Journal Subline

LNCS 8320

Transactions on Large-Scale Data- and Knowledge-Centered Systems XII

Abdelkader Hameurlain • Josef Küng • Roland Wagner
Editors-in-Chief

Lecture Notes in Computer Science 8320

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Abdelkader Hameurlain Josef Küng
Roland Wagner (Eds.)

Transactions on Large-Scale Data- and Knowledge-Centered Systems XII

Springer

Editors-in-Chief

Abdelkader Hameurlain
Paul Sabatier University, IRIT, Toulouse, France
E-mail: hameur@irit.fr

Josef Küng
Roland Wagner
University of Linz, FAW, Austria
E-mail: {jkueng, rrwagner}@faw.at

ISSN 0302-9743 (LNCS) e-ISSN 1611-3349 (LNCS)
ISSN 1869-1994 (TLDKS)
ISBN 978-3-642-45314-4 e-ISBN 978-3-642-45315-1
DOI 10.1007/978-3-642-45315-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013955032

CR Subject Classification (1998): H.2.8, H.2, I.2, H.3, F.2, J.1

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The LNCS journal TLDKS (Transactions on Large-Scale Data- and Knowledge-Centered Systems) is now an established journal on data management in large-scale environments. These environments are mainly characterized by high heterogeneity (in terms of models, infrastructures, and software) and large-scale distributed resources (i.e., computing resources and data sources).

This volume contains five revised selected regular papers. Its content covers a wide range of different and hot topics in the field of data and knowledge management, mainly: schema matching and schema mapping, update propagation in decision support systems, routing methods in peer-to-peer systems, distributed stream analytics, and dynamic data partitioning.

We would like to express our thanks to the external reviewers and Editorial Board for thoroughly refereeing the submitted papers and ensuring the high quality of this volume. Special thanks go to Gabriela Wagner for her availability and her valuable work in the realization of this TLDKS volume.

October 2013

Abdelkader Hameurlain
Josef Küng
Roland Wagner

Editorial Board

Reza Akbarinia	INRIA, France
Stéphane Bressan	National University of Singapore, Singapore
Francesco Buccafurri	Università Mediterranea di Reggio Calabria, Italy
Yuhan Cai	A9.com, USA
Qiming Chen	HP-Lab, USA
Tommaso Di Noia	Politecnico di Bari, Italy
Dirk Draheim	University of Innsbruck, Austria
Johann Eder	Alpen Adria University Klagenfurt, Austria
Stefan Fenz	Vienna University of Technology, Austria
Georg Gottlob	Oxford University, UK
Anastasios Gounaris	Aristotle University of Thessaloniki, Greece
Theo Härder	Technical University of Kaiserslautern, Germany
Dieter Kranzlmüller	Ludwig-Maximilians-Universität München, Germany
Philippe Lamarre	University of Nantes, France
Lenka Lhotská	Technical University of Prague, Czech Republic
Vladimir Marik	Technical University of Prague, Czech Republic
Mukesh Mohania	IBM India, India
Tetsuya Murai	Hokkaido University, Japan
Gultekin Ozsoyoglu	Case Western Reserve University, USA
Torben Bach Pedersen	Aalborg University, Denmark
Günther Pernul	University of Regensburg, Germany
Klaus-Dieter Schewe	University of Linz, Austria
Makoto Takizawa	Seikei University Tokyo, Japan
David Taniar	Monash University, Australia
A Min Tjoa	Vienna University of Technology, Austria
Chao Wang	Oak Ridge National Laboratory, USA

External Reviewers

Roberto De Virgilio	Roma Tre University, Rome, Italy
Philippe Joly	Paul Sabatier University, Toulouse, France
Franck Morvan	Paul Sabatier University, Toulouse, France
Marina Mongiello	Politecnico di Bari, Italy
Shaoi Yin	Paul Sabatier University, Toulouse, France

Table of Contents

EvoMatch: An Evolutionary Algorithm for Inferring Schematic Correspondences 1
Chenjuan Guo, Cornelia Hedeler, Norman W. Paton, and Alvaro A.A. Fernandes

Update Management in Decision Support Systems 27
Haitang Feng, Nicolas Lumineau, Mohand-Saïd Hacid, and Richard Domps

LRS: A Novel Learning Routing Scheme for Query Routing on Unstructured P2P Systems 54
Taoufik Yeferny, Khedija Arour, and Amel Bouzeghoub

Open Streaming Operation Patterns 83
Qiming Chen and Meichun Hsu

Dynamic Workload-Based Partitioning Algorithms for Continuously Growing Databases 105
Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez

Author Index 129

EvoMatch: An Evolutionary Algorithm for Inferring Schematic Correspondences

Chenjuan Guo, Cornelia Hedeler, Norman W. Paton, and Alvaro A.A. Fernandes

School of Computer Science, University of Manchester, M13 9PL, UK
{cguo,chedeler,norm,alvaro}@cs.man.ac.uk

Abstract. Schema matching provides an important foundation for both manual and semi-automatic derivation of mappings between sources. However, schema matchers typically return large numbers of potentially inconsistent matches that are neither conducive to automatic mapping generation nor readily digested by mapping developers. This paper presents a method, *EvoMatch*, for automatically inferring schematic correspondences, from which mappings can be generated directly. It aims to offer a more expressive characterization of the relationships between sources than matches identified by existing schema matching methods. In particular, the paper contributes: i) an evolutionary search method for inferring schematic correspondences; ii) an objective function for calculating the fitness value of a solution within the search space; and iii) an empirical evaluation assessing the effectiveness of *EvoMatch* for inferring schematic correspondences in comparison with well established existing techniques. In doing so, *EvoMatch* automatically identifies correspondences that can be used directly to bootstrap information integration systems, or to inform the manual refinement of mappings.

1 Introduction

The requirement to manage and query interrelated but heterogeneous data sources is widespread. Traditional data integration systems provide high-quality services but at a high cost [1], because, prior to offering services that build on the integration to users, precise mappings are required to describe the relationships between data sources, which tends to be a hard problem and needs upfront effort. The process for specifying mappings consists of *schema matching*, e.g., [2][3][4], and *schema mapping* (*view generation*), e.g., [5][6][7]. Schema matching methods identify matches, which relate elements of two data sources that show similar properties (e.g., names, instances and structures). Matches are then semi-automatically refined using schema mapping tools into declarative but executable mappings (e.g., in SQL or XSLT) to specify the relationships between the data sources. Most schema mapping techniques tend to include significant redundancy in their resulting mappings and require users to visually select or refine the mappings [8], and therefore human intervention is typically part of the process.

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 1–26, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The vision of *dataspaces* [9] was proposed to reduce the high upfront cost of traditional data integration following a *pay-as-you-go* approach, and to provide integration services to a diverse type of heterogeneous data sources (e.g., relational and XML databases, web services and text files). A possible solution could be achieved by automatically bootstrapping a data integration system to release experts from the time-consuming process of mapping specification. At a later stage, ordinary users may incrementally provide feedback on query results, thus helping to improve the quality of specified mappings (e.g., [10][11][12]). This idea motivates the results presented in the paper.

In this paper, we propose a method, called *EvoMatch*, for automatically refining matches between elements of two schemas into a type of correspondences, which represents relationships between elements more explicitly. By doing this, we provide more information about the relationships between the two schemas than matches, in order to inform the automatic generation of mappings [13]. Specifically, the type of correspondences inferred by *EvoMatch*, called schematic correspondences, is based on the classification of Kim *et al.* [14]. Note that the schematic correspondences are not mappings that specify relationships between elements using explicit expressions, but tend to offer more information than matches to serve as input to potential (semi-)automatic schema mapping methods.

Assume we have source and target data sources whose schemas $\mathcal{S}$ and $\mathcal{T}$ are shown in Fig. 1. *EvoMatch* takes as input source and target schemas $\mathcal{S}$ and $\mathcal{T}$ and matches $\mathcal{M}$ between them identified by some match system, e.g., COMA++ [2], as shown in Fig. 2(a).

Source Schema S

home_cust	overseas_cust	account
id	id	id
name	name	name
birth	birth	balance
phone	phone	tax

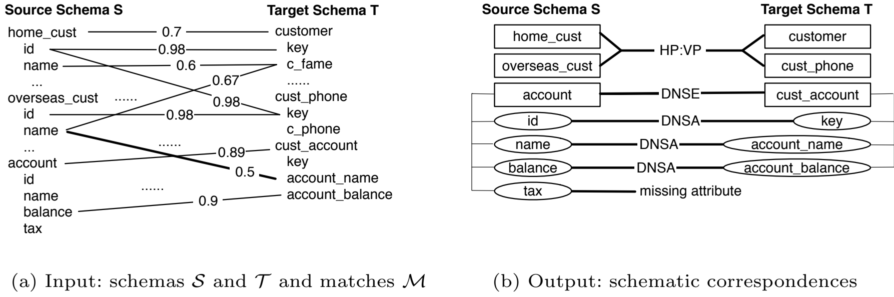
Target Schema T

customer	cust_phone	cust_account
key	key	key
c_fname	c_phone	account_name
c_lname		account_balance
c_birth		

Fig. 1. Example Source and Target Schemas $\mathcal{S}$ and $\mathcal{T}$

We do not dwell on the method for identifying the matches, as many schema matching approaches have been proposed [15], and we assume they are available. Note that matches only relate elements between $\mathcal{S}$ and $\mathcal{T}$ that show certain similarity, and thus sometimes associate elements that represent different real world information, e.g., the match between overseas_cust.name in $\mathcal{S}$ and cust_account.account_name in $\mathcal{T}$ (see Fig. 2(a)).

In contrast with the numerous, fine-grained matches, we can observe some coarser-grained relationships that explicitly indicate equivalence between elements of $\mathcal{S}$ and $\mathcal{T}$, as classified by the schematic correspondences of Kim *et al.* [14], which are shown in Fig. 2(b) and will be produced by *EvoMatch*.



(a) Input: schemas $\mathcal{S}$ and $\mathcal{T}$ and matches $\mathcal{M}$

(b) Output: schematic correspondences

Fig. 2. Input and output of *EvoMatch*

In particular, entity types account in $\mathcal{S}$ and cust_account in $\mathcal{T}$ represent equivalent information about *customer accounts*, though using different names. Their relationship is characterized as Different Names for the Same Entity (*DNSE*). Similarly, at the attribute level between account and cust_account, equivalent attributes, e.g., account.id and cust_account.key, may have different names, whose relationship is characterized as Different Names for the Same Attribute (*DNSA*). An attribute that does not have an equivalent attribute is called a missing attribute, such as account.tax.

The original information about *customers* is Horizontally Partitioned (*HP*) at the instance level in $\mathcal{S}$, thus giving rise to two tables home_cust and overseas_cust, each of which keeps the original schema information; whereas in $\mathcal{T}$, this information is Vertically Partitioned (*VP*) at the schema level into tables customer and cust_phone, each of which keeps a subset of attributes of the original information about customers. Entity types home_cust and overseas_cust in $\mathcal{S}$ together represent equivalent customers' information to tables customer and cust_phone in $\mathcal{T}$, and their explicit relationship is called *HP:VP*. Attribute-level schematic correspondences between {home_cust, overseas_cust} and {customer, cust_phone} are omitted in Fig. 2(b) for simplicity.

Principally, *EvoMatch* is expected: i) to require no user involvement, and thus users are not required to select or refine schematic correspondences from potential candidate correspondences; ii) not to need context-specific parameters, e.g., thresholds, because setting such parameters usually needs training data, which may be hard for users to obtain; iii) not to assume that external resources, e.g., domain specific ontologies, are available; and iv) not to assume that schemas contain referential constraints, as they may be missing in real applications, e.g., web tables [16]. Note that *EvoMatch* is designed as a fully automatic method, but of course it builds on input matches of variable dependability. Thus, there are cases where the results will be less sound or complete than those informed by expert manual input. However, the set of schematic correspondences returned as the *EvoMatch* result could be used as input to a semi-automatic or pay-as-you-go approach.

This paper makes the following contributions:

- An evolutionary search method, specifically a genetic algorithm, for inferring schematic correspondences, which allows potential sets of correspondences between two schemas to compete with each other to derive the resulting correspondences.
- An objective function used by the search that models the desirability of candidate entity-level schematic correspondences.
- An experimental analysis that evaluates the effectiveness of our approach for inferring schematic correspondences by comparing it with COMA++ [2], Similarity Flooding [3] and Harmony [17].

The remainder of the paper is structured as follows. Section 2 reviews related work. Section 3 defines the schematic correspondences of Kim *et al.* [14]. Section 4 overviews the *EvoMatch* method that utilizes the evolutionary search framework to infer entity-level schematic correspondences, followed by the objective function that models the requirements for inferring such correspondences in Section 5. Section 6 presents the method for inferring attribute-level schematic correspondences. Section 7 describes the experimental evaluation of our method. Section 8 concludes the paper.

2 Related Work

This section reviews work related to the proposed method, on classifications of correspondences, on methods for inferring complex correspondences, on using evolutionary algorithms for schema matching, and on methods for generating mappings automatically.

A classification of correspondences is presented in Table 1, including simple correspondences ($Type_1^s$ to $Type_4^s$) and complex correspondences ($Type_1^c$ to $Type_4^c$).

Table 1. A classification of correspondences

	simple	*complex*
1-to-1 entity-level	$Type_1^s$	$Type_1^c$
1-to-1 attribute-level	$Type_2^s$	$Type_2^c$
n-to-m entity-level	$Type_3^s$	$Type_3^c$
n-to-m attribute-level	$Type_4^s$	$Type_4^c$

The simple correspondences refer to equivalence associations between two (sets of) elements of schemas, while the complex correspondences refine such equivalence associations with additional information. Most existing schema

matching approaches (e.g., [2][3][18][19]) consider $Type_1^s$ and $Type_2^s$ correspondences and relate 1-to-1 equivalent elements. A few methods (e.g., [20][21][22]) address $Type_1^c$ and $Type_2^c$ correspondences and identify relationships, such as, *Is-a* [20] or *subsumption* [22], between 1-to-1 elements. $Type_3^s$ and $Type_4^s$ correspondences refer to the equivalence associations between two sets of elements. $Type_3^c$ and $Type_4^c$ correspondences not only indicate that the associated element sets are equivalent, but also specify an internal element association within each set (e.g., [23][4][24][25]). *EvoMatch* contributes to the inference of all types of complex correspondences, i.e., $Type_1^c$ to $Type_4^c$ in Table 1.

In terms of *inferring complex correspondences*, *SeMap* [20] identifies rich semantic relationships between 1-to-1 elements of different data models, i.e., $Type_1^c$ and $Type_2^c$ in Table 1. Both *EvoMatch* and *SeMap* infer 1-to-1 complex relationships using matches, but differ on the specific relationships. Particularly, *SeMap* infers *Has-a*, *Is-a*, *Associates* and *Equivalent* relationships, and *EvoMatch* infers *DNSE* and *DNSA* relationships. Thus, the two methods can be seen as being complementary to each other in inferring 1-to-1 complex correspondences. In addition, *EvoMatch* infers n-to-m relationships, thus handling a more challenging problem.

The methods proposed by Rizopoulos [22] and Giunchiglia *et al.* [21] identify complex 1-to-1 relationships between elements of schemas, such as *equivalence*, *subsumption*, *intersection*, *incompatibility* [22], and *more general*, *less general*, *disjointness* [21], referred to as $Type_1^c$ and $Type_2^c$ in Table 1. Rizopoulos compares the instance containment of elements to derive the relationships, while Giunchiglia *et al.* infer such relationships by determining the element name containment using WordNet [26]. Their work could be seen as complementary to our approach, or used to improve the quality of the input matches used by *EvoMatch*.

Xu *et al.* [23] propose a semi-automatic approach for inferring 1-to-1 and n-to-m complex relationships between schema elements, specifically, $Type_1^c$ to $Type_4^c$ in Table 1. In addition to identifying that two (sets of) elements are equivalent, Xu *et al.* apply operators over elements in the source (or target) set that specify their associations further, thus expressing an n-to-m relationship in the form of an algebra. Xu *et al.* make significant use of domain specific ontologies, which are not always available in real world scenarios, whereas *EvoMatch* does not rely on such external resources or human effort. Xu *et al.* utilize certain specific constraints of the employed schema model (i.e., the conceptual model) to infer the element associations in the source (or target) set; in contrast, we do not make use of any constraints of the relational model (e.g., primary and foreign key information) to identify the n-to-m relationships, and thus *EvoMatch* can be applied to contexts where schema constraints are not available. Furthermore, a single element may be a member of several complex relationships in the approach presented by Xu *et al.*, thus requiring the user to choose the desired relationships, whilst *EvoMatch* only associates an element with a single correspondence.

iMAP [4], Dai *et al.* [24] and Warren *et al.* [25] specialize in discovering complex n-to-m relationships at the attribute-level using instance data (i.e., $Type_4^c$ in Table 1). iMAP [4] detects various complex attribute matches using different

preset formulae that transform instances between the source and target attributes. Dai *et al.* [24] contribute to the identification of n-to-1 attribute matches where the concatenation of the n attributes is equivalent to the single attribute, and can handle the case that the n attributes have disjoint instances with the single attribute. The approach proposed by Warren *et al.* [25] also identifies n-to-1 matches for string attributes, and creates a transformation formula that concatenates the n attributes whose cardinality is unknown in advance into the single attribute. *EvoMatch* contributes more to the inference of complex relationships at the entity-level ($Type_3^c$), although we also implement a simple method for inferring n-to-m attribute relationships. Thus, iMAP [4], and methods proposed by Dai *et al.* [24] and Warren *et al.* [25] are seen as complementary to *EvoMatch*.

In terms of *using evolutionary algorithms for schema matching*, *EvoMatch* is not the first work that casts the schema matching problem as an evolutionary search problem. The method proposed by Elmeleegy *et al.* [27] also approaches the problem from this angle, but only contributes to the identification of 1-to-1 attribute-level relationships ($Type_2^s$). The method uses information extracted from query logs, and adapts the scoring functions proposed by Madhavan *et al.* [28] to calculate the similarity of attributes. It then employs a genetic algorithm to search for a particular set of 1-to-1 attribute matches. Although both Elmeleegy *et al.* and *EvoMatch* use genetic algorithms to infer correspondences, we design a novel objective function to complete a more complicated task (i.e., inferring $Type_1^c$ and $Type_3^c$). We infer equivalent 1-to-1 and n-to-m entities and discover particular entity associations within the source and target sets, i.e., horizontal or vertical partitioning, which, to the best of our knowledge, has not been done before.

In terms of *specifying mappings automatically*, the context of such work is relevant to the discussion. Established practice in schema mapping development involves designers in the use of a visual tool to select a subset of matches from which a collection of alternative mappings can be generated, for example as in Clio [29]. However, even where there is a small collection of user-selected matches, many alternative schema mappings may be able to be generated. This has led to additional research, for example in ++Spicy [8], which seeks to refine the collection of mappings generated by existing mapping tools. Another approach to mapping generation makes use of data instances to guide mapping generation (e.g. [30]). In this research, which complements mapping generation based on matchings alone, mapping designers provide data instances for source and target schemas and a search takes place for mappings, which can be further refined by users. The research on EvoMatch can be considered to complement the work on mapping generation, in that the correspondences identified by EvoMatch could be used as an input to schema mapping development. Experience with automatic mapping generation, both with and without data instances, suggests that going straight to expressive mappings from syntactic matches may be impractical.

3 Schematic Correspondences

We adopt the definitions of schematic correspondences of Kim *et al.* [14], which relate different symbolic representations of data that represent the same real world information. We do not claim that such relationships are uniquely suitable in some applications, but the following properties motivated their use in the proposed method: (i) the relationships between schema elements are characterized in a data model-independent manner; (ii) they include different cardinalities of correspondences at both the entity and attribute levels; (iii) and they identify cases that are common in practice, and that can be resolved using views whether constructed manually [14].

In this section, we follow mostly the schematic correspondence definitions of Kim *et al.*, and refine the definition of n-to-m entity correspondences into horizontal and vertical partitioning. In particular, we classify schematic correspondences using complex correspondences $Type_1^c$ to $Type_4^c$ in Table 1.

- $Type_1^c$ refers to correspondences that associate 1-to-1 equivalent entities, which carry additional information about their names. Equivalent entities whose names are different are called Different Names for the Same Entity (*DNSE*), such as account in $\mathcal{S}$ and cust_account in $\mathcal{T}$. Equivalent entities that have the same name are called the Same Name for the Same Entity (*SNSE*).
- $Type_2^c$ refers to correspondences of 1-to-1 attributes that represent the equivalent properties of equivalent entities, which carry additional information about their names. Equivalent attributes that have different names are called Different Names for the Same Attribute (*DNSA*), such as account.name in $\mathcal{S}$ and cust_account.account_name in $\mathcal{T}$. Equivalent attributes whose names are also the same are called the Same Name for the Same Attribute (*SNSA*).
- $Type_3^c$ refers to correspondences of two entity sets whose structures may be different but represent equivalent underlying information, e.g., {home_cust, overseas_cust} and {customer, cust_phone}. Inheriting terms from distributed database systems [31], we define the entity structure within a set as:
  - *horizontal partitioning* (*HP*), where an original entity is partitioned along its instances into new entities. As such, all attributes of the original entity are present in each of the new entities (e.g., home_cust and oversea_cust in $\mathcal{S}$).
  - *vertical partitioning* (*VP*), where an original entity is partitioned into new entities whose attributes are subsets of the original entity. As such, some attributes are present in each new entity, i.e., key attributes, whereas other attributes of the original entity are present only once across all the new entities (e.g., customer and cust_phone in $\mathcal{T}$).
- $Type_4^c$ refers to correspondences of two equivalent attribute sets. For example, home_cust.name in $\mathcal{S}$ represents the same information as the concatenation of customer.c_fname and customer.c_lname in $\mathcal{T}$.

4 Evolutionary Search

EvoMatch takes as input source and target schemas $\mathcal{S}$ and $\mathcal{T}$ and their matches $\mathcal{M}$, and infers schematic correspondences between $\mathcal{S}$ and $\mathcal{T}$. In Section 4.1, we present the overview of *EvoMatch*, which mostly utilizes an evolutionary search method, specifically a genetic algorithm [32], to complete the task; In Section 4.2, we introduce the important concepts used in the genetic algorithm, i.e., phenotype and genotype; and in Section 4.3, we describe the framework of the genetic algorithm.

4.1 Overview of EvoMatch

EvoMatch takes as input source and target schemas $\mathcal{S}$ and $\mathcal{T}$ and their matches $\mathcal{M}$, and infers schematic correspondences between $\mathcal{S}$ and $\mathcal{T}$ as output via two phases. A schema is defined in Definition 1, referring to both $\mathcal{S}$ and $\mathcal{T}$.

Definition 1 ***Schema.*** *A* **schema** $\mathcal{S}$=$\{S_1, ..., S_\mu\}$ *is a set of entities (e.g., relational tables), where each* $S_i \in \mathcal{S}$ *contains attributes* $S_i.A_1, ..., S_i.A_\alpha$*. An entity or an attribute is also called a* **construct** *of* $\mathcal{S}$*.*

As shown in Fig. 3, *EvoMatch* follows two phases to infer schematic correspondences. First, in the *entity-level* phase, *EvoMatch* searches for a set of entity-level schematic correspondences, i.e., $Type^c_1$ and $Type^c_3$ in Section 3, from candidate correspondences for which there is evidence from the input matches. Second, for each pair of entity sets in a resulting entity-level schematic correspondences, *EvoMatch* infers a set of attribute-level schematic correspondences, i.e., $Type^c_2$ and $Type^c_4$ in Section 3, in the *attribute-level* phase.

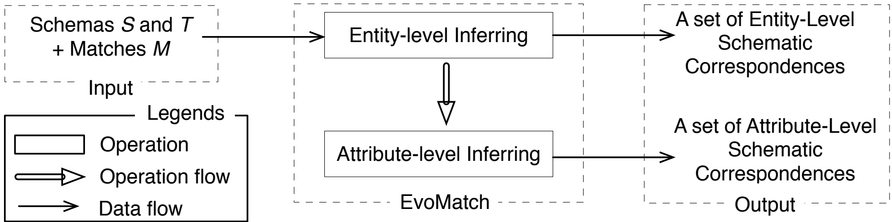


Fig. 3. Overview of *EvoMatch*

4.2 Phenotype and Genotype Representations of a Solution

The genetic algorithm uses the input matches $\mathcal{M}$ as evidence to explore the space of entity-level schematic correspondences between $\mathcal{S}$ and $\mathcal{T}$. Thus, the *search space* is a collection of candidate entity-level schematic correspondences. In the genetic algorithm, a solution has *phenotype* and *genotype* representations, where

the former provides the context for evaluating the fitness of a solution, and the latter supports the generation of alternative candidate solutions. A phenotype representation of a solution is defined in Definition 2.

Definition 2 ***Phenotype.*** *Given schemas* $\mathcal{S}$ *and* $\mathcal{T}$ *and matches* $\mathcal{M}$*, the* **phenotype** $\mathcal{P}$ *of a solution is a set of* **Entity-Level Relationships (ELRs)**. *An* $ELR_i{=}\langle ES_i^{\mathcal{S}}, ES_i^{\mathcal{T}}\rangle \in \mathcal{P}$*, where* $ES_i^{\mathcal{S}} \subseteq \mathcal{S}$ *and* $ES_i^{\mathcal{T}} \subseteq \mathcal{T}$ *are two entity sets with cardinality* $\geqslant 1$*, satisfies the following conditions: 1) there exists either an entity-level match or an attribute-level match* $m \in \mathcal{M}$ *between each entity in* $ES_i^{\mathcal{S}}$ *and each entity in* $ES_i^{\mathcal{T}}$*; and 2) for each* $ELR_{i'}{=}\langle ES_{i'}^{\mathcal{S}}, ES_{i'}^{\mathcal{T}}\rangle \in \mathcal{P}$, $i' \neq i$, $ES_i^{\mathcal{S}} \cap ES_{i'}^{\mathcal{S}} = \emptyset$ *and* $ES_i^{\mathcal{T}} \cap ES_{i'}^{\mathcal{T}} = \emptyset$*. Specifically, entity set* $ES_i^{\mathcal{S}}$ *(or* $ES_i^{\mathcal{T}}$*) is called an* **associated entity set** *of* ELR_i*. Each entity* $S_j \in \mathcal{S}$ *(or* $\in \mathcal{T}$*) that satisfies* $S_j \notin \cup ES_i^{\mathcal{S}}$ *(or* $\notin \cup ES_i^{\mathcal{T}}$*) is called an* **unassociated entity**.

We continue with the running example from Fig. 1 to illustrate phenotypes. Fig. 4 illustrates some example matches $\mathcal{M}$ between schemas $\mathcal{S}$ and $\mathcal{T}$. Specifically, a set of matches between two entities and between their attributes is represented by a single line in 4. Example phenotypes of candidate solutions are:

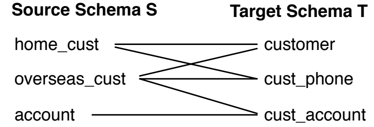


Fig. 4. Matches $\mathcal{M}$

$\mathcal{P}_1{=}\{ELR_1, ELR_2\}{=}\{\langle\{$home_cust, overseas_cust$\}, \{$customer, cust_phone$\}\rangle,$
$\langle\{$account$\}, \{$cust_account$\}\rangle\}$
$\mathcal{P}_2{=}\{ELR_3\}{=}\{\langle\{$overseas_cust$\}, \{$cust_account$\}\rangle\}$

The entity sets in $ELR_1 \in \mathcal{P}_1$ can be associated, as there is a match between each pair of entities in the two sets (Condition 1), as shown in Fig. 4. In $\mathcal{P}_1$, ELR_1 and ELR_2 are disjoint (Condition 2), indicating that each source or target entity only participates in a single entity-level schematic correspondence in a solution.

