooperative work.

General Terms
Design, Human Factors.

Keywords
Real-time collaborative programming, semantic consistency, semantic conflict, conflict prevention, locking, dependency relationship, dependency graph, locking state maintenance.

¹ This work is based on an earlier work: SAC ’12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03. http://doi.acm.org/10.1145/2245276.2245417

1. INTRODUCTION
The software industry is rapidly growing with increasing demands on sophisticated software with fast-growing size and complexity, requiring engineers to collaborate on almost every stage of software projects [2]. All activities in the software development process require collaboration by multiple engineers who are possibly geographically dispersed [4]. Collaborative software engineering has been studied by researchers and practitioners for the past decades [9][10][12][13][15][18][24][25], and a wide range of methodologies and tools for supporting collaborative software development has been proposed and produced.

One major tool for supporting collaborative software development is the version control system (VCS), which allows a team of programmers to access a common set of source code files, check out the programming elements from the repository to the local workspaces, complete their individual programming tasks in the local workspaces, and eventually check in and merge their work, under the support of version control mechanisms [23]. Sophisticated version control systems include Concurrent Versions System (CVS)², Subversion (SVN)³ and Rational ClearCase⁴, which have been widely adopted and used in the software industry.

Different from the traditional asynchronous (or non-real-time) collaboration supported by version control systems, real-time collaborative programming is an emerging technique that supports synchronous (real-time) collaboration: (1) multiple programmers are allowed to work jointly on the same source code document at the same time and their updates to the shared artifacts are instantly propagated to each other; (2) each programmer is able to view and edit any part of the shared source code document freely at any time, and changes performed by others can be noticed and merged in real-time; (3) jobs done by multiple programmers are integrated automatically during the real-time collaboration session. Experimental results have indicated that real-time collaborative programming is capable of accelerating the progress of problem-solving, creating better design and shorter code length, making programmers enjoy the work more, and thus increasing the productivity of programmers and improving the quality of software projects [3][14][19][25]. The potential benefits of this emerging technique result in rising interests in both research communities and software industry, and a number of real-

² http://www.nongnu.org/cvs
³ http://subversion.apache.org
⁴ http://www.ibm.com/software/awdtools/clearcase
2. EXPLORING SEMANTIC CONFLICTS IN REAL-TIME COLLABORATIVE PROGRAMMING SCENARIOS

2.1 Case Analysis of Semantic Conflicts

Following the general concept of semantic consistency in collaborative editing, semantic consistency in real-time collaborative programming is particularly concerned with whether the shared source code is correct with respect to programming language rules and problem-solving logic. In this paper, a classical programming task—Stack implementation in Java—is used to illustrate representative semantic inconsistencies (semantic conflicts) that may occur during real-time collaborative programming sessions. We implement the Stack as a Java class that stores integers as the stack elements and provides a set of simple stack functionalities, including: pushing an element onto the top of the stack; popping the top element out of the stack; retrieving the top element of the stack; and checking whether the stack is empty.

```java
package stack;

public class Stack {
    protected int store[];
    protected int max_length;
    protected int top;

    public Stack(int size) {
        store = new int[size];
        max_length = size - 1;
        top = -1;
    }

    public boolean push(int num) {
        if (top == max_length)
            return false;
        store[top] = num;
        return true;
    }

    public boolean pop() {
        if (top == -1)
            return false;
        top--;
        return true;
    }

    public int top() {
        return top;
    }

    public boolean isEmpty() {
        return (top == -1);
    }
}
```

Figure 1. Java source code of Stack implementation.

Suppose two programmers\(^8\) (denoted as \(P_1\) and \(P_2\)) are jointly conducting the task in a real-time collaboration session, where the syntactic consistency of the source code is guaranteed by means

---

\(^3\) [http://www.beweevee.com/subethaedit](http://www.beweevee.com/subethaedit)

\(^6\) [http://www.beweevee.com](http://www.beweevee.com)

\(^7\) [http://www.wave-vs.net](http://www.wave-vs.net)

---

\(^8\) Case studies in this paper use collaboration sessions involving two programmers in order to simplify the illustration, but all discussions also apply to scenarios with more programmers.
The assumption that the syntactic consistency of the shared source code document is maintained by means of the OT technique \cite{[22]}.

After a period of collaboration, the source code evolves into the state as presented in Figure 1. The constructor `Stack` takes a parameter that specifies the maximum size of the stack; the method `push` pushes an integer into the stack; the method `pop` pops the top element out of the stack; the method `top` returns the top element of the stack; and the method `isEmpty` tells whether the stack is empty. The basic idea of the stack implementation is to maintain a linear integer array `int store[]` as the internal data structure for storing stack elements. In addition, the field `int max_length` indicates the maximum size of the stack specified by the constructor and the field `int top` indicates the current index of the top element of the stack.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{stack.png}
\caption{Semantic conflict (Case 1).}
\end{figure}

\subsection*{2.1.1 Case 1: Semantic Conflict between Concurrent Editing Operations in a Self-Contained Source Code Segment}

As illustrated in Figure 2, \(P_1\) and \(P_2\) are concurrently editing the same method `push` at one moment. Initially, the source code segments at the two sites are the same and (syntactically) consistent. However, according to the problem-solving logic, there exists a programming error: the `top` index should be increased by one before placing the pushed integer into the `store` array. Suppose both of them discovered this error and attempted to fix it concurrently in different ways as follows:

1) \(P_1\) added an increment operator to the left of `top` at line 18. It correctly fixed the error in \(P_1\)`s copy of the source code;
2) \(P_2\) added a statement to increase the value of `top` at line 18. It also correctly fixed the error in \(P_2\)`s copy of the source code.

After the two concurrent modifications are propagated to each other\cite{[10]}, the two copies of the source code are identical (thus syntactically consistent), as shown in Figure 2. Unfortunately, the syntactically consistent result remains semantically incorrect: there is still a programming error because the `top` index has been increased twice after the two concurrent modifications. In this case, each of them could fix the original programming error when executed individually, but the two modifications are logically (semantically) incompatible and the merged result converts the original programming error into a new one. In general, such kind of semantic conflicts may occur if multiple programmers are concurrently modifying the same self-contained source code segment (e.g., a method, a field) in semantically incompatible ways. Concurrent changes in a self-contained source code segment may produce programming errors (semantic conflicts) even if each individual change is error-free.

\subsection*{2.1.2 Case 2: Semantic Conflict between Concurrent Editing Operations in a Method and a Referenced Field}

Different from Case 1, \(P_1\) and \(P_2\) are concurrently working in different segments of the same source code document, as shown in Figure 3:

1) \(P_1\) focuses on the definition of the field `store`;
2) \(P_2\) focuses on implementing the incomplete method `top`, which retrieves the top element of the stack.

Under such collaboration pattern, one programmer, while focusing on his/her own work, may not pay attention to where and what others are working. Workspaces of the collaborators may be far away from each other and could not be displayed together in the same screen, bringing adverse impact on the awareness of collaboration work. In this case, \(P_1\) is working at line 5 while \(P_2\) is
working between lines 29-31, so they may not be able to watch each other’s workspace. At one moment, $P_1$ and $P_2$ are performing the following tasks concurrently:

1) $P_1$ changed the internal data structure for the stack element storage (i.e., the field $store$) from an integer array into a List object provided by the standard Java library;

2) $P_2$ completed the implementation of the method $top$ based on the original definition (i.e., before the changes made by $P_1$) of the field $store$.

Clearly, the two concurrent changes are error-free when performed individually at each site, but their combined result becomes semantically incorrect: a programming error (i.e., a data type mismatch between the field $store$ and the method $top$) has been introduced. The root of this problem is that $P_1$ and $P_2$ are concurrently changing source code segments with a method-field-reference dependency relationship: the method $top$ depends on the field $store$ because the former references the latter.

2.1.3 Case 3: Semantic Conflict between Concurrent Editing Operations in a Method and another Invoked Method

Suppose an additional method, void $popList$ (int $size$), is added into the $Stack$ class, which provides the convenient functionality of removing $size$ consecutive elements at the top of the stack. The basic approach is to invoke the existing $pop$ method multiple times (determined by the $size$ parameter) in a loop. As illustrated in Figure 4, the initial implementation of this new method has a problem: the method $popList$ does not report any error message in case of invalid stack operations (e.g., the $pop$ operation is issued when the stack is empty). Upon discovering this problem, $P_2$ modifies the method $popList$ by changing the return type of $popList$ from void to boolean, checking the return value from the invoked method $pop$, and returning a false value if a false value has been returned from the invocation of the $pop$ method. Meanwhile, the programmer $P_1$, who is working on the implementation of the method $pop$, decides to adopt the Java exception mechanism to replace the boolean return value for reporting the error message: a Java exception object is thrown in case of an invalid stack operation issued.

This scenario is illustrated Figure 4. Once again, these concurrent changes are error-free when performed individually. However, a programming error has been introduced: there is a mismatch on the way of reporting invalid stack operation between the method $popList$ and the method $pop$. The root of the problem is that the two programmers are concurrently changing different source code segments with a method-method-invocation dependency relationship: the method $popList$ depends on the method $pop$ because the former invokes the latter.

2.2 General Conditions of Semantic Conflicts

The above cases are not exhaustive but representative in illustrating semantic conflicts in real-time collaborative programming sessions, and it can be derived that semantic conflicts may occur under the following general conditions:

1) Multiple programmers are performing concurrent editing operations;

2) The concurrent editing operations are performed in the same self-contained source code segment or in multiple source code segments with dependency relationships;

3) The editing operations are semantically incompatible.

All illustrated cases meet these conditions. The third condition, semantic incompatibility, may take different meanings. Firstly, semantic incompatibility may refer to incompatible problem-solving logic. Case 1 is an example of incompatible design of the interior logic of a self-contained source code segment. In general, the interior logic within a self-contained source code segment (e.g., a method) is often ad hoc and unstructured during the development phase. Concurrent editing within such source code segments may readily cause semantic conflicts. Secondly, semantic incompatibility may refer to incompatible usage of programming language rules (e.g., language syntax and mechanism). In Case 2, collaborators have different opinions on the definition of the field $store$; in Case 3, they adopt different error handling mechanisms.

For software projects that are modeled and structured by following good design principles (e.g., increasing modularity, reducing coupling between modules, and separation of concerns), semantic conflicts related to high-level components (e.g., modules, packages) are less likely to occur if collaborating programmers conduct

---

The $pop$ method reports the error message in the form of a boolean value so that the invoker could obtain whether the pop operation is successful or not by checking this return value.
their work by following design documentations. In contrast, collaborative development on low-level components (e.g., within the same class or method) are often conducted in an ad hoc and unstructured fashion, and more likely to produce source code with incompatible problem-solving logic and cause semantic conflicts. Our research focuses on solving conflict problems at this level.

3. DEALING WITH SEMANTIC CONFLICTS: A DEPENDENCY-BASED AUTOMATIC LOCKING APPROACH

3.1 Alternative Approaches

There are two general approaches in dealing with semantic conflicts: (1) *conflict resolution*, which allows semantic conflicts to occur, and then detects and resolves them; and (2) *conflict prevention*, which prohibits semantic conflicts from happening.

The *conflict resolution* approach is suitable for dealing with semantic conflicts that are easy to detect and cheap to resolve. For example, semantic conflicts resulting in violation of *programming language rules* (e.g., *Case 2* and *Case 3*) can be reported by the compiler or even automatically detected during the coding process in sophisticated integrated development environments (IDEs), which is supported by advanced techniques such as the incremental Java compiler in the Eclipse Java development tools (JDT)\(^\text{12}\). The major cost for resolving this kind of semantic conflicts is the additional time and effort spent in fixing the problem.

The *conflict prevention* approach is suitable for semantic conflicts that are difficult to detect and costly to resolve. For example, semantic conflicts resulting in *programming logic errors* (e.g., *Case 1*) are difficult to detect. Logic errors cannot be automatically detected during coding or compilation. They can only be detected by certain effort (e.g., source code reviews, software testing), which is time-consuming and costly. Some logic errors may even not be detected by reviews or testing, and it is very dangerous to deliver software products with such latent bugs. The additional complication of logic errors caused by semantic conflicts is that they are introduced by multiple programmers in concurrent work and the work looks error-free from the perspectives of individual programmers. Therefore, the conflict prevention approach is preferable for dealing with semantic conflicts in real-time collaborative programming.

Given the three necessary conditions of semantic conflicts as presented in Section 2.2, conflict prevention can be achieved by breaking any one of them:

\[^{12}\text{http://www.eclipse.org/jdt}\]
1) Preventing concurrent work, which disallows multiple programmers to work in the same source code document concurrently;

2) Preventing concurrent work in the same self-contained source code segment or in different source code segments with dependency relationships, which allows concurrent work in the same source code document as long as such kind of work does not occur in the same self-contained source code segment or in source code segments with dependency relationships;

3) Preventing incompatible work, which allows programmers to work concurrently in any part of the source code document as long as they never produce incompatible work.

The first approach is most easy to achieve (by simply locking the entire source code document for the unique programmer who is editing it), but too restrictive because it would eliminate the benefits of real-time collaboration. The third approach is most liberal in terms of maximum concurrency and flexibility, but infeasible to achieve because the system has no way to understand whether one source code segment is logically (semantically) compatible with another one. The second approach is most promising as it is able to support fine-grained conflict prevention while supporting concurrent work, and is technically achievable as well. Therefore, the approach of preventing concurrent work in the same self-contained source code segment or in different source code segments with dependency relationships is adopted to deal with semantic conflicts in real-time collaborative programming.

### 3.2 Dependency-based Automatic Locking

Locking is a standard technique for mutual exclusion and conflict prevention [1]. To achieve fine-grained conflict prevention in real-time collaborative programming, we propose a dependency-based locking approach: a programmer is not allowed to edit a source code segment until obtaining exclusive locks 1) on the source code segment to be edited and 2) on those source code segments that are depended on by the one to be edited.

An important design issue with the dependency-based locking is how to place locks selectively on those source code segments with dependency relationships in a real-time collaborative programming environment. One approach is manual locking, i.e., to require programmers to decide when and where to place/release locks. The programmer has to (1) manually specify the scope of the source code segment to be edited; (2) determine those depended source code segments and specify their scopes; (3) request all required locks from the system; and (4) wait until all requested locks are granted before continuing with the editing work. Upon completion of the editing work, the programmer has to manually release the locks. Such complicated process creates extra overheads to the programmers, which may cause the programmers to lose focus on their major programming work. Furthermore, they may also encounter difficulties in correctly analyzing the dependency relationships among multiple source code segments, or fail to specify the correct scopes for the source code segments.

To avoid these problems related to manual locking, we propose a Dependency-based Automatic Locking (DAL) approach: the DAL system, instead of the programmer, automatically determines the scope of the source code segment to be edited and the scopes of the depended source code segments, grants locks on them to the programmer, and releases those locks when the editing work is completed. With automatic locking, programmers will concentrate on their major work without caring about locking.

### 3.3 Basic Concepts of the DAL Approach

The self-contained source code segment is a basic element involved in the DAL approach. This element and several related concepts are defined and introduced in this section.

#### Definition 1: Basic Region and Open Area

A basic region is a continuous sequence of source code that forms a semantically meaningful and self-contained segment. An open area is a continuous sequence of source code outside all basic regions.

Alphabetic symbols are used to represent basic regions. For example in Figure 5, each method/field of the Stack class is identified as a basic region, and other parts of the source code are regarded as open areas.

#### Definition 2: Dependency Relationship

For any two basic regions A and B, if A depends on B in terms of semantics (problem-solving logic and/or programming language rules), then there is a dependency relationship from A to B, denoted as A→B, and B is called a depended region of A. If there exists no dependency relationship between A and B (neither A→B nor B→A), then A and B are independent. Dependency relationship is transitive: given three basic regions A, B and C, if A→B and B→C, then A→C.

Dependency relationships may vary in different application domains. There may exist a variety of reasonable methods in modeling basic regions and dependency relationships, but underlying DAL mechanisms and techniques are generic and independent of dependency-modeling methods. Without losing generality, we use a simple but representative dependency model to motivate and illustrate the design of basic mechanisms for the DAL approach. In this model, there are two types of basic regions—method and field, and two types of dependency relationships as follows:

1) Method-Field-Reference dependency relationship: if method M references field F, then M→F.