In $\mathcal{P}_2$, {overseas_cust} and {cust_account} are *associated entity sets*; the remaining entities, e.g., home_cust and customer, which are not contained in the entity sets of ELR_3 are called *unassociated entities.*

In the evolutionary search, a solution also has a *genotype* representation to facilitate the search process. The genotype is the encoding of the phenotype in the search space and is chosen as a binary string for simplicity, as presented in Definition 3.

Definition 3 ***Genotype.*** *Given source and target schemas* $\mathcal{S}$ *and* $\mathcal{T}$*, and match evidence* $\mathcal{M}$*, the* **genotype** *of a solution is a sequence of binary values* $\mathcal{G} = [x_1, x_2, ..., x_m]$*, where* $x_i \in \{0, 1\}$ $(i = 1, ..., m)$*. Each* x_i *represents a pair of entities* $< S_p, T_q >$*,* $S_p \in \mathcal{S}$*,* $T_q \in \mathcal{T}$*, such that every pairing supported by match evidence in* $\mathcal{M}$ *is represented. In particular,* $x_i = 1$ *(resp.* 0*) represents that the* i^{th} *pair of entities in* $\mathcal{G}$ *is (resp. is not) associated.*

Following on with the example, $\mathcal{G}_1 = [1, 1, 1, 1, 0, 1]$ and $\mathcal{G}_2 = [0, 0, 0, 0, 1, 0]$ are the genotypes for phenotypes $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Transformations between genotypes and phenotypes are referred to as encoding and decoding. The algorithms are straightforward, and thus are omitted here.

For a genotype consisting of m binary values, the size of the search space is 2^m, where m is the number of matched entities,

$$count(\{< s, t > | s \in \mathcal{S}, t \in \mathcal{T}, matched(s, t)\}),$$

where *matched* is true iff there exists a match between s and t. For example, if both schemas have 4 entities, where we assume there is at least one match between each pair of entities, the size of the search space is 2^{16} (i.e., the genotype is of size 4×4), and if both schemas have 10 entities each with at least one match to the other, the size of the search space is 2^{100}.

4.3 Search Framework

Fig. 5 presents a framework for the evolutionary search process [32], where a *generation* forms the basic unit of the search and holds a set of solutions, known as a *population*. The *population size* represents the number of solutions in a population or generation, and is fixed to a chosen size during the search. A *parent* is a member of a population, that has been chosen to go through mutation and crossover operators so as to produce new solutions (i.e., *offspring*). The *objective function* models requirements of the proposed search problem, and assigns a fitness value to a solution to evaluate its quality among all solutions with respect to the requirements, as presented in Section 5.

The evolutionary search starts from a set of random solutions called the *initial generation*, which are evaluated using the objective function. The following steps are then repeated until a termination condition (e.g., the elapsed CPU time or the number of fitness evaluations [32]) is satisfied:

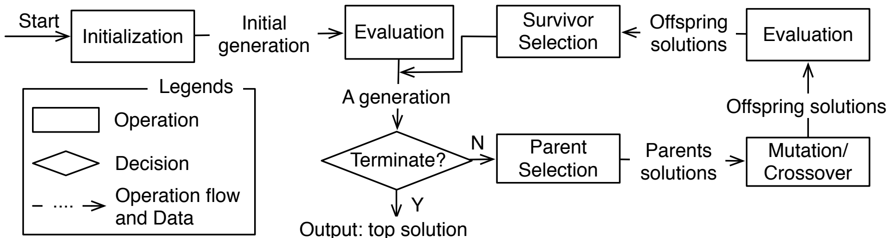


Fig. 5. The evolutionary search framework

- **Parent selection** selects from the current generation those that should be used as parents for the next generation with the aim of improving the quality of subsequent generations over time. We apply roulette-wheel selection, which maps the solutions from the current generation to segments in the wheel, such that solutions whose fitness values are greater occupy larger segments. This technique is applied so that solutions whose fitness values are comparatively high in the generation have a higher chance of being chosen.
- **Mutation** is a unary variation operator that manipulates a single parent genotype (a binary string) and is used to improve the diversity of a population [33]. It inverts each chosen bit in the binary string, with a probability called the *mutation rate* p_m, from 0 to 1 or from 1 to 0.
- **Crossover** is a binary variation operator on two parent genotypes and explicitly tries to combine "good" parts of the parents [33]. The one-point crossover is used for its simplicity. It splits both parents at the xth bit ($x <$ the genotype length) and exchanges their tails starting from the xth bit. The crossover is applied with a probability, called the *crossover rate* p_c.
- **Survivor selection** determines the next generation by choosing μ (i.e., population size) survivors that have the highest fitness values from μ parent solutions in the last generation and λ offspring produced from the last generation using mutation and crossover operators.

We chose to employ an evolutionary algorithm, specifically a genetic algorithm [32], because: (i) the search space is extremely large, precluding the use of an exhaustive search; (ii) several alternative search strategies, e.g., hill climbing, local search and simulated annealing, explore the search space based on a single solution, and thus the quality of the initial solution can influence the resulting solution significantly [34]; (iii) a genetic algorithm is a population-based search and thus can produce a collection of solutions rather than a single solution for further investigation (which is potentially useful both in pay-as-you-go and manual integration scenarios); and (iv) a genetic algorithm often uses less time to complete a task than other search methods [34].

5 Objective Function

The objective function acts on schematic correspondences at the entity-level, i.e., $Type_1^c$ and $Type_3^c$, and evaluates candidate solutions in their phenotype representation; we call such a candidate solution a *candidate phenotype*.

From Definition 2, each candidate phenotype $\mathcal{P}$ consists of a set of ELRs, $\mathcal{P} = \{ELR_1, ..., ELR_n\}$, where each ELR_i is a pair of associated entity sets $< ES_i^{\mathcal{S}}, ES_i^{\mathcal{T}} >$. The objective function builds on the vector space model from information retrieval [35] to estimate the similarity of the two entity sets in an *ELR*. The *associated entity sets* $ES_i^{\mathcal{S}}$ and $ES_i^{\mathcal{T}}$ of ELR_i give rise to sets of *terms*, $TS_i^{\mathcal{S}} = \{t_0^{\mathcal{S}}, ...t_q^{\mathcal{S}}\}$ and $TS_i^{\mathcal{T}} = \{t_0^{\mathcal{T}}, ...t_r^{\mathcal{T}}\}$, that represent entity and attribute names (as described in Section 5.1). In turn, each of these terms is associated with a weight computed using the widely used $tf \times idf$ function [35] (as described in Section 5.2).

The sets of terms $TS_i^{\mathcal{S}}$ and $TS_i^{\mathcal{T}}$ of ELR_i are represented as vectors $\boldsymbol{v}_i^{\mathcal{S}} = (w_0^{\mathcal{S}}, ...w_q^{\mathcal{S}})$ and $\boldsymbol{v}_i^{\mathcal{T}} = (w_0^{\mathcal{T}}, ...w_r^{\mathcal{T}})$, respectively (as described in Section 5.3), where each $w_j^{\mathcal{S}}$ and $w_k^{\mathcal{T}}$ is a weight, and the weights at position p, $w_p^{\mathcal{S}}$ and $w_p^{\mathcal{T}}$, in each of the vectors represent matched terms. The similarity score s_i of an ELR_i can then be computed as the cosine of the angle between the pair of vectors $\boldsymbol{v}_i^{\mathcal{S}}$ and $\boldsymbol{v}_i^{\mathcal{T}}$ (as described in Section 5.4).

Finally, the objective function aggregates ELR similarities $s_1, \ldots, s_n$ into the fitness the phenotype (as described in Section 5.5).

5.1 Representing Entity Sets Using Terms

Recall that an ELR is a pair of associated entity sets $< ES_i^{\mathcal{S}}, ES_i^{\mathcal{T}} >$; for example, the candidate phenotype $\mathcal{P}_2$ ={⟨{overseas_cust}, {cust_account}⟩} from Section 4.2 contains a single ELR. Thus each entity set represents a collection of one or more entity types, and we need a more detailed description that not only contains the names of potentially related entity types, but also includes attribute information. This more detailed description of an entity set is a set of *terms*, $TS_i = \{t_0, ...t_m\}$, where each t_j is a set of values that represent entity and attribute names, as described below.

Basic Entity Set. An entity set is called a *basic entity set* if it models an *associated entity set* with cardinality=1. An entity type S_i, containing attributes $S_i.A_1, ..., S_i.A_\alpha$, is modelled by the set of terms $TS_i = \{\{S_i\}, \{S_i.A_1\}, ..., \{S_i.A_\alpha\}\}$. For example, using $\mathcal{P}_2$, the *associated entity set* {cust_account} is modeled by the set of terms:

{{cust_account}, {cust_account.key}, {cust_account.account_name},
{cust_account.account_balance}}.

Equivalent attributes. To inform the creation of vectors that represent horizontally and vertically partitioned entity sets, we define equivalent attributes in Definition 4. The equivalent attributes make explicit different entities in a schema that have similar features (i.e., attributes that have similarity in their names or instances). For example, the three entities home_cust, overseas_cust and account in Figure 2 all have an attribute name.

Definition 4 ***Equivalent Attributes.*** *Given a schema $\mathcal{S}$ containing entities $S_1, \ldots, S_\mu$, a set of attributes $\mathcal{A}$=$\{A_1, \ldots, A_m\}$ in $\mathcal{S}$ is defined as a set of* **equivalent attributes** *if attributes in $\mathcal{A}$ satisfy the following conditions: 1) each attribute $A_i \in \mathcal{A}$ belongs to a distinct entity in $\mathcal{S}$; 2) each entity $S_j \in \mathcal{S}$ contains only a single attribute in $\mathcal{A}$; and 3) all attributes in $\mathcal{A}$ share similar features, e.g., names and instances.*

For example, the sets of equivalent attributes in schema $\mathcal{S}$ in Figure 2 are:

$\mathcal{A}_1$={home_cust.id, overseas_cust.id, account.id}
$\mathcal{A}_2$={home_cust.name, overseas_cust.name, account.name}

$\mathcal{A}_3$={home_cust.birth, overseas_cust.birth}
$\mathcal{A}_4$={home_cust.phone, overseas_cust.phone}

The only set of equivalent attributes in schema $\mathcal{T}$ is:

$\mathcal{A}_4$={customer.key, cust_phone.key, cust_account.key}

In the implementation of *EvoMatch*, we use matches between $\mathcal{S}$ and $\mathcal{T}$ to obtain equivalent attributes. The basic idea is that if some attributes that belong to different entities in $\mathcal{S}$ are matched to the same attribute in $\mathcal{T}$, we identify these attributes as equivalent attributes.

Horizontally Partitioned Entity Set. The basic idea of *horizontal partitioning* is that an original entity is partitioned along its instances into new entities, and as such all attributes of the original entity are present in each of the new entities.

Given an *associated entity set* with cardinality greater than 1, it is possible that this entity set represents a single entity type that has been horizontally partitioned. To capture this notion, an entity set in schema $\mathcal{S}$, $ES_i^{\mathcal{S}} = \{S_1, ..., S_\varrho\}$ is represented by the set of terms $TS^{\mathcal{S},H} = \{t_0^{\mathcal{S}}, ...t_m^{\mathcal{S}}\}$, where each $t_i^{\mathcal{S}}$ is a set of values that represent entity and attribute names, as described below.

If all entities $S_1, \ldots, S_\varrho$ have an equivalent attribute A_c, the term t_c that represents A_c is defined as $\{S_1.A_c, \ldots, S_\varrho.A_c\}$. If a subset of the entities, e.g., S_1 and S_2, have an equivalent attribute A_d, t_d is $\{S_1.A_d, S_2.A_d, S_3.\psi, \ldots, S_\varrho.\psi\}$, where $S_j.\psi$ ($j = 3, \ldots, \varrho$) means that the entity S_j does not contain an attribute equivalent to attribute A_d.

For example, in $\mathcal{P}_1$, the entity set $ES_1^{\mathcal{S}}$={home_cust, overseas_cust} can be represented as:

$TS_1^{\mathcal{S},H}$ ={{home_cust, overseas_cust}, {home_cust.id, overseas_cust.id},
{home_cust.name, overseas_cust.name},
{home_cust.birth, overseas_cust.birth},
{home_cust.phone, overseas_cust.phone}};

Equivalent attributes shared by all entities in the source set are represented as a single term (e.g., {home_cust.name, overseas_cust. name}). In the case of $TS_1^{\mathcal{S},H}$, interpreting the entity set as horizontally partitioned seems promising, as every attribute in one entity has an equivalent counterpart in the other.

As another example, in $\mathcal{P}_1$, the entity set $ES_1^{\mathcal{T}}$ = {customer, cust_phone}, can be represented as:

$TS_1^{\mathcal{T},H}$ ={{customer, cust_phone}, {customer.key, cust_phone.key},
{customer.c_fname, cust_phone.ψ},
{customer.c_lname, cust_phone.ψ},
{customer.c_birth, cust_phone.ψ},
{customer.ψ, cust_phone.c_phone}};

In this case, there are several attributes that are not shared by all entities (e.g., customer.c_fname), as indicated by the ψ attribute, which suggests that this entity set may not be horizontally partitioned.

Vertically Partitioned Entity Sets. The basic idea of *vertical partitioning* is that an original entity is partitioned into new entities such that the original entity

is reconstructed by joining its partitions. Thus, some attributes are present in each of the partitions, referred to as key attributes, whereas other attributes of the original entity are present only once across all the partitions.

Given an *associated entity set* with cardinality greater than 1, it is possible that this entity set represents a single entity type that has been vertically partitioned. To capture this notion, an entity set in schema $\mathcal{S}$, $ES_i^{\mathcal{S}} = \{S_1, ..., S_\varrho\}$ is represented by the set of terms $TS_i^{\mathcal{S},V} = \{t_0^{\mathcal{S}}, ...t_m^{\mathcal{S}}\}$, where each $t_i^{\mathcal{S}}$ is a set of values that represent entity and attribute names, as described below.

If all entities $S_1, \ldots, S_\varrho$ share an equivalent attribute A_c, term t_c that represents A_c is defined as $\{S_1.A_c, \ldots, S_\varrho.A_c\}$, representing a key attributes of all the entities. If an attribute $S_i.A_d$ is not shared by all entities, then t_d is defined as $\{S_i.A_d\}$, indicating that it does not meet the condition to be a key attribute in vertical partitioning.

For example, in $\mathcal{P}_1$, the entity set $ES_1^{\mathcal{T}} = \{$customer, cust_phone$\}$, can be represented as:

$TS_1^{\mathcal{T},V}$ =$\{\{$customer, cust_phone$\}$, $\{$customer.key, cust_phone.key$\}$,
$\{$customer.c_fname$\}$, $\{$customer.c_lname$\}$,
$\{$customer.c_birth$\}$, $\{$cust_phone.c_phone$\}\}$.

Equivalent attributes shared by all the entities are represented as a single term (i.e., $\{$customer.key, cust_phone.key$\}$), and are identified as key attributes. There are also attributes (e.g., customer.c_fname) that are not shared by all the entities, and each of them is represented as a term (e.g., $\{$customer.c_fname$\}$). In this case, there is a single term that meets the requirement for a key term, and thus the entity set may be considered to be a strong candidate as a vertical partition.

5.2 Weight Calculation

The objective function utilizes the well-known $tf \times idf$ function [35] to calculate term weights. Assume that the entity set $ES_j^{\mathcal{S}}$ in the schema $\mathcal{S}$ is represented by the set of terms $TS_j^{\mathcal{S}}$, which contains the term $t_i^{\mathcal{S}}$. The weight $w_i^{\mathcal{S}}$ representing $t_i^{\mathcal{S}}$ is defined as $w_i^{\mathcal{S}} = tf \times idf$ with $tf = \frac{|t_i^{\mathcal{S}}|}{|TS_j^{\mathcal{S}}|}$ and $idf = log\frac{N}{n_i}$, where

- $|t_i^{\mathcal{S}}|$ is the cardinality of $t_i^{\mathcal{S}}$, which essentially represents the frequency of occurrence of an entity or attribute name in an entity set.
- $|TS_j^{\mathcal{S}}|$ denotes the total number of terms in $ES_j^{\mathcal{S}}$.
- N is the total number of entity sets in the schema $\mathcal{S}$.
- n_i denotes the number of entity sets in the schema $\mathcal{S}$ that contain terms that are equivalent to $t_i^{\mathcal{S}}$.

For example, assume we have the *associated entity sets* $ES_1^{\mathcal{S}}$=$\{$home_cust, overseas_cust$\}$ and $ES_2^{\mathcal{S}}$=$\{$account$\}$ in $\mathcal{P}_1$ for which the following sets of terms have been derived:

$TS_1^{\mathcal{S},H}$ =$\{\{$home_cust, overseas_cust$\}$, $\{$home_cust.id, overseas_cust.id$\}$,
$\{$home_cust.name, overseas_cust.name$\}$,

{home_cust.birth, overseas_cust.birth},
{home_cust.phone, overseas_cust.phone}};
$TS_2^{\mathcal{S}}$ ={{account}, {account.id}, {account.name}, {account.balance},
{account.tax}}.

The term weight $w_i^{\mathcal{S}}$ for term {home_cust.id, overseas_cust.id} in $TS_1^{\mathcal{S},H}$ can be calculated as: $|t_1^{\mathcal{S}}|$=2 for the two attributes in the term; $|TS_1^{\mathcal{S}}| = 10$ for the total number of constructs (i.e., entities and attributes) in $TS_1^{\mathcal{S},H}$; N=2 as there are 2 entity sets {home_cust, overseas_cust} and {account}; n_i=2 as both entity sets contain terms that have id attributes.

5.3 Aligning Entity Sets

Recall that an ELR is a pair of associated entity sets $< ES_i^{\mathcal{S}}, ES_i^{\mathcal{T}} >$, for which we need to derive a similarity measure using the vector space model. Section 5.1 has described how sets of terms can be derived from entity sets. In essence, if an entity set represents a single entity type, it is represented as a set of terms the members of which correspond to the name of the entity type and the names of its attributes. If an entity set represents several entity types, then two alternative sets of terms are generated, representing the alternative interpretations that the entity set is horizontally or vertically partitioned. In all cases, we need to align the sets of terms representing $ES_i^{\mathcal{S}}$ and $ES_i^{\mathcal{T}}$, so that their similarities can be measured.

Definition 5 ***Matched Terms.*** *Assume there are two sets of terms* $TS^{\mathcal{S}} = \{t_0^{\mathcal{S}}, ..., t_q^{\mathcal{S}}\}$ *and* $TS^{\mathcal{T}} = \{t_0^{\mathcal{T}}, ..., t_r^{\mathcal{T}}\}$. *For any two terms* $t_i^{\mathcal{S}}$ *and* $t_j^{\mathcal{T}}$, *the* **term similarity** $ts(t_i^{\mathcal{S}}, t_j^{\mathcal{T}})$ *is defined as the (average) match similarity of the two (sets of) constructs that* $t_i^{\mathcal{S}}$ *and* $t_j^{\mathcal{T}}$ *represent.* $t_i^{\mathcal{S}}$ *and* $t_j^{\mathcal{T}}$ *are called* **matched terms** *if they satisfy the following conditions: 1)* $ts(t_i^{\mathcal{S}}, t_j^{\mathcal{T}}) > 0$; *2)* $ts(t_i^{\mathcal{S}}, t_j^{\mathcal{T}}) > ts(t_i^{\mathcal{S}}, t_{j'}^{\mathcal{T}})$ $(0 \leqslant j' \leqslant n$ *and* $j \neq j')$; *and 3)* $t_{i'}^{\mathcal{S}}$ $(0 \leqslant i' \leqslant m$ *and* $i' \neq i)$ *and* $t_j^{\mathcal{T}}$ *are not matched terms.*

For example, let $TS_1^{\mathcal{S},H}$ represent the terms for the candidate horizontal partitioning of the entity set {home_cust, overseas_cust} in $\mathcal{P}_1$, and let $TS_1^{\mathcal{T},V}$ represent the terms for the candidate vertical partitioning of the entity set {customer, cust_phone} in $\mathcal{P}_1$. The matched terms (denoted by $\Leftrightarrow$ below) between $TS_1^{\mathcal{S},H}$ and $TS_1^{\mathcal{T},V}$ are presented in Table 2.

Assume that there are $m+1$ matched terms between $TS_i^{\mathcal{S}}$ and $TS_i^{\mathcal{T}}$. We reorder terms in $TS_i^{\mathcal{S}}$ and $TS_i^{\mathcal{T}}$, so that the matched terms occupy the first $m+1$ positions, and assign the first $m+1$ positions of the vectors $\boldsymbol{v}_i^{\mathcal{S}}$ and $\boldsymbol{v}_i^{\mathcal{T}}$ with the tf/idf weights of the corresponding terms. Then for the a unmatched terms in $\boldsymbol{v}_i^{\mathcal{S}}$ we insert the tf/idf weights of the terms to the next a positions in $\boldsymbol{v}_i^{\mathcal{S}}$ and 0 into the next a positions in $\boldsymbol{v}_i^{\mathcal{T}}$. Then for the b unmatched terms in $\boldsymbol{v}_i^{\mathcal{T}}$ we insert the tf/idf weights of the terms to the next b posititions in $\boldsymbol{v}_i^{\mathcal{T}}$ and 0 into the next b positions in $\boldsymbol{v}^{\mathcal{S}}$, to give:

Table 2. Matched terms between $TS_1^{\mathcal{S},H}$ and $TS_1^{\mathcal{T},V}$

$TS_1^{\mathcal{S},H}$		$TS_1^{\mathcal{T},V}$
{home_cust, overseas_cust}	⟺	{customer, cust_phone}
{home_cust.id, overseas_cust.id}	⟺	{customer.key, cust_phone.key}
{home_cust.name, overseas_cust.name}	⟺	{customer.c_fname}
		{customer.c_lname}
{home_cust.birth, overseas_cust.birth}	⟺	{customer.c_birth}
{home_cust.phone, overseas_cust.phone}	⟺	{cust_phone.c_phone}

$$\boldsymbol{v}^{\mathcal{S}} = (w_0^{\mathcal{S}}, ..., w_m^{\mathcal{S}}, w_{m+1}^{\mathcal{S}}, ..., w_{m+a}^{\mathcal{S}}, w_{m+a+1}^{\mathcal{S}}, ...w_{m+a+b}^{\mathcal{S}})$$

where $w_{m+a+1}^{\mathcal{S}}$ to $w_{m+a+b}^{\mathcal{S}}$ all have the value 0, indicating that there is no term in $TS_i^{\mathcal{S}}$ associated with those positions in the vector, and

$$\boldsymbol{v}^{\mathcal{T}} = (w_0^{\mathcal{T}}, ..., w_m^{\mathcal{T}}, w_{m+1}^{\mathcal{T}}, ..., w_{m+a}^{\mathcal{T}}, w_{m+a+1}^{\mathcal{T}}, ...w_{m+a+b}^{\mathcal{T}})$$

where $w_{m+1}^{\mathcal{T}}$ to $w_{m+a}^{\mathcal{T}}$ all have the value 0 indicating that there is no term in $TS_i^{\mathcal{T}}$ associated with those positions in the vector.

5.4 Vector Similarity

The cosine of the angle between vectors is widely used as a similarity function in information retrieval [35]. Building on this approach, the similarity of vectors $\boldsymbol{v}^{\mathcal{S}}$ and $\boldsymbol{v}^{\mathcal{T}}$ is computed using Function 1.

$$sim(\boldsymbol{v}^{\mathcal{S}}, \boldsymbol{v}^{\mathcal{T}}) = \frac{\sum_{i=0}^{k} ts(t_i^{\mathcal{S}}, t_i^{\mathcal{T}}) \times w_i^{\mathcal{S}} \times w_i^{\mathcal{T}}}{\sqrt{\sum_{i=0}^{k} (w_i^{\mathcal{S}})^2} \times \sqrt{\sum_{i=0}^{k} (w_i^{\mathcal{T}})^2}} \tag{1}$$

where k is the length of the vectors $\boldsymbol{v}^{\mathcal{S}}$ and $\boldsymbol{v}^{\mathcal{T}}$, $t_i^{\mathcal{S}}$ (resp., $t_i^{\mathcal{T}}$) are the terms corresponding to the weights $w_i^{\mathcal{S}}$ (resp., $w_i^{\mathcal{T}}$), and ts is a function that derives the similarity of the terms from match evidence. The introduction of the term similarity measure into the standard cosine function is to allow for the fact that matched terms may have been matched with different scores.

Function 1 can be used to infer the partitioning type of an *ELR*. For example, ELR_1=⟨{home_cust, overseas_cust},{customer, cust_phone}⟩ $\in \mathcal{P}_1$ is represented as a pair of source vectors (i.e, $\boldsymbol{v}_1^{\mathcal{S},H}$ and $\boldsymbol{v}_1^{\mathcal{S},V}$) and a pair of target vectors (i.e, $\boldsymbol{v}_1^{\mathcal{T},H}$ and $\boldsymbol{v}_1^{\mathcal{T},V}$). We calculate $s_{H:H} = sim(\boldsymbol{v}_1^{\mathcal{S},H}, \boldsymbol{v}_1^{\mathcal{T},H})$, $s_{H:V} = sim(\boldsymbol{v}_1^{\mathcal{S},H}, \boldsymbol{v}_1^{\mathcal{T},V})$, $s_{V:H} = sim(\boldsymbol{v}_1^{\mathcal{S},V}, \boldsymbol{v}_1^{\mathcal{T},H})$ and $s_{V:V} = sim(\boldsymbol{v}_1^{\mathcal{S},V}, \boldsymbol{v}_1^{\mathcal{T},V})$ to denote the fitness of ELR_1 that are *HP* and *HP*, *HP* and *VP*, *VP* and *HP*, and *VP* and *VP*, respectively. We consider the similarity of ELR_1 as the maximum value

of $s_{H:H}$, $s_{H:V}$, $s_{V:H}$, and $s_{V:V}$, and report the corresponding partitioning type. The vector similarity function alone doesn't capture all the properties of the partitioning types; where different partitioning strategies give rise to the same vector similarity, heuristics can be used to establish the most suitable type.

5.5 Aggregation

Given a candidate phenotype $\mathcal{P} = \{ELR_1, ..., ELR_n\}$, each $ELR_i \in \mathcal{P}$ is assigned a similarity, denoted as s_i, as presented in Section 5.4. In this section, we describe the last step of the objective function that calculates the fitness value for the phenotype $\mathcal{P}$ by aggregating the *ELR* similarities. We expect the aggregation function (Function 2) to assign a higher value to a phenotype if more *ELRs* with comparatively high similarities are contained in the phenotype. Note that we do not define a similarity threshold that arbitrarily denotes whether the pair of entity sets associated by an *ELR* is similar or not, and that we do not assume in advance that we know how many *ELRs* should be contained in a solution.

$$
\begin{aligned}
&sum(s_i \times c_i) \times average(s_i \times c_i) \\
&= [\sum_{i=1}^{n}(s_i \times c_i)] \times \frac{\sum_{i=1}^{n}(s_i \times c_i)}{\sum_{i=1}^{n} c_i} = \frac{(\sum_{i=1}^{n} s_i \times c_i)^2}{\sum_{i=1}^{n} c_i}
\end{aligned}
\tag{2}
$$

where s_i is the similarity of $ELR_i = <ES_i^{\mathcal{S}}, ES_i^{\mathcal{T}}>$, and $c_i = avg(|ES_i^{\mathcal{S}}|, |ES_i^{\mathcal{T}}|)$ is its coverage, where $|ES_i^{\mathcal{S}}|$ and $|ES_i^{\mathcal{T}}|$ represent the number of entities in the sets $ES_i^{\mathcal{S}}$ and $ES_i^{\mathcal{T}}$, respectively.

6 Attribute-Level Schematic Correspondences

As presented in Sections 4 and 5, a phenotype $\mathcal{P} = \{ELR_1, ..., ELR_n\}$ that has the highest fitness value is returned. Each $ELR_i \in \mathcal{P}$ maps to an entity-level schematic correspondence. Inferring attribute-level schematic correspondences is a post-processing step after the evolutionary search.

Similar to the entity level, we use a set of Attribute-Level Relationships (*ALRs*) to model the set of attribute-level schematic correspondences, and require that each two *ALRs* identified between the pair of entity sets in $ELR_i \in \mathcal{P}$ are disjoint, as indicated in Definition 6.

Definition 6 ***Attribute-Level Relationships.*** *Given schemas $\mathcal{S}$ and $\mathcal{T}$ and an $ELR = \langle ES^{\mathcal{S}}, ES^{\mathcal{T}} \rangle$, a set of Attribute-Level Relationships (***ALRs***), i.e., $\{ALR_1, ..., ALR_n\}$, indicates a set of attribute-level schematic correspondences between the entity sets $ES^{\mathcal{S}}$ and $ES^{\mathcal{T}}$. $ALR_i = \langle AS_i^{\mathcal{S}}, AS_i^{\mathcal{T}} \rangle$, $i = 1, ..., n$, where $AS_i^{\mathcal{S}}$ and $AS_i^{\mathcal{T}}$ are source and target attribute sets with cardinalities greater than or equal to 1, satisfy both: i) any attribute $\in AS_i^{\mathcal{S}}$ is also an attribute of an entity $\in ES^{\mathcal{S}}$, and any attribute $\in AS_i^{\mathcal{T}}$ is also an attribute of an entity $\in ES^{\mathcal{T}}$; and ii) for each $ALR_{i'} = \langle AS_{i'}^{\mathcal{S}}, AS_{i'}^{\mathcal{T}} \rangle$, $i' = 1, ..., n$ such that $i' \neq i$, $AS_i^{\mathcal{S}} \cap AS_{i'}^{\mathcal{S}} = \emptyset$ and $AS_i^{\mathcal{T}} \cap AS_{i'}^{\mathcal{T}} = \emptyset$.*

The method for inferring *ALRs* is quite straightforward. Assume $\mathcal{P}_1$ is the phenotype returned from the evolutionary search. Let us use $ELR_1 \in \mathcal{P}_1$=⟨{home_cust, overseas_cust},{customer, cust_phone}⟩ as an example. Recall Table 2, which illustrates an *HP:VP* entity correspondence. The matched terms identified between $TS_1^{\mathcal{S},H}$ and $TS_1^{\mathcal{T},V}$ are shown below.

ALR_1=⟨{home_cust.id, overseas_cust.id}, {customer.key, cust_phone.key}⟩,
ALR_2=⟨{home_cust.name, overseas_cust.name}, {customer.c_fname}⟩,
ALR_3=⟨{home_cust.birth, overseas_cust.birth}, {customer.c_birth}⟩,
ALR_4=⟨{home_cust.phone, overseas_cust.phone}, {cust_phone.c_phone}⟩.

EvoMatch also includes a simple algorithm for identifying complex attribute n-to-1 correspondences for each $ELR_i \in \mathcal{P}$ (i.e., $Type_4^c$). Similar to previous methods, e.g., iMAP [4], *EvoMatch* uses formulae to transform instances of a set of attributes into instances of a single attribute, thus inferring the complex n-to-1 attribute correspondences. The method currently covers common formulae enumerated by Kim *et al.* [14], such as concat (*first_name*, *last_name*) = *name* and concat (*day*/*month*/*year*) = *date*, which are extendable. Thus, a new *ALR*=

⟨{home_cust.name, overseas_cust.name}, {customer.c_fname, customer.c_lname}⟩

is identified to replace ALR_2 above.

7 Experimental Evaluation

This section presents experimental studies of the evolutionary search method, which evaluate its effectiveness for inferring schematic correspondences using the collection of MatchBench scenarios[1] and a pair of real world relational databases provided by the Amalgam benchmark [36]. In particular, our method is compared with COMA++ [2], Similarity Flooding [3] and Harmony [17] matchers.

7.1 Experimental Setting

MatchBench is a benchmark for evaluating the effectiveness of schema matching systems for inferring the schematic correspondences of Kim *et al.* [14]. To generate the scenarios, it starts from source and target relational databases where a pair of entities (i.e., tables) are exactly the same, and systematically injects different types of schematic correspondences into the equivalent entities.

In this paper, we show the results of *EvoMatch* for three collections of scenarios. In collection C_1, called *1-to-1 equivalent entities scenarios*, the 1-to-1 entities are equivalent, but entity (attribute) names are changed or some attributes are removed, combined with changes at the instance-level. In collection C_2, called *n-to-m equivalent entities scenarios*, the equivalent 1-to-1 entities are horizontally or vertically partitioned into equivalent entity sets, combined with renaming attributes. Collection C_3, called *n-to-1 equivalent attributes scenarios*, offers scenarios where a set of attributes and a single attribute represent the

[1] Available from `http://code.google.com/p/matchbench/`

same real world information. The numbers of ground truth correspondences of $Type^c_1$ to $Type^c_4$ in collections C_1 to C_3 are presented in Table 3.

The Amalgam benchmark [36] contains real world relational databases from the bibliographic domain devised by different designers. We asked three experts who have a good understanding of the bibliographic domain and schematic heterogeneities to manually identify ground truth schematic correspondences between pairs of Amalgam databases. We chose to evaluate our method on a pair that represents most types of schematic heterogeneities defined in Section 3. The numbers of $Type^c_1$ to $Type^c_4$ correspondences in Amalgam are presented in Table 3 as well.

The effectiveness of different systems is measured by comparing their results with the ground truth (correct correspondences), thus determining the true positives (TP), i.e., correspondences correctly identified; the false positives (FP), i.e., correspondences incorrectly identified; and the false negatives (FN), correspondences incorrectly missed. Given cardinalities of the above sets, we follow the standard definitions to evaluate the effectiveness:

- $Precision = \frac{|TP|}{|TP|+|FP|}$ specifies the fraction of correct correspondences among all detected correspondences;
- $Recall = \frac{|TP|}{|TP|+|FN|}$ specifies the fraction of correct correspondences among all detectable correspondences;
- $F\text{-}measure = 2 * \frac{Precision * Recall}{Precision + Recall}$ is the harmonic mean of *Precision* and *Recall*.

7.2 Settings of Systems

Our method for inferring schematic correspondences takes as input two schemas and different sets of matches denoting various types of element similarity, e.g., names, instances or data types, between the two schemas. In particular, we chose the *Name* and *Content-based* matchers of COMA++ [2], which have been demonstrated to be effective [2][37] and are publicly available, to compute the name similarities and the instance similarities of elements, because these two types of similarities mostly represent the commonalities and differences between the elements. Note that other types of similarity evidence, e.g., data type or synonym similarity, can easily be added for further support, although they are not applied in this paper.

By investigating the two matchers, we have noticed that they match almost all pairs of elements of the schemas even though some elements have no commonalities. We did a sensitivity analysis and decided to apply a threshold of 0.3 for both *Name* and *Content-based* matchers. This is not because our method requires them but because the two matchers provided by COMA++ associate attributes that have no commonalities with low similarity scores, which results in a large number of input matches, and therefore a large search space. The threshold of 0.3 for the matchers remains the same for all scenario contexts.

Running an evolutionary search (specifically, using a genetic algorithm) requires parameters (Section 4). We follow suggestions from literature (e.g., [32]), and thus set population size as 30, offspring size as 30, mutation rate p_m as $1/n$ (n represents the length of the genotype), and crossover rate p_c as 0.9. We have also carried out a sensitivity analysis and concluded that results are not highly sensitive to these values, or to the specific (random) initial generation used. Usually, if the search goes through more generations, it is more likely to obtain the global optimal solution. Therefore, for the small and medium scale schemas, we terminate the search when 500 generations have been produced.

To evaluate the effectiveness of *EvoMatch*, we compare it with three well-known and publicly available matching systems. We mainly follow the advice of authors to configure the systems and do what we can to help them perform well.

For COMA++ [2], we applied *AllContext* as the matching strategy, selected matchers *Name*, *NamePath*, *Leaves* and *Parents* at the schema-level and *Content-based* at the instance-level, and employed *Average* for aggregation, *Both* for direction, *Threshold+MaxDelta* for selection and *Average* for combination, as they are demonstrated to be effective in published experimental evaluations [2]. As experience with COMA++ has not given rise to consistent recommendations for Threshold and Delta [38][2], we decided to employ the default settings of Threshold and Delta (i.e., 0.1 + 0.01) that are provided with the COMA++ tool.

Initial matches of Similarity Flooding (SF) [3] are produced by combining results of its own matcher (i.e., the *NGram* matcher) and an instance matcher (i.e., the *Content-based* matcher of COMA++), rather than using its own matcher alone, which acts only on schema level data. This approach enables Similarity Flooding to make use of instance-level information for the initial matches, which turns out to be important for identifying schematic correspondences.

Harmony [17] is a schema matching tool provided in OpenII, whose *EditDistance*, *Documentation* and *Exact* matchers are chosen for the evaluation. Given candidate matches, we select the top matches associated with each element as result matches while not restricting the number of matches associated with an element, as recommended by the authors. As Harmony only works at the schema level, we combine it with the *Content-based* matcher of COMA++, to provide the same basis in terms of instance-based matches as COMA++, Similarity Flooding and *EvoMatch*.

These matching systems are principally designed for identifying $Type_1^s$ and $Type_2^s$ correspondences, but are chosen to compare with *EvoMatch* because they are the publicly available systems for identifying correspondences. Thus, we did an extra step on top of the 1-to-1 matches identified by these systems to derive $Type_1^c$ to $Type_4^c$ correspondences.

A pair of n-to-m elements is derived as a simple n-to-m correspondence (i.e., $Type_3^s$ or $Type_4^s$), if it satisfies conditions: i) there exists a match between all pairs of elements; and ii) there does not exist an additional element, so that the pair of (n+1)-to-m or n-to-(m+1) elements satisfies condition i). Thus, as these systems provide no mechanisms for establishing the horizontal or vertical

partitioning, we simply consider the derived $Type_3^s$ and $Type_4^s$ correspondences as $Type_3^c$ and $Type_4^c$ correspondences, and assume that any correspondence of the correct size is of the correct type. In experiments, EvoMatch is set the more challenging task of also identifying the correct type. The remaining $Type_1^s$ and $Type_2^s$ correspondences not used for deriving $Type_3^c$ and $Type_4^c$ correspondence are considered as $Type_1^c$ and $Type_2^c$ correspondences, because identifying whether equivalent entities or attributes have the same name is an easy task for them.

7.3 Experimental Results

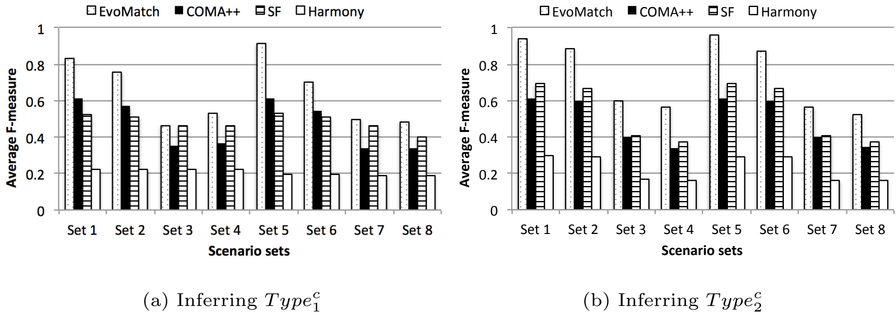
Table 3 presents the number of ground truth values for each type of schematic correspondence (i.e., $Type_1^c$ to $Type_4^c$) in MatchBench scenario collections C_1 to C_3 and in an Amalgam scenario, respectively. Table 3 also lists the number of correspondences correctly identified, i.e., true positives, by *EvoMatch*, COMA++, Similarity Flooding (SF) and Harmony.

Table 3. The number of ground truth and the number of true positives in each type of schematic correspondence

	Types		Ground Truth	EvoMatch	COMA++	SF	Harmony
MatchBench	C_1	$Type_1^c$	207	162	147	183	87
		$Type_2^c$	1479	1366	1267	1213	542
	C_2	$Type_3^c$	96	72	15	0	18
		$Type_2^c$	992	838	703	389	355
	C_3	$Type_4^c$	9	2	0	0	4
Amalgam	$Type_1^c$		1	1	1	1	0
	$Type_2^c$		10	10	2	2	0
	$Type_3^c$		1	1	0	0	0
	$Type_4^c$		0	0	0	0	0

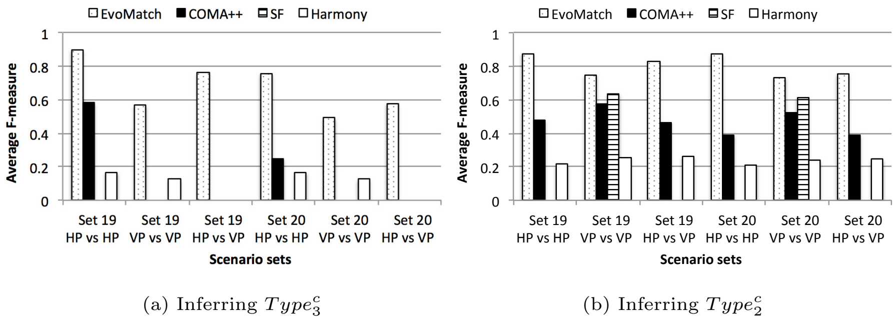
The F-measures of the four systems for inferring schematic correspondences in MatchBench scenario collections C_1 and C_2 are presented in Fig. 6 and 7. In Fig. 6(a) and (b), the 1-to-1 equivalent entity scenarios in collection C_1 are categorized into scenario sets *1* to *8*; in Fig. 7(a) and (b), the n-to-m equivalent entity scenarios in collection C_2 are in sets *19* and *20*, whose particular partitioning types, i.e., horizontal partitioning (HP) and vertical partitioning (VP), are further specified. In the Amalgam scenario, the F-measure of entity-level schematic correspondences ($Type_1^c$ and $Type_3^c$) for *EvoMatch*, COMA++, SF and Harmony is 0.667, 0.5, 0.39 and 0.0, respectively; the F-measure of attribute-level schematic correspondences ($Type_2^c$) is 0.93, 0.17, 0.16 and 0.0, respectively.

In terms of *the expressiveness of the correspondences*, *EvoMatch* has advantages in relation to the other three systems in that it provides direct support for n-to-m correspondences. As shown in Table 3, *EvoMatch* has identified a larger number of true positives than COMA++, SF and Harmony in most scenarios.



(a) Inferring $Type_1^c$ (b) Inferring $Type_2^c$

Fig. 6. F-measure in MatchBench C_1 scenarios



(a) Inferring $Type_3^c$ (b) Inferring $Type_2^c$

Fig. 7. F-measure in MatchBench C_2 scenarios

This is because *EvoMatch* is designed for inferring more expressive correspondences, i.e., schematic correspondences, than the three systems, whereas the other systems associate 1-to-1 elements whose features, e.g., names or instances, are same or similar and typically support a greater range of correspondence types.

The other three systems are generally designed for identifying 1-to-1 matches, especially SF that only identifies the 1-to-1 top match for each element, and therefore have been reasonably successful in $Type_1^c$ and $Type_2^c$. However, *EvoMatch* has still been able to identify more $Type_1^c$ and $Type_2^c$ true positives than them, because *EvoMatch* assigns distinguishable similarity scores between pairwise equivalent entities and pairwise different entities that coincidentally have similar attributes using the vector space model (VSM), and uses an aggregation function to select equivalent entities.

In contrast, COMA++ chooses limited top matches above a threshold for each element, and thus removes a few true positives. Harmony does not restrict the number of matches associated with an element and keeps a candidate match as long as it is the top match for either of its associated elements, and therefore

more matches are inferred as n-to-m correspondences rather than 1-to-1. *EvoMatch* has performed satisfactorily in inferring $Type_3^c$, thus indicating that the evolutionary algorithm is effective at fulfilling its purpose. Again, as the three systems compared are not designed for inferring $Type_3^c$, their performance is rather patchy in identifying these correspondences; SF fails completely because it only identifies 1-to-1 correspondences.

EvoMatch has not performed well in $Type_4^c$ correspondences because the input match evidence used for the inference is not sufficient, whereas Harmony has identified a few $Type_4^c$ when the n attributes and the *1* attribute have similar names, because their matches are usually the top matches for the n attributes, thus being selected. However, this result does not indicate that Harmony has been able to identify $Type_4^c$ correspondences, as the match evidence that OpenII uses comes from attribute names rather than instances.

In terms of *overall effectiveness*, higher F-measures are reported for *EvoMatch* (Fig. 6 and 7). In addition to identifying more true positives, *EvoMatch* has also been able to effectively remove entity-level false positives without using thresholds. This is because: (i) following the intuition of VSM the objective function assigns rather low similarities for false positives between different (sets of) entities; (ii) when the similarity between equivalent entities is high, the aggregation function is able to assign a lower fitness value to a solution that contains both false positives and the true positive than to the solution that only contains the true positive, thus helping to remove false positives.

COMA++ relies on a threshold to remove false positives, and thus those with similarity scores above the threshold are kept. SF tries to associate each element with a 1-to-1 match, even for elements that do not have equivalent elements. As stated above, Harmony keeps top matches of an element, thus resulting in a large number of false positives.

EvoMatch has also outperformed the other systems in inferring attribute-level correspondences ($Type_2^c$). It follows a top-down process, and identifies equivalent attributes only between entities that have been identified as being equivalent, therefore, once equivalent entities are correctly associated, the chances that equivalent attributes can be matched are high. This approach also helps to exclude false positives that associate attributes in unassociated entities.

8 Conclusions

This paper has presented a method, *EvoMatch*, for inferring schematic correspondences between source and target schemas using an evolutionary search, specifically a genetic algorithm. *EvoMatch* utilizes matches denoting similarity of schema elements and searches for a particular set of entity-level relationships (*ELRs*) as entity-level schematic correspondences. For each derived *ELR*, *EvoMatch* further identifies a set of attribute-level relationships (*ALRs*) as attribute-level correspondences.

The paper contributes to: i) an evolutionary search method for inferring entity-level schematic correspondences, ii) an objective function for modeling the requirement of entity-level schematic correspondences, and iii) an experimental evaluation that demonstrates the effectiveness of *EvoMatch*. To the best of our knowledge, this is the first work for inferring complex n-to-m entity-level correspondences without using specific schema information (e.g., integrity constraints), external resources (e.g., ontology) and context-sensitive rules (e.g., thresholds) and without requiring user engagement. In contrast, existing methods typically identify 1-to-1 complex correspondences (e.g., *SeMap* [20]) or require specific schema information to identify n-to-m complex correspondences (e.g., Xu *et al.* [23])

EvoMatch differs from the three evaluated schema matching systems in the focus of *EvoMatch* on the inference of mutually consistent higher-level correspondences from lower-level matches. In support of this higher-level perspective, the objective function is able to select collections of matches that together represent coherent correspondences, rather than using more primitive match selection methods that act more independently (in COMA++ or Harmony) or less purposefully (as in Similarity Flooding). The objective function incorporates several features that resulted from empirical evaluation with real and synthetic data sets: the use of tf/idf was important for keeping apart entity types that have commonly recurring attributes (such as name or address), and the aggregation technique in Section 5.5 judiciously trades-off coverage and similarity.

EvoMatch and schema mapping tools, e.g., Clio [5], are similar in using matches as evidence to infer expressive relationships of schemas. However, such tools require user participation and schema constraints to compensate for the limitations of the matchings. Although *EvoMatch* does not specify declarative expressions at the same level as such tools, by inferring more information than the underlying matches it provides a richer foundation on which mapping tools can build. As an interesting future work, EvoMatch can be extended to on the large scale data using MapReduce techniques [39][40][41][42].

References

1. Halevy, A.Y., Rajaraman, A., Ordille, J.J.: Data integration: The teenage years. In: VLDB, pp. 9–16 (2006)
2. Do, H., Rahm, E.: Matching large schemas: Approaches and evaluation. Information Systems 32(6), 857–885 (2007)
3. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and itsapplication to schema matching. In: ICDE, pp. 117–128 (2002)
4. Dhamankar, R., Lee, Y., Doan, A., Halevy, A.Y., Domingos, P.: imap: Discovering complex mappings between database schemas. In: SIGMOD Conference, pp. 383–394 (2004)
5. Fagin, R., Haas, L.M., Hernández, M., Miller, R.J., Popa, L., Velegrakis, Y.: Clio: Schema mapping creation and data exchange. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 198–236. Springer, Heidelberg (2009)

6. Bonifati, A., Chang, E.Q., Ho, T., Lakshmanan, L.V.S., Pottinger, R., Chung, Y.: Schema mapping and query translation in heterogeneous p2p xml databases. VLDB J. 19(2), 231–256 (2010)
7. Bonifati, A., Mecca, G., Pappalardo, A., Raunich, S., Summa, G.: Schema mapping verification: the spicy way. In: EDBT, pp. 85–96 (2008)
8. Marnette, B., Mecca, G., Papotti, P., Raunich, S., Santoro, D.: ++spicy: an opensource tool for second-generation schema mapping and data exchange. PVLDB 4(12), 1438–1441 (2011)
9. Franklin, M., Halevy, A., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Record 34(4), 27–33 (2005)
10. Belhajjame, K., Paton, N.W., Embury, S.M., Fernandes, A.A.A., Hedeler, C.: Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In: EDBT, pp. 573–584 (2010)
11. Salles, M.A.V., Dittrich, J.-P., Karakashian, S.K., Girard, O.R., Blunschi, L.: itrails: Pay-as-you-go information integration in dataspaces. In: VLDB, pp. 663–674 (2007)
12. Sarma, A.D., Dong, X., Halevy, A.Y.: Bootstrapping pay-as-you-go data integration systems. In: SIGMOD Conference, pp. 861–874 (2008)
13. Mao, L., Belhajjame, K., Paton, N.W., Fernandes, A.A.A.: Defining and using schematic correspondences for automatically generating schema mappings. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 79–93. Springer, Heidelberg (2009)
14. Kim, W., Seo, J.: Classifying schematic and data heterogeneity in multidatabase systems. IEEE Computer 24(12), 12–18 (1991)
15. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. PVLDB 4(11), 695–701 (2011)
16. Cafarella, M.J., Halevy, A.Y., Wang, D.Z., Wu, E., Zhang, Y.: Webtables: exploring the power of tables on the web. PVLDB 1(1), 538–549 (2008)
17. Smith, K., Morse, M., Mork, P., Li, M.H., Rosenthal, A., Allen, D., Seligman, L.: The role of schema matching in large enterprises. In: CIDR (2009)
18. Kang, J., Naughton, J.F.: Schema matching using interattribute dependencies. IEEE Trans. Knowl. Data Eng. 20(10), 1393–1407 (2008)
19. Bilke, A., Naumann, F.: Schema matching using duplicates. In: ICDE, pp. 69–80 (2005)
20. Wang, T., Pottinger, R.: Semap: a generic mapping construction system. In: EDBT, pp. 97–108 (2008)
21. Giunchiglia, F., Yatskevich, M., Shvaiko, P.: Semantic matching: Algorithms and implementation. J. Data Semantics 9, 1–38 (2007)
22. Rizopoulos, N.: Automatic discovery of semantic relationships between schema elements. In: ICEIS, vol. (1), pp. 3–8 (2004)
23. Xu, L., Embley, D.W.: A composite approach to automating direct and indirect schema mappings. Inf. Syst. 31(8), 697–732 (2006)
24. Dai, B.T., Koudas, N., Srivastava, D., Tung, A.K.H., Venkatasubramanian, S.: Validating multi-column schema matchings by type. In: ICDE, pp. 120–129 (2008)
25. Warren, R.H., Tompa, F.W.: Multi-column substring matching for database schema translation. In: VLDB, pp. 331–342 (2006)
26. Miller, G.A.: Wordnet: A lexical database for english, Commun. ACM 38(11), 39–41 (1995)
27. Elmeleegy, H., Ouzzani, M., Elmagarmid, A.K.: Usage-based schema matching. In: ICDE, pp. 20–29 (2008)

28. Madhavan, J., Bernstein, P.A., Doan, A., Halevy, A.Y.: Corpus-based schema matching. In: ICDE, pp. 57–68 (2005)
29. Haas, L., Hernández, M., Ho, H., Popa, L., Roth, M.: Clio grows up: from research prototype to industrial tool. In: ACM SIGMOD, pp. 805–810 (2005)
30. Alexe, B., Chiticariu, L., Miller, R.J., Tan, W.C.: Muse: Mapping understanding and design by example. In: ICDE, pp. 10–19 (2008)
31. Ozsu, M.T., Valduriez, P.: Principles of distributed database systems. Addison-Wesley, Reading (1989)
32. Eiben, A., Smith, J.: Introduction to evolutionary computing. Springer (2003)
33. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Comput. Surv. 35(3), 268–308 (2003)
34. Michalewicz, Z., Fogel, D.: How to solve it: modern heuristics. Springer-Verlag New York Inc. (2004)
35. Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern information retrieval. ACM Press, New York (1999)
36. Miller, R.J., Fisla, D., Huang, M., Kymlicka, D., Ku, F., Lee, V.: The Amalgam Schema and Data Integration Test Suite (2001), `http://www.cs.toronto.edu/~miller/amalgam`
37. Engmann, D., Maßmann, S.: Instance matching with coma++. In: BTW Workshops, pp. 28–37 (2007)
38. Massmann, S., Engmann, D., Rahm, E.: Coma++: Results for the ontology alignment contest oaei, Ontology Matching (2006)
39. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5(8), 716–727 (2012)
40. Yuan, P., Sha, C., Wang, X., Yang, B., Zhou, A., Yang, S.: Xml structural similarity search using mapreduce. In: Chen, L., Tang, C., Yang, J., Gao, Y. (eds.) WAIM 2010. LNCS, vol. 6184, pp. 169–181. Springer, Heidelberg (2010)
41. Kolb, L., Thor, A., Rahm, E.: Load balancing for mapreduce-based entity resolution. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 618–629. IEEE (2012)
42. Ma, Q., Yang, B., Qian, W., Zhou, A.: Query processing of massive trajectory data based on mapreduce. In: CloudDb, pp. 9–16 (2009)

Update Management in Decision Support Systems*

Haitang Feng[1,2], Nicolas Lumineau[1], Mohand-Saïd Hacid[1], and Richard Domps[2]

[1] Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622, France
firstname.lastname@liris.cnrs.fr
[2] Anticipeo, 4 bis impasse Courteline, 94800, Villejuif, France
firstname.lastname@anticipeo.com

Abstract. Forecasting is the process of making statements about events whose actual outcomes have not yet been observed. It is used for decades in different fields like climate, crime, health, business... Although the purpose of different forecasting systems is not the same, in general, they help decision-makers to make appropriate plans for future likely events. As the nature of forecasting methods and measures are often quantitative, these predictive analytics systems usually use a data warehouse to store data and OLAP tools to visualize query/simulation results. A specific feature of forecasting systems regarding predictions analysis is backward propagation of updates, **which is the computation of the impact, on raw data, of modifications performed on summaries**. In data warehouses, some methods propagate updates over hierarchies when modifications are performed on data sources. However, so far, very few works have been devoted to update propagation from summaries to raw data. This paper proposes an algorithm called PAM (Propagation of Aggregate-based Modification), to efficiently propagate modifications performed on summaries to raw data, and then to other summaries. Experiments have been conducted on an operational application[1].