2) Method-Method-Invocation dependency relationship: if method M invokes method M', then M→M'.

All basic regions and dependency relationships embedded in a source code can be represented by a dependency graph as follows.

#### Definition 3: Dependency Graph (DG)

A dependency graph (DG) is a directed graph, in which: (1) a node represents a basic region in a source code document; and (2) an arrow from node A to node B represents a dependency relationship A→B.

Based on the dependency model, all basic regions and dependency relationships for the Stack source code document are identified and illustrated as a DG in the right part of Figure 5.

### 3.4 DAL Permission Check

When a programmer is permitted to edit a source code region that is not locked by any programmer (regarded as a free region), the DAL system automatically places exclusive locks on this working region and its depended regions. Consequently, no one else can work in any of these locked regions until those locks are released.
This basic DAL mechanism ensures the effectiveness of semantic conflict prevention by prohibiting concurrent access on selected source code regions (i.e., in the same source code region or in different source code regions with dependency relationships).

Under the basic DAL scheme, for each local editing operation to be performed on the source code, the permission check procedure is applied to examine and grant/deny the editing permission, according to the following permission check conditions.

Definition 4: Permission Check Conditions

Permission to a local editing operation \( O \) is granted only if:

1) \( O \)'s location falls in an open area (free for all collaborating programmers to edit concurrently without requiring locks); or

2) \( O \)'s location falls in a basic region \( W \), and:

   a) \( W \) is already locked by this programmer (granted and recorded previously) as a working region; or

   b) Neither \( W \) nor \( W \)'s depended region is locked by any other collaborating programmer.

To illustrate how the DAL permission check works, see the example in Figure 5. If one of the collaborating programmers, namely \( P_1 \), starts to edit the method \( pop \) while no one is currently holding any lock on it, then the DAL system automatically locks this region (denoted as basic region \( F \)) and its depended region \( int\ top \) (denoted as basic region \( C \)) for \( P_1 \). Consequently, no other programmer can edit any of these regions (\( F \) and \( C \)) because they have been exclusively locked for \( P_1 \). Furthermore, even if another programmer, namely \( P_2 \), attempts to edit a source code region outside these locked regions such as the method \( popList \), the editing would also be denied by the DAL system because the method \( pop \) and the field \( int\ top \) are depended regions of the working region \( popList \), have already been exclusively locked. \( P_2 \) cannot obtain all required locks (on \( I, F \) and \( C \)) at that moment, and therefore has to wait until those locks held by \( P_1 \) are released.

4. TECHNICAL ISSUES AND SOLUTIONS

4.1 DG and Locking State Maintenance

4.1.1 General Requirements

One fundamental issue in technically realizing the DAL approach is the DG maintenance, including the determination of the location and scope of each basic region in the source code, as well as the dependency relationships among them. The DG must always be consistent with the source code: for each method or field in the source code, there must be a corresponding basic region (with correctly recorded location and scope of the basic region) in the DG, and vice versa; for each method-field-reference or method-field-access relationship.
method-invocation relationship in the source code, there must be a corresponding dependency relationship between corresponding basic regions in the DG, and vice versa. Moreover, because the source code content is continuously updated during the programming process, the DG must be continuously updated accordingly as well. In addition, the interactive and dynamic nature of the working environment imposes a real-time efficiency requirement: the DAL system must support efficient and incremental DG updating in order to keep the DG staying consistent with the source code in real-time.

Another related issue is the locking state maintenance, which is concerned with keeping track of which regions are locked by whom. The **locking state** is the key data structure for performing permission check: whenever a programmer issues a local editing operation on the source code, the maintained locking state is consulted to examine whether the programmer is permitted to work in the targeted source code region. It is not necessary for the locking state to cover all basic regions in the source code (i.e., covering the whole DG). Only those source code regions that are currently locked by programmers should be recorded in the locking state in order to facilitate the DAL permission check. Similar to the DG, the locking state is also dynamic in nature, which may be changed from time to time as the source code structure (i.e., in terms of basic regions and dependency relationships) changes or any programmer’s working region changes. Such changes performed on the locking state are regarded as a process of locking state update.

### 4.1.2 The Explicit Maintenance Approach

One possible approach to maintaining the DG and locking state is the **explicit maintenance** approach, which records and maintains the DG and locking state separately from the source code. For example, an explicit DG representation is illustrated in the right part of Figure 5. Under the explicit maintenance approach, whenever a local editing operation is performed on the source code, corresponding update operations must be performed on the DG data structure immediately to maintain the correct mapping between the DG and the source code. Under this approach, the locking state can be integrated with the DG by simply associating relevant DG nodes (that are locked by certain programmers) with locking information. The explicit maintenance approach is intuitive to understand, but based on some technical experiments and analyses, it has several major problems.

Firstly, the complexity involved in the consistency maintenance is overwhelming. As aforementioned, one technical requirement is the consistency between the DG and the source code, which can be named as the DG-SC (dependency graph vs. source code) consistency. In a distributed real-time collaborative programming environment, replication of the DG brings another consistency issue named DG-DG (dependency graph vs. dependency graph) consistency: multiple replicas of a DG must be consistent over all collaborating sites in real-time, which is similar to the syntactic consistency among replicas of text documents in generic real-time collaborative editing systems [22]. Past research has derived that maintaining consistency among document replicas is a challenging task, which requires sophisticated techniques such as the **operational transformation**. However, these existing syntactic consistency maintenance techniques cannot be directly applied to DG replicas; and inventing another technique to maintain the DG-DG consistency is non-trivial.

Secondly, the space and time cost for explicit maintenance is high. In essence, an explicit DG is a separate graphic representation of the source code structure, which may take substantial memory space. On the other hand, editing operations may change both the source code content and the separately maintained DG, which brings significant overhead in executing every editing operation. Such extra overhead could be even higher if an editing operation has impacts on the scopes of multiple source code regions and the dependency relationships among them. In addition, some editing operations may also trigger the process of locking state update on the working region and its depended regions. It is a significant technical challenge to perform all these DG update operations and locking state update procedures on a potentially large and complex DG (with locking state data structure) for every editing operation efficiently in real-time without causing noticeable latency.

### 4.1.3 The Implicit Derivation Approach

In recognizing the problems related to **explicit maintenance**, we propose an **implicit derivation** approach. Under this approach, the DAL system does not maintain the DG separately, but relies on dependency derivation techniques to analyze and derive dependency relationships among specified regions within the source code document on demand and in real-time. It is **implicit** in the sense that the DG is implicitly embedded inside the source code. When a programmer issues an editing operation at a certain location within the source code, the dependency derivation technique is invoked to derive the working region of the editing operation as well as its depended regions, which are necessary in performing permission check and locking state update.

Given any location within the source code, the dependency derivation technique is able to determine whether this location falls in a basic region (e.g., method/field) or an open area. If the location is within a basic region, the dependency derivation technique is able to further determine the scope of this working region, as well as the locations and scopes of its depended regions. This basic derivation functionality can be recursively invoked to derive all depended regions with respect to a given working region within the source code.

Compared to the **explicit maintenance** approach proposed in Section 4.1.2, the **implicit derivation** approach has significant advantages. Firstly, it is able to achieve both DG-SC consistency and DG-DG consistency altogether. The DG-SC consistency can be avoided because all dependency relationships among source code regions are always derived from the latest source code dynamically, and thus the DG-SC consistency is automatically guaranteed provided that the dependency derivation technique works properly and correctly. On the other hand, the DG-DG inconsistency can be eliminated for free because the consistency of the source code replicas has been guaranteed by syntactic consistency maintenance techniques such as the OT, and thus the DG-DG consistency is automatically achieved provided that the same dependency derivation technique is applied at all collaborating sites. Secondly, the **implicit derivation** approach is capable of avoiding most of the space and time overheads under the **explicit maintenance** approach. The space cost can be avoided simply because there is no separate DG maintained; and the time cost on updating the DG is avoided because an editing operation performed on the source code will automatically affect and update the embedded DG, and thus there is no need to perform additional DG updating operations.
Despite these advantages, the *implicit derivation* approach has several technical issues. The first issue is the design and implementation of the dependency derivation technique that is able to analyze the source code regions and dependency relationships efficiently in real-time. In our research prototype implementation (to be discussed in Section 5), we have achieved this functionality by incorporating real-time source code syntax analyzing features that are commonly available in sophisticated IDEs such as the C# editor in the Microsoft Visual Studio\(^\text{13}\) and the Java editor in the Eclipse\(^\text{14}\). The second technical issue is the need for separate representation of the locking state. Under the *implicit derivation* approach, dependency relationships among source code regions are embedded inside the source code, but the locking state is external to the source code, and has to be separately maintained. In essence, the DAL locking state is actually a partial DG associated with locking information. Compared to maintaining the whole DG under the *explicit maintenance* approach, the cost of maintaining a partial DG is much lower. Furthermore, within this partial DG, the DAL system does not need to maintain the location and scope information for those locked regions and update them for every editing operation in real-time, because such location and scope information can be automatically derived from the latest source code by the dependency derivation technique. However, since the locking state is replicated (together with the source code replica) at all collaborating sites, consistency maintenance among multiple locking state replicas requires additional techniques to achieve.

### 4.1.4 DAL Locking State Data Structures

As discussed, under the *implicit derivation* approach, the locking state is separately maintained outside the source code, which will be (1) consulted by the permission check procedure and (2) maintained by the locking state update procedure.

The major information to record in the locking state is which source code regions are locked by which users. As illustrated in the left part of Figure 6, a three-level hierarchical data structure is devised for the DAL locking state: (1) the source code is associated with a *DAL Table* at the root level; (2) the *DAL Table* consists of a list of *DAL Regions* at the middle level; and (3) each *DAL Region* contains a set of *DAL Locks* held by different users at the leaf level\(^\text{15}\).

Within the locking state data structures, the *DAL Lock* represents a lock granted to a particular user on a certain source code region, which is expressed as \(<Owner ID, Region Type>\), where the Region Type indicates whether the locked region is a working region (denoted as W) or a depend region (denoted as D) with respect to the owner. The *DAL Region* represents a source code region (e.g., method/field) that is currently locked by programmers, which is expressed as \(<Region Reference, Lock List>\), where the Region Reference relates the DAL Region to the corresponding segment within the source code content, and the Lock List stores a list of *DAL Locks* placed on this source code region.

The right part of Figure 6 illustrates an instance of the DAL locking state data structures at one moment when two programmers \(P_1\) and \(P_2\) are editing basic regions \(A\) and \(E\) respectively in a real-time collaborative programming session. In the *DAL Table*, it is clearly indicated that: (1) locks on working region \(A\) and depend regions \(B, C, F\) are granted to \(P_1\); and (2) locks on working region \(E\) and depend regions \(C, F\) are granted to \(P_2\). As presented in the DG, five source code regions \(\{A, B, C, E, F\}\) are currently locked by programmers (and thus involved in the DAL mechanism), so there are five corresponding *DAL Regions* within the *DAL Table*. As aforementioned, the DAL locking state is essentially a partial DG, which does not cover source code regions that are not locked (such as basic region \(D\) in this example).

\[\text{13}\] \text{http://www.microsoft.com/visualstudio}

\[\text{14}\] \text{http://www.eclipse.org}

\[\text{15}\] The data structure of one *DAL Region* is able to accommodate multiple *DAL Locks*, which has been designed to allow multiple locks held by different collaborating programmers to be placed on the same source code region for supporting future extensions to the DAL scheme. Under the basic DAL scheme presented in this paper, at most one *DAL Lock* can be granted on one *DAL Region*, which is fully exclusive.

---

**Figure 6.** Data structures for DAL locking state maintenance.
4.2 Key Algorithms in the Basic DAL Scheme

4.2.1 Algorithm for Permission Check Procedure

Based on the permission check conditions defined in Definition 4 and the DAL locking state data structures devised in Section 4.1.4, the algorithm for the permission check procedure can be designed accordingly. For simplification in presenting the algorithm, several utility procedures are designed as follows:

- **GetWorkingRegion(O):** to derive the working region of a given editing operation O. It returns a Region Reference (which points to the corresponding segment in the source code) if O falls in the scope of a basic region, or returns NULL if O falls in an open area.
- **GetDepRegionSet(W):** to derive the set of all depended regions that the working region W depends on.
- **CheckOwnership(R, U, S, T):** to check whether a given source code region R is locked by the user U or by others (specified by S = SELF/OTHERS) as a working region or depended region, or without regarding the region type (specified by T = WORKING/DEPENDED/ALL).

Algorithm 1 presents the permission check procedure for examining the editing permission for a given local editing operation O issued by the local user U: PermissionCheck(O, U). If O falls in an open area, PERMIT_OA is returned to indicate that the permission is granted for an editing operation in an open area (no lock is required); if O falls in a basic region, the procedure further examines the locking state of the working region and its depended regions. If the targeted working region W is found being a working region that is already locked by the local user U, PERMIT_WR is returned to indicate that the permission is granted for an existing working region. If W is found being locked by other users, REJECT is returned to deny the editing permission. If W is not found in the DAL Table, it must be a free region, and W's depended regions are further derived and examined. If any of the depended regions is found being locked by other users, REJECT is returned to deny the editing permission; otherwise, PERMIT_FR is returned to grant the permission for a free region.

**Algorithm 1. PermissionCheck(O, U)**

```
W := GetWorkingRegion(O);
if W = NULL
    return PERMIT_OA;
for each DAL_DepRegion X in DAL_Table {
    if X.RegionReference = O
        if CheckOwnership(W, U, SELF, WORKING) = TRUE
            return PERMIT_WR;
        if CheckOwnership(W, U, OTHERS, ALL) = TRUE
            return REJECT;
    }
DS := GetDepRegionSet(W);
for each Basic_Dependent_R D in DS {
    if CheckOwnership(D, U, OTHERS, ALL) = TRUE
        return REJECT;
}
return PERMIT_FR;
```

4.2.2 Algorithm for Locking State Update Procedure

Based on the locking state data structures, the algorithm for the locking state update procedure is designed accordingly. Two utility procedures are first designed for the locking state update:

- **GrantLocks(W, DS, U):** to place locks on the working region W and the depended region set DS for the user U in the DAL Table.
- **ReleaseLocks(U):** to remove all existing locks held by the user U from the DAL Table.

Whenever an editing operation triggers any change of the locking state, the above two locking operations will be invoked to perform locking state update. For instance, when a programmer starts the editing work in a free region, this region (as the working region) and its depended regions will be locked for this programmer. In another case where the programmer switches the workspace from a locked working region to another free region, the existing locks held by this programmer will be released first, and then a set of new locks on the new working region and depended regions will be granted to this programmer.

In addition, for each local locking operation triggered by a local editing operation performed at a local site, besides its local execution on the local locking state, it is also necessary to propagate the locking operation to all remote sites for execution in order to achieve consistent locking state update over all collaborating sites. Under the basic DAL scheme, this consistent locking state update effect can be achieved by the following measures:

1) At a local site, whenever the locking state update procedure grants or releases a set of locks with respect to an editing operation, a locking operation tag LOCK or RELEASE is additionally attached to the editing operation and then propagated together with the editing operation to all remote sites;

2) At a remote site, whenever a remote editing operation arrives with a locking operation tag (LOCK or RELEASE) attached, the locking state update procedure is then performed accordingly, which updates the locking state consistently with the operation’s local site behaviors.

Algorithm 2 presents the procedure LSUpdate(S, R, U, O) for updating the locking state with respect to an editing operation O issued by the user U. The parameter S specifies whether the update is related to a local editing operation (specified by S = LOCAL) or a remote editing operation (specified by S = REMOTE). The parameter R passes the returned result of the permission check procedure for this editing operation to the locking state update procedure, if the editing operation is a local one.

For a local editing operation O, the locking state update procedure acts according to the result of the permission check: (1) if O is permitted for an open area, all existing locks held by U are released (and no lock is required for the editing in an open area); (2) if O is permitted for a free region, all existing locks held by U are released, and then the new set of required locks are derived and granted; (3) if O is permitted for the existing locked working region, no locking state update action is needed.

For a remote editing operation O, the locking state update procedure acts according to the locking operation tag attached to the editing operation. If the RELEASE tag is attached, all existing locks held by the remote user U are released. If the LOCK tag is
attached, the working region (at the location of the remote editing operation \( O \)) and its depended regions are derived, and locks are granted on these involved regions to the remote user \( U \).