Keywords: OLAP, Forecasting applications, Update propagation, Materialized views, Decision support systems.

1 Introduction

A forecasting system is a set of techniques or tools that are used for analysis of historical data, selection of most appropriate modeling structure for the computation of forecasts, model validation, development of forecasts, and monitoring and adjustment of forecasts[2].

* Research partially supported by the French Agency ANRT (www.anrt.asso.fr) and Anticipeo (www.anticipeo.com)

[1] A sales forecasting system called Anticipeo (http://www.anticipeo.com)

[2] See http://www.businessdictionary.com/definition/forecasting-system.html

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 27–53, 2013.
© Springer-Verlag Berlin Heidelberg 2013

In daily life, various forecasting systems are used in many areas. We introduce some important forecasting systems in the following paragraphs.

Environmental forecasting is one of the most frequently and earliest used forecasting application. Many countries and trans-boundary agencies achieve predictions with derived statistical models specific to their domains. For instance, the European Center for Medium-Range Weather Forecasts [5] provides operational medium- and extended-range forecasts and a state-of-the-art super-computing facility for scientific research. The National Centers for Environmental Prediction [24] of the United States provides national and global weather, water, climate and space weather guidance, forecasts, warnings and analyses to their partners and external user communities. Environmental forecasting systems respond to user needs to protect life and property, enhance the nation's economy and support the nation's growing need for environmental information.

Another well-known forecasting system is used for the traffic estimation and prediction to improve traffic conditions and reduce travel delays by facilitating the utilization of available transportation facilities. Singapore is the first country in the world that implemented the practical application of congestion pricing that is currently based on [22]. The objective is to be able to predict the levels of congestion over preset durations (from ten minutes up to an hour) in advance [14].

Other forecasting systems appeared more recently to respond to new demands. Tourism forecasting systems provide forecasts of tourism demand, which are prerequisites to the decision-making process in the organizations of the private or public sector, involved in the tourism industry, helping decision-makers to plan more effectively and efficiently [27]. Stock forecasting systems [30] and sales forecasting systems are among useful financial forecasting systems for investors and enterprise managers to reduce logistics and stock cost and to improve the income of enterprises.

The field of forecasting is concerned with approaches to determining what the future holds. It is also concerned with the proper presentation and use of forecasts. The terms "forecast", "prediction", "projection", "plan", and "prognosis" are typically used interchangeably. The field of forecasting includes the study and application of judgment as well as of quantitative (statistical) methods[1].

The basic functionalities a forecasting system supports are: **computation**, **visualization** and **modification**. The first functionality, computation of forecasts, uses specific methods (typically statistical models) to derive forecasts. The second functionality, visualization of computed forecasts, uses OLAP (online analytical processing) tools to visualize and interact with data stored in warehouses. However, the third functionality, modification of computed forecasts during visualization, is a specific problem which is not well investigated in the data warehousing environment. In forecasting systems, raw data are composed of historical data and predictive data. Unlike historical data which represent achieved facts and do not evolve over time, predictive data can be dynamic and can be updated. Experts of the domain could make some modifications to adjust computed forecasts to some specific situations. They could also make

some simulations in order to visualize an objective. These modifications occur on summarized data and should be propagated to raw data (computed forecasts) and then to other summarized data. This process implies two directions of modifications. However, the work in the data warehouse domain focuses only on propagating source data modification to summarized data.

Related Work. The focus of our work is on how to efficiently propagate an update performed on a summary to all data, including raw data and other summaries computed from the update summary. More precisely, the problem that we deal is about the modification of raw data, the modification of aggregated data and the visualization of new aggregated data in a forecasting system.

Forecasting Methods. Forecasting methods can be classified as either subjective or objective [1]. Objective methods include extrapolation (such as moving averages [34], linear regression against time, or exponential smoothing [31]) and econometric methods [18] (typically using regression techniques [13] to estimate the effects of causal variables). Our work focuses on the presentation layer of forecasting results and does not consider the way to calculate forecasts. We assume it is done by a relevant component that includes required methods and that we consider as a "black-box".

OLAP (Online Analytical Processing) and BI (Business Intelligence). Visualization and data analysis tools represent the dataset in an N-dimensional space. OLAP and BI tools perform "dimensionality reduction", often by summarizing data along dimensions of interest. Along with summarization and dimensionality reduction, data analysis applications extensively use constructs such as histogram, cross-tabulation, subtotals, roll-up and drill-down [11, 12, 28, 33]. In our context, the aggregated data are calculated and stored before the visualization in order to provide an immediate access. When a modification of an aggregate occurs, recomputing is necessary. However, as the modifications are frequent and require an immediate recalculation, the solutions consisting in the recomputing of all aggregated data could be not satisfactory.

Materialized Views. The aggregated data we consider are stored as materialized views in a data warehouse. Change impact management can then be considered as the issue of (materialized) view maintenance. The materialized view maintenance problem has been widely discussed in data warehouse architectures. Solutions about how to efficiently update materialized views in relational databases are adapted for this area. The combination of "materialized view log" and "fast refresh" applied in Oracle [26] shows a good performance in certain contexts. Approaches to view maintenance in data warehouses are concerned with different directions. In [36], the authors propose "lazy" maintenance of materialized views. In order to reduce the view maintenance cost, this paper suggests to postpone maintenance of a view until the system has free cycles or the view is referenced by a query rather than update materialized views when source data

change. [20, 23] propose solutions of incremental view maintenance. These solutions create differential files, which keep the differences of the relevant tuples and calculate new views based on these differential files instead of calculating complete materialized views. [2, 25] discuss multi-view maintenance and their consistency problems over distributed data sources. There exist many others (see the research-oriented bibliography on Data Warehouse and OLAP[3] and Jacob Hammer's web bibliography[4]).

Other works focus on the optimization of OLAP operators such as pivot and unpivot [3]. They propose rewriting rules, combination rules and propagation rules for such operators and also design a novel view maintenance framework for applying these rules to obtain an efficient maintenance plan.

Note that the main context of these approaches is the propagation of updates occurring on sources to materialized views. In our context, the updates take place on summarized data, in other words, directly on materialized views. We need to propagate the modification to raw data and also to other materialized views.

The problem of updating summaries and computing the effect on raw data has been recently investigated by [16, 17]. A formal solution composed of five steps is suggested. Firstly, a copy phase involves to copy data stemming from previous years on which the future calculation will be based. Then a deleting phase allows to clean irrelevant data according to potentially data evolution observed by decision makers. A reevaluation phase applies the necessary transformation to move from last year data to current data. A disaggregation phase distributes the data updates from aggregated level to individual fact level. Finally, a forecast computation phase allows to produce forecast raw data based on the transformation functions previously applied. This solution is not able to use complex model to provide users with accurate forecasts.

What-if Analysis/Simulation. What-if analysis is simulation analysis in which key quantitative assumptions and computations (underlying a decision, estimate, or project) are systematically changed to assess their effect on the final outcome. Used most of the time in evaluation of overall risk or in identification of critical factors, it attempts to predict alternative outcomes of the same course of action [10, 29]. In comparison with our work, what-if analysis changes variables' values to inspect the impacts. When variables' values are changed, a new calculation is required using the simulation model to evaluate the impact. In our work, decision makers perform changes in the forecasting results produced by simulation models. Nevertheless, propagating the modification does not require a new calculation with simulation models. The impact is directly evaluated at different levels of hierarchies in different dimensions regarding some predefined rules. As the forecasting results are stored as materialized views, our issue is data consistency, in other words, maintenance of materialized views. The approach considered in the "Back-write" functionality on the MS Analysis Services OLAP server and MS Excel[21] is not relevant.

[3] `http://lemire.me/OLAP/`

[4] `http://www.cise.ufl.edu/~jhammer/online-bib.htm`

Contribution. In this paper, we propose an algorithm, PAM (Propagation of Aggregate-based Modification) to propagate modifications performed on aggregates. Given a modification of an aggregate, this algorithm identifies the exact sets of impacted raw tuples and summaries to update. It performs the update by creating a temporary table of raw data impacted by the aggregate modification. We also describe an optimized version of PAM that achieves better performance when the use of additional semantics (e.g., dependencies) is possible.

This paper is a consolidation and extension of material presented in [6–8]. In this paper, we extended the PAM algorithm in such way it accommodates semantic dependencies. This extension leads to a better performance. Also, we conducted extensive experiments on real data sets (see Section 6).

Paper Organization. The rest of this paper is organized as follows : Section 2 defines the problem and states the motivations. Section 3 introduces some notations and definitions. Section 4 discusses the current and ad-hoc solutions for aggregate-based update propagation. In Section 5, we describe the algorithm (and its extension) which rests on a convenient exploitation of dimension-hierarchies. Section 6 shows the evaluation and the experimental results of both the existing approach and our algorithms. We conclude in Section 7.

2 Problem Statement and Motivations

Our research motivations come from a typical predictive analytics system, a sales forecasting system. A sales forecasting system, also called a business forecasting system, is a forecasting system allowing achievable sales revenue, based on historical sales data, analysis of market surveys and trends, and salespersons' estimates[5]. The goal is to predict the forthcoming stages of sales of some company or organization. The sales forecasting is one of the most difficult areas of management, where a lot of experience and knowledge is required for accurate prediction. It is done through detailed analysis of all the available information regarding the different aspects of sales. This future prediction will help the company to calculate profits, to make decisions on investments, and to launch new products and services. The implementation of sales forecasting systems will help the company to improve the methods in targeting new customers, thereby giving greater sales output, and supreme customer service. Also, it will help to reach maximum efficiency through proper scheduling of its various activities. An effective sales forecast can have a positive impact on: financing and valuation, inventory management, order management, sales headcount capacity planning, sales revenue, visibility into sales activities [9]. The sales forecasting process is managed by a key person who can be: a sales or financial analyst; a sales operations manager; a sales finance manager or similar other positions. The other intended users of the forecast can be staff of other departments than sales or marketing as discussed above.

[5] See `http://www.businessdictionary.com/definition/sales-forecast.html`

To clearly define our problem, we first review how dimensions, hierarchies and the basic data schema are used by visualization tools of OLAP systems [4, 15, 19].

OLAP systems employ multidimensional data models to structure raw data into multidimensional structures in which each data entry is shaped into a fact with associated measure(s) and descriptive dimensions that characterize the fact. The values within a dimension can be further organized in a containment type hierarchy to support multiple granularities.

In the example shown in Figure 1, we present the model to describe the dimensions together with their alternative hierarchies used in a sales forecasting system. This dimension-hierarchy data model is based on one fact table and three different dimensions. The fact table contains four measures: *turnover*, *quantity*, *price*, and *profit*. We would like to mention that in the fact table of a forecasting system, there are not only "facts", which are achieved results, but also predictions. The three dimensions refer to *customer*, *product* and *time*. Each dimension has its hierarchies to describe the organization of data. The customer dimension, the product dimension and the time dimension have 4 hierarchies, 4 hierarchies, 3 hierarchies, respectively. For instance, the second hierarchy, "Hierarchy Geography", of customer dimension is a geographical hierarchy for analyzing sales by area of sales. Customers are grouped by city for level 1, by department for level 2 and by country for level 3. Sales are aggregated at each level according to this geographical organization when they are analyzed through this hierarchy.

Regarding the visualization, OLAP systems employ materialized views to store derived information in order to avoid extra response time. In the example of sales, derived instances for customers and products are added to represent elements in superior hierarchy levels, such as the creation of a derived customer instance for the city of Lyon, a second one for the Rhône department and a third one for the country France. Thus, the system has three new entries in the customer dimension and accordingly some aggregated sales in the fact table

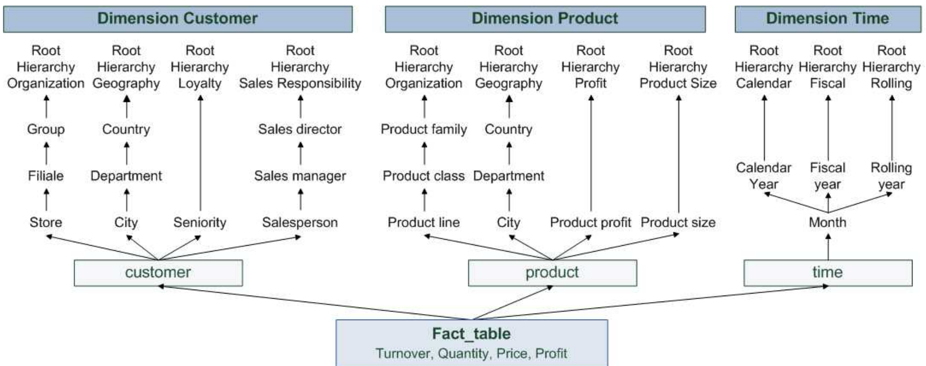


Fig. 1. Example of a fact table with different hierarchies of three dimensions which are used to analyze raw data

corresponding to these newly created derived instances. Finally, all the elements of every hierarchy level from every dimension are aggregated and added to the dimension and fact tables. This pre-calculation guarantees an immediate access to any direct aggregated information, while users perform visualization demands.

However, the visualization in a forecasting system is not the last operation as in other OLAP systems. The systems only produce an initial version of the sales forecasts, which are then reviewed by experienced salespersons. Salespersons check these mathematically generated sales forecasts, take into account some issues not considered by the system and perform some necessary adjustments. For example, promotional offers can lead to higher turnover during the concerned period, but can also cause a decrease in turnover for the next few days because of the carried inventory. Salespersons should make some modifications for these two periods. In other cases, sales managers can also perform some modifications in order to simulate a new marketing objective. They make an estimation on a level of one hierarchy and analyze the modification impacts on other levels, e.g., the detailed customer level, to decide whether the target is achievable. The fact that this update takes place on an aggregated level constitutes the major specific feature of sales forecasting systems. Compared to traditional OLAP systems in which source data are considered to be static, data in sales forecasting systems could be modified many times to obtain a final result.

Hence, sales forecasting systems need to have the ability to quickly react to data modification on an aggregated level. **The problem we need to deal with can be generalized to include how to efficiently update aggregated data through a dimension-hierarchy structure**.

The motivating case study we consider is a real operational sales forecasting system called Anticipeo. In all cases, the current solution calculates new results for raw data and reconstructs all the summaries from scratch. This solution clearly leads to serious optimization issues.

3 Notations and Definitions

In the presentation of the algorithms, we use some notations and predicates. In this section, we introduce some definitions. In the following sections:

- $\mathbb{T}$ stands for all raw tuples
- $\mathbb{A}$ stands for all the aggregates in the materialized view
- α is a distributive aggregate function (e.g., SUM)
- A=α_T is an aggregate of A that employs the aggregate function α on a set of tuples $T \subseteq \mathbb{T}$

Definitions

*Definition 1 (**tuple dependency**).* Given an aggregate A=α_T and a set of raw tuples T', A is said to depend on T' iff $T \cap T' \neq \varnothing$.

*Definition 2 (**tuple dependency predicate**).* **dep(A, T')** returns true if the aggregate A depends on the set of raw tuples T', false otherwise.

*Definition 3 (**impacted tuple**).* A tuple t is said to be impacted by the modification performed on the aggregate A=α_T iff A depends on the tuple t.

*Definition 4 (**aggregate dependency**).* An aggregate A=α_T is said to depend on the aggregate A'=$\alpha_{T'}$ iff A depends on T'.

*Definition 5 (**impacted aggregate**).* An aggregate A=α_T is said to be impacted by the modification on the aggregate A'=$\alpha_{T'}$ iff A depends on A'.

*Definition 6 (**aggregate impact predicate**).* **imp(A, A')** returns true iff the aggregate A is impacted by the modification of the aggregate A', false otherwise.

Let us show on an example how an aggregation-level modification can impact other data by using these definitions and predicates (see Figure 2).

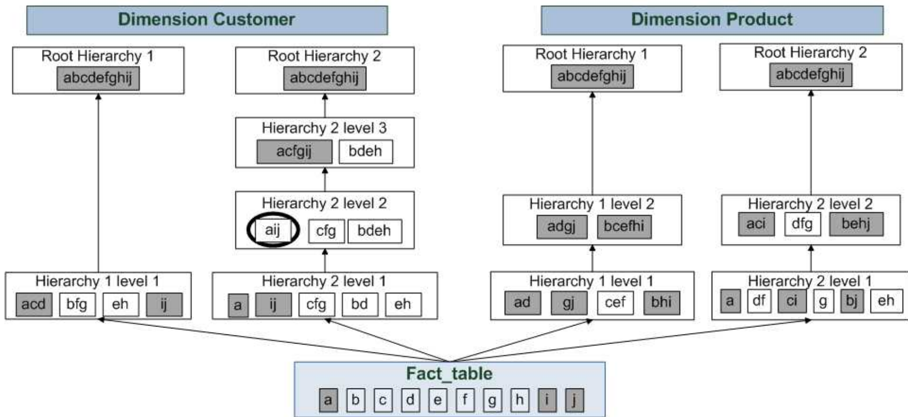


Fig. 2. Example of data modification on an aggregated level in a dimension-hierarchy structure and the impact of the modification

In this example and for the sake of simplicity, we consider only two hierarchies for the customer dimension and the product dimension respectively. In the fact table, we consider only 10 raw tuples: named from a to j. Aggregates at superior hierarchy levels are presented by rectangles including the raw tuples which generate corresponding aggregates. For instance, the circled rectangle of level 2 of hierarchy 2 in the customer dimension represents the aggregate $\alpha_{\{a,i,j\}}$. This presentation denotes that the aggregate $\alpha_{\{a,i,j\}}$ depends on the set of raw

tuples {a,i,j}. In the specific case of a sales forecasting system, the result of the aggregate $\alpha_{\{a,i,j\}}$ is the sum of the base sales a, i and j. Other aggregates are presented in the same manner. The root rectangles of every hierarchy stand for all the sales, therefore, they have the same results in spite of different hierarchies.

Figure 2 depicts the underlying data structure when the system presents the prediction result to sales managers. Sales managers analyze the sales and then decide to modify the value of an aggregate, for example the aggregate $\alpha_{\{a,i,j\}}$ (i.e., to evaluate beforehand the impact of a strategical or tactical move). As the aggregate $\alpha_{\{a,i,j\}}$ is generated from a, i and j, if its value is modified, the results of the three tuples should be updated afterwards. Meanwhile, these three tuples are also the raw tuples that are involved in the calculation of other aggregates in hierarchies of some other dimensions, e.g., the aggregate $\alpha_{\{a,c,d\}}$ of level 1 of hierarchy 1 in the customer dimension and the aggregate $\alpha_{\{b,e,h,j\}}$ of level 2 of hierarchy 2 in the product dimension. Hence, all the aggregates containing any of these three tuples in their composition should be updated as well. These aggregates impacted by the modification on the aggregate $\alpha_{\{a,i,j\}}$ in this example are darkened in Figure 2.

4 Current Solution: Principles and Limitations

A current solution consists in identifying approaches to similar problems and building on the implemented solutions. In this system, methods to calculate the aggregates are already well defined. The current solution uses these methods to calculate new results. The steps of the current solution which consists in recomputing everything are the following:

1. calculate the raw tuples wrt the modification and the decomposition rules,
2. recompute all the aggregates.

To illustrate this process, consider the example shown in Figure 2. We assume the actual result of the aggregate $\alpha_{\{a,i,j\}}$ is 500 000 euros. The sales manager has a new marketing plan, estimated to achieve 600 000 euros sales. The result of $\alpha_{\{a,i,j\}}$ is updated, and the sales manager needs to evaluate the impact on other aggregates in order to determine whether this new plan is achievable in different angles. This example introduces two different values of the aggregate $\alpha_{\{a,i,j\}}$. We denote by val($\alpha_{\{a,i,j\}}$) the value before the modification and by val'($\alpha_{\{a,i,j\}}$) the value after the modification. In this example, val($\alpha_{\{a,i,j\}}$) = 500 000 and val'($\alpha_{\{a,i,j\}}$) = 600 000. Assume that the distribution of sales on raw tuples a, i and j is 100 000 euros, 200 000 euros and 200 000 euros, respectively. We then denote by val(t) the value of the attribute considered in the computation for a tuple. We have, in this example, val(a) = 100 000, val(i) = 200 000 and val(j) = 200 000. Here, we note that each of the raw tuples does not contribute equally to the result of the aggregate. We should consider the contribution of each raw tuple while calculating their new results.

*Definition 7 (**tuple weight**).* A tuple weight is a measure to evaluate the contribution of a tuple to the calculation of an aggregate. It does not depend neither

on the value of the raw tuple nor on the value of the aggregate. A tuple weight could be defined as a constant or as a variable relating to some criteria. In this case, where the result of an aggregate is the simple sum of raw tuples, the tuple weight is defined as a variable and it can be determined as follows:

$weight(t, A) = \frac{val(t)}{val(A)}$,
where t is a tuple and A is an aggregate depending on t.
By considering our example, we have:

$$weight(a, \alpha_{\{a,i,j\}}) = \frac{val(a)}{val(\alpha_{\{a,i,j\}})} = \frac{100\ 000}{500\ 000} = 0.2$$

$$weight(i, \alpha_{\{a,i,j\}}) = \frac{val(i)}{val(\alpha_{\{a,i,j\}})} = \frac{200\ 000}{500\ 000} = 0.4$$

$$weight(j, \alpha_{\{a,i,j\}}) = \frac{val(j)}{val(\alpha_{\{a,i,j\}})} = \frac{200\ 000}{500\ 000} = 0.4$$

Please note that the total weight of all the raw tuples composing an aggregate should be equal to 1. Then, the propagation of the modification using the current solution is processed as follows:

Step 1: calculation of new values of raw tuples
We aim to compute new values of each raw tuple impacted by the modification of the aggregate. Then, the formula to calculate the new result for a tuple t is:

$$\forall t \in T : val'(t) = val(t) + (val'(\alpha_T) - val(\alpha_T)) * weight(t, \alpha_T)$$

In our example, the new values for T={a,i,j} are:

$$val'(a) = 100\ 000 + (600\ 000 - 500\ 000) * 0.2 = 120\ 000$$

$$val'(i) = 200\ 000 + (600\ 000 - 500\ 000) * 0.4 = 240\ 000$$

$$val'(j) = 200\ 000 + (600\ 000 - 500\ 000) * 0.4 = 240\ 000$$

Step 2: recalculation of aggregated information
The second step consists in recomputing the aggregates of all levels for all hierarchies of all dimensions. We follow the same process as when the aggregates were previously created for the hierarchies, i.e., a new execution of the query associated with the materialized views. For instance, the aggregate $\alpha_{\{a,c,d\}}$ is an aggregate of the raw tuples a, c and d; so its new result is calculated by summing the sales of a, c and d with their updated values.

Following this straightforward solution, we can regenerate all the hierarchies of the whole schema with updated data.

5 Proposed Algorithms

The current solution advocates the calculation of all the aggregates of all the hierarchies. However, this solution performs some useless work. If we look closely at the recomputed aggregates in Figure 2, only the dark ones are concerned with the modification and need to be updated, that is, **19** aggregates out of **33**. Hence, the current solution leads to the calculation of **14** aggregates in vain. The key

idea is thus to be able to identify and recompute only the concerned elements. By considering the dependencies between aggregates and raw tuples, we can identify the exact aggregates to modify and hence avoid useless work.

Another drawback of the current solution is its heavy recomputing procedure. Operations of removing and adding aggregates ask for heavy maintenance of index tables and physical storage. Nevertheless, our approach can keep the aggregates at their logical and physical location and avoid extra effort.

Moreover, it is worth underlining that the solution proposed by [16] is quite different from our solution. Our solution integrates a complex forecasting engine which is able to plan future results with satisfactory precision. It helps decision makers not to make their own plans, but only to adjust these autocalculated results. That is the reason why we need to use an interactive adjustment solution with almost no latency.

5.1 PAM Algorithm

In this section, we explain how the PAM algorithm (Propagation of Aggregate-based Modification) [8] identifies and updates the relevant sets of aggregates. We also present its utilization in more complex data schema with multiple hierarchies. The time complexity is also calculated to show its scalability.

Description of the Algorithm. A coarse-grained description of our algorithm is composed of the following steps:

1. retrieval of participating raw tuples to the modified aggregate;
   creation of a temporary table for the raw tuples to be updated;
   and calculation of the differences for raw tuples resulting from the old values and the new ones;
2. update of impacted raw tuples;
3. identification of impacted aggregates;
   and update of impacted aggregates based on previously calculated differences of raw tuples.

In the following, δ for a tuple or an aggregate stands for the difference of the value of a tuple or the result of an aggregate before and after the modification.

The algorithm for the update propagation through a dimension-hierarchy architecture is shown in Table 1. The description of this algorithm uses the notations defined in Section 3. Line 1 to line 4 identify the raw tuples involved in the modification and calculate their differences. Line 5 allows to update these raw tuples. Line 6 to line 10 identify impacted aggregates and perform the update.

Let us take the previous example (Section 4) to illustrate the approach. A sales manager changes the sales of the aggregate $\alpha_{\{a,i,j\}}$ from 500 000 euros to 600 000 euros. Once the modification is confirmed, the system will proceed using the algorithm in Table 1.

Table 1. Algorithm PAM for the update propagation of an aggregate modification

Algorithm PAM (Propagation of Aggregate-based Modification)

Input: Schema S, aggregate A=α_T, the current result CR of T and the updated result UR of A
Output: An updated schema S' of all hierarchies
Algorithm:
1: Calculate the modification of the aggregate A:
$\delta = UR - CR$
2: Retrieve participating raw tuples of A :
$T = \{x_1, x_2, ..., x_n\}$
3: Create a temporary table ΔX for T containing:
element identifier, keys of the dimensions and delta δ_i.
4: Calculate the difference for every raw tuple:
$\forall x_i \in T$: $\delta_i = \delta * weight(x_i)$
Add update attribute δ_i of table ΔX for each tuple x_i
5: Update all the impacted raw tuples:
$\forall bt_i \in T$: $val'(bt_i) = val(bt_i) + \delta_{bt_i}$
6: For each level of each hierarchy of each dimension
7: Identify impacted aggregates A' in all aggregates $\mathbb{A}$:
$A' = \{A_i \in \mathbb{A} | imp(A, A_i)\}$
8: Calculate the difference for every aggregate:
$\forall A_i \in A'$: $\delta_{A_i} = \sum_{x_i \in \{t \in T | dep(A_i, t)\}} (\delta_{x_i})$
9: Update the impacted aggregates:
$\forall A_i \in A'$: $val'(A_i) = val(A_i) + \delta_{A_i}$
10: End for

Step 1: retrieval of the participating tuples to the aggregate, creation of a temporary table and calculation of differences
Retrieve the composition of the aggregate $\alpha_{\{a,i,j\}}$: sales of the aggregate $\alpha_{\{a,i,j\}}$ is the sum of a, i and j. Hence, the composing tuples are a, i and j.

Create a temporary table ΔX for the raw tuples that are identified.

Calculate the δ for the aggregate $\alpha_{\{a,i,j\}}$: δ = 600 000 - 500 000 = 100 000.

Calculate the difference for every tuple using the tuple weight.

$\delta_a = \delta * weight(a) = 100\ 000 * \frac{100\ 000}{500\ 000} = 20\ 000$

$\delta_i = \delta * weight(i) = 100\ 000 * \frac{200\ 000}{500\ 000} = 40\ 000$

$\delta_j = \delta * weight(j) = 100\ 000 * \frac{200\ 000}{500\ 000} = 40\ 000$

The resulting differences of raw tuples are added to the temporary table. This table also contains the dependency information to higher hierarchical levels (shown in Table 2).