Algorithm 2. \( LSUpdate(S, R, U, O) \)

\[
\begin{align*}
\text{if } S &= \text{LOCAL} \{ \\
&\quad \text{if } R = \text{PERMIT\_OA} \{ \\
&\quad\quad \text{ReleaseLocks}(O); \\
&\quad\quad \text{AttachTag}(O, \text{RELEASE}); \\
&\quad\} \\
&\quad \text{else if } R = \text{PERMIT\_FR} \{ \\
&\quad\quad \text{ReleaseLocks}(O); \\
&\quad\quad \text{AttachTag}(O, \text{RELEASE}); \\
&\quad\quad \text{W} := \text{GetWorkingRegion}(O); \\
&\quad\quad \text{DS} := \text{GetDepRegionSet}(W); \\
&\quad\quad \text{GrantLocks}(W, DS, U); \\
&\quad\quad \text{AttachTag}(O, \text{LOCK}); \\
&\quad\} \\
&\quad \text{else if } S = \text{REMOTE} \{ \\
&\quad\quad \text{if } \text{CheckTag}(O, \text{RELEASE}) = \text{TRUE} \\
&\quad\quad\quad \text{ReleaseLocks}(O); \\
&\quad\quad \text{if } \text{CheckTag}(O, \text{LOCK}) = \text{TRUE} \{ \\
&\quad\quad\quad\quad \text{W} := \text{GetWorkingRegion}(O); \\
&\quad\quad\quad\quad \text{DS} := \text{GetDepRegionSet}(W); \\
&\quad\quad\quad\quad \text{GrantLocks}(W, DS, U); \\
&\quad\quad\} \\
&\quad\} \\
\end{align*}
\]

It is worth pointing out that the DAL mechanism is deadlock-free. Deadlock among multiple users (collaborating sites) is impossible because the permission check procedure is performed in an unblocking manner, thus avoiding the possibility of the “hold and wait” condition, which is a necessary condition for any deadlock. The permission check procedure always returns with either a success (i.e., the editing is permitted and locks are updated accordingly if necessary) or a failure (i.e., the editing is denied). In case that the editing permission is denied, the user is notified of the result but not blocked, and therefore s/he can temporarily continue the programming work in other parts (e.g., free regions or open areas) of the source code and later switch the workspace to this region again if the previously placed locks have been released.

4.2.3 Integrated Operation Handlers for Processing Editing and Locking Operations

Algorithm 3 presents the local operation handler \( LOH(O, U) \) for processing a local editing operation \( O \) issued by the local user \( U \). The handler first invokes the permission check procedure to examine the editing permission. If the permission is denied, \( O \) is prohibited from execution, and \( U \) is notified of the failure; otherwise, it is executed on the source code, and the \( LSUpdate \) procedure is then invoked to update the locking state accordingly. Finally, the operation is propagated to remote sites for execution.

Algorithm 4 presents the remote operation handler \( ROH(O, U) \) for processing a remote editing operation \( O \) issued by a remote user \( U \). \( O \) is first executed without permission check because \( O \) has been permitted at its local site and its permission at a remote site is guaranteed by nature. After the execution of \( O \) on the source code, the locking state is updated accordingly if necessary.

5. PROTOTYPE IMPLEMENTATION

To validate the feasibility of the DAL approach and techniques, a prototype system named \( CoEclipse \) has been implemented \[8\]. It has converted the single-user Eclipse Java editor into a real-time collaborative programming system and supports multiple programmers to conduct real-time collaborative programming with DAL-based semantic conflict prevention features. The \( CoEclipse \) client application runs as a plug-in with the Eclipse IDE, which is distributed at multiple sites connected to a \( CoEclipse \) central server via communication networks.

Figure 7 presents a snapshot\(^{16}\) of the \( CoEclipse \) user interface in a real-time collaborative programming session participated by two collaborating programmers who are concurrently editing the \( Stack \) source code illustrated in Section 2. The DAL mechanism works (including permission check, notification, and locking state update) automatically without manual effort from programmers. In addition to the locking mechanism, \( CoEclipse \) also provides advanced user interface features for supporting collaboration awareness regarding the DAL locking state. The system assigns distinctive background colors to different source code segments for supporting locking state awareness. Unlocked source code segments (e.g., open areas and free regions) take the default color (i.e., uncolored). Source code regions locked for a user are highlighted by the particular color assigned to this user, and the working region among them is further differentiated by a color bar displayed to the left of the vertical ruler column with line numbers.

For instance, in this snapshot, the class constructor \( Stack \) is a free region (uncolored); the local programmer is currently holding locks on source code regions \{push, int store[], int max_length, int top\}, with the working region push highlighted by the color bar in the left; the remote programmer is working on the method \( popList \) with locks granted on source code regions \{popList, pop, int top\}; particularly, the field \( int top \) is highlighted by a special color to indicate that it is currently locked by both of them\(^{17}\). With

\(^{16}\) The snapshot is best to view in color.

\(^{17}\) Allowing multiple locks to be placed on one source code region is an important extension to the basic DAL scheme, which has been addressed in our follow-up research work. This significant extension will be presented in a future paper for reporting a comprehensive package of contributions including motivations, design rationales, supporting techniques, and implementations.
these awareness features, programmers can intuitively observe which segments of the source code are locked by whom, and thus avoid editing source code regions locked by others. If a programmer accidently or intentionally issues an editing operation in a source code region that is locked by others, the editing permission will be denied, and the programmer is notified of the failure.

The CoEclipse prototype system has provided a proof-of-concept for the DAL approach and technical solutions. Preliminary experiments have confirmed that the local responsiveness of the real-time collaborative programming system is as good as the single-user Eclipse IDE while consistency maintenance features are incorporated, and meanwhile, the effect of real-time remote notification is good as well because of the small message sizes and low bandwidth requirements.

6. CONCLUSIONS AND FUTURE WORK
In this paper, we have contributed a novel approach for preventing semantic conflicts in real-time collaborative programming environments, which has initiated a new research direction for supporting real-time collaborative programming. By analyzing a set of representative programming scenarios for understanding the nature and general conditions of semantic conflicts, it is derived that semantic conflicts may occur when multiple programmers are performing concurrent and incompatible editing work in the same self-contained source code region or in different source code regions with dependency relationships. Based on these analyses of semantic conflicts, a Dependency-based Automatic Locking (DAL) approach is proposed, and major technical issues and solutions for supporting the DAL approach are discussed and presented, in
cluding dependency graph (DG) and locking state maintenance, permission check, locking state update, and integrated handlers for processing editing and locking operations. The novelty of the DAL approach lies in its capabilities of supporting automatic and fine-grained locking on selected source code regions with dependency relationships to balance conflict prevention, concurrent work, and programmer convenience. As an important part of the contributions, the DAL approach and techniques have been implemented in a prototype system named CoEclipse, which has provided a proof-of-concept for the DAL approach and validated the feasibility of the technical solutions.

This paper has focused on the basic DAL scheme and techniques under two constraints: (1) the DG structure of the source code remains static during real-time collaborative programming sessions, in the sense that editing operations may change the textual content of the source code, but never result in DG structural changes (e.g., creation/deletion of basic regions and dependency relationships among them); and (2) there exists no regional overlapping relationship among concurrent locking operations. Our future work will extend the basic DAL scheme to fully support concurrent and overlapping editing operations that may dynamically change the DG structure of the source code, and the CoEclipse prototype will be continuously extended and used as a tool for further exploration and evaluation.

7. REFERENCES
ABOUT THE AUTHORS:

Hongfei Fan received the BEng degree in software engineering from the School of Software Engineering, Tongji University, China, in 2007. He is currently a PhD candidate in the School of Computer Engineering, Nanyang Technological University, Singapore. His PhD research focuses on innovative techniques for supporting collaborative software development, and his research interests include distributed computing, computer supported cooperative work, and software engineering. He has received two Best Paper Awards, one in the 2012 ACM Symposium on Applied Computing and another in the 2012 IEEE International Conference on Computer Supported Cooperative Work in Design, where he published the first two papers based on his PhD research work. He is a member of the ACM SIGAPP.

Chengzheng Sun is currently a Professor in the School of Computer Engineering, Nanyang Technological University, Singapore. He received the PhD degree in computer engineering from National University of Defense Technology, China, in 1987, and a second PhD degree in computer science from University of Amsterdam, The Netherlands, in 1992. His current research focuses on collaborative Internet/cloud computing, which lies at the intersection of computer supported cooperative work, distributed systems and computer communication networks, human-computer interaction, and software engineering.
A Unified Framework for Document Clustering with Dual Supervision

Yeming Hu
Dalhousie University
Faculty of Computer Science
6050 University Avenue
Halifax, Canada
yeming@cs.dal.ca

Evangelos E. Milios
Dalhousie University
Faculty of Computer Science
6050 University Avenue
Halifax, Canada
eem@cs.dal.ca

James Blustein
Dalhousie University
Faculty of Computer Science
and School of Information Management
jamie@cs.dal.ca

ABSTRACT

Semi-supervised clustering algorithms for general problems use a small amount of labeled instances or pairwise instance constraints to aid the unsupervised clustering. However, user supervision can also be provided in alternative forms for document clustering, such as labeling a feature by associating it with a document or a cluster. Besides labeled documents, this paper also explores labeled features to generate cluster seeds to seed the unsupervised clustering. In this paper, we present a unified framework in which one can use both labeled documents and features in terms of seeding clusters and refine this information using intermediate clusters. We introduce two methods of using labeled features to generate cluster seeds. Experimental results on several real-world data sets demonstrate that constraining the clustering by both documents and features seeding can significantly improve document clustering performance over random seeding and document only seeding. We also demonstrate that the clustering performance can be improved even with only a fraction of clusters being seeded compared to unsupervised clustering.1

Categories and Subject Descriptors

H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—Clustering; I.5.4 [Pattern Recognition]: Application—Text Processing

General Terms

Algorithm, Document Clustering, Features

Keywords

User Supervision, Feature Supervision, Seeding, Text Cloud

1. INTRODUCTION

Traditional document clustering is an unsupervised categorization that partitions a given document collection into clusters so that topically similar documents are placed into the same clusters. However, given the same document collection, different users may want to organize it in their own point of view instead of a universal one, which is addressed to some extent by incorporating document supervision [3]. In this paper, we have two types of user supervision, namely,

---

**Figure 1:** Text Cloud of a Document about Canadian Basketball

Figure 1: Text Cloud of a Document about Canadian Basketball

---

document supervision and feature supervision for document clustering. Document Supervision involves labeling documents, i.e., assigning a document to a cluster. Feature Supervision involves labeling features, i.e., associating a feature with a document if that feature describes the topic of that document.

Most prior semi-supervised clustering algorithms use user supervision in the form of document supervision such as labeled instances [3] or instance pairwise constraints [26] for general clustering problems. However, user supervision can also be provided in alternative forms such as labeling features (words) for document clustering in addition to labeling instances (documents). Since this paper focuses on document clustering, we may use instance and document, feature and word interchangeably. Labeling documents and words can be performed at the same time, with little additional effort for labeling words, if an appropriate document visualization is used, such as text clouds [19]. While the user assigns a document to a cluster based on the document’s text cloud, the words appearing in the text cloud can also be labeled by being clicked or highlighted.

**Example 1.** Consider a collection of news articles about international sports. While the user labels the document displayed as text cloud (Fig. 1) to a cluster, the words associating the document with the specific cluster can also be labeled by being clicked or highlighted. In one scenario, the document (Fig. 1) can be labeled to cluster “Canada”, in which the words “Canada”, “Canadians” should labeled (associated) with the document. In another scenario, the document would be labeled to cluster “Basketball”, in which the words “basketball”, “points” should be associated with the document.

---

1This work is based on an earlier work: SAC ‘12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03. http://doi.acm.org/10.1145/2245276.2245306
Example 2. Assume we have two papers and one talks about programming languages while the other is about software debugging. One human user can assign them into the same cluster “software engineering” while another one would like to put them into two clusters, i.e., “languages” and “debugging”. Clearly, the keywords (features) assigned for the two cases will be different too.

Therefore, different labeled words reflect different organizations and the user forms his point of view based on the perception of the words in the text clouds. By using the text cloud for labeling documents, the user can not only label documents to seed the clustering but also label the words discriminating among clusters. It has been argued that document supervision and feature supervision are complementary rather than completely redundant and this joint use has been called dual supervision [1].

In this paper, we assume that the user labels a document by reading its content. At the same time, the user can label a word by indicating (e.g. highlighting) whether it is associated with the document or the specific cluster. The text cloud could be used to visualize the document content and enhance the labeling. We extend two methods incorporating the labeled features from document classification to document clustering, namely, feature-vote-model [9] which uses labeled features to vote for cluster label of an unlabeled document, and feature-generative-model [21] which uses labeled features to infer a multinomial generative model. In (semi-supervised) document classification, labeled documents and features are required for each category. However, knowledge of the relevant categories is incomplete in many domains. Semi-supervised document clustering can group documents into partial clusters with labeled documents and features, as well as extend and modify the existing set of clusters to reflect other topical groupings in document collection [3]. In this paper, we propose a clustering model built from both the labeled documents and the labeled features can be used to guide the clustering process. At the same time, the knowledge from the labeled documents and features will be refined by intermediate clusters in an iterative manner. To this end, we present a unified framework which combines knowledge from labeled documents, labeled features, and unlabeled documents by an iterative clustering process. Finally, we demonstrate the effectiveness of the framework on several real-world data sets.

The rest of this paper is organized as follows. Related work on semi-supervised clustering and feature supervision is discussed in Section 2. In Section 3, we introduce the models to incorporate the labeled features and present the unified framework to combine knowledge from labeled documents, labeled features and intermediate clusters. The details of the experimental results on several real-world text datasets are presented and discussed in Section 4. We conclude this paper and discuss the future work in Section 5.

2. RELATED WORK

Existing semi-supervised clustering techniques, employing user supervision in the form of instance-level constraints, are generally grouped into four categories. First, constraints are used to modify the loss function [4, 18, 25]. Second, cluster seeds derived from the constraints initialize the cluster centers [3]. Third, constraints are employed to learn adaptive distance metrics using metric learning techniques [2, 6]. Finally, the original high-dimensional feature space can be projected into low-dimensional feature subspaces guided by constraints [24]. However, alternative forms of user supervision exist when we apply semi-supervised clustering algorithms to group documents. In this paper, we explore words labeled by being associated with a document when the document is assigned to a cluster.

Liu et al. [20] propose to ask the user to label features for each class and use the set of features labeled for each class to label a set of documents for training classifiers. Druck et al. [9] use labeled features for each class to constrain the probabilistic model estimation on unlabeled instances instead of creating pseudo-instances as done in other approaches. Sindhwani and Melville [23] present a novel semi-supervised sentiment prediction method which use both labeled documents and features for each class to train the classifiers. Raghavan et al. [22] make use of feature feedback in the active learning with support vector machine by up-weighting the accepted features. Unlike the above classification methods which require labeled documents and/or features for each class, our framework can deal with partial clusters with labeled documents and/or features. In addition, it explores the unlabeled documents to refine the prior knowledge provided by the user. Huang and Mitchell [17] propose a generative probabilistic framework to incorporate various types of user feedback including feedback on features. In their work, the user needs to assign a feature to an intermediate cluster, which requires the user browse the intermediate clusters and understand them. In our framework, the user associates the features with documents through text clouds, which is much easier and more convenient than understanding intermediate clusters. Hu et al. [11] propose an interactive framework for feature selection for document clustering, in which the user only indicates whether a feature is suitable for clustering. However, they ask the user to label features from a standalone ranked list of features, which requires extra effort for labeling. In addition, they did not explore the usefulness of integrating labeling documents and features together or compare feature supervision with document supervision for clustering.

3. METHODOLOGY

In this section, we first briefly describe basic KMeans algorithm and then present a unified framework to combine the document supervision, feature supervision, and unlabeled documents.

3.1 Background

KMeans [5] is a clustering algorithm based on iterative assignments of data points to clusters and partitions a dataset into K clusters so that the average squared distance between the data points and the closest cluster centers are locally minimized. For a dataset with data points \( X = \{x_1, x_2, \ldots, x_N\}, x_i \in \mathbb{R}^d \), KMeans algorithm generates K clusters \( \{X_i\}_{i=1}^K \) of \( X \) so that the objective function

\[
J = \sum_{i=1}^{K} \sum_{x_i \in X_i} ||x_i - \mu_i||^2
\]  

is locally minimized and \( \{\mu_1, \mu_2, \ldots, \mu_K\} \) represents the cen-
3.2 Algorithms

In this section, we first introduce document supervision and feature supervision in the form of document seeding and feature seeding separately. Then, we present two methods to model feature seeding. At the end, we describe a unified framework to incorporate both document seeding and feature seeding into the KMeans algorithm, namely, DualSeededKMeans.

3.2.1 Document Seeding

Given a dataset \( \mathcal{X} \), as previously described, KMeans can partition it into \( K \) clusters \( \{ \mathcal{X}_i \}_{i=1}^K \). Then, we can define the document seed set \( D^d \subseteq \mathcal{X} \) as the following subset of data points: for each \( x_i \in D^d \), the user provides the cluster \( \mathcal{X}_i \) to which it belongs. We assume that there is at least one data point \( x_i \) for each cluster \( \mathcal{X}_i \). Note that there is a \( K \)-disjoint partitioning \( \{ D_i^k \}_{i=1}^K \) of the seed set \( D^d \) such that all \( x_i \in D_i^k \) belong to \( \mathcal{X}_i \) according to the supervision. We define the centers of the document seed set \( \{ D_i^k \}_{i=1}^K \) as \( \{ \mu_i^d \}_{i=1}^K \):

\[
\mu_i^d = \frac{\sum_{x_i \in D_i^k} x_i}{|D_i^k|}
\]  

(2)

Those seed centers can be used to both initialize the clustering algorithms and guide the clustering process.

3.2.2 Feature Seeding

Similar to document seed set \( D^d \), we can define the feature seed set \( W^l \) as the following subset of features: for each \( w_i \in W^l \), the user indirectly associates it with the cluster \( \mathcal{X}_i \) through document \( x_i \in \mathcal{X}_i \) in which \( w_i \) occurs and is labeled from. We assume that each cluster has a topic and at least one feature is associated with it. Note that there does not exist a \( K \)-disjoint partitioning \( \{ W_i^l \}_{i=1}^K \) of the feature seed set because one feature can be associated with multiple clusters. We define the centers of the feature seed set \( \{ W_i^l \}_{i=1}^K \) as \( \{ \mu_i^l \}_{i=1}^K \), which can be derived from either feature-vote-model (see Section 3.2.4 for details) or feature-generative-model (see Section 3.2.5 for details). And then those seed centers can be used to both initialize the clustering algorithms and guide the clustering process.

3.2.3 Feature supervision

A document \( d \) can be considered as a list of words in the order in which the words appear in the document, i.e., \( < w_1, w_2, \ldots, w_d > \), where \( |d| \) is the length of the document in terms of the number of words. Note that \( w_i \) might be the same as \( w_j \) where \( i \neq j \), \( 1 \leq i \leq |d| \) and \( 1 \leq j \leq |d| \). To label a document, we assume that the user needs to read at least a fraction of the document content, i.e., \( < w_s, w_{s+1}, \ldots, w_e > \), where \( 1 \leq s \leq |d| \) and \( s \leq e \leq |d| \). While reading a document, the user is assumed to be able to recognize useful words for clustering. The useful words should describe the cluster topic of the document from which they are labeled. The fraction of the document content could be displayed as a text cloud and the user could label words by highlighting them through double-clicking on the text clouds. The user labels a feature if it is a good description of the topic of a cluster and discriminated the cluster from others. Then, a labeled feature is associated with a cluster indirectly through the labeled documents from which it is labeled. After a cluster being created, additional features can be associated with a cluster directly by being assigned into the cluster.

3.2.4 Feature-Vote-Model

In this method, we use the labeled features in the feature seed set to vote on cluster labels for the unlabeled documents. A similar approach was introduced for document classification [9, 27]. For each labeled feature \( w \) in a document \( x \), it contributes one vote for each of its cluster labels (could be associated with multiple clusters). Then, we normalize the vote totals to get a probabilistic distribution over the cluster labels for each document, i.e., \( \{ P_i \} \) for document \( x \) and cluster \( \mathcal{X}_i \). Assume document \( x_i \) contains \( n_{il} \) labeled features for cluster \( \mathcal{X}_i \), we define:

\[
P_i = \frac{n_{il}}{\sum_{k=1}^K n_{ik}}
\]  

(3)

With this soft labeled documents, we can derive the center of \( \mu_i^f \) from the feature seed set as:

\[
\mu_i^f = \sum_{x_i \in \mathcal{X}_i} P_i x_i
\]  

(4)

where \( x_i \) is the vector of TFIDF values of the features selected for clustering.

3.2.5 Feature-Generative-Model

This model was introduced for binary sentiment analysis [21] and we extend it for document clustering with multiple clusters. In this method, we generate each cluster center from the feature seed set directly. We choose to represent the cluster center as a multinomial distribution which generates documents for the corresponding cluster. Without losing generality, we derive the cluster center for cluster \( \mathcal{X}_i \) and \( \mu_i^d \)- and \( \mu_i^l \)-words and features are used interchangeably. We define the following notations to aid our derivations:

- \( V \) – set of words used for clustering, including both labeled and unlabeled words
- \( \mathcal{P}_{\mathcal{X}_i} \) – set of words labeled for cluster \( \mathcal{X}_i \)
- \( \mathcal{N}_{\mathcal{X}_i} \) – set of words labeled for the other clusters
- \( \mathcal{U} \) – set of unlabeled words used for clustering
- \( m \) – size of vocabulary, i.e. \(|V|\)
- \( p_{\mathcal{P}_{\mathcal{X}_i}} \) – number of words labeled for cluster \( \mathcal{X}_i \), i.e. \(|\mathcal{P}_{\mathcal{X}_i}|\)
- \( n_{\mathcal{N}_{\mathcal{X}_i}} \) – number of words labeled for the other clusters, i.e. \(|\mathcal{N}_{\mathcal{X}_i}|\)

In order to derive the multinomial distribution for cluster center of \( \mathcal{X}_i \), we assume the following properties about the relationships between words and clusters.

Property 1: All words in \( \mathcal{P}_{\mathcal{X}_i} \) are equally likely to occur in a document from cluster \( \mathcal{X}_i \).

\[
P(w_i | \mathcal{X}_i) = P(w_j | \mathcal{X}_i), \forall w_i, w_j \in \mathcal{P}_{\mathcal{X}_i}
\]  

(5)

We refer to the probability of any word in \( \mathcal{P}_{\mathcal{X}_i} \) appearing in a document from cluster \( \mathcal{X}_i \) simply as \( P(w_i | \mathcal{X}_i) \).

Property 2: All words in \( \mathcal{N}_{\mathcal{X}_i} \) are equally likely to occur in a document from cluster \( \mathcal{X}_i \).

\[
P(w_i | \mathcal{X}_i) = P(w_j | \mathcal{X}_i), \forall w_i, w_j \in \mathcal{N}_{\mathcal{X}_i}
\]  

(6)

We refer to the probability of any word in \( \mathcal{N}_{\mathcal{X}_i} \) appearing in a document from cluster \( \mathcal{X}_i \) simply as \( P(w_i | \mathcal{X}_i) \).

Property 3: The unlabeled words are treated equally in each cluster.

\[
P(w_i | \mathcal{X}_i) = P(w_j | \mathcal{X}_i), \forall w_i, w_j \in \mathcal{U}
\]  

(7)
We refer to the probability of any word in \( U \) appearing in a document from cluster \( X_i \) simply as \( P(w_i|X_i) \).

Property 4: A document from cluster \( X_i \) is more likely to contain a word from \( P_{X_i} \) than a word from \( \mathcal{N}_{X_i} \)

\[
P(w_i|P_{X_i}) = r \times P(w_i|X_i)
\]

where \( r \) is referred to as polarity level, which measures how much more likely a word in \( P_{X_i} \) occurs in a document from cluster \( X_i \) compared with a word in \( \mathcal{N}_{X_i} \). Since a word in \( P_{X_i} \) is more likely occurs in a document from cluster \( X_i \), we have \( 0 < 1/r \leq 1 \).

Property 5: The multinomial probability distribution learned from labeled features for each cluster is constrained by summing to one.

\[
\sum_{i=1}^{m} P(w_i|X_i) = 1
\]

We use property 5 as constraints to derive the appropriate probability distribution based on labeled features. By Eq. 9 it follows that

\[
pP(w_i|P_{X_i}) + nP(w_i|\mathcal{P}_{X_i}) + (m-p-n)P(w_i|\mathcal{N}_{X_i}) = 1
\]

which gives us the following inequality using Eq. 8,

\[
pP(w_i|X_i) + nP(w_i|X_i) \leq 1
\]

\[
\Rightarrow pP(w_i|X_i) + n P(w_i|X_i) \leq 1
\]

Since \( 0 < 1/r \leq 1 \), it follows that,

\[
P(w_i|X_i) \leq \frac{1}{p+n}
\]

By assigning the maximum probability mass to the known words, \( P(w_i|X_i) \) it is set to the maximum value possible, i.e.

\[
P(w_i|X_i) = \frac{1}{p+n}
\]

Now, it follows from Eq. 8,

\[
P(w_i|X_i) = \frac{1}{p+n} \times \frac{1}{r}
\]

Now, solving Eq. 10, we can have the probabilities for the unlabeled words:

\[
P(w_i|X_i) = \frac{n(1-1/r)}{(p+n)(m-p-n)}
\]

Finally, we use Eqs. 11, 12 and 13 to derive the center \( \mu_i^w \) of cluster \( X_i \). The cluster center \( \mu_i^w \) is defined as a vector, whose elements are the probabilities of words in \( V \) given the cluster \( X_i \), namely,

\[
\mu_i^w = (P(w_1|X_i), P(w_2|X_i), \ldots, P(w_m|X_i))
\]

where \( w_i \in V \) and \( m = |V| \) as previously defined.

In our experiments, we set \( r = 100 \) based on previous experimental results [21].

### 3.2.6 Combining Multiple Centers

**Opinion pool** is a general approach to combine information from multiple sources, such as the centers derived from document seed set and feature seed set in our document clustering problem. Particularly, we use linear opinion pool approach to aggregate multiple centers, which was used to combine probability distributions for text classification [21]. In this approach, the aggregated (pooling) center is defined as

\[
\mu_i = \sum_{s=1}^{S} \alpha_s \mu_i^s
\]

where \( S \) is the number of sources we have.

In addition, we compute the weights \( \alpha_i \) s of individual sources based on their error in labeling the document seed set. In particular, we use the same weighting scheme as [21]:

\[
\alpha_s = \log \frac{1}{err_s}
\]

where \( err_s \) is the classification error of the source \( s \) when the derived centers based on the information provided by the source \( s \) are used to classify the documents in the document seed set. All \( \alpha_s \)'s are normalized to one.

### 3.2.7 Dual Semi-supervised KMeans

In DualSeededKMeans, both the document seeds and feature seeds are used to initialize the KMeans algorithm through derived cluster centers. To this end, the center of the \( t^{th} \) cluster is initialized with the pooling center derived from both \( \mu_i^d \) and \( \mu_i^w \) (Eq. 15) before the clustering starts. During the clustering, the cluster centers are refined using the information contained in the intermediate clusters. This information is expressed in the form of intermediate cluster centers \( \mu_i^t \)

\[
\mu_i^t = \frac{\sum_{x_i \in X_i^t} x_i}{|X_i^t|}
\]

where \( X_i^t \) is the \( t^{th} \) intermediate cluster. Then, we can incorporate \( \mu_i^t \) to the DualSeededKMeans algorithm using the linear opinion pool technique (Eq. 15). The algorithm is described in details in Alg. 1. Note that DualSeededKMeans can be specialized to DocumentSeededKMeans when feature seed set is empty and FeatureSeededKMeans when document seed set is empty.
3.3 Oracles
Most research involving labeling documents simulates human input by a document oracle that uses the underlying class labels of documents in the dataset [1, 3, 4, 6, 16, 15, 18, 24]. However, in the case of features, we do not have a gold-standard set of feature labels. Ideally, we should have a human expert in the loop labeling the selected features. However, such a manual process is not feasible for repetitive large-scale experiments. Therefore, we construct a feature oracle similar to the method described by [9, 10, 11, 13, 14]. Using the document labels, the oracle computes the $\chi^2$ value of each feature with cluster/class label, and accept a feature if the $\chi^2$ value is above a threshold $\beta$. In this paper, the $\beta$ value is the mean of the top $f$ most predictive features, where $f = 100K$, namely, 100 times the number of clusters. If accepted, the feature oracle labels a feature with the cluster in which it occurs the most and any other clusters in which the feature occurs at least half of the most occurrences.

4. EXPERIMENTAL RESULTS

4.1 Datasets
We conducted our experiments on several real-word datasets of different sizes and also consisting of different types of text documents. We derive three datasets of different sizes from the 20-Newsgroup corpus and three more datasets from webkb, industry sector, and reuters21578 separately. The datasets are different from each other in terms of sizes of the datasets and types of documents, i.e., webpages, newsgroup messages, etc. The descriptions and details of the datasets are summarized in Table 1.

We pre-processed each document by tokenizing the text into bags-of-words. Then, we removed the stop words and stemmed all the remaining words. Next, we selected the top 2000

---

Table 1: Six Datasets from the 20-newsgroups, Webkb, Industry Sectors and Reuters21578

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th>Categories included</th>
<th>Category Doc.</th>
<th>Total Doc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>news-similar-3-100 (D1)</td>
<td>The 20-Newsgroup dataset consists of 20 different Usenet newsgroups, each of which has approximately 1000 newsgroup messages.</td>
<td>3:comp.graphics,comp.os.ms-windows,misc.comp.windows.x</td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>news-multi-r-100 (D2)</td>
<td>webpages from different universities</td>
<td>10:alt.atheism,comp.sys.mac.hardware,misc.forsale,rec.sport.hockey,sci.crypt,-talk.politics.guns.soc.religion.christian</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>news-multi-10-100 (D3)</td>
<td>news articles from different industrial sectors</td>
<td>1:comp.graphics,comp.os.ms-windows,misc.comp.windows.x</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>webkb-sfcp-4-250 (D4)</td>
<td>webpages from different universities</td>
<td>4:student, faculty, course, project</td>
<td>250</td>
<td>1000</td>
</tr>
<tr>
<td>sector-multi-10-100 (D5)</td>
<td>webpages from different industrial sectors</td>
<td>10:basic.materials,capital.goods,consumer.cyclical,investment.services,biotechnology.and.drugs,hotels.and.motels,communications.equipment,railroad,water,utilities</td>
<td>100 (railroad-95)</td>
<td>995</td>
</tr>
<tr>
<td>reuters-multi-10-100 (D6)</td>
<td>news articles from Reuters21578. We use the top 10 most frequent categories, documents of which does not have multiple labels.</td>
<td>10:acq, coffee, crude, earn, gold, interest, money,fx, ship, sugar, trade</td>
<td>100 (gold-90)</td>
<td>990</td>
</tr>
</tbody>
</table>

---

4.2 Evaluation Measures
In this paper, we employed normalized mutual information (NMI) [8] as the clustering evaluation measure. NMI measures the share information between the cluster assignments $S$ and class labels $L$ of documents. It is defined as:

$$NMI(S, L) = \frac{I(S, L)}{(H(S) + H(L))/2}$$  \hspace{1cm} (18)

where $I(S, L)$, $H(S)$, and $H(L)$ denote the mutual information between $S$ and $L$, the entropy of $S$, and the entropy of $L$ respectively. The range of NMI values is 0 to 1.

4.3 Analysis of Results
First, we have two sets of comparisons in our experiments. The first set of comparisons is designed to see whether the user provided information can be refined by the intermediate clusters, i.e., clustering models incorporating unlabeled documents categorize documents better than classification models which only use labeled information. The clustering and classification models are defined as:

- DualSeededKMeans, or its specialized algorithms when one of the seed set is empty, i.e., DocumentSeededKMeans and FeatureSeededKMeans. Note that FeatureSeededKMeans has two variants, namely, Feature-Vote-Model and Feature-Generative-Model to derive cluster centers.