Step 2: update of raw tuples
Update the raw tuples impacted by the aggregate modification.

The new values of these raw tuples are computed by their actual values and the differences calculated in step 1.

Table 2. Temporary table ΔX created to store impacted raw tuples

element identifier	**customer key**	**product key**	**delta δ_x**
a	customer_key$_a$	product_key$_a$	20 000
i	customer_key$_i$	product_key$_i$	40 000
j	customer_key$_j$	product_key$_j$	40 000

$val'(t) = val(t) + \delta_t$

In this case, the three impacted raw tuples are updated to:

$val'(a) = val(a) + \delta_a = 100000 + 20000 = 120000$
$val'(i) = val(i) + \delta_i = 200000 + 40000 = 240000$
$val'(j) = val(j) + \delta_j = 200000 + 40000 = 240000$

Step 3: identification of impacted aggregates and update of impacted aggregates

Identify level by level all the aggregates impacted by the modification of the result of the aggregate $\alpha_{\{a,i,j\}}$ by using the dependencies between aggregates and registered raw tuples in the temporary table ΔX. In this case, we identify all the dark rectangles in Figure 2.

Propagate the changes to every impacted aggregate. Let us illustrate this issue with the customer dimension hierarchy 1. We loop for every level of the hierarchy. For level 1, two aggregates to be updated are identified: $\alpha_{\{a,c,d\}}$ and $\alpha_{\{i,j\}}$ because they have at least one of the registered raw tuples in their composition. The aggregate $\alpha_{\{a,c,d\}}$ depends on a, c and d and among these raw tuples, only one is registered in the table ΔX, namely, the raw tuple a. Hence, the value of $\alpha_{\{a,c,d\}}$ is changed only by adding δ_a (here 20 000).

$$val'(\alpha_{\{a,c,d\}}) = val(\alpha_{\{a,c,d\}}) + \delta_a$$
$$= val(\alpha_{\{a,c,d\}}) + 20\ 000$$

The new value of the other aggregate $\alpha_{\{i,j\}}$ at level 1 is then

$$val'(\alpha_{\{i,j\}}) = val(\alpha_{\{i,j\}}) + \delta_i + \delta_j$$
$$= val(\alpha_{\{i,j\}}) + 40\ 000 + 40\ 000$$

The root aggregate $\alpha_{\{a,b,c,d,e,f,g,h,i,j\}}$ at level 2 of the same hierarchy can be calculated in a similar way with only the differences of depending raw tuples which are registered in ΔX, a, i and j:

$$val'(\alpha, \{a, b, c, d, e, f, g, h, i, j\})$$
$$= val(\alpha_{\{a,b,c,d,e,f,g,h,i,j\}}) + \delta_a + \delta_i + \delta_j$$
$$= val(\alpha_{\{a,b,c,d,e,f,g,h,i,j\}}) + 20\ 000 + 40\ 000 + 40\ 000$$

Doing this way, we update only the aggregates impacted by the modification for hierarchy 1 of the customer dimension. The propagation to other hierarchies are processed in the same manner. Finally, we obtain updated data over the entire schema.

PAM in the case of multiple hierarchies
In the example that illustrates the PAM algorithm, the aggregate, subject to a modification, results from only one hierarchy. Meanwhile, a modification can take place on an aggregate resulting from multiple hierarchies, for example, the sales of the product category “office furniture” for the city of “Lyon”. The PAM algorithm can also be used in these cases, i.e., aggregates resulting from multiple hierarchies are subject to a modification. Compared with the cases in which one hierarchy is involved, only the queries in the identification of raw tuples are different. There are more restrictions when retrieving participating tuples. With one hierarchy, we select raw tuples whose hierarchical classification is the modified aggregate regarding the hierarchy. With multiple hierarchies, we select raw tuples whose every hierarchical classification corresponds to the modified aggregate. In the example of the sales of the product category “office furniture” for the city of “Lyon”, the impacted raw tuples are the intersection of raw tuples belonging to the product category “office furniture” and the ones corresponding to the city of “Lyon”.

Main differences with the Jaecksch's approach
The solution proposed by Jaecksch [16] is composed of 5 steps. In step 1, the copy phase involves to copy data stemming from previous years on which the future calculation will be based. In our approach, data are straightly produced by a statistical model based on sales over three last years. In step 2, the deleting phase allows to clean irrelevant data. In step 3, the reevaluation phase allows to apply the necessary transformation to move from last year data to current data. These transformation functions are not modeled in our solution because these transformations are implicitly produced by a statistical model. That is why we do not have equivalent functions in our solution. In step 4, the disaggregation phase distributes the data updates from aggregated customer level to individual fact level. This point is the core of our solution and we propose an additional optimization phase to efficiently improve the update of all impacted aggregates. In step 5, the forecast computation phase allows to produce forecast raw data based on the transformation functions previously applied. The forecasting data generation is a time-consuming process. In our context, we need to provide users with updated results within nearly no delay. It is not relevant to run the forecasting data generation after each modification of forecasting data. Therefore, in our approach, we consider this “forecast generator” as a black-box and we do not concern the production process of forecast data. We focus on the efficient propagation of updates. Our entire solution allows to straightly produce forecasts by a technically advanced statistical model, to manually modify an aggregate based on forecast raw data for specific needs and to efficiently update all impacted forecast raw data and related aggregates.

Time Complexity. In order to estimate the scalability of the PAM algorithm, we evaluate its performance.

Let n be the number of raw tuples impacted by the aggregate modification, k be the total number of levels for all hierarchies and m be the average number of aggregates to be updated in a given level. Let t_i be the time unit consumed by the actions carried out in line i of the algorithm given in Table 1, then line 1 is considered to consume time t_1, line 2 uses $n * t_2$ and so forth. The total time required to run this algorithm can be estimated as:

$$T = t_1 + n * t_2 + n * t_3 + n * t_4 + n * t_5 + k * (n * t_7 + n * t_8 + m * t_9)$$

Subsequently the time complexity of the PAM algorithm is estimated. In practical cases, as the value of n is much larger than m, the time complexity can be approximated by Ω(k*n). We see that Ω(k*n) is polynomial in k and n, hence the PAM algorithm is polynomial (in time).

5.2 PAM II Algorithm

In a second stage, we propose the PAM II algorithm, which is an extended version of PAM algorithm. The main difference between PAM and PAMII is the extra data we need to store in order to efficiently propagate the updates. The PAM II algorithm uses additional semantics (e.g., dependencies between raw tuples and aggregates) in order to improve the performance when propagating the aggregate modification. In the following paragraphs, we will describe the PAM II algorithm and show the difference with the PAM algorithm. In the PAM algorithm, we perform a loop on each level of each hierarchy to identify the aggregates to update. This means that we have to execute one SQL query per level per hierarchy. For the example of hierarchies shown in Figure 1, we have to execute 17 queries for customer dimension, 12 queries for product dimension and 7 queries for time dimension. If these similar queries can be grouped into a single query, the execution time will be reduced.

The dependencies between aggregates and raw tuples are already fixed when the dimensional schema is determined. The idea of this derivative is to provide direct access from all aggregates to raw tuples by employing meta-tables which contain their dependency information. In addition, the temporary table ΔX (Table 2) contains the keys of the dimensions (one key per dimension). If the identification of aggregates through dependency information by providing informations on raw tuples is possible, we can reduce the size of this temporary table by not storing the keys of the dimensions.

The meta-tables are persistent tables and are created when the dimension schema is determined. They need to be maintained up-to-date afterwards when the schema is modified. One meta-table is created for one materialized view to limit the size of the meta-table for the sake of further efficient search. There are two attributes in these tables: keys of aggregates and keys of their depending raw tuples. Figure 3 depicts the database schema where the meta-table “dependency_info” links materialized views and the fact table sales in a sales forecasting system.

The general approach of the PAM II algorithm remains the same as the PAM algorithm. We first identify and update involved raw tuples and then identify and

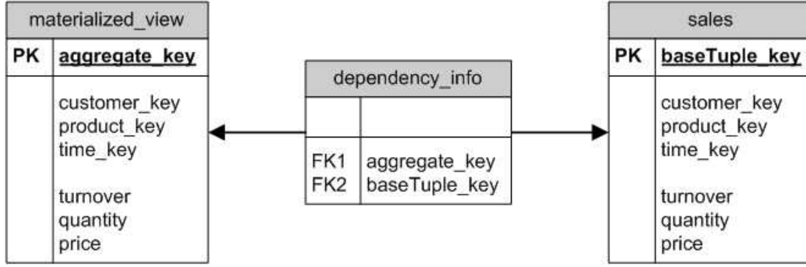


Fig. 3. The database schema for meta-table storing dependency information

update impacted aggregates by an intermediate temporary table. Nonetheless, the detailed processing of the creation of temporary table and the identification of aggregates is not the same. Since the dependency information already exists in the database, we do not need to store the keys of the dimensions in the temporary table. The temporary table has now only two attributes: the element identifier and the delta for this element. The size of this temporary table is reduced. Regarding the identification of the impacted aggregates, instead of running through the dimension tables to identify impacted aggregates level by level, we can identify them directly through the dependency meta-table at one time.

Compared to the original algorithm PAM described in Table 1, the changes of the derived algorithm PAM II mainly concern the lines 3, 6 and 10. The instruction given in line 3 creates a temporary table with less attributes than the one created by the original algorithm. For the update part of the aggregate, it is not necessary to loop through the dimensions and levels to perform the aggregate updates because we can identify all the aggregates at one time by dependency information in the meta-table. Line 6 and line 10 which intended to loop on levels of hierarchies are removed. The PAM II algorithm is shown in Table 3.

There are some further advantages with the meta-tables. These tables give direct dependency information between aggregates and raw tuples. This can serve not only the aggregates, which can be directly deduced from raw tuples via dimension hierarchy structure, but also the aggregates satisfying some specific conditions, e.g., the sum of sales for retail stores whose turnover is more than 100 000 euros. Hence, the PAM II algorithm can be applied more widely to any similar domain that needs to update raw tuples and other materialized views from an aggregate modification.

Time Complexity. The performance of the PAM II algorithm is also calculated to determine its scalability.

Consider n to be the number of raw tuples that are impacted by the aggregate modification, k to be the total number of levels for all hierarchies and m to be the total number of aggregates that are influenced by the modification in the entire schema. As for PAM algorithm, we use the same method to estimate the

Table 3. Algorithm PAM II for the update propagation of a modification

Algorithm PAM II (Propagation of Aggregate Modification - II)

Input: Schema S, aggregate A=α_T, the current result CR of T, dependency meta-table D and the updated result UR of A
Output: An updated schema S' of all hierarchies
Algorithm:
1: Calculate the modification of the aggregate A:
$\delta = UR - CR$
2: Retrieve participating raw tuples of A :
$T = \{x_1, x_2, ..., x_n\}$
3: Create a temporary table ΔX for T containing:
element identifier and delta δ_i.
4: Calculate the difference for every raw tuple:
$\forall x_i \in T$: $\delta_i = \delta * weight(x_i)$
Add update attribute δ_i of table ΔX for each tuple x_i
5: Update all the impacted raw tuples:
$\forall bt_i \in T$: $val'(bt_i) = val(bt_i) + \delta_{bt_i}$
6: Identify impacted aggregates A' in all aggregates $\mathbb{A}$:
$A' = \{A_i \in \mathbb{A} | imp(A, A_i)\}$
7: Calculate the difference for every aggregate:
$\forall A_i \in A'$: $\delta_{A_i} = \sum_{x_i \in \{t \in T | dep(A_i, t)\}} (\delta_{x_i})$
8: Update the impacted aggregates:
$\forall A_i \in A'$: $val'(A_i) = val(A_i) + \delta_{A_i}$

time complexity of the PAM II algorithm. The total time required to run this algorithm is:

$$T = t_1 + n * t_2 + n * t_3 + n * t_4 + n * t_5 + n * t_6 + k * n * t_7 + m * t_8$$

In practice, as the value of n is much larger than m, the time complexity can be approximated by **O(k*n)**. We see that O(k*n) is polynomial in k and n, hence the PAM II algorithm is polynomial (in time).

5.3 Other Aggregate Functions

The aggregate functions are generally divided into three classes [11]: distributive, algebraic and holistic. Distributive aggregate functions can be computed by partitioning their input into disjoint sets, aggregating each set individually and obtaining the final result by further aggregating the partial results. Among the aggregate functions, COUNT, SUM, MIN and MAX found in standard SQL, belong to this category. For example, COUNT can be computed by summing partial counts. Algebraic aggregate functions can be expressed as a scalar function of distributive aggregate functions. AVERAGE, for example, is an algebraic function since it can be expressed as SUM / COUNT. Holistic aggregate functions (e.g., MEDIAN) cannot be computed by dividing the input into parts.

We have introduced the PAM algorithm and its extension PAM II by using the aggregate function SUM. These algorithms are also applicable with other aggregate functions, except that in this work, we do not consider the holistic aggregate functions.

COUNT. Actually, the result of COUNT for higher hierarchical levels is the sum of the partial results corresponding to lower hierarchical levels. The PAM and PAM II algorithms for the COUNT aggregate function are similar to the algorithms used for the SUM function. We identify raw tuples involved in the calculation of the modified aggregate, which is the result of COUNT. We calculate the delta for each of those raw tuples and update them. Then, we identify aggregates impacted by this modification and update those aggregates. The only difference is the calculation of the delta δ for each raw tuple. The numbers used in SUM can be decimal numbers, but the result of COUNT should only contain natural numbers. We slightly modify the calculation mechanism in step 1, the calculation of delta, of the PAM and PAM II algorithms by adding a prune phase to the temporary table ΔX. Once the delta δ of each raw tuple is calculated by their contribution weight of the result, it will be rounded to integer if necessary. The rules are the following:

- if δ is an integer, it will be recorded as such.
- if δ is a decimal, the sum of fractional part of all decimal δ is 1, so
  - the raw tuple having the biggest fractional part will get 1.
  - in the case of equality for fractional part, the raw tuple having the biggest integer part will get 1.
  - in the case of equality for both integer and fractional parts, the first raw tuple registered in the table ΔX will get 1.

AVG. We assume that if a view contains the AVG aggregate function, the materialized view will contain instead the SUM and COUNT functions. The PAM and PAM II algorithms for AVG aggregate function are then reduced to the combination of algorithms for SUM and COUNT functions. The only difference is that, for the function AVG, we slightly modify the structure of the temporary table ΔX. Instead of storing one column for the delta (δ), two columns are created: one for storing the delta of SUM (δ_{sum}), and the other one for storing the delta of COUNT (δ_{count}). The propagation of the aggregate modification, i.e. update of raw tuples involved and update of impacted aggregates, is processed with the modification of results of SUM and COUNT functions. The algorithms roughly remain the same.

MAX and MIN. The above functions, SUM, COUNT, AVG, generate new tuples. However, the aggregate functions MAX and MIN do not generate new tuples. Their results correspond to selected raw tuples. When the result of MAX or MIN is modified, it is the value of the raw tuple (or raw tuples in the case of equality) that is modified. We do not need to identify raw tuples involved in the modification, because they are already known. We assume that we store the MAX/MIN raw tuple(s) and their followers in the materialized views. The PAM

and PAM II algorithms only need to identify the impacted aggregates, whose underlying modified raw tuple(s) are the same as those of MAX/MIN or as their followers. When the value of a MAX or a MIN raw tuple is modified, we compare directly the follower with the new value. If the result after modification is bigger than its follower in the case of MAX or smaller than the follower in the case of MIN, the aggregate result does not need to be updated. If not, we replace the MAX/MIN tuple by its follower. The followers' information needs to be updated consequently.

6 Experiments

The main technical features of the server on which we conducted the evaluation are: two Intel Quad core Xeon-based 2.4 GHz, 16 GB RAM and one SAS disk of 500 GB, 15000 rotations per second. The operating system is a 64-bit Linux Debian system using the EXT3 file system. Our evaluation has been performed on real data (copy of Anticipeo database) implemented on MySQL. The total size of the database is 50 GB, out of which 50% is used in the computation engine, 45% for result visualization and 5% for the web framework. The problem we deal with is concerned with the result visualization. Our test only focuses on the data used by the update: one fact table and dimension tables.

6.1 Evaluation of Different Methods with Two Dimensions

In this first data schema, there are only two dimensions: *customer* and *product*. The fact table containing the keys of the dimensions and forecasts measures has about 300 MB, with 257.8 MB of data and 40.1 MB of indexes. There are 688 419 raw tuples in this fact table. As we know, materializing all aggregates of a data cube is not applicable in a real application. In this experiment, we materialized aggregates resulting from one hierarchy of one dimension, that represents 6 861 aggregates. The customer dimension table contains 5240 real customers and 1319 derived customer instances (6559 in total) and the product dimension table contains 8256 real products and 404 derived product instances (8660 in total) (ref. see Section 2 for the definition of derived customer and product instances).

Each of these dimension tables is composed of 4 hierarchies. It presents a similar structure to the one depicted in Figure 1 with different numbers of levels in each hierarchy (from 2 to 4 levels). Note that the time dimension is investigated within the fact table for some performance issues [6, 7]. Hence, only two explicit dimensions are materialized in dimension tables.

In this section, we will show the evaluation results of different methods in a two-dimensional environment The objective of the evaluation is to show the time of updating the whole schema using the current solution and our PAM and PAM II algorithms. We also validate the estimation of their complexity. Different tests are performed by considering various modifications. This refers to aggregate

modifications which take place on each level of 3 hierarchies, which have 2, 3 and 4 levels, respectively. In our evaluation, we modify one aggregate from each level of each of these 3 hierarchies to compare the evaluation time resulting from the current solution and from our approaches. The number of raw tuples involved in the aggregate modification is shown in Table 4. In other words, this is the number of tuples stored in the temporary table for the PAM and PAM II algorithms.

Table 4. Number of raw tuples involved by the aggregate modification on the appropriate level of hierarchies in the two-dimensional schema

	Hierarchy H1		Hierarchy H2			Hierarchy H3			
	level 1	level 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
Number	64 308	688 419	61 567	61 580	688 419	4 739	50 071	262 771	688 419

We first perform tests with the current solution. The result is shown in Table 5. In this table, we see that when the modification occurs at level 1 of the Hierarchy H1, it takes 0.9 second to perform the step 1, to update raw tuples and 179.5 seconds to perform step 2, to delete and reconstruct all the aggregates. The total time spent for the update of the entire schema caused by this modification is 180.4 seconds. This table shows time spent for updates of the whole schema when modifications occur at different level of different hierarchies. We note that the time devoted to step 2 stays almost the same for different hierarchies. That is because it is concerned with the destruction and the recomputation of the whole schema each time. This operation is also the source of the latency of the current solution.

Table 5. Evaluation time of updating the whole schema following an aggregate modification by using the current solution in a two-dimensional data warehouse

	Hierarchy H1		Hierarchy H2			Hierarchy H3			
(seconds)	level 1	level 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
Step 1*	0.9	7.9	0.9	1.0	7.5	0.08	0.8	2.9	7.8
Step 2*	179.5	182.1	185.7	181.4	188.4	181.1	179.6	179.9	176.6
Total	180.4	190.0	186.6	182.4	195.9	181.2	180.4	182.8	184.4

* Step 1: updating raw tuples;

* Step 2: deleting outdated aggregates and constructing updated aggregates

The same tests are performed with our PAM algorithm. The result is shown in Table 6. We take the same example introduced within the current solution. When we modify an aggregate at level 1 of the Hierarchy H1, it takes 0.3 second to perform stage 1, to create a temporary table containing raw tuples information; 1.0 second to perform stage 2, to update raw tuples and 4.4 seconds to perform

Table 6. Evaluation time of updating the whole schema following an aggregate modification by using our PAM algorithm in a two-dimensional data warehouse

	Hierarchy H1		Hierarchy H2			Hierarchy H3			
(seconds)	level 1	level 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
Step 1*	0.3	3.0	0.3	0.3	2.6	0.05	0.3	1.4	3.0
Step 2*	1.0	8.1	0.9	0.9	7.9	0.1	0.8	3.3	8.3
Step 3*	4.4	47.2	4.4	4.4	49.4	0.5	3.5	17.9	47.4
Total	5.8	58.3	5.6	5.6	59.8	0.7	4.5	22.6	58.7

* Step 1: creating a temporary table of four attributes;
* Step 2: updating raw tuples;
* Step 3: propagating modifications to impacted aggregates.

stage 3, to propagate modifications to all impacted aggregates. In total, we spend 5.8 seconds to update the entire schema.

If we analyze the results of different levels of one hierarchy, we can see that they roughly correspond to our estimation of first time complexity criterion, i.e., number of raw tuples involved in a modification. When a modification occurs in a high level, the number of raw tuples involved in the modification could be large. Then, the execution of the algorithm takes more time. In contrast, a modification on a low level impacts less raw tuples and thus less time is required to update the whole schema. That is why in this table, we note that the time spent to deal with a higher level is greater than the time required to deal with lower levels of the same hierarchy.

To validate the PAM II algorithm, we perform the same tests as we did with the current solution and the PAM algorithm. The result is shown in Table 7. We consider the same example introduced within the current solution and the PAM algorithm. When we modify an aggregate at level 1 of the Hierarchy H1, it takes 0.3 second to perform stage 1, to create a temporary table containing raw tuples

Table 7. Evaluation time of updating the whole schema following an aggregate modification by using our derived PAM II algorithm in a two-dimensional data warehouse

	Hierarchy H1		Hierarchy H2			Hierarchy H3			
(seconds)	level 1	level 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
Stage 1*	0.3	2.7	0.3	0.3	2.5	0.05	0.3	1.2	2.7
Stage 2*	1.0	3.9	0.9	0.9	3.5	0.1	0.7	2.6	3.5
Stage 3*	5.9	29.2	3.3	3.4	28.8	0.3	2.4	11.2	29.6
Total	7.2	35.7	4.5	4.6	34.8	0.4	3.4	15.1	35.9

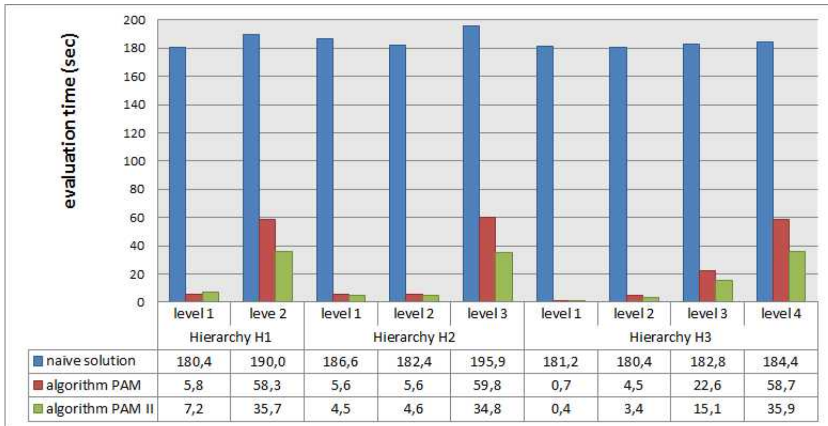
* Stage 1: creating a temporary table of two attributes;
* Stage 2: updating raw tuples;
* Stage 3: propagating modifications to impacted aggregates.

information; 1.0 second to perform stage 2, to update raw tuples and 5.9 seconds to perform stage 3, to propagate modifications to all impacted aggregates. In total, we spent 7.2 seconds to update the entire schema. Compared to 5.8 seconds using the PAM algorithm, this extended version does not show much effect of performance improvement for low level modifications. High level modifications show that the PAM II algorithm is better when compared to the other solutions. For example, only 35.7 seconds are needed to propagate a modification occurring on level 2 of the hierarchy H1. Using the PAM algorithm, we should spend 58.3 seconds for the same operation.

The results of different levels of one hierarchy also confirm our estimation of first time complexity criterion, i.e., number of raw tuples involved in a modification. High level modification takes more time as the number of raw tuples involved in the modification might be large. In contrast, a modification on a low level impacts less raw tuples and requires less time to update the whole schema. That is why in this table, the time devoted to a higher level is more important than the time required for a lower level of the same hierarchy.

6.2 Comparison of the Three Methods

We compare the total evaluation time using the three solutions in one chart shown in Figure 4.



	Hierarchy H1		Hierarchy H2			Hierarchy H3			
	level 1	leve 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
naive solution	180,4	190,0	186,6	182,4	195,9	181,2	180,4	182,8	184,4
algorithm PAM	5,8	58,3	5,6	5,6	59,8	0,7	4,5	22,6	58,7
algorithm PAM II	7,2	35,7	4,5	4,6	34,8	0,4	3,4	15,1	35,9

Fig. 4. Comparison of evaluation time using the current solution, the PAM and PAM II algorithms

Roughly speaking, the new algorithms display much better performance than the current solution. In most cases, the evaluation time is significantly reduced. For example, for the modification at level 1 of hierarchy H3, the propagation time is only 0.7 second using the PAM algorithm and 0.4 second using the PAM II algorithm. Compared to 181.2 seconds spent by the current solution, the PAM

and PAM II algorithms are 257 times faster and 452 times faster respectively. Even in the worst case where the root aggregate (the single aggregate at top level of every hierarchy) is subject to modifications, we get a nearly 220% and 437% better performance using the PAM and PAM II algorithms. The result confirms that, instead of recalculating all the aggregates as the current solution does, our solutions are more efficient by identifying and updating the **exact** set of aggregates impacted by the modification.

Regarding the comparison between our algorithms, PAM and PAM II, PAM II shows an average of 40% better performance. In particular, higher levels benefit more from the existence of the meta-tables by avoiding complex joins. Nevertheless, we need more space. In this test, one meta-table is created to contain dependencies between raw tuples and hierarchical aggregates. There are 688 419 raw tuples and 6 861 aggregates in this test database. Even if the number of tuples in the meta-table is not the Cartesian product of raw tuples and aggregates, more precisely 688 419 * 6 861, there are still 17 711 504 tuples created in this meta-table. This represents **630 MB** of data and **627 MB** of indexes in terms of physical storage. For a database of **50 GB**, the meta-table of **1.23 GB** is relatively large. In addition, if other materialized views need to be updated in the same way, additional meta-tables should be created. Hence, when the physical storage is not a constraint, we recommend the PAM II algorithm. Otherwise, the PAM algorithm is a good candidate.

6.3 Evaluation of Different Methods with Three Dimensions

In the second data schema, we investigate the performance with three dimensions: *customer*, *product* and *time*. In Section 6.1, we introduce the fact that the time dimension of this application is merged into the fact table. In this section, we make the time dimension explicit to create an environment of three dimensions with real data. The customer dimension table and the product dimension table are the same as the ones used in the schema with two dimensions. The time dimension table has 60 basic lines for 60 months and 13 derived time instances for corresponding different hierarchical years. The customer dimension and the product dimension are both composed of 4 hierarchies and the time dimension is composed of 2 hierarchies. The fact table containing the keys of the dimensions and forecasts measures has about **985.5 MB** with **453 MB** of data and **532.5 MB** of indexes. There are 6 995 465 raw tuples in this fact table. Like in the experiment with two dimensions, we only materialized aggregates resulting from one hierarchy of one dimension, which represents 6 897 aggregates.

Similar tests are performed. The number of raw tuples involved in the modification is shown in Table 8.

We compare the total evaluation time using the three solutions in a three-dimensional environment in one chart shown in Figure 5.