- SupervisedKMeans, which performs clustering by assigning documents to nearest cluster centers inferred from either document seed set or feature seed set or both. It can be achieved by running the DualSeededKMeans or its specialized cases, i.e., DocumentSeededKMeans and FeatureSeededKMeans, with only one iteration. Correspondingly, we have DualSupervisedKMeans, DocumentSupervisedKMeans, and FeatureSupervisedKMeans.

We did thorough pair comparisons (Table 2) to demonstrate
Table 2: Supervised KMeans compared to peer algorithms refined by intermediate clusters. 10 documents are labeled for each cluster and features are labeled by feature oracle from the labeled documents. We did two-tailed paired t-test with \( p = 0.05 \) for comparing pairs of algorithms. In this table, we compare algorithms by pairs, i.e., DocumentSeeded KMeans vs. DocumentSupervised KMeans, FeatureSeeded KMeans vs. FeatureSupervised KMeans using Feature-Vote-Model and Feature-Generative-Model. DualSeeded KMeans vs. DualSupervised KMeans using Feature-Vote-Model and Feature-Generative-Model. All algorithms refined by intermediate clusters works significantly better than peer Supervised KMeans algorithm except FeatureSeeded KMeans and FeatureSupervised KMeans using Feature-Vote-Model on D3 (news-multi-10-100) and DualSeeded KMeans and DualSupervised KMeans using Feature-Vote-Model on D1 (news-similar-3-100) indicated by *.

<table>
<thead>
<tr>
<th>Supervision</th>
<th>Algorithm</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Supervision</td>
<td>Basic KMeans</td>
<td>0.069</td>
<td>0.523</td>
<td>0.468</td>
<td>0.341</td>
<td>0.710</td>
<td>0.350</td>
</tr>
<tr>
<td>Document Only</td>
<td>DocumentSeeded KMeans</td>
<td>0.276</td>
<td>0.692</td>
<td>0.686</td>
<td>0.397</td>
<td>0.815</td>
<td>0.637</td>
</tr>
<tr>
<td>Feature Only</td>
<td>FeatureSeeded KMeans</td>
<td>0.551</td>
<td>0.770</td>
<td>0.820*</td>
<td>0.464</td>
<td>0.795</td>
<td>0.649</td>
</tr>
<tr>
<td>Dual Supervision</td>
<td>FeatureSupervised KMeans</td>
<td>0.548</td>
<td>0.766</td>
<td>0.820*</td>
<td>0.428</td>
<td>0.791</td>
<td>0.634</td>
</tr>
<tr>
<td>Feature-Vote-Model</td>
<td>FeatureSeeded KMeans</td>
<td>0.512</td>
<td>0.681</td>
<td>0.747*</td>
<td>0.413</td>
<td>0.734</td>
<td>0.660</td>
</tr>
<tr>
<td>Feature-Generative-Model</td>
<td>FeatureSeeded KMeans</td>
<td>0.482*</td>
<td>0.757</td>
<td>0.783*</td>
<td>0.421</td>
<td>0.822</td>
<td>0.687</td>
</tr>
<tr>
<td>Dual Supervision</td>
<td>FeatureSupervised KMeans</td>
<td>0.482*</td>
<td>0.745</td>
<td>0.765*</td>
<td>0.372</td>
<td>0.815</td>
<td>0.660</td>
</tr>
<tr>
<td>Feature-Vote-Model</td>
<td>DualSeeded KMeans</td>
<td>0.423</td>
<td>0.732</td>
<td>0.738*</td>
<td>0.443</td>
<td>0.824</td>
<td>0.684</td>
</tr>
<tr>
<td>Feature-Generative-Model</td>
<td>DualSeeded KMeans</td>
<td>0.421</td>
<td>0.703</td>
<td>0.700*</td>
<td>0.391</td>
<td>0.812</td>
<td>0.642</td>
</tr>
</tbody>
</table>

Figure 2: Performance as a function of the Number of Unseeded Clusters. 5 Documents Are Labeled for Each Seeded Cluster where FeatureSeeded KMeans works better than DocumentSeeded KMeans and DualSeeded KMeans.
Table 3: Comparison of algorithms with dual supervision to algorithms with any single supervision. 20 documents are labeled for each cluster and features are labeled by feature oracle from those labeled documents. We did two-tailed paired t-test with p = 0.05 for comparing pairs of algorithms. In this table, we compared DualSeeded KMeans with DocumentSeeded KMeans, DualSeeded KMeans with FeatureSeeded KMeans using Feature-Vote-Model or Feature-Generative-Model. DualSeeded KMeans works better than DocumentSeeded KMeans on all datasets. DualSeeded KMeans works better than FeatureSeeded KMeans on all datasets except D1 (news-similar-3-100) and D4 (webkb-sfcp-4-250) with Feature-Generative-Model indicated by *.

<table>
<thead>
<tr>
<th>Feature Model</th>
<th>Algorithm</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Supervision</td>
<td>Basic KMeans</td>
<td>0.69</td>
<td>0.52</td>
<td>0.46</td>
<td>0.34</td>
<td>0.71</td>
<td>0.35</td>
</tr>
<tr>
<td>Document Only</td>
<td>DocumentSeeded KMeans</td>
<td>0.41</td>
<td>0.70</td>
<td>0.80</td>
<td>0.46</td>
<td>0.84</td>
<td>0.76</td>
</tr>
<tr>
<td>Feature-Vote-Model</td>
<td>Feature Only FeatureSeeded KMeans</td>
<td>0.56</td>
<td>0.71</td>
<td>0.81</td>
<td>0.48</td>
<td>0.79</td>
<td>0.67</td>
</tr>
<tr>
<td>Feature-Generative-Model</td>
<td>Dual Supervision</td>
<td>0.56</td>
<td>0.81</td>
<td>0.83</td>
<td>0.48</td>
<td>0.84</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>Dual Seeded KMeans</td>
<td>0.50</td>
<td>0.80</td>
<td>0.82</td>
<td>0.50</td>
<td>0.89</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Figure 3: Performance as a Function of the Number of Unseeded Clusters. 20 Documents Are Labeled for Each Seeded Cluster where DualSeeded KMeans works better than DocumentSeeded KMeans and FeatureSeeded KMeans.
Figure 4: Performance as a Function of the Number of Labeled Documents. The more documents labeled, the more features labeled and the better performance. The usefulness of labeled features are more obvious when there are only a few documents labeled, e.g., < 10. In fact, the feature supervision even works better than dual supervision at the beginning of the curves, indicating that feature supervision is more reliable when only few documents are labeled.
that incorporating unlabeled documents can refine the information provided by the user and produce better clusters than only using labeled information. Concretely, we compared the following pairs of algorithms:

- DocumentSeededKMeans vs. DocumentSupervisedKMeans
- FeatureSeededKMeans vs. FeatureSupervisedKMeans
- DualSeededKMeans vs. DualSupervisedKMeans
- DualSeededKMeans vs. DocumentSeededKMeans using Feature-Vote-Model or Feature-Generative-Model.

From Table 2, we can tell that all algorithms with refinement by intermediate clusters improve its clustering performance over the peer algorithms of SupervisedKMeans except when Feature-Vote-Model with only feature supervision works on dataset D3 (news-multi-10-100) and DualSeededKMeans and DualSupervisedKMeans using Feature-Vote-Model on D1 (news-similar-3-100) (indicated by * in Table 2), which is only 2 out of 30 comparisons. Therefore, intermediate clusters are helpful in improving clustering performance in addition to labeled information.

The second set of comparisons is designed to see whether dual supervision performs better than any single supervision. Thus, we compare DualSeededKMeans with DocumentSeededKMeans, and FeatureSeededKMeans. Again, we have two variants when feature seed set is involved. From Table 3, we can tell that dual supervision with both document and feature generally improve the clustering performance over any single supervision except with feature-generative-model on D1 (news-similar-3-100) and D4 (webkb-sfcp-4-250) indicated by * in Table 3, which is only 2 out of 24 comparisons. Note that algorithms with dual supervision works better than document only supervision on all datasets. Therefore, it is worth labeling features.

Second, we ran experiments with incomplete seeding, namely, only a fraction of categories are seeded by labeled documents, or labeled features, or both (Fig. 2 and Fig. 3). It can be seen that the performances decreases with increase number of unseeded clusters. However, the performances do not decrease substantially, showing that DualSeededKMeans can extend the seeded clusters and generate more clusters to fit the regularities in the dataset. Therefore, not all clusters have to be seeded by labeled information.

Finally, we study the behaviors of the DualSeededKMeans with different numbers of document seeds. Note that the more document seeds labeled, the more feature seeds labeled because the feature seeds are labeled while a document is being labeled. We have the following observations from Fig. 4.

- DualSeededKMeans always works better than DocumentSeededKMeans. However, the performances of the two algorithms are getting close when more documents are provided. It suggests that the feature labeling is more useful when there are few documents labeled, i.e., little effort. One of the possible explanations is that few labeled documents cannot represent the cluster very well, which can be enhanced by the labeled features at the beginning of the learning curve. However, with enough documents labeled, the cluster structures can be represented pretty well with only documents so that the dual supervision has similar performance to document supervision only.

- When there are only a few documents labeled, FeatureSeededKMeans (fewer feature seeds) performs better than DualSeededKMeans and DocumentSeededKMeans. It suggests that feature supervision is more reliable than document supervision when only little supervision can be provided. However, DualSeededKMeans and DocumentSeededKMeans improve their performances quickly than FeatureSeededKMeans when more document seeds labeled. When there are enough document seeds labeled, both DualSeededKMeans and DocumentSeededKMeans performs better than FeatureSeededKMeans. Our explanation is that a few labeled features can represent the cluster structures better than a few documents, which also contains other non-discriminating features. Therefore, it is better to label features than documents if only limited user supervision is available.

- Learning curves of FeatureSeededKMeans are steep at the beginning but become flat quickly. Our explanation is that enough feature seeds are labeled after a few document seeds labeled at the beginning. The number of feature seeds labeled does not change much when more document seeds are labeled later.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we incorporate document supervision and feature supervision in the form of feature seeding. DualSeededKMeans is a unified framework to combine document supervision, feature supervision and unlabeled documents in the form of seeding. DocumentSeededKMeans and FeatureSeededKMeans are two specialized cases of DualSeededKMeans. Experimental results demonstrate that unlabeled documents can help to refine the information provided by the user and feature supervision is worth the effort to improve the clustering performance further compared to document supervision only and much more helpful when only few documents can be labeled due to manually cost.

The research presented in this paper is in the context of a document management system that support user-driven organization of document collections. Evaluation of the effectiveness of the system through user studies is available in [12].

6. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their insightful comments. This research was supported in part by the NSERC (Natural Sciences and Engineering Research Council) Business Intelligence Network, and by the MITACS NCE.

References


Yeming Hu is a PhD candidate at Faculty of Computer Science, Dalhousie University (Canada). He got his Master of Engineering from Dalhousie University in 2006 and his B.Sc. from Sun Yat-sen University (China) in 2004. His PhD thesis focuses on interactive document clustering for producing personalized clusters for users.

Evangelos E. Milios received a diploma in Electrical Engineering from the NTUA, Athens, Greece, and Master's and Ph.D. degrees from the Massachusetts Institute of Technology. Since July of 1998 he has been with the Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia. He served as Director of the Graduate Program (1999-2002) and he is currently Associate Dean, Research. He is a Senior Member of the IEEE. He was a member of the ACM Dissertation Award committee (1990-1992), a member of the AAAI/SIGART Doctoral Consortium Committee (1997-2001) and he is co-editor-in-chief of Computational Intelligence. At Dalhousie, he held a Killam Chair of Computer Science. He has published on the interpretation of visual and range signals for landmark-based navigation and map construction in robotics. He currently works on modelling and mining of content and link structure of Networked Information Spaces.

Dr. James Blustein is an Associate professor in both Dalhousie's Faculty of Computer Science and School of Information Management. His overall goal is to help people find and use information more effectively. His doctorate in Computer Science was co-supervised by the late Jean Tague-Sutcliffe.
ABSTRACT
The use of dictionaries is a common practice among those applications performing on huge RDF datasets. It allows long terms occurring in the RDF triples to be replaced by short IDs which reference them. This decision greatly compacts the dataset and mitigates the scalability issues underlying its management. However, the dictionary size is not negligible and the techniques used for its representation also suffer from scalability limitations. This paper focuses on this scenario by adapting compression techniques for string dictionaries to the case of RDF. We propose a novel technique: $\mathcal{D}_{\text{comp}}$, which can be tuned to represent the dictionary in compressed space ($22 - 64\%$) and to perform basic lookup operations in a few microseconds ($1 - 50\mu s$). In addition, we propose $\mathcal{D}_{\text{comp}}$ as a basis for specific SPARQL query optimizations leveraging its ability for early FILTER resolution$^1$.

Categories and Subject Descriptors
E.4 [Coding and Information Theory]; H.3.1 [Information Storage and Retrieval]: Dictionaries

General Terms
Algorithms, Design, Performance

Keywords
RDF Dictionaries, Scalability, Compression, SPARQL

1. INTRODUCTION
Nowadays, the so-called Web of Data materializes the basic principles of the Semantic Web [9]. It interconnects datasets from diverse fields of knowledge within a cloud of data-to-data hyperlinks which enables a ubiquitous and seamless data integration to the lowest level of granularity. As the Web of Data grows in popularity, the number (and scale) of semantic applications in use increases, more data are linked together and larger datasets are increasingly obtained. Performance and scalability arise as major issues in this scenario and their resolution is closely related to the efficient storage and retrieval of semantic data. Both issues must be analyzed under the World Wide Web Consortium (W3C) Recommendations of the RDF[2] (Resource Description Framework) data model for conceptual description and SPARQL[3] querying language.

RDF provides a graph-based model for structuring and linking data which describes facts of the world [10]. It is based on atomic triples comprising a subject ‘$s$’, (the resource being described), a predicate ‘$p$’, (the property), and an object ‘$o$’ (the property value). A set of RDF triples makes up an RDF graph in which the knowledge is represented through the different terms stored in nodes and edges. This term collection (called vocabulary) comprises elements drawn from three disjoint classes: Uniform Resource Identifiers (URIs), blank nodes ($bnodes$), and literals.

The vocabulary of terms is commonly indexed through a bijective function (called dictionary) which maps the strings representing the terms and the integer values (IDs) which identify them. Thus, a dictionary must provide at least two complementary operations: (i) the string-to-ID which returns the ID of a given string, and (ii) the ID-to-string which retrieves the string identified by a given ID. Both operations are exhaustively used by SPARQL engines during the query resolution process.

The use of dictionaries is a simple but effective decision for managing RDF, because all triples in the dataset can be rewritten by replacing the terms with their corresponding ID. Thus, the original dataset is now modeled through the dictionary and the resultant ID-triples representation. It allows high compression ratios to be achieved, and also enables great simplifications for the query processor, which can now perform on the ID-triples representation [20].

Despite of the undeniable contribution of dictionaries for improving scalability, their use is also compromised in the current scenario. As shown in Section 5.1, the space required by the dictionaries is even larger than that used for the resulting ID-triples representations, so managing their scalability is an open issue. Whereas much research work has been developed for compression and/or indexing of the ID-triples representations (see section 3), specific compressed dictionary representations are not covered in the previous literature to the best of our knowledge. Our main contributions in this scenario are:

- An introduction to the problem of effective representations of RDF dictionaries together with an empirical study characterizing their main features.

- A practical deployment of how compressed string dictionaries are used for representing RDF vocabularies.
A configurable technique (called $D_{comp}$) which achieves highly-compressed RDF dictionaries and very efficient performance for basic lookup operations.

A basic method leveraging $D_{comp}$ features for early FILTER resolution in SPARQL.

The paper is structured as follows. Section 2 shows different approaches for compressed string dictionaries, and Section 3 describes RDF dictionaries and reviews their state-of-the-art. Our approach for compressed RDF dictionaries (called $D_{comp}$) is explained in Section 4, providing specific details on lookup and filtering operations. Then, Section 5 shows an empirical study of RDF dictionaries and analyzes the $D_{comp}$ performance for five real-world datasets. Finally, Section 6 gives conclusions on the current results and devises future lines of work in this field.

2. COMPRESSED STRING DICTIONARIES

String dictionaries are the natural precedent of RDF dictionaries. A string dictionary $D = \{s_1, s_2, \ldots, s_n\}$ contains all different strings (vocabulary) representing the terms used in a dataset. Its basic management is reduced to the efficient resolution of two queries: locate($s_i$) which maps the string $s_i$ into its ID in $D$ (string-to-ID), and extract($i$) which returns the string $s_i$ identified as $i$ in $D$ (ID-to-string).

Classical approaches for string dictionaries, like hashing [13], use much space. It dissuades applications handling large vocabularies (for instance, those running in the Web of Data) from using it, because of the limited size of available memory. The use of B-tree [7] based solutions is the alternative, considering their optimization for large scale disk representations. However, their efficiency is compromised by the I/O costs derived from disk transfers. In this scenario, compression arises as the natural solution for increasing the amount of data which can be efficiently managed in memory.

Bender et al. [8] propose a cache-oblivious string B-tree performing Front-Coding [30] compression in the leaves. It is improved by the compressed permuterm [18], which gives efficient support for locate and extract and also resolves some substring-based operations in a compressed space. A more recent work, by Brisaboa et al. [11], revisits the problem from an eminently practical perspective. It proposes compressed variants of well-known solutions and introduces some novel ones. These are studied for emergent applications (for instance, they test a dictionary of URIs) and their results guarantee their interest in the RDF scenario.

We consider three types of techniques to give a more complete coverage of the current scenario: (§2.1) Hashing as representative of solutions traditionally used for managing string dictionaries; (§2.2) Front-Coding, because it excels for representing vocabularies in which long common prefixes are shared between many strings; and (§2.3) Self-indexes because they arise as a competitive choice for achieving compressed indexes of any kind of general string collection.

All these techniques are shown by following their original description [11]. Thus, we regard the dictionary as a text $T_{dict}$ which concatenates all strings ended by a special $\$ symbol (it is, in practice, the ASCII zero code).

2.1 Hashing

Hashing [13] is a natural choice for representing key-value structures like that required for a dictionary. It excels for locate, because the hash function is a natural way to transform a string into an ID, but it has no primitive mechanism to answer extract. Besides, hashing does not achieve compression by itself: i) it needs space for storing all $m$ strings in the dictionary, and ii) extra storage space is required for representing the hash table itself $H[1,n]$. The concept of load factor $m/n$ ($m < n$) is worth noting because it influences the space usage and the time performance.

We consider the technique named HashB-dh in [11] (we rename it as Hash) because it yields the best space/time trade-offs from among the hashing-based ones. It achieves compression through two basic decisions:

- It stores the hash table in a compact form by removing all the empty cells: $H'[1,m]$. A bitmap structure $B[1,n]$ is now required: $B[i] = 1$ if $H[i]$ is a non-empty cell and $B[i] = 0$ if $H[i]$ is empty.
- It compresses $T_{dict}$ with Huffman [22] and performs the hash function over the compressed strings.

Besides, this technique integrates a primitive resolution for extract. It sorts $T_{dict}$ to store the strings in the same order used in $H$. Thus, extract is answered by directly accessing the corresponding cell in the hash table.

2.2 Front-Coding

Front-Coding [30] is a technique traditionally used for compressing lexicographically sorted dictionaries. It leverages that consecutive strings are likely to share a common prefix and they can be differentially encoded with respect to their preceding string. This differential encoding concatenates, for each string, the integer representing the common prefix length and the substring which represents the remaining suffix.

Front-Coding partitions the sorted dictionary into buckets of $b$ strings to allow efficient searching: in each bucket, the first string is explicitly stored, whereas the other $b−1$ ones are differentially encoded according to the method previously explained. Given a string in the locate operation, a fast bucket location is first performed, comparing the first string of each bucket (e.g. through a binary search). Then, the location of the string within the correct bucket requires decoding the differences. It is easily implemented by performing a sequential bucket scan, which is also used in the extract operation. In this last case, the process locates the bucket representing the given ID: $[ID/b]$ and then decodes all strings until the required one. This decodification performance depends on the value of $b$, hence it allows different space/time tradeoffs to be yielded.

The Plain Front-Coding (PFC) technique [11] uses VByte [29] for encoding the differential representation of each string (both the prefix length and the remaining suffix). This decision allows all queries to be completed by exclusively running fast byte-wise operations. Focusing on spatial effectiveness, the Hu-Tucker Front-Coding (HTFC) compresses the byte-stream with Hu-Tucker [23] and performs all op-
erations over this compressed representation. This decision allows significative size reductions to be achieved at the price of slightly increasing querying times.

These techniques, performed on a lexicographic $T_{dct}$ ordering, are especially suitable for vocabularies containing strings with long common prefixes, such as the case of the URIs contained in RDF datasets.

2.3 Self-Indexing

Self-indexes [25] take advantage of the compressibility of a text to represent it in a structure that uses space closer to the compressed text, providing search functionality and containing enough information to reproduce any text substring. Thus, a self-index can replace the text. The FM-Index [17] is a self-index modeling the text on the Burrows-Wheeler Transform (BWT) [12]. This transformation is the core of the well-known compressor bzip2, so it gives a notion of the FM-Index effectiveness for compressing general texts.

An FM-Index (FMI) based approach is proposed in [11] for compressing string dictionaries. It also performs on a lexicographic $T_{dct}$ ordering and achieves effective compressed representations for all studied scenarios at the price of a less competitive performance for locate and extract.

In the current scenario, RDF literals are an example of general texts in which any kind of knowledge (natural language summaries, biological sequences, geographic information, among many other types) can be represented.

3. RDF DICTIONARIES

An RDF dictionary organizes all different terms used in a dataset. They come from three disjoint classes:

- **URIs** (U) identify resources in the WWW, so these are the identifiers used for data integration in the Web of Data. Many terms in this URI set share long prefixes.

- **Bnodes** (B) name anonymous nodes in the RDF graph and usually serve as parent nodes to a grouping of data. Naming of bnodes can matter in some treatments. Thus, canonical representations of RDF are due to the structure of bnodes which are in general tricky to achieve. For our purpose, we consider the bnode naming convention of N3.

- **Literals** (L) can be considered as “end nodes” in RDF graphs because they exclusively play the object role. Although literals can be tagged with an optional language or datatype, no general features can be assumed about their content. It is strongly related to the knowledge represented in the dataset.

The RDF model does not allow the subject to be a literal and the predicate must be an URI. All classes can take the role of an object. Thus, $(s, p, o) \in (U \cup B) \times (U \cup B \cup L)$.

**SPARQL and RDF Dictionaries.** The Web of Data popularity is the basis for the development of RDF management systems (RDF stores) that provide efficient storage and lookup infrastructure. As previously explained, dictionaries are used for compression purposes, but their representation is also an issue for querying: SPARQL engines make use of dictionary indexes, in conjunction with evaluation and histogram indexes, for physical optimization [20].

SPARQL resolution and RDF dictionaries are clearly related. SPARQL considers triple patterns (i.e. RDF triples $(s, p, o)$ in which $s$, $p$, or $o$ may be a variable) as atomic queries for building more complex ones. Thus, the engine 1) locates the IDs associated to the terms provided in the triple patterns; 2) the transformed query is performed, and the resulting ID values are bound to the variables given in this query; and 3) the final result is obtained by extracting the terms associated to these resulting IDs. This basic process implies that locate and extract are unevenly used. Whereas extract is used many times as results are returned for each variable in the query, the use of locate is limited to the number of terms bounded in the query. Thus, extract is overused in comparison to locate and should be optimized.

**State-of-the-art.** Many real-world RDF stores, such as the C-store based on [5], Hexastore [28], or RDF-3X [26] among others, maintain dictionary indexes. However, none implements optimized dictionary solutions. All of them use two independent structures, giving ineffective solutions which double the space required for representing the dictionary. B+-tree disk-oriented based solutions map terms to IDs for efficient locate resolution. In some cases, Front-Coding compression is performed on the leaves. For extract, structures supporting constant-time direct access (arrays or memory-mapped files) are used for mapping IDs to terms.

Dictionaries are also a core element for RDF exchanging. The W3C Member Submission HDT[4] is an RDF data-mapped files) are used for mapping IDs to terms. State-of-the-art. Many real-world RDF stores, such as the C-store based on [5], Hexastore [28], or RDF-3X [26] among others, maintain dictionary indexes. However, none implements optimized dictionary solutions. All of them use two independent structures, giving ineffective solutions which double the space required for representing the dictionary. B+-tree disk-oriented based solutions map terms to IDs for efficient locate resolution. In some cases, Front-Coding compression is performed on the leaves. For extract, structures supporting constant-time direct access (arrays or memory-mapped files) are used for mapping IDs to terms.

Dictionaries are also a core element for RDF exchanging. The W3C Member Submission HDT[4] is an RDF data-mapped files) are used for mapping IDs to terms. State-of-the-art. Many real-world RDF stores, such as the C-store based on [5], Hexastore [28], or RDF-3X [26] among others, maintain dictionary indexes. However, none implements optimized dictionary solutions. All of them use two independent structures, giving ineffective solutions which double the space required for representing the dictionary. B+-tree disk-oriented based solutions map terms to IDs for efficient locate resolution. In some cases, Front-Coding compression is performed on the leaves. For extract, structures supporting constant-time direct access (arrays or memory-mapped files) are used for mapping IDs to terms.

Some other specific applications within the Web of Data demand efficient RDF dictionaries. We emphasize its use in reasoning applications. In this scenario, Hogan [21] claims that a dictionary of URIs requires a prohibitive amount of memory to be stored and its compression would help to increase the in-memory capacity. It is very relevant for our objectives to consider that in-memory representations can handle a higher degree of reasoning.

4. OUR APPROACH: $D_{COMP}$

This section describes our approach (referred to as $D_{comp}$) for representing and querying compressed RDF dictionaries. First, we set up its structural organization (§4.1) as the basis for our compression purposes. We describe, in (§4.2), the locate and extract operations over the proposed organization. Finally, we show that the $D_{comp}$ organization also enables some kinds of SPARQL filtering (§4.3).

**Running Example.** Our explanation is guided by a running example which illustrates how the vocabulary from an RDF excerpt is modeled using $D_{comp}$. As shown, it comprises 11 triples stating that the “Symposium on Applied
Running example: RDF excerpt describing the “Symposium on Applied Computing” and “Riva del Garda”.

```
<http://dbpedia.org/resource/Riva_del_Garda> <http://www.myexample.org/ontology/city> "Riva del Garda is a town and comune in the northern Italian..." @en .
```

This role-based organization allows $D_{comp}$ to perform on three ID-range mapping terms in $[1, |S|] + |O|$ (subjects), $[1, |S|] + |O|$ (objects), and $[1, |P|]$ (predicates). Although a given ID can belong to different ranges, ambiguity issues cannot arise in extract because the general role (subject, object or predicate) is always known and can be provided together with the term ID.

Each partition is then subdivided by attending to the classes ($U, B, L$) that they can store. It allows the technique which best adjusts each class to be chosen in accordance to its features and its application requirements. That is, a specific subdictionary is used to represent the terms for each class within each partition. This implies that each subdictionary handles its specific mapping. Thus, each term is locally identified within its corresponding subdictionary.

Figure 2 illustrates the resulting organization. As can be seen, the partitions SO and S are split into URIs and bnodes. Moreover, O contains URIs and bnodes, but also an area for literals in which specific representations for general literals, lang literals (tagged with a specific language), and datatype literals (tagged with its specific datatype) are maintained. Tagged-literals are divided again to classify all different languages and datatypes used in the dataset. This hierarchy defines the ranges for a given language or datatype, allowing string tags to be removed in the final literal representation. Finally, the partition P only contains URIs.

Figure 3 shows the $D_{comp}$ organization for the RDF excerpt...
described in the running example. Note that bnode subdictionaries are empty in non-shared subjects and objects.

As stated, each subdictionary owns a local mapping and terms are locally identified within them. However, the RDF graph after ID replacement (that is, the ID-triples) must be unambiguously represented through global IDs. Thus, \( D_{\text{comp}} \) has to implement a mechanism for translating global and local IDs. This mechanism, referred to as \( ptrs \), is a very small multi-level mapping structure (shown in Figure 2 and in practice in Figure 3). Each cell in \( ptrs \) stores two elements: 1) a pointer to the corresponding class subdictionary, and 2) an integer value representing the number of terms previously organized in the corresponding role. That is, the \( i^{th} \) cell in the first level of \( ptrs \) stores the value \( ptrs[i] = ptrs[i-1] + t_{i-1} \), where \( t_{i-1} \) is the number of terms organized in the subdictionary \( i - 1 \). Some exceptions must be considered:

- \( ptrs[1] = 0 \) because \( D_{\text{comp}} \) forbids any term from being represented before the URIs in the partition 0.
- \( ptrs[8] = 0 \) because predicates are identified within their exclusive range of IDs.
- \( ptrs[5] = ptrs[3] = ptrs[2] + t_2 \) because both cells store the number of terms represented in the partition 0. For instance, in the running example (Figure 3), \( ptrs[5] = ptrs[3] = 2 \) because there are two previous terms in the 0 partition.

The second-level of \( ptrs \) is required for managing the literal subpartitions in 0. Three additional cells are used:

- The first cell points to the general literals subdictionary and stores the value \( ptrs[7,1] = ptrs[7] \).
- The second cell points to the lang-tagged literals representation and stores the value \( ptrs[7,2] = ptrs[7,1] + t_{7,1} \), where \( t_{7,1} \) is the number of general literals in \( D_{\text{comp}} \).
- Finally, the third cell points to the datatype-tagged literals representation and stores the value \( ptrs[7,3] = ptrs[7,2] + t_{7,2} \), where \( t_{7,2} \) is the number of lang-tagged literals in \( D_{\text{comp}} \).

In addition, \( ptrs \) stores two simple indexes for language-tagged literals, \( \text{lang} \), and another for datatype-tagged literals, \( \text{dtype} \). These indexes respectively point to the beginning of each language and datatype subdictionary. They store, respectively, the datatype and language keys allowing them to be deleted in each literal. This decision saves a lot of space because each different tag is represented once.

In the running example (Figure 3), \( \text{general literals} \) stores a single term, whereas there are three \( \text{language-tagged literals}: \) two English (en) and one Italian (it), and four \( \text{datatype-tagged literals}: \) one date, two floats and one integer. As can be seen, all these tags are indexed and represented once.

**Transforming local and global IDs.** \( Ptrs \) is the key structure for transforming local IDs into global IDs and vice versa. We introduce a simple notation to explain both translation operation. We refer to \( l_i \) the \( i^{th} \) local ID in the \( j^{th} \) subdictionary. For instance, \( \text{blank_node_edition_2012} \) is identified in the example through the local ID 1 in the 2\(^{th} \) subdictionary, and it is named as \( 1_2 \).

In the first operation, \( \text{local-to-global} \), a local ID \( l_j \) is transformed into its global counterpart as \( g(l_j) = l + ptrs[j] \).

For instance, \( \text{blank_node_edition_2012} \) is translated to its global ID as: \( g(1_2) = 1 + ptrs[2] = 1 + 1 = 2 \). An exception occurs for the subdictionary representing literals in the partition 0. In this case, the process is slightly different:

- If \( l_j \) belongs to the subdictionary of \( \text{general literals} \), the corresponding global ID is obtained as \( g(l_j) = l + ptrs[7,1] \).
- If \( l_j \) belongs to the subdictionary of \( \text{language-tagged literals} \), we first look for the language key in the index \( \text{lang} \) and retrieve the value stored for it: \( \text{lang}[x] \). The global ID is finally obtained as \( g(l_j) = l + \text{lang}[x] \).
- If \( l_j \) belongs to the subdictionary of \( \text{datatype-tagged literals} \), the translation is performed as in the previous case; the datatype key is found in its index: \( \text{dtype}[y] \), and the global ID is obtained as \( g(l_j) = l + \text{dtype}[y] \).

For instance, the term \( \text{Riva del Garda} \) is a town and commune in the northern Italian... is locally identified, in the running example, as \( 2_2 \) in the first division (en) of \( \text{lang-tagged literals} \). In this case, the corresponding global ID is obtained as \( g(2_2) = 2 + \text{lang}[1] = 2 + 4 = 6 \).