Table 8. Number of raw tuples involved in the modification of each test

	Hierarchy H1		Hierarchy H2			Hierarchy H3			
	level 1	level 2	level 1	level 2	level 3	level 1	level 2	level 3	level 4
Number	1 245 321	6 995 465	825 955	826 106	6 995 465	98 190	498 173	2 647 289	6 995 465

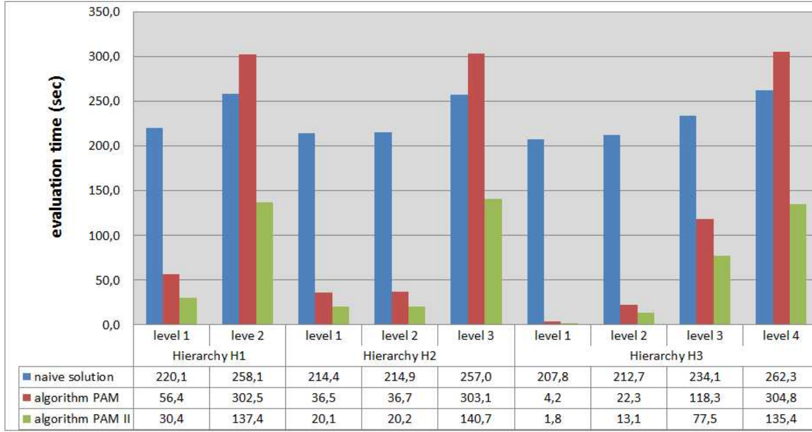


Fig. 5. Comparison of evaluation time using the current solution, the PAM and PAM II algorithms in a three-dimensional schema

In most cases, proposed algorithms present a better performance than the current solution. We take the example of the level 1 of hierarchy H1, time spending to update the whole schema is reduced from 220.1 seconds using current solution to 56.4 seconds using PAM algorithm and 30.4 seconds using PAM II algorithm, which is a gain of 290% and 625% respectively for PAM and PAM II. In the case of modifying an aggregate, which impacts less raw tuples, the gain of PAM and PAM II is more important. As for the example of the level 1 of hierarchy H3, the PAM and PAM II algorithms get fifty times and hundredfold improvement respectively.

However, we notice that in the worst case where the root aggregate (the single aggregate at top level of every hierarchy) is subject to modifications, the current solution of reconstructing all the aggregates is more efficient than the PAM algorithm. Applying the PAM algorithm on this data is not always optimal. Hence, when implementing the PAM algorithm in the real application, we propose an alternative. As mentioned previously, the PAM algorithm is linear to the number of raw tuples involved in an aggregate modification. We can compute the average time spent on a single raw tuple by dividing the total time by the number of raw tuples involved. In the case where the PAM algorithm is less efficient in time than the current solution, we switch to the current solution. The threshold is easy to determine. The execution time of the current solution is known, the average time spent on a single raw tuple by PAM is also known.

Their division is the threshold under which PAM is more efficient. Therefore, when propagating an aggregate modification to the whole schema, we estimate the number of raw tuples that should be updated and make the decision of which solution to adopt.

In this schema, the meta-table of PAM II, which contains the dependencies between aggregates and raw tuples has 191 279 805 tuples. This represents **15 GB** including **9.7 GB** of data and **5.3 GB** of indexes. In the case where the physical storage is not a constraint, the PAM II is the optimal solution.

7 Conclusion

In this paper, we discussed the problem of efficiently propagating the impact of modifications performed on aggregates through dimension hierarchies. A current solution naively recomputes all the aggregates of all the hierarchies, which is time-consuming and does not fulfill the performance needs. We proposed the PAM algorithm and its extension to reduce the propagation cost. Our algorithm is based on the dependencies that may exist between aggregates and raw data. It identifies the exact sets of aggregates to be updated and calculates the delta for each aggregate. We conducted experiments that show that with our approach, the update propagation time can be significantly reduced compared to the current solution implemented in a real application.

As further work, we will take into consideration the scalability of the algorithms. We have shown in this paper that the algorithms are polynomial in time. We will be facing performance issues when databases reach a certain size. Our idea is to potentially create groups over similar raw data, identify dependencies between aggregates and groups and then adapt the algorithms to be able to manipulate groups instead of raw data. Another future way to optimize our algorithm is to identify how some solutions like Star-Cubing [35] may improve our performance. Doing this way, we expect to reduce the time complexity so as to reach a better performance on large datasets. Another direction of research is to evaluate the performance of the propagation of the aggregate-based modification in a column-storage database [32].

References

1. Armstrong, S., Collopy, F., Graefe, A., Green, K.C.: Answers to frequently asked questions (FAQ) in forecasting (2004), `http://repository.upenn.edu/marketing_papers/156/` (last updated November 24, 2004)
2. Chen, S., Liu, B., Rundensteiner, E.A.: Multiversion-based view maintenance over distributed data sources. ACM Trans. Database Syst. 29(4), 675–709 (2004)
3. Chen, S., Rundensteiner, E.A.: Gpivot: Efficient incremental maintenance of complex rolap views. In: ICDE 2005: Proceedings of the 21st International Conference on Data Engineering, pp. 552–563 (2005)
4. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP (on-line analytical processing) to user-analysis: An it mandate. Codd and Date 32, 31 (1993)

5. European Centre for Medium-Range Weather Forecasts: European centre for medium-range weather forecasts (2012), http://www.ecmwf.int/ (accessed January 18, 2012)
6. Feng, H.: Performance problems of forecasting systems. In: ADBIS 2011: Proceedings II of the 15th International Conference on Advances in Databases and Information Systems (PhD Symposium), pp. 254–261 (2011)
7. Feng, H., Lumineau, N., Hacid, M.S., Domps, R.: Data management in forecasting systems: Case study - performance problems and preliminary results. In: BDA 2011: Actes of the 27èmes Journées Bases de Données Avancées - Informal Proceedings (2011)
8. Feng, H., Lumineau, N., Hacid, M.S., Domps, R.: Hierarchy-based update propagation in decision support systems. In: Lee, S.-G., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part II. LNCS, vol. 7239, pp. 261–271. Springer, Heidelberg (2012)
9. Gilmore Lewis LLC.: How to Develop an Effective Sales Forecast. White paper, Gilmore Lewis LLC, 9 pages (July 2006), http://www.gilmorelewis.com/storage/salesforecast.pdf
10. Golfarelli, M., Rizzi, S., Proli, A.: Designing what-if analysis: towards a methodology. In: DOLAP 2006: Proceedings of the ACM 9th International Workshop on Data Warehousing and OLAP, pp. 51–58 (2006)
11. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Min. Knowl. Discov. 1(1), 29–53 (1997)
12. Gupta, H.: Selection of views to materialize in a data warehouse. In: Afrati, F.N., Kolaitis, P.G. (eds.) ICDT 1997. LNCS, vol. 1186, pp. 98–112. Springer, Heidelberg (1996)
13. Hoffman, M.S.: The world almanac and book of facts 1993, 125th Annv. edn. Pharos Books (1993)
14. IBM Press release: IBM and singapore's land transport authority pilot innovative traffic prediction tool (2007), http://www-03.ibm.com/press/us/en/pressrelease/21971.wss (accessed January 18, 2012)
15. Inmon, W.H.: Building the Data Warehouse, 4th edn. John Wiley & Sons, Inc., New York (2005)
16. Jaecksch, B., Lehner, W.: The planning olap model - a multidimensional model with planning support. In: Hameurlain, A., Küng, J., Wagner, R., Cuzzocrea, A., Dayal, U. (eds.) TLDKS VIII. LNCS, vol. 7790, pp. 32–52. Springer, Heidelberg (2013)
17. Jaecksch, B., Lehner, W.: The planning olap model - a multidimensional model with planning support. T. Large-Scale Data- and Knowledge-Centered Systems 8, 32–52 (2013)
18. Johnston, J., Dinardo, J.E.: Econometric Methodes, 4th edn. McGraw-Hill, New York (2007)
19. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd edn. John Wiley & Sons, Inc. (2002)
20. Leung, C.K.-S., Lee, W.: Efficient update of data warehouse views with generalised referential integrity differential files. In: Bell, D.A., Hong, J. (eds.) BNCOD 2006. LNCS, vol. 4042, pp. 199–211. Springer, Heidelberg (2006)
21. Microsoft: Enabling write-back to an olap cube at cell level in excel 2010 (2010), http://msdn.microsoft.com/en-us/library/office/gg521158(v=office.14).aspx (accessed June 1, 2013)

22. Ministry of Transport, Singapore Government: Electronic road pricing (2012), http://app.mot.gov.sg/Land_Transport/Managing_Road_Use/Electronic_Road_Pricing.aspx (accessed January 18, 2012)
23. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenance of data cubes and summary tables in a warehouse. In: Special Interest Group on Management of Data (SIGMOD), pp. 100–111 (1997)
24. National Oceanic and Atmospheric Administration, US Government: National centers for environmental prediction (2012), http://www.ncep.noaa.gov/ (accessed January 18, 2012)
25. Nica, A., Lee, A.J., Rundensteiner, E.A.: The cvs algorithm for view synchronization in evolvable large-scale information systems. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 359–373. Springer, Heidelberg (1998)
26. Oracle: Materialized view concepts and architecture (2012), http://docs.oracle.com/cd/B10501_01/server.920/a96567/repmview.htm (accessed June 1, 2012)
27. Petropoulos, C., Metaxiotis, K., Nikolopoulos, K., Assimakopoulos, V., Patelis, A.: Sftis: a decision support system for tourism demand forecasting. J. of Comput. Inf. Syst. 44(1), 21–32 (2003)
28. Ross, K.A., Srivastava, D., Chatziantoniou, D.: Complex aggregation at multiple granularities. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 263–277. Springer, Heidelberg (1998)
29. Saltelli, A.: Global sensitivity analysis: the primer. John Wiley & Sons, Inc. (2008)
30. Schumaker, R.P., Chen, H.: A quantitative stock prediction system based on financial news. Inf. Process. and Manag. 45, 571–583 (2009)
31. of Standards, N.I., of America, T.N.: NIST/SEMATECH e-Handbook of Statistical Methods (2011), http://www.itl.nist.gov/div898/handbook/index.htm (last updated April 1, 2011)
32. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented dbms. In: Very Large Data Bases (VLDB), pp. 553–564 (2005)
33. Theodoratos, D.: Exploiting hierarchical clustering in evaluating multidimensional aggregation queries. In: DOLAP 2003: Proceedings of the ACM 6th International Workshop on Data Warehousing and OLAP, pp. 63–70 (2003)
34. Wikipedia: Moving average (2012), http://en.wikipedia.org/wiki/Moving_average (accessed June 6, 2012)
35. Xin, D., Han, J., Li, X., Wah, B.W.: Star-cubing: computing iceberg cubes by top-down and bottom-up integration. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 476–487. VLDB Endowment (2003), http://dl.acm.org/citation.cfm?id=1315451.1315493
36. Zhou, J., Larson, P.Å., Elmongui, H.G.: Lazy maintenance of materialized views. In: VLDB 2007: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 231–242 (2007)

LRS: A Novel Learning Routing Scheme for Query Routing on Unstructured P2P Systems

Taoufik Yeferny[1,3], Khedija Arour[2], and Amel Bouzeghoub[3]

[1] Dept. of Computer Science, Faculty of Sciences of Tunis,
Tunisia LISI Research Group
Taoufik.Yeferny@it-sudparis.eu

[2] Dept. of Computer Science, National Institute of Applied Sciences and Technology of Tunis, Tunisia URPAH Research Group
Khedija.arour@issatm.rnu.tn

[3] Dept. of Computer Science, TELECOM Sudparis, France
SAMOVAR Research Group
Amel.Bouzeghoub@it-sudparis.eu

Abstract. Query routing is a fundamental problem in unstructured Peer-to-Peer systems. Recently, researches in this area have focused on methods based on query-oriented routing indices. These methods use the historical information of past queries and query hits to build a local knowledge base per peer, which represents the user's interests or profile. Existing approaches represent the user's profile only by some statistics about past queries and they have not addressed two difficult challenging problems: (*i*) the bootstraping (*ii*) the unsuccessful relevant peers search. Indeed, when a peer selects an insufficient number of relevant peers from its local knowledge base, it floods the query through the network, which badly affects the routing efficiency and effectiveness. To tackle these problems, we introduce a novel Learning Routing Scheme (*LRS*). We implemented the proposed scheme and compared its routing efficiency and retrieval effectiveness with a broadcasting scheme (without learning) and a learning scheme taken from the literature. Experimental results show that our scheme carries out better than other ones with respect to accuracy.

Keywords: P2P, Learning routing methods, Clustering.

1 Introduction

Peer-to-peer systems have emerged as platforms for users to search and share information over the Internet. In fact, thanks to these systems, each user can share various resources (e.g. documents, music, etc.), send queries to search and locate resources shared by other users. There are different kinds of P2P system architectures that can be classified into structured, unstructured and hybrid architectures [13, 7]. Nowadays, unstructured P2P systems are the most frequently used on the Internet since there is no fixed topology for peers. Each peer is randomly connected to a set of peers named neighbors and self-maintains links with

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 54–82, 2013.
© Springer-Verlag Berlin Heidelberg 2013

them. It also imposes no constraints on data placement. In these systems, if a peer wants to find a desired resource in the network, it floods a query through the network to find as many peers as possible that share the pertinent resources. Query flooding strategy does not guarantee that the queries will always be resolved. In addition, this strategy generates a very large number of messages and cannot quickly locate the requested resources. In unstructured P2P systems, researches' efficiency and effectiveness can be improved by making smart decisions for query routing: selecting the best peers to which a given query should be forwarded in order to retrieve the best search results. In the literature, several approaches have attempted to improve query routing in unstructured P2P systems by adding some semantic aspects. The semantic routing methods can be classified into content-oriented routing indices methods [15, 5, 19, 18], cluster-based routing [30, 11, 2, 22, 10] methods and query-oriented routing indices methods [14, 3, 23, 4]. Content-oriented routing indices methods are based on peers shared contents. Each peer maintains routing indices (or peer-content synopses) that describe the shared content of other peers in the network. When a peer routes a given query, it uses its routing indices to select the best processing peers to forward the query to. Such methods improve the search efficiency and effectiveness. However, they incur a higher storage cost since more indices need to be stored at a peer and a higher update cost for these indices, which badly affect the system scalability. Furthermore, due to the network dynamicity, these indices may be obsolete or inconsistent. Cluster-based routing methods organize the P2P network into clusters of peers sharing similar preferences (semantic overlay). The preferences of the user can be deduced from his shared documents, past queries, etc. In semantic overlay network, each peer makes connections with peers having closer interests (i.e., friend peers). In those methods, a query is first routed directly to a related cluster and then to the peers in that cluster. The existing methods differ from each other in the way they build the clusters. Those methods improve the search effectiveness and efficiency when a node issues queries similar to its semantic (i.e., shared content). Nevertheless, when issuing queries irrelevant with their semantic, those methods work poorly since each node knows only a few long semantic links. Query-oriented routing indices methods use information about past queries and query hits to route future queries. Indeed, the observation of the past information is used to create a knowledge base per peer that represents the user's interests or profile. When a peer propagates a given query among computing peers, it evaluates it against its local knowledge base in order to select a set of relevant peers to whom the query will be routed. If the number of relevant peers is below a certain threshold, a random set of peers will be added from the neighbors table. These methods are more advantageous than content-oriented and cluster-based routing methods, since no excessive network overhead is necessary for the construction of the routing indices. Few query-oriented routing indices methods has been proposed in the literature. They improve the search efficiency and effectiveness [23, 4] of traditional routing approach. However, they suffer from three main limits:

1. ***User profile representation***: The user profile is represented by some statistics about past queries (e.g. query keywords, hits number per peer, etc.) but it does not exploit repetition rate for keywords seen in sent queries and relationships between them as well [14, 3, 23, 4].
2. ***Unsuccessful relevant peers search***: The existing approaches have not addressed the unsuccessful relevant peers search problem [14, 3, 23, 4, 26]. Indeed, when a peer selects an insufficient number of relevant peers from its local knowledge base, it floods the query through the network which badly affect the routing efficiency and effectiveness.
3. ***bootstrapping problem***: Upon joining the P2P network, a peer has no prior knowledge; therefore, it is impossible to make smart routing decisions [27]. For this reason, the newly joined peer has to flood a given number of queries and records the returned responses in order to build the initial knowledge base. This phase is called *training phase*. Indeed, methods based on queries history achieve slow improvement in routing efficiency and effectiveness especially during the *training phase* [14, 3, 23, 4, 26].

To avoid these drawbacks, we introduce a new hybrid scheme Learning Routing Scheme (LRS) that combines the query-oriented and cluster-based approaches. In our scheme LRS, each peer maintains a local knowledge base that contains a set of user interests deduced from the past queries and query hits. In addition, the P2P network is organized into clusters of peers sharing similar knowledge bases (semantic overlay). Hence, each peer makes connections with peers having closer interests, named friend peers.

The main contributions of this paper are the following: First, LRS exploits repetition rate for keywords seen in sent queries and relationships between them as well in order to build a knowledge base per peer that represents the user's profile. In LRS, the user's profile is a correlation between sent queries and positive peers or sent queries and query terms. Second, to palliate the unsuccessful relevant peers search problem, we propose a new search mechanism avoiding the random selection based upon semantic overlay network. We propose a community construction method that establishes connections between peers sharing similar knowledge bases, named friend peers. Indeed, the proposed network topology allows each peer which selects from its local knowledge base an insufficient number of relevant peers to forward the query according to its content to the best friend peers, rather than random neighbors. Third, to tackle the bootstrapping problem, we propose a proactive initial knowledge base building method to improve the efficiency and effectiveness of query routing in the training phase. Indeed, LRS predicts the user's profile based on shared documents and builds an initial knowledge base before a peer sends its first query.

The remainder of this paper is organized as follows: Section 2 presents the requirements of our approach. In Section 3, we present a critical overview of query routing methods in P2P systems. Section 4 discusses LRS approach. In Section 5, we report the results of our experimental evaluation. Section 6 concludes with some proposed direction for further works.

2 Query Routing in P2P Systems: An Overview

Before we examine the various existing routing methods, we shall look at some of the main requirements that our proposed scheme must satisfy. We describe these requirements in the following. Then, we compare the existing methods based on these requirements:

- R1. ***(Search efficiency)***. A routing method is efficient whenever it requires a small number of messages to be routed in order to locate information. For method that requires a huge amount of messages, the bandwidth consumption will be high, hampering by the way the system scalability.
- R2. ***(Search effectiveness)***. A routing method is effective whenever it can locate more pertinent resources.
- R3. ***(Cost of building routing indices)***. Semantic routing methods maintain at each peer routing indices, which are used in directing the search space. If an excessive network overhead is necessary for the construction of the routing indices, the routing method is not scalable. Our goal is to design a scalable routing scheme that requires a low cost for building routing indices.

Efficient query routing in unstructured P2P requires intelligent decisions: selecting the best peers to which a given query should be forwarded for retrieving related resources. The first query routing method is based on query flooding as in Gnutella system [7]. In order to find pertinent resources, a peer sends a query to all its neighbors on the overlay, which, in turn, forward the query to all of their neighbors and so on, until the query Time-To-Live (TTL) expires. Although this solution is straightforward and robust, it generates a very large number of messages and it cannot quickly locate the requested resources. Researchers have mainly focused on improving this naive method by using controlled flooding such as expanded ring search with random walks [25], iterative deepening [25], directed BFS [7] and random breadth first search RBFS [14]. The flaw in these methods is that efficiency of their routing is very low because they do not take into account the query string. Hence, a large number of irrelative nodes have to be visited during the search process.

Several works have attempted to improve these traditional query routing methods in unstructured systems by introducing semantics in the process of query propagation [3]. The existing semantic methods can be classified into content-oriented routing indices methods [15, 5, 19, 18], cluster-based routing methods [30, 11, 2, 22, 10] and query-oriented routing indices methods [14, 3, 23, 4]. In the following, we briefly summarize main works for each category.

2.1 Content-Oriented Routing Indices Methods

Content-oriented routing indices methods are based on the shared contents of peers. Each peer maintains routing indices (or peer-content synopses) that describe the shared content of other peers in the networks. Hence, the peers to

which a query is forwarded are chosen based on the content similarity between the query and the data held by the target candidate peers (or the corresponding peer synopses), rather than random selection. In CORI (Collection Retrieval Inference Network) [1] and GLOSS (generalized Glossary-of-Servers Server) [16] the collection of each neighbor is represented by a *superdocument*. The set of all *superdocuments* forms a special purpose collection that is used to identify the most promising collections for a given query. In RI (Routing indices for peer-to-peer systems) [5], the contents are classified under "topics" and peers index the number of documents under each topic reachable through each neighbor (path) and the number of documents along each path. To forward a given query, the forwarder peer computes the "goodness" of each node for a query then, propagates the query to the peers having the highest "goodness". GLOSS and CORI assume that we are selecting among a set of collections, whereas RI assumes that we are selecting among a set of "paths" that lead to a set of collections. The information retrieval system PlanetP [19] represents the content of each peer in the network in a compactly Bloom filter [21]. These Bloom filters are distributed across the network using a Gossiping algorithm. The set of all Bloom filters forms a global index to give the peer a partial and approximate view of the network content. When receiving a query, a peer searches at first in its local index. If it is not possible to answer this query, it calculates a score of peers from the global index and propagates the query to the peers which have the greatest score. PlanetP may yield better results than CORI, GLOSS and RI. However, the storage space necessary for the global index is very important and a huge amount of messages need to be exchanged in order to build and update this index. In Sunrise [18] each peer exposes the data it wants to share with other peers according to a local ontology in order to make a rich representation of the shared data. Each peer P_j that peer P is connected to, peer P builds a semantic mapping $MP[P_j]$, which defines how to represent the schema ($MySchema$) of P in terms of P_j's schema ($MySchemaj$) vocabulary. Then, it associates each concept in $MySchema$ to a corresponding concept in $MySchema_j$ according to a score that denotes the semantic similarity grade between the two concepts. Based on the similarity between the concepts in its schema and the concepts in the schema of each of its neighbors, peer P builds a semantic routing index (SRI) that suggests the relevance of the data that can be reached in each direction starting from P. When a peer receives a query, it exploits its SRI scores to rank its neighborhood, thus, it identifies the most promising direction to follow in the network. To accomplish the forwarding step, each peer reformulates the received query over the destination peers schema according to the corresponding mapping. Sunrise uses more semantics than the previous presented approaches [1, 16, 5, 19]. It offers new potentialities for query formulation and, consequently, new challenges for query routing.

The existing content-oriented routing indices methods improve the search effectiveness and efficiency. However, they incur either a higher storage cost since more indices need to be stored at a peer. In addition, a huge amount number of messages need to be exchanged to build and refresh these indices, which badly

affect the system scalability. Furthermore, due to the network dynamicity, the indices may be obsolete or inconsistent. Hence, content-oriented routing indices methods do not satisfy requirement $R3$.

2.2 Cluster-Based Routing Methods

Cluster-based routing methods organize the P2P network into clusters of peers sharing similar preferences (semantic overlay). The preferences of the user can be deduced from his shared documents, past queries, etc. In semantic overlay network, each peer makes connections with peers having closer interests (i.e., friend peers). Several schemes have been proposed to reduce the querying overhead through clustering of peers. In those schemes, a query is first routed directly to a related cluster and then to the peers in that cluster. The schemes differ from each other in the way they build the clusters. GES [30] and CSS [11] summarize all the documents in each node into an average term vector (named node vector) based on VSM (vector space model). By introducing VSM, GES forms semantic clusters (i.e., nodes are organized into clusters according to their node vectors). A query is first routed directly to a related group, and then flooded inside that group. To make searching more efficient, CSS extends GES by clustering all documents on a node into different classes. The drawback in both GES and CSS is the flooding protocol through semantic groups, whose search cost is very high. Carchiolo et al [2] propose a model for the growth and evolution of a peer-to-peer network inspired by the social networks dynamics and behaviors, called PROSA (P2P Resource Organisation by Social Acquaintances). In PROSA, nodes may establish "Fully Semantic Link" (FSL) or "Temporary Semantic Link" (TSL) or "Acquaintance Link" (AL) according to the degree of knowing each other. Each link in PROSA is associated with a compact representation of target peer's knowledge, when available: in the case of ALs, no such information is available (modeled with an empty set $\emptyset$). To forward a given query, the forwarder peer computes the relevance between the query and its neighbors with TSL or FSL links. Thereafter, it selects the peer that is connected with the link having the highest relevance value.If the forwarder peer has only ALs links, the next peer is selected at random, which badly affect the search efficiency and effectiveness. SKIP [22] uses vector space model (VSM) and relevance ranking algorithms to construct an overlay network. The key idea of SKIP is to reorder the semantic neighbors of nodes according to relevant scores. The search mechanism replaces flooding protocol in GES with K-iteration preference. Hung-Chang et al. [10] present a clustering method that organizes the P2P network as a semantic small-world random graph. Here, semantic small-world networks refer to the fact that probability of peer j being the neighbor of peer i increases if j shares more common interests with i. The goal of the proposed clustering methods is to restructure the P2P network to satisfy the following properties: 1) High clustering, 2) Low diameter and 3) Progressive. The first property, assumes that each peer connects the most similar peers in the networks. The second property, imposes that should exist at least one overlay path connecting two peers u and v. The hop count of the path should be as small as possible, enabling a query message

to be rapidly propagated from u to v. The third property, imposes that should exist an overlay path P connecting a peer that issues a query s and the peer that can resolve the query d such that for any two neighboring peers u and v on P, upon receiving a query message, u forwards the message to v that is more similar to d than u. The proposed network formation algorithm performs very well with rigorously mathematical guarantees. However, to satisfy these three properties, the clustering algorithms need a very large number of messages.

Cluster-based routing methods establish links among most semantic similar nodes. Query routing is first done by global routing through long links which are established between semantic dissimilar nodes, and then, perform flooding by local routing through short links which are established between semantic similar nodes (peers in the same cluster). Those methods improve the search effectiveness and efficiency when a node issues queries similar to its semantic (i.e., shared content). However, when issuing queries irrelevant with their semantic, those methods work poorly since each node knows only few peers belonging to other clusters. Hence, cluster-based routing methods do not usually satisfy the search effectiveness requirement $R2$.

2.3 Query-Oriented Routing Indices Methods

Query-oriented routing indices methods exploit the historical information of past queries and query hits to route future queries. In directed BFS [14], each node maintains some statistics of its neighbors such as the number of times previous queries can be answered through a neighbor node, the number of results obtained for the queries and the latency in receiving the results. The authors of this method developed a number of heuristics that are based on these statistics to select the best P_{max} neighbors to send the query (P_{max} is a specified user threshold). The main drawback of this technique is that statistics maintained by each peer about its neighborhood is not wealthy enough. These statistics do not contain the information related to the query content. Kalogeraki et al. [14] propose a similar method to BFS called intelligent search (IS). In this method, each peer builds a profile of its neighbors and uses the profile to find peers, which are likely to answer a forthcoming query. The profile contains the list of the most recent past queries and peers that supplied answers. The authors represent each profile by a single queries table. Each entry is a couple (Q, N), where N is the node that supplied answers to query Q. The node accumulates the list of past queries by two different mechanisms. In the first mechanism, the peer is continuously monitoring and recording the query and the corresponding query-hit messages it receives. In the second, each peer, when replying to a query message, broadcasts this information to its neighbor peers. This operation increases the accuracy of the system, at the expense of $O(d)$ extra messages (where d is the average degree of the network) per answering node. To decide to which peers a query will be sent, a peer p ranks all its neighbors with respect to the given query; then, it forwards it to the first P_{max} peers. Indeed, the peer p compares the query to previously seen queries and finds the most similar ones in the repository. The query is then forwarded to the nodes that supplied

answers to the closest queries. Unlike directed BFS, IS takes into account the query terms to select the best neighbors to forward the query to. However, in IS, each node exports its profile to its neighbors, which implies a high bandwidth consumption. $REMINDIN$ [3] scheme exploits social metaphors to define a strategy of query routing. In this scheme, each peer maintains a set of RDF statements in a local peer repository (ontology). It stores metadata about these statements in order to memorize where the statement cames from and how much resource-specific confidence have. Overall confidence is put into these statements and peers. To select promising peers for a given query, a peer evaluates the query against the local node repository in order to select a set of statements matching the query. For each statement, $REMINDIN$ retrieves its metadata and specific confidence. Thereafter, promising peers are sorted according to their strength. Up to P_{max} best peers are returned as targets for the query. A major problem for $REMINDIN$ is the selection of pertinent information to store. Indeed, the user manually determines which information to store in the local peer repository. However, the system must provide an automatic mechanism to select the pertinent information. In $LBQR$ [23], every peer assigns for its connections a set of weights. Then, it uses these weights to choose the neighbors to route the query. Indeed, every peer maintains routing indices, which contain two main parameters, *weight* and *visit*, per neighbor. *Weight*, depicts the estimation of the number of returned contents when a query is forwarded to the neighbor. *Visit*, records the statistical information of query forwarding. When the query hits arrive, they are combined to form the *feedback* and are used to update the weight. The peer will choose P_{max} connection having the highest forwarding probability. In $LBQR$, routing indices are updated online (when a peer receives query hits), which generates an important computational load. In addition, $LBQR$ does not take into account multi-keywords queries. Learning Query Routing Method [26] builds a knowledge base per peer by learning the implicit behavior of users that is deducted from query history. The knowledge base contains a set of user profiles that represent the past query terms and peers whose provided answers. When a peer forwards a given query it exploits the local knowledge base to select P_{max} relevant peers to forward the query to. In Route Learning [4], a peer tries to assess the neighbors that will most likely reply to queries. Peers compute this estimation based on the knowledge that accumulates gradually from query and query hit messages sent to and received from neighbors. In Route Learning, every peer in the network maintains a feature space that is created for each neighbor. Each point k of the feature space is mapped to keywords seen in sent queries and it has two numbers. The first one (answer-count) is the number of answers returned through that neighbor for keywords mapping to point k, and the second one (query-count) is the number of queries made to point k. When a peer forwards a given query, it tries to select P_{max} neighbors by applying the Parzen Windows estimation which is used to solve classification problem. Indeed, Route Learning is an adaptation of a classification problem to routing, in unstructured P2P systems, each peer having n neighbors corresponding to n classes. In a classification problem, the classifier tries to classify an object using

the feature of each class. By analogy, for a given query, Route Learning selects P_{max} neighbors according to their futures spaces and the Parzen Windows estimation. This scheme requires more memory space at a peer; thus, it should be used in platforms where memory capacity is not very low.

The idea underlying all the existing query-oriented routing indices methods is to replace the classical routing method (spread by flooding) used for example in Gnutella [7], by a semantic routing method based on the historical information about past queries. The historical information are used to build a knowledge base per peer in order to guide the process of peers' selection. These methods are more advantageous than content-oriented and cluster-based methods since no excessive network overhead is necessary for the construction of the routing indices. However, they suffer from the unsuccessful relevant peers search and cold-start problems. The first problem appears when a peer selects an insufficient number of relevant peers (i.e., below a certain threshold P_{max}) from its local knowledge base. It floods the query through the network which badly affect the search efficiency and effectiveness. In [28] we proposed a generic solution for this problem which can be adapted for any query-oriented routing indices method. The second problem appears when a new peer joins the P2P network. It has no prior knowledge. Therefore it is impossible to make smart routing decisions [27]. The newly joined peer has to flood a given number of queries and to record the returned responses in order to build the initial knowledge base. This phase is called *training phase*. Indeed, query-oriented routing indices methods achieve slow improvement in search efficiency and effectiveness especially during the *training phase*.

After this review of existing query-oriented routing indices methods, we conclude that they do not satisfy the search efficiency ($R1$) and effectiveness ($R2$) requirements during the training phase or when a peer selects an insufficient number of relevant peers, from its local knowledge base, for a given query.

2.4 Synthesis on Query Routing Methods

Table 1 summarizes the surveyed routing approaches and compares them against the predefined requirements. In the previous section, we showed that none of the existing routing approaches supports the three predefined requirements $R1$, $R2$ and $R3$. Indeed, content-oriented routing indices methods need an unavoidable excessive network overhead for the construction of the routing indices. Thus, they do not support the requirement ($R3$). Cluster-based routing methods do not satisfy the search effectiveness ($R2$) requirements, when a peer issues queries dissimilar to its semantic. Query-oriented routing indices methods do not satisfy the search efficiency ($R1$) and effectiveness ($R2$) requirements during the training phase or when a peer selects an insufficient number of relevant peers, from its local knowledge base, for a given query.

In this paper, we propose a new hybrid scheme *LRS* that combines the query-oriented and cluster-based approaches in order to support the three predefined requirements. We aim to take advantage of the surveyed query-oriented and cluster-based approaches and improve their weaknesses. In *LRS*, each peer maintains a local knowledge base that contains a set of user interests deduced from the past queries and query hits. We exploit repetition rate for keywords seen in past queries and the relationships between them. Hence, the user profile represents a correlation between *past queries*, their *terms* and the *peers* who provided answers. *LRS* stores less routing indices and updates them offline periodically in order to reduce the cost of the updating operation. To tackle the unsuccessful peer selection problem in query-oriented methods, we propose a new search mechanism avoiding the random selection based upon semantic overlay network construction. Our method organizes the P2P overlay network into semantic clusters of peers sharing similar knowledge bases. Indeed, each peer p_i in the network makes new connections to link friend peers. Consequently, if a peer p_i selects from its local knowledge base an insufficient number of relevant peers, it forwards the query according to its content to the best friend peers, rather than random chosen peers. The chosen friend peers are able to find in their knowledge bases relevant peers. We based our idea on the following social networks assumption: *it is common that two of your friends would have a greater probability of knowing each other than two people randomly chosen from the population, on account of their common acquaintance with you.* To address the cold-start problem of query-oriented routing indices methods, our scheme predicts the user profile using the shared documents and builds an initial knowledge base before the peer sends its first query.

3 *LRS* Approach

Our scheme *LRS* is designed for unstructured P2P networks like Gnutella. The idea underlying our proposal is to replace flooding routing method used in Gnutella [7] by a semantic routing method based on the user's profile. The key contribution is the observation of the past queries, which are used to create for each peer p_i a knowledge base B_i to guide the process of peers' selection. Moreover, to alleviate the unsuccessful peer selection problem, *LRS* organizes the P2P overlay network into semantic clusters of peers sharing similar knowledge bases. Indeed, in our proposal, each peer p_i makes new connections to link friend peers Fr_i. Consequently, if a peer p_i selects from its local knowledge base an insufficient number of relevant peers, it forwards the query according to its content, to the best friend peers rather than randomly chosen peers like in the existing query-oriented routing indices approaches. Friend peers are able to find in their knowledge bases relevant peers.

Table 1. Comparative study of existing routing methods

	Content-oriented routing indices methods: CORI, GLOSS, PlanetP, Sunrise, etc.	**Cluster-based routing methods: GES, CSS, PROSA, SKIP, etc**	**Query-oriented routing indices methods: BFS, IS, REMINDIN, LBQR, Route Learning, etc.**	**Our hybrid approach: LRS**
R1	Efficient search	Efficient search	Poor search efficiency during training phase or when a peer selects an insufficient number of relevant peers, from its local knowledge base.	Efficient search
R2	Effective search	Poor search effectiveness when a peer issues queries dissimilar to its semantic.	Poor search effectiveness during training phase or when a peer selects an insufficient number of relevant peers, from its local knowledge base.	Effective search
R3	An unavoidable excessive network overhead for the construction of the routing indices	Acceptable network overhead for the construction of the routing indices	No network overhead for the construction of the routing indices	Acceptable network overhead for the construction of the routing indices

The global architecture of *LRS* is composed of the five following layers:

1. *Log file management layer* that runs when receiving responses. It manages the log file of peer.
2. *User profile management layer* that runs periodically. It builds the knowledge base from a log file.
3. *Clustering layer* that runs when a peer updates its knowledge base. It organizes the P2P network into clusters of peers sharing similar knowledge. Indeed, each peer in the network makes new connections to link friend peers that share similar knowledge. Hence, peers are dynamically organized in new semantic clusters.
4. *Bootstrapping layer* that runs one time when a new peer joins the P2P network. It builds an initial knowledge base before that the peer sends its first query.
5. *Semantic query spreading layer*: that runs when a peer forwards a given query. It uses the local knowledge base and the semantic links (i.e., links with friend peers) in order to route the query to promising peers.

3.1 Log File Management Layer

Each peer stores information about past queries in a local log file. This file contains raw data collected from the answered queries. When a peer receives responses for a query, this layer updates the log file by adding information related to this query like the identifier of the query, its terms, the downloaded documents and associated peers (peers who have answered the query).

3.2 User Profile Management Layer

This layer aims to build a knowledge base per peer that will be used by the *semantic query propagation layer* in order to select relevant peers. In our scheme, the knowledge base contains a set of interests that represents the user's profile. Our goal is to generate a set of interests that define semantic relationships between *sent queries*, their *terms* and positive *peers*. Otherwise, each interest I_i must represents a group of closest past queries having a set of common terms and answered by a set of positive peers. In LRS, each interest is a triplet $I_i(E_i, F_i, G_i)$, that expresses a correlation between a subset of queries E_i, their common terms F_i and the set of associated peers G_i. Indeed, to generate any user profile, the first step clusters queries which have common terms into a single set E_i. The second step builds the set G_i, which contains positive peers that answered to all queries in E_i. In most cases G_i is empty; so to alleviate this case, we propose to build a set G_i containing peers that answered to the majority of queries in the set E_i. The question that arises is how to find these correlations?

In our case, we adapted a formal approach based on Formal Concepts Analysis [6] in order to generate these correlations. In fact, due to the huge quantity of available information, it is necessary to eliminate the redundant data and find the interesting correlations. In this context, a method for data correlations based on the formal concept analysis is proposed. In what follows, we present a brief description of Formal Concepts Analysis.

Formal Concepts Analysis (FCA). Formal Concept Analysis is a branch of applied mathematics. Based on a mathematization of concept and concept hierarchy, it activates mathematical methods for conceptual data analysis and knowledge processing [6].

Definition 1. *A* **formal context** $K = (G, M, I)$ *consists of two sets* G, M *and a relation* I *between* G *and* M. *The elements of* G *are referred to as objects and the elements of* M *as attributes and we assume that* $G \cap M = \emptyset$. *A context may be depicted as a* $|G| \times |M|$ *binary matrix, where the objects of* G *form row labels and the objects* M *form column labels. Let* $mat(K)$ *denotes the matrix representation of* K *; then, we may fully specify the entries of this matrix as:*

$$mat(K)_{ij} = \begin{cases} 1 & if \quad g_i I m_j \\ 0 & otherwise \end{cases} \quad (1)$$

For a set $A \subseteq G$, called an **object-set**, *we define*

$$A^{'} = \{m \in M \ \ |gIm, \ \ \forall \ \ g \in A\} \tag{2}$$

For a set $B \subseteq M$, called an **attribute-set** *we have :*

$$B^{'} = \{g \in G \ \ |gIm \ \ \forall \ \ m \in B\} \tag{3}$$

Definition 2. *A* **Concept** *of the context (G, M, I) is a pair $C = (A, B)$ with $A \subseteq G$, $B \subseteq M$, such that $A^{'} = B$ and $B^{'} = A$. We call $A = Ext(C)$ the* extent *and $B = Int(C)$ the* intent *of the concept $C = (A, B)$.*

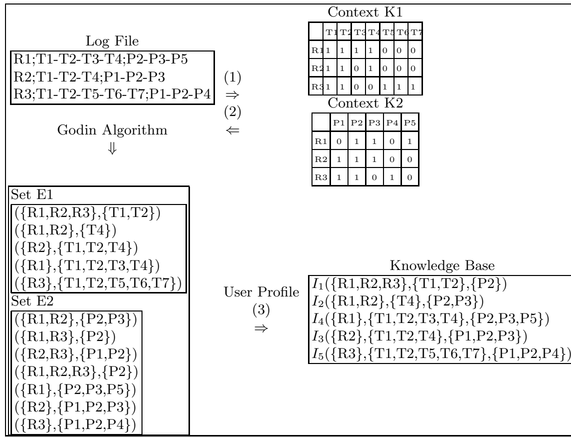


Fig. 1. The different steps for building a knowledge base

FCA Approach for Knowledge Base Building. To apply FCA approach, we used a context $K_1 = (QIds, T, I)$ that represents the semantic relationship between sent queries and their terms. The objects of K_1 are the queries identifiers and the attributes are the queries terms. Thereafter, an algorithm of formal concepts generation (i.e., Godin algorithm [8]) is applied to generate a set of concepts, noted E_1. The concepts of E_1 will be under the following form $c_1(o_1, p_1)$, where o_1 denotes a subset of sent queries and p_1 is a set of common terms of queries in o_1. Hence, o_1 is the *Extent* and p_1 is the *Intent* of the concept $c_1(o_1, p_1)$.

Moreover, to represent the semantic relationship between sent queries and positive peers, we used a second context $K_2(QIds, P, I)$. The objects of K_2 are the queries identifiers and the attributes are the positive peers. We applied the same algorithm in order to generate a set of concepts E_2 that represents the semantic relationship between sent queries and positive peers. The concepts of E_2, will be under the following form $c_2(o_2, p_2)$, where o_2 is a subset of sent queries and p_2 is a set of peers which answered to all queries in o_2.

Finally, we applied *User Profile* algorithm (see Algorithm 1) to generate a set of interests. The inputs of this algorithm are the sets E_1 and E_2. It generates a knowledge base B that contains a set of interests. The $getCloseConcept(c_1, E_2)$ function in Algorithm 1 (see line 3 of Algorithm 1) returns the closest concept to c_1 (i.e., concept having the highest similarity value) from the concepts set E_2. The similarity value between a concept $c_i \in E_1$ and a concept $c_j \in E_2$ is evaluated as follows:

$$Sim(c_i, c_j) = \frac{|Ext(c_i) \cap Ext(c_j)|}{|Ext(c_i) \cup Ext(c_j)|} \tag{4}$$

Where $Ext(c)$ represents the extent of the concept c. Figure 1 represents the different steps for building a knowledge base from a log file.

Figure 1 represents the different steps for building the knowledge base. The first step of this figure represents context generation. The step consists of applying any formal concept generator to obtain the concept sets E_1 and E_2. Finally, the third step consists in applying *User Profile* algorithm (see Algorithm 1) to generate the knowledge base.

Algorithm 1. User Profile

Input:
E_1 : Set of concepts
E_2 : Set of concepts
Output:
B : Knowledge base containing a set of interests

```
1 begin
2     for each c1 ∈ E1 do
3         c2 = getCloseConcept(c1, E2)
4         B = B ∪ {I(Ext(c1), Int(c1), Int(c2))}
5     return (B)
6 end
```

The knowledge bases are periodically updated with information about the new queries. We have defined an incremental strategy to maintain the knowledge bases. It consists in generating a set of interests B' from the history of queries issued after the last update operation. Thereafter, we add this set of interests to the old knowledge base.

3.3 Clustering Layer

The goal of this layer is to organize the P2P network into clusters of peers sharing similar knowledge bases or interests. Each peer p_i in the network establishes

connections with a set of friend peers Fr_i that share similar knowledge bases. The underlying idea of our approach is to represent the knowledge base B_i of a given peer p_i by a set of representative vectors R_i to describe its interests. Thereafter, peers that share similar interests (knowledge bases) make a friendship relation between them. To illustrate this idea, consider Figure 2 which shows a P2P network before applying our clustering method. In this example, peers are connected randomly and each of them has a knowledge base described by two representative vectors. Each representative vector is represented by a shape (i.e., triangle, square,...). After running our clustering method, each peer establishes connections with peers sharing similar knowledge (see Figure 3). By this way, we build a semantic overlay network, wherein peers sharing similar knowledge are "clustered" together.

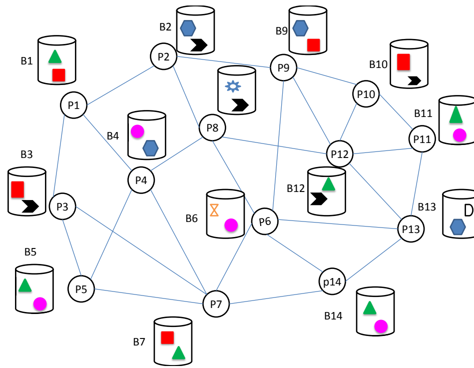


Fig. 2. Random overlay network (P2P network before clustering)

Before describing our peers clustering method, we present the following definitions:

Definition 3. *Representative Vectors* R_i Each peer $p_i \in P$ selects a representative vectors set R_i to describe its knowledge base content. We define, the cluster centroid of a specific past queries set belonging to the p_i' knowledge base as a representative vector $r_{i_j} \in R_i$.

Definition 4. $Distance(r_{i_k}, r_{j_v})$ is the distance measure between $r_{i_k} \in R_i$ and $r_{j_v} \in R_j$; in other words, it is the similarity between two particular clusters belonging to two different knowledge bases B_i and B_j. We use the Euclidean distance between the centroid of two clusters represented by r_{i_k} and r_{j_v}. Let r_{i_k} and r_{j_v} be two representative vectors, the Euclidean distance is defined as follows:

$$Distance(r_{i_k}, r_{j_v}) = \sqrt{\sum (x_i - y_i)^2}$$

where x_i , y_i are respectively the ith components of r_{i_k} and r_{j_v}. Notice that each ith component is the weight of the ith term in the vector.

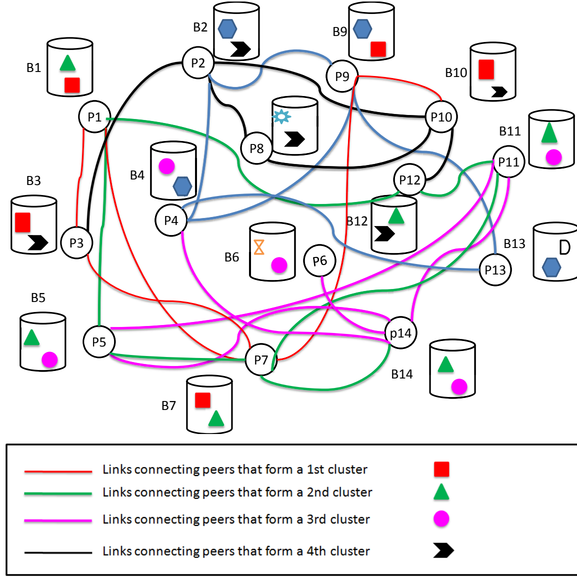


Fig. 3. Semantic overlay network (P2P network after clustering)

Based on the above definitions, we introduce our peer clustering algorithm. It involves two phases *(i)* Computing representative vectors and *(ii)* Community Construction.

Computing Representative Vectors. In our case, each representative vector $r_{i_k} \in R_i$ is a cluster centroid of past queries set belonging to the knowledge base B_i. Each vector can be used to indicate a user's interest. This task is executed when a peer builds or updates its knowledge base in order to update the representative vectors accordingly.

Community Construction. After computing the set R_i of its representative vectors, the peer p_i must search its friend peers in the network. The friends of p_i form a set Fr_i. We define formally this set as follows:

$$Fr_i = \{p_j \in P \ \ / \exists r_{i_k} \in R_i, \ \ r_{j_v} \in R_j \\ such\ that \ \ i \neq j \ \ and \ \ Distance(r_{i_k}, r_{j_v}) < \theta\}$$

Where θ denotes a specified user threshold. To build the set Fr_i, the peer p_i floods a search query containing its representative vectors set R_i within a certain Time To Live (TTL). It sends a query, named *search_Friends*(R_i, TTL), similar to that of ping-pong messages in Gnutella. When a peer p_j receives this query, it computes the distance between each of their representative vectors (see $ComputeSimilarity$ from Algorithm 2); then, p_j answers the query by sending

Algorithm 2. COMPUTESIMILARITY

```
Input:
    R_i : Representative vector of peer p_i.
    R_j : Representative vector of peer p_j.
    θ : A specific user threshold.
Output:
    < r, d >: Couple, where r is the representative vector and d is a
    distance.
1 begin
2    < r, d >=< null, +∞ >
3    for each r1 ∈ R_i do
4        for each r2 ∈ R_j do
5            s = Distance(r1, r2)
6            if s < θ and s < d then
7                r = r2
8                d = s
9 end
```

both its closest representative vector, if it exists, and the distance value. When p_i receives the representative vectors of the closest peers, it selects the best k peers having minimum distance.

Each peer in the network periodically runs this community construction algorithm. Hence, peers are organized dynamically in new semantic clusters. This task is periodically executed offline.

3.4 Bootstrapping Layer

The goal of this layer is to build an initial knowledge for each peer in the network, before the user sends his first search query through the network. Thus, the *training phase* is implicitly executed. To build an initial knowledge base B_{i_0} for a peer p_i, we need to flood some queries that have a semantic relationship with user's intended queries. To generate queries that have a semantic relationship with user's intended queries (user intention), we exploit the shared documents collection to extract few representative queries. In real world, there is a snugness connection between search queries and shared documents. For example, a user who shares documents related to mathematics probably looks for documents in the same domain.

Query Extraction. The idea underlying our proposal is to extract a query Q from any shared document d. The document d and the query Q must have the highest similarity value. In addition, query terms must be representative of the document d. Therefore, we aim to maximize the similarity value document query $Sim(Q, d)$, where Sim is the similarity function between a query Q and a document d. In our case, we chose the Cosine similarity function [17]. Indeed, for

Algorithm 3. Query Extraction

```
   Input:
     C : Documents collection;
     t_max : Length each query;
   Output:
     queriesList : List of selected queries
1  begin
2    d: document
3    t: term
4    w_t: weight of the term t
5    Q: query
6    index : list containing tuples with this form < t, w_t >
7    queriesList = ∅
8    for each d ∈ C do
9      index =<>
10     for each t ∈ d do
11       w_t = computeWeight(t, d, C)
12       index.add(< t, w_t >)
13     Q = generateQuery(index, t_max)
14     queriesList = queriesList + Q
15   return (queriesList)
16 end
```

a given document d from a collection C of shared documents, we compute the weight w_t of each term $t \in d$ using the $computeWeight(t, d, C)$ function called in Algorithm 3. The weight of each term is computed according to the *tf-idf* measure [17]. Thereafter, the first t_{max} terms having the highest weight form a query Q (see line 13 of Algorithm 3).

Building the Initial Knowledge Base. After applying the Query Extraction algorithm, each peer in the network randomly chooses K queries from the generated queries. The value of K will be proportional to the number of shared documents. Thereafter, chosen queries are sent implicitly using the classical flooding techniques [7]. Finally, each peer builds its initial knowledge base from the returned results.

3.5 Semantic Query Spreading Layer

This layer firstly exploits local knowledge base to select relevant peers which are most likely to provide an answer for a forthcoming query. Thereafter, if the number of relevant peers is below a given threshold P_{max}, it adds the best *friend peers*, where we are sure that they are able to select promising peers from their knowledge bases. Hence, the peer selection process is totally semantic. Indeed, when a peer receives a query Q, it performs *QueryRouting* algorithm

(see Algorithm 4) for selecting a set of relevant peers to forward the query. Firstly, this algorithm calls the ($PeerSelection$) algorithm for selecting a set of relevant peers from the local knowledge base (see line 2 of Algorithm 4). Thereafter, if the number of selected peers is below a threshold P_{max}, it calls $addSemantic()$ algorithm (see line 5 of Algorithm 4) that adds a list of the best *friend peers*.

Algorithm 4. QUERY ROUTING ALGORITHM

Input:
B : Knowledge base.
Q: Query to forward.
F_r : Friend list.
P_{max} : The maximum number of peers to be selected.

```
1 begin
2   finalList = PeerSelection(B, Q, P_max)
3   if (|finalList| < P_max) then
4     N = P_max − |finalList|
5     addSemantic(finalList, N, F_r, Q)
6   Forward(Q, finalList)
7 end
```

***PeerSelection* Algorithm.** To choose the relevant peers, $PeerSelection$ is based on a knowledge base B generated by the management user profile layer. Indeed, for a given query Q, our algorithm extracts from B the closest user interest $I(E, F, G)$ to Q using the following similarity function (see line 4 of Algorithm 5).

$$Sim(I(E, F, G), Q) = \frac{F \bigcap Q}{F \bigcup Q}$$

Thereafter, from the set G we determine the list of promising peers (see line 6 of Algorithm 5). If the number of selected peers is lower than a threshold P_{max}, we extract the closest user interest to Q from $B \setminus \{I(E, F, G)\}$ (see lines 7 and 8 of Algorithm 5) and we repeat the same steps.

***addSemantic* Algorithm.** This algorithm involves three steps:

1. *Similarity computing*: in this step we compute the similarity between the representative vector of each friend peers and the query Q;
2. *Sorting step*: this step sorts the list F_r of friend peers according to the similarity value;
3. *Adding*: this step adds to $finalList$ the best N friends having the highest similarity values.

Algorithm 5. PEER SELECTION ALGORITHM

Input:
B : Knowledge base (Set of interests)
Q : Query
P_{max} : The maximum number of peers to be selected for query
Output:
$peers$: List of selected peers

```
1  begin
2      I : Interest
3      peers =<>
4      I = getClosetInterst(Q, B)
5      while I and |peers| < P_max do
6          peers = peers + I.getPeers()
7          B = B \ {I}
8          I = getClosetInterst(Q, B)
9      Return (peers)
10 end
```

4 Experiments

Our experimental study aims to validate the following challenges, discussed in this paper:

1. Starting with an initial knowledge base, instead of cold-startup, to achieve better results during the training phase. To validate this proposal, we studied the retrieval effectiveness and the routing efficiency of our scheme LRS when it starts with or without an initial knowledge base. In the later case, LRS is denoted LRS';
2. Exploiting correlations between past queries and positive peers to represent the user profile, instead of a flat representation based on some statistics about past queries. To validate this proposal, we compared the effectiveness and the efficiency of LRS to an existing query-oriented routing method "Intelligent Search" (IS) and a classical routing method *Gnutella*. We point out again that, IS represents the user profile by past query keywords and associate peers, without taking into account the relationships between them;
3. Avoiding the fail search problem for better retrieval effectiveness and routing efficiency. To validate this proposal, we compared LRS to IS, which does not address the fail search problem.

Several parameters can be considered to simulate the different routing methods. In our case, we simulated the queries and documents distribution models, the knowledge evolution and the overlay size. Indeed, various test scenarios can be performed to detect the impact of these different parameters:

S1. Impact of the initial knowledge base on the routing efficiency and the retrieval effectiveness of *LRS*;
S2. Impact of the knowledge evolution on the routing efficiency and the retrieval effectiveness of *IS* and *LRS*;
S3. Impact of the queries and the documents distribution models on the routing efficiency and the retrieval effectiveness of *Gnutella*, *IS* and *LRS*;
S4. Impact of overlay size on the routing efficiency and the retrieval effectiveness of *Gnutella*, *IS* and *LRS*;
S5. Impact of the maintenance cost of routing indices on the routing efficiency of *LRS* and *IS*.

4.1 Environment

To simulate our approach, we have chosen the PeerSim simulator [12], which is an open source Java tool. The simulation is based on the following parameters:

- *TTL*: The maximum number of hops that a query is allowed to travel (the horizon of the query), initialized to 4.
- P_{max}: The maximum number of peers to be selected for a query, initialized to 4.
- *Overlay size*: For the three first scenarios (**S1, S2 and S3**), we initialised the overlay size to 810 peers. However, we varied the number of peers from 500 to 2300 in order to study the fourth scenario (**S4**).