The opposite transformation: \( \text{global-to-local} \), is also implemented over \( ptrs \). Given a global ID \( i \), the first step determines the \( j^{th} \) subdictionary in which \( i \) is represented. The value of \( j \) is obtained through the condition: \( ptrs[j] < i \leq ptrs[j+1] \). Once \( j \) is known, the local value is obtained as \( l(i) = i - ptrs[j] \) for the general case. An exception occurs for \( j = 7 \). In this case, the global ID refers to a literal term and its corresponding subdictionary must be determined in the second level of \( ptrs \) through the condition: \( ptrs[7,k] < i \leq ptrs[7,k+1] \). Three subcases arise in this situation:

- If \( k = 1 \), the term is a \( \text{general literal} \): its local ID is obtained as \( l(i) = i - ptrs[7,1] \).
- If \( k = 2 \), the term is a \( \text{language-tagged literal} \) and we must find its language key in the corresponding index. In this case, the key \( x \) is determined as \( \text{lang}[x] < i \leq \text{lang}[x+1] \), and the local ID is obtained as \( l(i) = i - \text{lang}[x] \).
- If \( k = 3 \), the term is a \( \text{datatype-tagged literal} \) and we must determine its datatype key as in the previous case. The local ID is finally obtained as \( l(i) = i - \text{dtype}[y] \).

Let us consider the global ID \( i = 6 \). Comparing the values in the first level of \( ptrs \), we verify that it is represented in the \( 7^{th} \) subdictionary, because 6 is still greater than \( ptrs[7] = 3 \). Then, the global ID 6 is a literal term playing as object.
We now determine the corresponding class of literal in the second-level of \textit{ptrs} and find that it is a language-tagged literal because \( \text{ptrs}[7, 2] = 4 < 6 \leq 7 = \text{ptrs}[7, 3] \). The language is finally looked for in the index \textit{lang}. It is an English-tagged literal because the ID is represented in the subdictionary related to the first language key: \( \text{lang}[1] = 4 < 6 \leq 6 = \text{lang}[2] \). Finally, the local ID is obtained as \( l(6) = 6 - \text{lang}[1] = 6 - 4 = 2 \), which corresponds to the English literal term \textit{Riva del Garda is a town and comune in the northern Italian...}.

\textbf{Ptrs Implementation.} \textit{Ptrs} is implemented using basic data structures, and its size is negligible for real-world RDF dictionaries. On the one hand, the first two levels of \textit{ptrs} are stored through an array of 11 cells: 8 for the first level, and 3 for the second one. On the other hand, the number of different languages and datatypes modeled in an RDF dictionary depends on the dataset features. However, this number is very small in practice (only several tens, in the worst case), and these indexes can be efficiently implemented through two lexicographically sorted arrays which enable efficient searches for key and global ID.

Finally, it is worth noting that any markup symbol is explicitly stored in the corresponding subdictionary. As can be seen, symbols \(<\) and \(>\) enclosing URIs, \_: prefixing bnodes, and quotes delimiting literals are not stored in \( D_{\text{comp}} \) because all them can be deducted according to the class of terms represented in each subdictionary. It also allows small spatial savings to be achieved.

4.2 Basic Lookup Operations

The minimal functionality required for an RDF dictionary is reduced to the efficient implementation of two basic lookup operations: \textit{locate} and \textit{extract}. We detail below how \( D_{\text{comp}} \) provides this functionality within a generic SPARQL querying process. We consider that a query \( Q \) comprises one or more triple patterns and each one contains \textit{terms} drawn from URI, bnode and literal classes: \( C = \{U, B, L\} \), and \textit{variables}. Both terms and variables can play the roles of subject, predicate or object: \( R = \{S, P, O\} \). We define \( \mathcal{T} \) and \( \mathcal{V} \) as the sets which, respectively, contain the terms and variables in \( Q \).

A SPARQL processor firstly parses \( Q \) to obtain the corresponding sets \( \mathcal{T} \) and \( \mathcal{V} \). Let us suppose that we ask for the descriptive comment of all Italian towns in our running example. This query is expressed as follows:

```
SELECT ?town ?comment
WHERE
{
}
```

Therefore, the SPARQL processor obtains the set of terms: \( \mathcal{T} = \{\text{http://dbpedia.org/ontology/country}, \text{http://dbpedia.org/resource/Italy}, \text{http://www.w3.org/2000/01/rdf-schema#comment}\} \), and the variables one: \( \mathcal{V} = \{\text{?town}, \text{?comment}\} \). The next step consists of locating the ID corresponding to each term \( t_i \in \mathcal{T} \). It requires many \textit{locate} lookups as terms in the set \( \mathcal{T} \). Once all terms are transformed into their corresponding IDs, the SPARQL query is rewritten, and it is run over the ID-triples representation. The query resolution outputs a series of ID values matching for the variables in \( \mathcal{V} \). Thus, the last step performs many \textit{extract} operations as results are obtained, for each variable in \( \mathcal{V} \), and the corresponding terms are reported within the final query result. In the previous example query, this step extracts the terms binded to the ID-results for the variables \textit{?town} and \textit{?comment}.

We detail below how the location and extraction processes are implemented in \( D_{\text{comp}} \) and illustrate them using the example query above.

\textbf{Locate.} This operation implements the translation \texttt{string-to-id}. Thus, it requires the term \( t_i \in \mathcal{T} \) as key for accessing to \( D_{\text{comp}} \). However, it uses two additional properties about
Let us analyze how the terms in our query are located:

- \( t_1 = \langle \text{http://dbpedia.org/ontology/country} \rangle \) is an URI playing the predicate role in the first triple pattern, so the locate operation is invoked with the term \( t_1 \), and the parameters \( r_1 = P \) and \( c_1 = U \). The subdictionary of predicates is queried and the global ID 1 is returned.

- \( t_2 = \langle \text{http://dbpedia.org/resource/Italy} \rangle \) is an URI which plays the object role in the first triple pattern, so the locate operation is invoked with the term \( t_2 \), and the parameters \( r_2 = O \) and \( c_2 = U \). The term is firstly searched in the subdictionary of URI within the partition SO, but it is not found. The locate operation is repeated in the URIs subdictionary in \( D \), obtaining the local ID 1. It is transformed into the corresponding global ID: \( 1 + \text{ptrs}[5] = 3 \) which is used for replacement in the original SPARQL query.

- Finally, \( t_3 = \langle \text{http://www.w3.org/2000/01/rdf-schema#comment} \rangle \) is also an URI playing as predicate role in the second triple pattern, so the locate operation is invoked with the term \( t_3 \), and the parameters \( r_3 = P \) and \( c_4 = U \). The subdictionary of predicates is queried again and the global ID 7 is returned.

After these operations, the query is rewritten as: \( \text{SELECT ?town ?comment WHERE ( (?town 1 3) ( ?town 7 ?comment) )} \), and this is the representation provided for the SPARQL engine. The expected result is the URI of Riva del Garda for the town and the literals, in English and Italian, representing comments about them. These results are returned as follows: \( \langle \text{http://dbpedia.org/resource/Riva_del_Garda} \rangle \) with local ID 1, so the literal is extracted from this subdictionary: \( \text{extract(1)} \). Therefore, the final query result comprises the English and Italian comments about Riva del Garda:

- \( \langle \text{http://dbpedia.org/resource/Riva_del_Garda} \rangle \), “Riva del Garda is a town and comune in the northern Italian…”@en.

- \( \langle \text{http://dbpedia.org/resource/Riva_del_Garda} \rangle \), “Riva del Garda è un comune della provincia di Trento…”@it.

4.3 Filter Resolution

A generic RDF dictionary provides locate and extract resolution, because it holds the mappings between terms and IDs. However, our dictionary organization enables more advanced SPARQL functionality. This is the case of the FILTER clause, which restricts solutions to those for which the filter expression evaluates to true [3].

In the following, we show how unary SPARQL filters can be directly resolved over the proposed dictionary. Two types of SPARQL filtering are distinguished:

- **Tests** are used for checking if a query result is drawn from a given term class. Thus, three different tests are available for filtering: isIRI, isBlank, and isLiteral.

- **Accessors** use specific internal term information for filtering. Three different accessors are distinguished:

  - **str**: returns the lexical form of a given term. In practice, this accessor is used for retrieving the string version of the argument passed to it [14].

  - **lang**: returns the language tag of a given literal, if it has one. In other case, it returns an empty string.

  - **datatype**: returns the datatype tag of a given literal. If it is a simple (general) literal, or it is tagged with any language information, datatype returns the string tag (xsd:string).

In both cases, \( D_{\text{comp}} \) provides direct filter resolution over the dictionary, allowing the query processor to push-up fil-
ter evaluation. This means, in general, to reduce the number of triples to be explored in the query and thereby to improve the overall query performance when \( D_{comp} \) is used in conjunction with any RDF store in the state-of-the-art. These improvements are highly interesting in real-world scenarios by considering that roughly the 50% of the queries perform any kind of filtering [6].

**SPARQL tests.** As explained above, these filters rely on checking the term class. We illustrate their implementation by restricting that all bindings for the variable `town` must be URIs:

```sparql
SELECT ?town ?comment WHERE {
  FILTER (lang(?comment) = "en")
}
```

The traditional non-early test evaluation (also available in \( D_{comp} \)) runs the SPARQL query by matching the triple patterns against all triples in the dataset. Then, the result set must be checked, one-on-one, with respect to the filter condition, obtaining the final resultant bindings. Note that, thanks to the organization of \( D_{comp} \), this evaluation of the condition can be performed directly on the IDs, i.e., without the need of extraction of the literal mapped to each ID. This is due to the fact that each partition (subdictionary) only holds a type of term; URIs, Blank nodes or Literals. For instance, in the example query, the resolution of the triple patterns binds the variable `?town` to the ID 1, and we then verify that 1 is in an URI partition by accessing `ptrs`. In this case, `ptrs[1] = 0 < 1 ≤ 1 = ptrs[2]`, thus the ID 1 is in the first subdictionary: URIs playing as SO.

In contrast, early test evaluation is resolved in a more efficient way because it leverages the information recorded in the `ptrs` structure for reducing the space of triples to be explored by the query engine. This evaluation algorithm retrieves, from the first-level of `ptrs`, the ranges of contiguous IDs associated to a given class term, and provokes them to the SPARQL engine. This decision allows the engine to only explore the triples represented in this space of possible results, directly discarding all the triples out of these ranges because they do not match the filter condition. In the example query, the filtered variable: `?town`, plays as subject and the filter condition restricts its bindings to URIs. Thus, the space of possible results is first limited to the triples whose subject is identified within the ranges \([1,1]\) and \([3,3]\), because these are the ranges assigned to the URIs playing as SO and S in our running example. These ranges are provided to the engine which only search for matching results in them, and return the ID 1 as a valid result.

**SPARQL accessors.** These filters extract specific information about the terms. We reformulate the original query to only retrieve comments expressed in English about Riva del Garda:

```sparql
SELECT ?town ?comment WHERE {
  FILTER (lang(?comment) = "en")
}
```

In the traditional non-early evaluation method, again, the query is first run over the full dataset and the result set must be individually checked. However, not all accessors can be resolved without term extraction:

- **str** needs checking the lexical value of each returned ID against the string provided in the filter. Thus, the ID is firstly extracted and then compared with respect to the string in the filter.

- **lang** and **datatype** are resolved without extraction because each ID can be directly compared against the range assigned to the corresponding language or datatype. In this case, the resolution requires querying the second level of `ptrs` and the indexes `lang` and `dtype`. In the current query, comments are restricted to those expressed in English, so they must identified in the range \([5,6]\) assigned to this language. Considering that two bindings are returned: \([6,7]\), only the first one is a result because 7 is not in the valid range.

The early evaluation algorithm proceeds as in the test case. That is, the ranges of possible results are firstly obtained and the query is exclusively performed over them. This way, the set of returned results is already filtered. In our example query, we firstly access to the `lang` index and retrieves the range \([5,6]\) assigned to the English-tagged literals. This range is provided to the engine which only searches possible results among those triples containing an object ID in this range. In this case, the returned result contains the value 6, because it is the only binding found in the valid range.

It is worth noting that early resolution of other filters, like `regex` or those based on arithmetic operations, remains still opened. On the one hand, `regex` could be addressed leveraging specific features for substring resolution [11]. On the other hand, the case of arithmetic filters seems more difficult by considering that \( D_{comp} \) relies on techniques optimized for string dictionaries, and these operations demand efficient management of numerical values. These topic has been recently studied [15], and the solutions proposed for compact indexing of real numbers are an interesting choice for addressing this issue.

### 5. Experimental Setup

This section studies the problem of the compression of RDF dictionaries on a real-world setup. To do this, we choose five real-world datasets to achieve a heterogeneous setup containing data from different application domains. The amount of triples in each dataset is also considered in our choice. Three different datasets are extracted from the Billion Triples Challenge 2010\(^2\); in particular, `geonames` gathers geographic concepts, `dbtune` contains biological information mainly focused on proteins; `wikipedia\(^3\)` stands for the English Wikipedia links between pages transformed to RDF, and `dbpedia\(^4\)` is a community effort with the aim of making this type of information semantically available on the Web.

\(^2\)http://km.mibf.kit.edu/projects/btc-2010/

\(^3\)http://labs.systemone.at

\(^4\)http://wiki.dbpedia.org/Downloads36
We firstly characterize the RDF dictionaries extracted from these datasets (§5.1). We focus on the impact of the dictionary size within a dataset and also study their more relevant statistics to the current problem. Then (§5.2), we test compressed string dictionaries in the RDF scenario and extract conclusions for $D_{\text{comp}}$. It is studied in (§5.3) through two functional configurations: $D_{\text{comp}}^C$ is focused on compression effectiveness and $D_{\text{comp}}^Q$ is optimized for querying.

All querying tests are performed on a computer using an Intel Core2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running Linux kernel 2.6:24-28. User times are reported for all experiments. Prototypes are developed in C++ using structures from libcds [1] Plain (referred to as RG [19]) and compressed (referred to as RRR [27]) bitmaps are tested in our experiments; they can be parameterized with a sampling value. All sources are compiled on g++ 4.2.4 with options -O9 and -m64.

### 5.1 Characterizing RDF Dictionaries

The dictionary size is the issue addressed in this paper under the consideration that it is a significant fraction of the dataset size. This assumption can be confirmed by transforming the original dataset into an equivalent raw representation of dictionary and ID-triples: the dictionary concatenates all terms ended by a separator symbol and the ID-triples representation replaces the terms by their corresponding ID and performs a na"{i}ve encoding which uses log$(n)$ bits/element ($n$ is the number of total subjects, predicates or objects respectively). We assume that dictionary and ID-triples representations use $c_d$ and $c_t$ bytes respectively, so the dataset size is $C = c_d + c_t$ bytes.

Table 1 (left) shows this comparison. Columns $O$ and $C$, respectively, contain the size (in GB) of the original N3 dataset and that obtained through the dictionary-based representation. As can be seen, the use of dictionaries allows for large compression (e.g. for wikipedia, the dictionary-based representation is 11.80 times smaller than the original). The next two columns measure the impact of the dictionary size in this representation (values in parentheses correspond to $c_d/C$ and $c_t/C$ respectively). As can be seen, $c_d$ is always larger than $c_t$ (up to $\approx 3$ times for geonames or dbpedia).

Thus, the dictionary always takes more space than the most-na"{i}ve ID-based representation. This fact supports the need for effective dictionary representations that can be used in conjunction with more-advanced techniques for ID-triples.

Table 1 (right) describes the studied datasets. The columns triples and elements, respectively, show the amount of triples in the dataset and the number of terms contained in the vocabulary. As can be seen, this vocabulary grows with the dataset size, but the proportion depends on the design and purpose of the dataset. The next columns contain the distribution of terms playing different roles in the dataset. Again, their proportion depends on the dataset features but two significative conclusions can be extracted: i) the SO partition allows terms playing roles of subject and object to be represented a single time. As can be seen, this means a significative improvement for all datasets (except for geonames), saving up to 60.74% of the terms for dbtune; and ii) the proportion of predicates is always a less significative fraction of the dictionary.

### 5.2 Analyzing Compressed String Dictionaries for RDF

This evaluation is carried out on the dictionaries obtained from the datasets studied above. We analyze space/time tradeoffs for each technique and for URI, Bnode, and Literal dictionaries. Plain (referred to as RG [19]) and compressed (referred to as RRR [27]) bitmaps are studied in these tests.

The Hash technique reserves a table with an overhead of 10% ($n = 1.1 * m$) and compacts it by using a bitmap RG configured with sampling 20. Tests performed on other load factors report comparable results. PFC and HTFC techniques are configured on different bucket sizes: $b = 2^s$, for all $x \in [1, 10]$. Thus, we obtain results for buckets containing from $2^1$ to $2^{10}$ terms. Finally, the FMI technique is implemented by using plain (FMI-RG) and compressed (FMI-RRR) bitmaps. FMI-RG is parameterized with sampling values $s = \{4, 20, 40\}$, and FMI-RRR with $s = \{16, 64, 128\}$.

**Compression.** Table 2 summarizes the compression results achieved for the datasets. Compression ratios are calculated with $c_{\text{comp}}/c_d$, where $c_{\text{comp}}$ and $c_d$ are the compressed and the original dictionary sizes respectively. This table organizes the results for URIs, bnodes (only dbtune uses them), and literals. We give the best and the worst ratios for all parameterizable techniques. Note that, for the FMI technique, the FMI-RRR variant always obtains the most compressed representations (for $s = 128$) and FMI-RG the worst ones (for $s = 4$). The well-known compressor gzip is also considered as a reference of our compression achievements.

Results for URI vocabularies give a clear situation. On the one hand, Hash achieves a poor compression: of around 80% of the original raw size. This result is mainly due to Huffman code performs a character-based compression and it cannot take advantage of longer-range correlations existing between the terms in the vocabulary. This discourages its use for large URI vocabularies. On the other hand, HTFC obtains the best ratios for all datasets because the Front-Coding algorithm is able to detect the long common prefixes shared by

---

**Table 1: Size comparison (sizes are expressed in GB) and role-based configurations for all datasets.**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>dataset size</th>
<th>$C$</th>
<th>$O$</th>
<th>dictionary size: $c_d$</th>
<th>ID-triples size: $c_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>geonames</td>
<td>1.09</td>
<td>0.19</td>
<td>0.14</td>
<td>(73.66%)</td>
<td>0.05 (26.32%)</td>
</tr>
<tr>
<td>wikipedia</td>
<td>6.72</td>
<td>0.57</td>
<td>0.30</td>
<td>(52.63%)</td>
<td>0.27 (47.37%)</td>
</tr>
<tr>
<td>uniprot</td>
<td>9.11</td>
<td>1.21</td>
<td>0.75</td>
<td>(61.99%)</td>
<td>0.16 (38.01%)</td>
</tr>
<tr>
<td>dbtune</td>
<td>9.34</td>
<td>1.37</td>
<td>0.98</td>
<td>(71.53%)</td>
<td>0.39 (28.47%)</td>
</tr>
<tr>
<td>dbpedia</td>
<td>33.12</td>
<td>6.96</td>
<td>5.15</td>
<td>(73.99%)</td>
<td>1.81 (26.01%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>triples</th>
<th>elements</th>
<th>SO</th>
<th>S</th>
<th>O</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>9,415,253</td>
<td>5,141,366</td>
<td>1.83</td>
<td>41.03%</td>
<td>57.14%</td>
<td>0.39 x 10^{-3}%</td>
</tr>
<tr>
<td>47,054,407</td>
<td>8,869,064</td>
<td>17.61%</td>
<td>6.77%</td>
<td>75.62%</td>
<td>0.10 x 10^{-5}%</td>
</tr>
<tr>
<td>72,460,981</td>
<td>14,842,666</td>
<td>43.33%</td>
<td>38.79%</td>
<td>17.88%</td>
<td>0.65 x 10^{-5}%</td>
</tr>
<tr>
<td>58,926,361</td>
<td>16,589,644</td>
<td>60.74%</td>
<td>14.01%</td>
<td>25.24%</td>
<td>0.02 x 10^{-3}%</td>
</tr>
</tbody>
</table>
the terms. HTFC outperforms PFC thanks to the Hu-Tucker compression. Both maximize their effectiveness for increasing bucket sizes. As can be seen, the HTFC representations take between 20.08% (for dbtune) and 5.04% (for uniprot) of the raw size. In the latter case, it even surpasses the effectiveness of gzip. This is a very significant achievement because it demonstrates that these techniques can represent the vocabulary in a space close to that used by a universal compressor and also answer locate and extract. PFC and FMI are less effective in this scenario. This analysis can be extended to the Bnodes scenario.

A less clear situation arises for Literals. As can be seen, HTFC is the best choice for geonames and wikipedia, whereas FMI is the most effective for the other datasets. However, the effectiveness of FMI is the most uniform. Experiments show that FMI-RRR largely outperforms PFC and gzip, and larger sampling values improve compression in both cases. In turn, PFC and Hash obtain poor results for literals. This fits our initial expectations: literal vocabularies show less regularities than URIs or bnodes, and their compression is greatly complicated. The effectiveness of gzip verifies this fact: its better ratios are always greater than 20%, whereas this was the upper limit for URIs.

These results show that URIs and bnodes can be highly compressed and HTFC is the most effective choice. However, optimizations for literals are more complicated because they can contain any type of information, and prefix-based compression is not always sufficient. FMI-RRR arises as an interesting solution for literals, outperforming HTFC in some cases. The classic Front-Coding (PFC) achieves limited success for URI compression, whereas Hashing is clearly discouraged when compact representations are required.

**Querying.** We design specific micro-benchmarks for testing querying operations: i) locate is studied through a batch of 10,000 terms randomly chosen for each vocabulary, and ii) another batch containing 10,000 random IDs are used for extract. We run 50 independent executions of each batch and average total times to isolate our measurements of external events. These averaged times per batch are then

### Table 2: Comparison of general techniques for string dictionaries (s, is the dictionary raw size in MB).

<table>
<thead>
<tr>
<th>URIs</th>
<th>s, (MB)</th>
<th>gzip</th>
<th>Hash</th>
<th>PFC</th>
<th>HTFC</th>
<th>FMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>geonames</td>
<td>102.77</td>
<td>9.74</td>
<td>82.23</td>
<td>19.17</td>
<td>64.10</td>
<td>11.87</td>
</tr>
<tr>
<td>wikipedia</td>
<td>212.65</td>
<td>13.62</td>
<td>77.68</td>
<td>22.18</td>
<td>64.62</td>
<td>15.37</td>
</tr>
<tr>
<td>uniprot</td>
<td>520.90</td>
<td>7.20</td>
<td>80.94</td>
<td>8.71</td>
<td>58.40</td>
<td>5.04</td>
</tr>
<tr>
<td>dbtune</td>
<td>281.54</td>
<td>19.47</td>
<td>78.44</td>
<td>30.88</td>
<td>68.30</td>
<td>20.08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bnodes</th>
<th>s, (MB)</th>
<th>gzip</th>
<th>Hash</th>
<th>PFC</th>
<th>HTFC</th>
<th>FMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbtune</td>
<td>623.73</td>
<td>9.09</td>
<td>71.14</td>
<td>22.83</td>
<td>64.08</td>
<td>14.50</td>
</tr>
</tbody>
</table>

![Figure 4: locate and extract times for URIs (top) and literals (bottom) from dbpedia.](image-url)
divided by the number of queries (10,000) to obtain the times per query finally reported in the graphics below.

Figure 4 compares locate (left) and extract (right) performance for the URI (upper) and literal (bottom) vocabularies from dbpedia. Each graphic draws compression ratios on the X axis and querying times (in µs/query) on the Y axis (logscale). All the conclusions below can be extended to the other datasets in the current setup, but their graphics are not shown due to lack of space.

All graphics share a general result: the space/time tradeoffs for Hash are never the best choice, neither for compression nor at querying times. The results reported for URIs are very clear: PFC always outperforms HTFC in querying because the latter pays the price of Hu-Tucker decompression. However, as commented above, PFC pays a spatial overhead with respect to HTFC. Its compression ratio is 8 percentage points better than that obtained by PFC, but its temporal improvement is less than 20 µs/query. Thus, HTFC is well-suited for scenarios focused on compression, but PFC is the better choice if spatial requirements are relaxed. Finally, FMI performance is not competitive for URIs.

The analysis for literals is quite complex. PFC achieves excellent times (5 – 10µs/query), but its space is up to 3 times larger than that used by the most effective technique: FMI-RRR. In turn, HTFC largely improves PFC compression, but querying times evolve to 10 – 60 µs/query for competitive tradeoffs. Finally, FMI takes between 200 and 300 µs per extraction and location respectively. Thus, FMI-RRR must be chosen for optimizing space and PFC may be the choice in scenarios where time prevails. However, FMI is still the only choice when more sophisticated queries (such as substring-based ones) are desired [11]. This accomplishes with the line of future work devised for filter resolution.

### 5.3 $D_{comp}$ Performance

As explained above, two functional configurations for $D_{comp}$ are studied. We choose parameters optimizing the desired dimension within a competitive space/time tradeoff:

- $D_{comp}^{(C)}$ is optimized for compression. It implements URI and bnode dictionaries on HTFC ($b = 16$), and represents literals by using FMI-RRR with sampling 128.

- $D_{comp}^{(Q)}$ is optimized for querying. It implements URI and bnode dictionaries on PFC ($b = 8$), and represents literals by using FMI-RG with sampling 4.

Table 3 shows compression effectiveness for $D_{comp}$. We include the sizes of the dictionaries used in RDF-3X [26] to compare our results with respect to a real-world solution (note that we measure the space that $D_{comp}$ takes in memory, but RDF-3X size is measured on disk, so additional space is required to be loaded in memory). As can be seen, $D_{comp}^{(Q)}$ configuration takes approximately twice the space used by $D_{comp}^{(C)}$. The broad difference existing between them allows for some other configurations whose size can be tuned in accordance to specific application requirements. The comparison of our two variants with respect to RDF-3X gives a magnitude of our achievements with regard to the representation of RDF dictionaries. Whereas RDF-3X always uses more space than the original raw dictionary (remember that it combines two data structures for the dictionary), our worst $D_{comp}^{(Q)}$ result (for dbpedia) uses 64.11% of the original space, while the best one for $D_{comp}^{(C)}$ is only 30.32%. Thus, $D_{comp}$ reduces the space taken by RDF-3X between 2 and 7 times for the studied datasets.

These results guarantee that $D_{comp}$ can be tuned to achieve highly-compressed dictionaries. This saves processing resources and enables larger size dictionaries to be managed in a fixed main memory, but also achieves very efficient querying performance. It is studied through a heterogeneous set of real-world dbpedia queries from the log of the USEWOD’2011 Challenge\(^5\). We designed a batch of 10,000 queries chosen at random, and executed it in 50 independent repetitions. Times reported are averaged by following the procedure used in the previous experiments.

Figure 5 shows locate and extract times. As can be seen, $D_{comp}^{(Q)}$ always outperforms $D_{comp}^{(C)}$. It is worth noting that times obtained by our two variants are always less than 10µs per query except for literals. In this case, the use of a more general representation (like FMI) slightly reduces the performance achievable through the other techniques. Note that extract is faster than locate in all cases. Thus, a better performance is achieved for the most used operation in SPARQL engines. The RDF-3X performance is also analyzed. We run the query batch and measure the time that it uses for locate and extract in two different scenarios: “cold” (no data is preloaded in the system main memory) and “warm” (the required data are available in the main memory). The comparison is unfair in the cold scenario because RDF-3X needs data to be transferred from disk; these operations are performed in some milliseconds (one order of magnitude above our technique). As can be seen in figure 5, the test in the warm scenario reduces the times to the level of microseconds, but it never improves our approaches for locate, and only surpasses $D_{comp}^{(Q)}$ for extract (literals). However, RDF3X is unable to handle tagged literals, whereas our approaches give specific support for them.

### 6. CONCLUSIONS AND FUTURE WORK

This paper addresses compressed representations for RDF dictionaries. We apply existing techniques for string dictionaries to the specific case of RDF and obtain simple compressed representations for URI, blank node and literal dictionaries. This experience is integrated within a novel compressed technique, called $D_{comp}$, which compressed the original dictionary up to 22–64% of its original size and answered queries in 1–50µs. These results represent an improvement\(^6\).


---

### Table 3: Compression results for RDF dictionaries.

<table>
<thead>
<tr>
<th>Dict.</th>
<th>$s_2$ (MB)</th>
<th>RDF-3X</th>
<th>$D_{comp}^{(Q)}$</th>
<th>$D_{comp}^{(C)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>geonames</td>
<td>135.07</td>
<td>195.47%</td>
<td>27.09%</td>
<td>50.94%</td>
</tr>
<tr>
<td>wikipedia</td>
<td>295.00</td>
<td>177.64%</td>
<td>29.37%</td>
<td>55.27%</td>
</tr>
<tr>
<td>uniprot</td>
<td>734.33</td>
<td>152.12%</td>
<td>27.61%</td>
<td>45.21%</td>
</tr>
<tr>
<td>dbtune</td>
<td>964.66</td>
<td>142.34%</td>
<td>21.99%</td>
<td>41.08%</td>
</tr>
<tr>
<td>dbpedia</td>
<td>5213.76</td>
<td>115.93%</td>
<td>30.32%</td>
<td>64.11%</td>
</tr>
</tbody>
</table>

---
on the state-of-the-art (studied through dictionaries modeled in RDF3X). \( D_{\text{comp}} \) i) uses between 2–7 times less space, and ii) answers queries in more efficient time for all cases except for literal extraction. However, \( D_{\text{comp}} \) gives advanced support for managing tagged literals. This experience has been evidenced in an innovative technique providing efficient SPARQL resolution in compressed space [24].

Our future work firstly focuses on integrating \( D_{\text{comp}} \) as a dictionary index within an existing SPARQL engine and test in both space and time improvements over the current solutions. Besides, the use of \( D_{\text{comp}} \) within a SPARQL engine provides interesting features for filtering. Our main line of future work is to optimize these features for early evaluation. We are also working to support more advanced operations on \( D_{\text{comp}} \). Prefix-based searches are easily implementable for PFC and HTFC, and general substring matching can be supported in FMI [11]. Note that this latter feature is essential for the expressive regex filtering. Achieving this goal can also influence SPARQL querying performance by integrating early resolution on physical optimization plans.

An additional line of future work focuses on evolving \( D_{\text{comp}} \) to support dynamic operations of insert, delete, and update. These are essential to integrate \( D_{\text{comp}} \) in semantic databases in which dictionaries evolve according to triples management.

Acknowledgments

This work was funded by the MICINN (Spain): TIN2009-14009-C02-02. The second author holds grants from Eras-

muse Mundus, the Regional Government of Castilla y León (Spain) and the European Social Fund. The third author is now a member of the NICTA which is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

We would especially like to thank Nieves R. Brisaboa, Francisco Claude and Gonzalo Navarro for their advice about compressed string dictionaries, and Claudio Gutierrez for his continued support and his magistral lessons about the Web of Data. We would also like to thank the Database Lab (Univ. of A Coruña, Spain) for lending us its servers for our experiments.

7. REFERENCES


ABOUT THE AUTHORS:

Miguel A. Martínez-Prieto is a Lecturer in the Department of Computer Science at the University of Valladolid, Spain. He completed his Ph.D in Computer Science from the same University in 2010. His research interests are mainly related to data compression and its application to the efficient representation, management, and querying of Big Data. He is co-author of HDT, the binary format acknowledged by the W3C for publication and exchange of huge RDF in the Web of Data.

Javier D. Fernández is a PhD candidate in the Departments of Computer Science at the University of Valladolid, Spain, and the University of Chile, Chile. He started his double diploma thanks to an Erasmus Mundus grant in 2010. He is co-author of the HDT W3C Member Submission and his main interests include scalable representations and indexes for querying the Web of Data, data compression and efficient management of Big (semantic) Data.

Rodrigo Cánovas is a PhD Candidate in the Department of Computing and Information System at The University of Melbourne in Victoria, Australia. He completed his BS and MS degrees in 2007 and 2010, respectively, at the Department of Computer Science at the University of Chile in Santiago, Chile. His research interests include design and analysis of algorithms, data structures, data compression, and bioinformatics.