In addition, as a data source, we used the "Big Dataset" collection developed under the RARE project [20]. This collection is obtained from a statistical analysis on Gnutella system data [9] and the TREC collection [24], which allow us to simulate our algorithm in real conditions. Big Dataset is composed of 25000 documents and 5000 queries. To distribute documents and queries among the set of peers, we used the *Benchmarking Framework for P2PIR* [29]. This framework is configurable and allows the user to define some parameters (e.g. number of peers, distribution method of documents and queries, replication rate of queries and documents, etc.) and provides XML files describing the peers, the associated documents and the queries to be issued. To consider the performance of our approach in the worst case, we have chosen a query replication rate equal to 7 (i.e., the total number of issued queries is 35000) and without documents replication. Furthermore, we have chosen the following distribution models:

- *Uniform*: this method distributes documents according to the uniform law among the peers set. The obtained dataset is named *uniform benchmark* (*UB*).
- *Random*: this method distributes randomly documents among the peers set. The obtained dataset is named *random benchmark* (*RB*).
- *Clustering*: this method clusters similar documents together according to their common terms. Hence, each document's cluster will be affected to one peer. The obtained dataset is named *clustering benchmark* (*CB*).

To evaluate the retrieval *effectiveness* of our approach, we are used the Recall (R) and Precision (P) metrics defined as follows for a given query q [17]:

$$R(q) = \frac{RRD}{RLD} \quad (5)$$

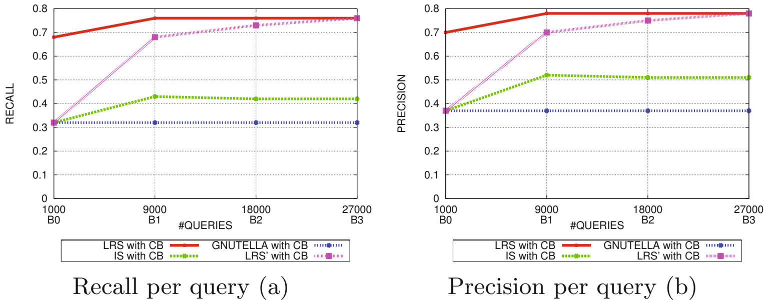
$$P(q)@k = \frac{RRD@k}{RTD} \quad (6)$$

Where, RRD denotes the number of relevant retrieved documents, RLD the number of relevant documents, $RRD@K$ is the number of relevant retrieved documents in the first K rank positions (in our case we fixed K to 3) and RTD denotes the number of retrieved documents.

Besides, we are interested in assessing the routing *efficiency* of our approach by computing the number of visited peers (VP) and the number of messages (MT) per query.

4.2 Results

In this section, we report the results of the five test scenarios presented above.

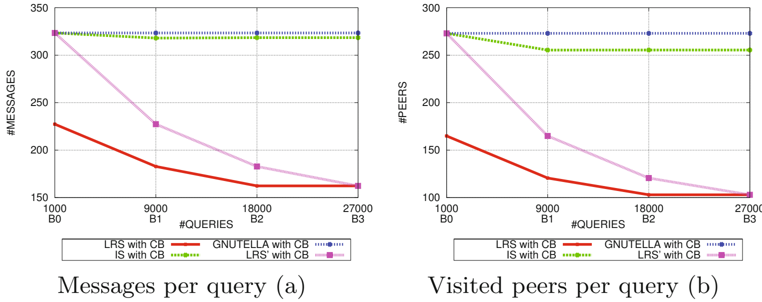


Recall per query (a) Precision per query (b)

Fig. 4. Average recall and precision per query of *Gnutella*, IS, LRS' and LRS

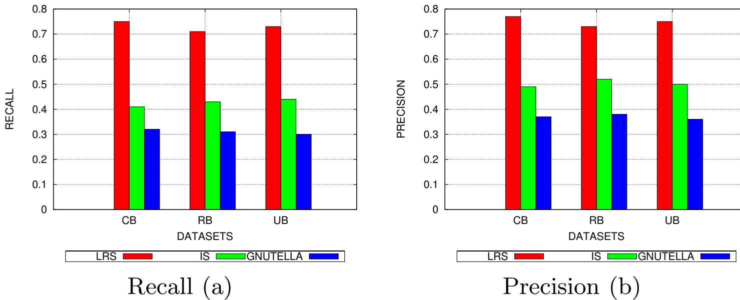
S1. Impact of the Initial Knowledge Base on the Routing Efficiency and the Retrieval Effectiveness of *LRS*. To study the impact of the initial knowledge base, we compared the retrieval effectiveness and the routing efficiency of LRS and LRS'. We note that, LRS starts with an initial knowledge base B_0 generated by the bootstrapping layer. However, LRS' starts with an empty knowledge base. During the simulation task, the knowledge bases have been updated three times for building B_1, B_2 and B_3 per peer.

To compare the retrieval effectiveness and the routing efficiency of LRS and LRS', we computed the average recall and precision, (respectively the average number of messages and visited peers), by intervals of 9000 queries sent from different peers in the P2P network. The simulation of the two methods is based on the "*Clustering Benchmark*" (CB).



Messages per query (a) Visited peers per query (b)

Fig. 5. Average number of messages and visited peers per query of *Gnutella*, *IS*, LRS' and *LRS*

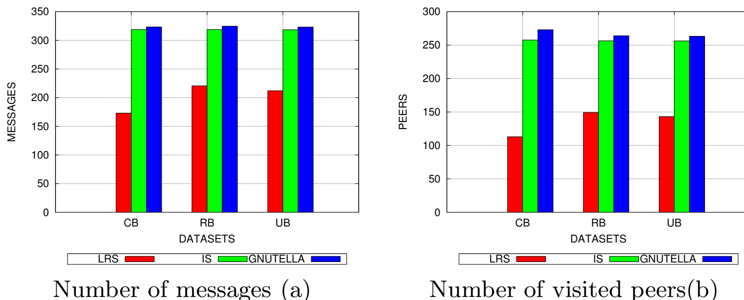


Recall (a) Precision (b)

Fig. 6. Average recall and precision according to *CB*, *RB* and *UB* datasets

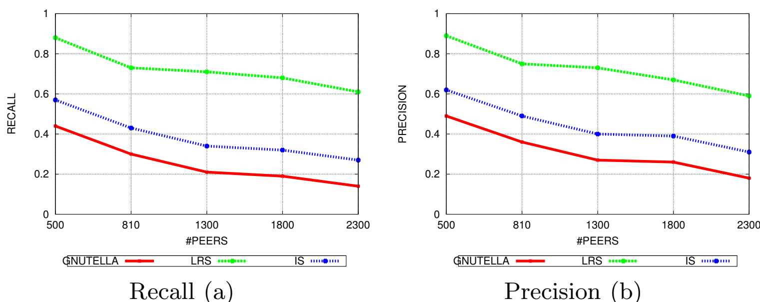
Retrieval effectiveness of LRS' ***and*** *LRS*: Figures 4 (a) and 4 (b) show that *LRS* gives better results than LRS' in terms of recall and precision. Indeed, at the system startup, the average recall and precision for *LRS* are respectively around 0.68 and 0.7, while they are respectively around 0.32 and 0.37 for LRS'. Hence, we deduce that with an initial knowledge base we increase the recall and precision by 112% during the *training phase*, which proves the effectiveness of our solution to the cold-start problem. These results are very encouraging in that they give users good appreciation about the system. Most significantly, users are not obliged to wait for the system until it builds the initial knowledge base.

Routing efficiency of LRS' ***and*** *LRS*: Figures 5 (a) and 5 (b) show that at the system startup, *LRS* carries out better than LRS' in terms of the average number of messages and visited peers. Hence, the average number of messages and visited peers for *LRS* are respectively around 227 and 164, while they are respectively around 323 and 273 for LRS'. Indeed, by starting with an initial knowledge base we decrease significantly the number of messages (around 30%) and the number of visited peers (around 40%), which demonstrates the routing efficiency of our solution for the cold-start problem.



Number of messages (a) Number of visited peers(b)

Fig. 7. Average number of messages and visited peers according to CB, RB and UB datasets



Recall (a) Precision (b)

Fig. 8. Average recall and precision under various numbers of peers

S2. Impact of the Knowledge Evolution on the Routing Efficiency and the Retrieval Effectiveness of IS and LRS. To study the impact of the knowledge evolution on the retrieval effectiveness and the routing efficiency of IS and LRS, we compared the average recall and precision, (respectively the average number of messages and visited peers) by intervals of 9000 queries sent from different peers in the P2P network. The tests are carried out with the CB dataset.

Retrieval effectiveness of IS and LRS: At the beginning, LRS starts with an initial knowledge base prepared by the bootstrapping layer. However, IS starts with a flooding technique since the profiles are not yet learned. Hence, our scheme gives better results than IS in terms of recall and precision during the training phase. Indeed, Figures 4 (a) and 4 (b) show that at the system startup the average recall and precision of LRS are respectively around 0.68 and 0.7, while they are respectively around 0.32 and 0.37 for IS. In addition, we observe that the recall and precision of IS improve over time, as peer profiles are learned. They are increased from 0.30 to 0.46, and respectively from 0.37 to 0.53 after sending 9000 queries. Thereafter, they remain stable respectively at 0.46 and 0.53 because the profiles table of each peers have reached the maximum size. Furthermore, we show that recall and precision of LRS increase after each

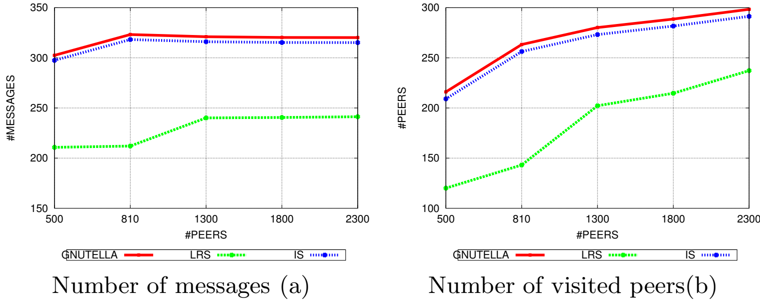


Fig. 9. Average number of messages and visited peers under various numbers of peers

update operation of knowledge bases. Figure 4 (a) shows that the recall of LRS increases from 0.68 to 0.76. Similarly, figure 4 (b) shows that the precision of LRS increases from 0.7 to 0.78. Indeed, throughout the simulation task, our scheme is more effective than IS.

Routing efficiency of IS ***and*** LRS: Figures 5 (a) and 5 (b) show that the number of messages and the number of visited peers of IS decreased from 323 to 318, respectively from 273 to 255 after sending 9000 queries. We observe as well that the number of messages and the number of visited peers of LRS decrease after the two first updates operations of the knowledge base. Figure 5 (a) shows that the number of messages of LRS decreases from 227 to 182, by using B_1, to 162, by using B_2. Similarly, figure 5 (a) shows that the number of visited peers of LRS decreases from 164 to 120 by using B_1, to 102 by using B_2. Hence, we deduce that our scheme is more efficient than IS throughout the simulation task.

S3. Impact of the Queries and the Documents Distribution Models on the Routing Efficiency and the Retrieval Effectiveness of *Gnutella*, *IS* and *LRS*. To study the impact of the queries and the documents distribution models on the routing efficiency and the retrieval effectiveness of *Gnutella*, IS and LRS, we compared their average recall and precision, (respectively their average number of messages and visited peers), according to the CB, UB and RB datasets.

Retrieval effectiveness of *Gnutella*, IS ***and*** LRS: Figure 6 (a) shows that the average recall and precision of *Gnutella* are respectively around 0.32 and 0.37 according to the different datasets. In addition, we observe that the average recall and precision of IS are respectively around 0.43 and 0.5 according to the different datasets. Indeed, IS improves the results of *Gnutella* and the three distribution models do not have an impact on their retrieval effectiveness. Furthermore, Figure 6 (a) shows that LRS increases the recall of IS by 83%, 64% and 64% respectively according the CB, RB and UB datasets. Similarly, Figure 6 (b) shows that LRS increases the precision of IS by 55%, 40% and 49% respectively according the CB, RB and UB datasets. These results confirm that LRS is more effective than IS according to the three datasets.

In addition, we remark that the recall and precision of LRS are close with UB and RB datasets. However, we observe a slight increase (around 5%) of the recall and precision with CB dataset. This increase is due to the nature of the CB dataset (each peer shares a set of similar documents). Hence, when LRS routes a given query to a pertinent peer, this peer can provide more relevant documents, which contributes to the increase of the recall and the precision.

***Routing efficiency of** Gnutella, IS **and** LRS*: Figure 7 (a) shows that IS slightly decreases the number of messages and the number of visited peers of *Gnutella*. Furthermore, we observe that LRS decreases the number of messages of IS by 45%, 30% and 33%, respectively according the CB, RB and UB datasets. Similarly, Figure 7 (b) shows that LRS decreases the number of visited peers of IS by 56%, 41% and 44%, respectively according the CB, RB and UB datasets. These results prove that LRS is more efficient than IS according to the three datasets. In addition, like the recall and precision, we observe that the number of messages and the number of visited peers of LRS are near with a statistical datasets RB and UB. However, there is an important decrease (around 24%) of messages and visited peers, with CB dataset. Indeed, in CB dataset the local collection of each peer is homogeneous. Consequently, LRS swiftly identifies relevant peers for a given query. These peers process once time the same query, which contributes to the decrease of the number of messages and visited peers.

Synthesis: Figures 6 and 7 prove that by (i) exploiting the correlations between past queries and positive peers to represent the user profile, (ii) avoiding the cold-start problem and (iii) avoiding the fail search problem, our hybrid routing scheme LRS improves the retrieval effectiveness and the routing efficiency of the query-oriented routing method IS. Indeed, LRS achieves 83% recall rate while using less than 45% of the number of messages of the IS, with CB dataset. Furthermore, it achieves 64% recall rate while using less than 30% of the number of messages of the IS algorithm, with UB and RB datasets.

S4. Impact of Overlay Size on the Routing Efficiency and the Retrieval Effectiveness of *Gnutella*, *IS* and *LRS*. To evaluate the scalability of *Gnutella*, IS and LRS, we compared their routing efficiency and their retrieval effectiveness by varying the overlay size. The simulation of the different methods is based on the UB dataset.

***Routing effectiveness of** Gnutella, IS **and** LRS*: Figures 8 (a) and 8 (b) show that the recall and the precision of the three methods decreased by increasing the number of peers in the network. This is logical, since by increasing the number of peers in the network the likelihood to find relevant peers decreases, which negatively affects the recall and precision. Furthermore, Figure 8 shows that the recall, (respectively the precision), of *Gnutella* significantly decreased (around 68%) from 0.44 with 500 peers to 0.14 with 2300 peers (respectively from 0.49 to 0.18), which confirms that this classic method is not scalable. Similarly, we observe a decrease (around 52%) of recall and precision of IS from 0.57 to 0.27, (respectively from 0.62 to 0.31). Indeed, with 2300 peers IS is not effective.

In addition, we observe that the recall, (respectively the precision), of *LRS* decreases from 0.88 to 0.61, (respectively from 0.89 to 0.59). Although this decrease is around 30%, recall and precision of *LRS* remain high (around 0.60). We point out again that, in our tests we chose the worst case, since in the *UB* dataset the documents are not replicated. These results are very encouraging and prove that our scheme *LRS* is scalable.

***Retrieval efficiency of* Gnutella*, IS and* LRS**: Figure 8 shows that the number of messages, (respectively the number of visited peers), of *Gnutella* increases from 302 with 500 peers to 320 with 2300 peers, (respectively from 216 to 298). Similarly, we observe that the number of messages (respectively the number of visited peers), of *IS* increases from 297 with 500 peers to 315 with 2300 peers, (respectively from 209 to 291). We deduce that the number of messages and the number of visited peers with a small network are lower than those with a larger network. Indeed, in a small network there is a high probability that peers receive several times the same queries. In this case, they stop the propagation, which reduces the number of messages and the number of visited peers. In addition, Figure 8 shows that the number of messages and the number of visited peers of *LRS* remain acceptable even if the overlay size increases (respectively around 241 and 237), which prove the scalability of *LRS*.

S5. Impact of the Maintenance Cost of Routing Indices on the Routing Efficiency of the *LRS* and *IS*. In our experimental study, *LRS* uses a single representative vector to represent the knowledge base of each peer. Indeed, during the simulation task, each peer flooded 2 representative vectors for searching and maintaining its list of friend peers. The cost of flooding of the 2 representative vectors is 646 (i.e., 2×323) messages per peer. Thus, the cost of the clustering algorithm (i.e., maintenance cost of routing indices) of *LRS* is 523260 messages (i.e., $number\ of\ messages\ per\ peers \times number\ of\ peers = 646 \times 810$). However, *IS* does not require a communication cost to maintain peers routing indices. Indeed, in our experimental study, we have chosen the first mechanism that *IS* propose to build the routing indices.

Figure 7 shows that the average number of messages per query of *IS* and *LRS* are respectively 319 and 173. Hence, *LRS* decreases the number of messages per query of *IS* by 146 messages. Knowing that the number of queries issued by all peers is 35000, we deduce that *LRS* forwards 5110000 (i.e., 146×35000) less messages than *IS*. By considering the cost of building and maintenance of the routing routing indices, *LRS* uses 4586740 (i.e., $5110000 - 523260$) less messages than *IS* (i.e., less than 131 message per query). These results show that the maintenance cost of routing indices of *LRS* does not affect its routing efficiency.

5 Conclusion and Future Works

In this paper we presented a new learning query routing scheme *LRS* for unstructured P2P systems. The proposed scheme is fully distributed and scales

well with the size of the network. To route more efficiently future queries, *LRS* selects pertinent peers based on the user's profile (user interests), which is generated from the past queries. Unlike the proposed schemes in the literature, firstly *LRS* exploits repetition rate for either keywords seen in sent queries or relationships between them to represent the user interests. Secondly, it addresses the bootstrapping problem by building an initial knowledge base founded on the shared documents. Finally, to palliate the unsuccessful relevant peers search problem, our scheme organizes the P2P network into clusters of peers sharing similar knowledge. The experimental results highlights the retrieval effectiveness and the routing efficiency of our scheme. Obvious pointers for future work include the proposition of a learning routing scheme that take into account other dimensions of the user context like location, time and device.

References

[1] Callan, J.P., Lu, Z., Croft, W.B.: Searching distributed collections with inference networks. In: The 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 21–28 (1995)
[2] Carchiolo, V., Malgeri, M., Mangioni, G., Nicosia, V.: Emerging structures of p2p networks induced by social relationships. Comput. Commun. 31(3), 620–628 (2008)
[3] Christoph, T., Steffen, S., Adrian, W.: Semantic query routing in peer-to-peer networks based on social metaphors. In: 13th International World Wide Web Conference (WWW 2004), New York City, USA, pp. 55–68 (2004)
[4] Ciraci, S., Körpeoglu, I., Ulusoy, O.: Reducing query overhead through route learning in unstructured peer-to-peer network. J. Netw. Comput. Appl. 32(3), 550–567 (2009)
[5] Crespo, A., Garcia-Molina, H.: Routing indices for peer-to-peer systems. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria, pp. 23–30 (2002)
[6] Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, New York (1997)
[7] Gnutella: Gnutella Web site (March 2009), `http://www.gnutella.com`
[8] Godin, R., Missaoui, R., Alaoui, H.: Incremental concept formation algorithms based on galois (concept) lattices. Computational Intelligence 11(2), 246–267 (1995)
[9] Goh, S.-T., Kalnis, P., Bakiras, S., Tan, K.-L.: Real datasets for file-sharing peer-to-peer systems. In: Zhou, L.-Z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, pp. 201–213. Springer, Heidelberg (2005)
[10] Hsiao, H.C., Su, H.W.: On optimizing overlay topologies for search in unstructured peer-to-peer networks. IEEE Transactions on Parallel and Distributed Systems 23, 924–935 (2012)
[11] Huang, J., Li, X., Wu, J.: A class-based search system in unstructured p2p networks. In: Proceedings of the 21st International Conference on Advanced Networking and Applications (AINA 2007), pp. 76–83 (2007)
[12] Jelasity, M., Montresor, A., Jesi, G.P., Voulgaris, S.: The peersim simulator (March 2010), `http://peersim.sf.net`

[13] Jin, H., Ning, X., Chen, H., Yin, Z.: Efficient query routing for information retrieval in semantic overlays. In: Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006), Dijon, France, pp. 23–27 (2006)
[14] Kalogeraki Vana, G.D., Zeinalipour-Yazti, D.: A local search mechanism for peer-to-peer networks. In: Proceedings of the Eleventh International Conference on Information and Knowledge Management (CIKM 2002), McLean, Virginia, USA, pp. 300–307 (2002)
[15] Kumar, A., Xu, J., Zegura, E.W.: Efficient and scalable query routing for unstructured peer-to-peer networks. In: (INFOCOM 2005), Miami, USA, pp. 1162–1173 (2005)
[16] Luis, G., Hector, G., Anthony, T.: The effectiveness of gloss for the text database discovery problem. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (1994)
[17] Makhoul, J., Kubala, F., Schwartz, R., Weischedel, R.: Performance measures for information extraction. In: Proceedings of DARPA Broadcast News Workshop (DARPA 1999), Herndon, VA, pp. 249–252 (1999)
[18] Mandreoli, F., Martoglia, R., Penzo, W., Sassatelli, S.: Data-sharing p2p networks with semantic approximation capabilities. IEEE Internet Computing 13, 60–70 (2009)
[19] Raja, C., Bruno, D., Georges, H.: Définition et diffusion de signatures sémantiques dans les systèmes pair-à-pair. In: Extraction et gestion des connaissances (EGC 2006), Lille, France, pp. 463–468 (2006)
[20] RARE: Rare project. (March 2010), `http://www-inf.it-sudparis.eu`
[21] Sergio, A.P., Carlos, B., Nuno, P., David, H.: Scalable bloom filters. Inf. Process. Lett. 101
[22] Shen, W.W., Su, S., Shuang, K., Yang, F.C.: Skip: an efficient search mechanism in unstructured p2p networks. The Journal of China Universities of Posts and Telecommunications 17(5), 64–71 (2011)
[23] Shi, C., Han, D., Liu, Y., Meng, S., Yu, Y.: A dynamic routing protocol for keyword search in unstructured peer-to-peer networks. Computer Communications 31(2), 318–331 (2008)
[24] TREC: Text REtrival Conference (March 2010), `http://trec.nist.gov`
[25] Yang, B., Garcia-Molina, H.: Improving search in peer-to-peer networks. In: The 22nd International Conference on Distributed Computing Systems (ICDCS 2002), Vienna, Austria, pp. 5–14 (2002)
[26] Yeferny, T., Arour, K.: Learningpeerselection: A query routing approach for information retrieval in p2p systems. In: International Conference on Internet and Web Applications and Services (ICIW 2010), Barcelona, Spain, pp. 235–241 (2010)
[27] Yeferny, T., Arour, K.: Efficient routing method in p2p systems based upon training knowledge. In: The Eighth International Symposium on Frontiers of Information Systems and Network Applications, in Conjunction with AINA (WAINA 2012), Fukuoka, Japan, pp. 300–305 (2012)
[28] Yeferny, T., Bouzeghoub, A., Arour, K.: A query learning routing approach based on semantic clusters. International Journal of Advanced Information Technology (IJAIT) 1(6) (2011)
[29] Zammali, S., Arour, K.: *P2PIRB*: Benchmarking framework for *P2PIR*. In: Hameurlain, A., Morvan, F., Tjoa, A.M. (eds.) Globe 2010. LNCS, vol. 6265, pp. 100–111. Springer, Heidelberg (2010)
[30] Zhu, Y., Yang, X., Hu, Y.: Making search efficient on gnutella-like p2p systems. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), Washington, DC, USA, pp. 1–56 (2005)

Open Streaming Operation Patterns

Qiming Chen and Meichun Hsu

HP Labs Palo Alto, California, USA
Hewlett Packard Co.
{qiming.chen,meichun.hsu}@hp.com

Abstract. We describe our *canonical dataflow operator framework* for distributed stream analytics. This framework is characterized by the notion of *open-executors*. A dataflow process is composed by chained operators which form a graph-structured topology, with each logical operator executed by multiple physical instances running in parallel over distributed server nodes. An open executor supports the streaming operations with specific characteristics and running pattern, but is *open* for the application logic to be plugged-in. This framework allows us to provide automated and systematic support for executing, parallelizing and granulizing the continuous operations.

We illustrate the power of this approach by solving the following problems: first, how to categorize the meta-properties of stream operators such as the I/O, blocking, data grouping characteristics, for providing unified and automated system support; next, how to elastically and correctly parallelize a stateful operator that is history-sensitive, relying on the prior state and data processing results; how to analyze unbounded stream granularly to ensure sound semantics (e.g. aggregation); and further, how to deal with parallel sliding window based stream processing systematically. These capabilities are not systematically supported in the current generation of stream processing systems, but left to user programs which can result in fragile code, disappointing performance and incorrect results. Instead, solving these problems using open-executors benefits many applications with system guaranteed semantics and reliability.

In general, with the proposed canonical dataflow operator framework we can *standardize* the operator execution patterns, and to support these patterns systematically and automatically. The value of our approach in real-time, continuous, elastic data-parallel and topological stream analytics has been revealed by the experiment results.

1 Introduction

Real-time stream analytics has increasingly gained popularity since enterprises need to capture and update business information just-in-time, analyze continuously generated "moving data" from sensors, mobile devices, social media of all types, and gain live business intelligence.

We have built a stream analytics platform with code name ***Fontainebleau*** for dealing with *continuous, real-time* data-flow with *graph-structured* topology. This platform is *parallel* and *distributed* with each logical operator executed by multiple

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 83–104, 2013.
© Springer-Verlag Berlin Heidelberg 2013

physical instances running in parallel over distributed server nodes. The stream analysis operators are defined by users flexibly. From stream abstraction point of view, our stream analytics cluster is positioned in the same space of System S(IBM), Dryad(MS), Storm(Tweeter), etc. However, this work aims to advance the state of art by providing canonical execution support for stream analysis operators.

1.1 The Challenges

A stream analytics process is composed by multiple operators and pipes connecting these operators. The operators for stream analysis have certain meta-properties representing their I/O characteristics, blocking characteristics, data grouping characteristics, etc, as well as the functionalities common to various types of applications, which can be categorized for introducing unified system support. Categorizing stream operators and their running patterns to provide automatic support accordingly, can ensure the operators to be executed optimally and consistently, as well as ease user's effort for dealing with these properties manually which is often tedious and risky. Unfortunately, this issue has been missed by the existing stream processing systems.

There exist several key requirements in stream processing which demand automated and systematic support. First, to scale out, the data-parallel execution of operators must be taken into account, where how to ensure the correctness of data-parallelism is the key issue which requires the appropriate system protocol to guarantee; particularly in parallelizing stateful stream operators where the stream data partitioning and data buffering must be consistent. Next, stream processing is often made in granule. For example, to provide sound aggregation semantics (e.g. sum), the infinite input data stream must be processed chunk by chunk where each operator may punctuate data based on different chunking criteria such as in 1-minute or 1-hour time windows (certain constraints apply, e.g. the frame of a downstream operator must be the same as, or some integral number of, the frame of its upstream operator). Granulizing dataflow analytics represents another kind of common behavior of stream operators which also need to be supported systematically.

Current large-scale data processing tools, such as Map-Reduce, Dryad, Storm, etc, do not address these issues in a canonical way. As a result, the programmers have to deal with them on their own, which can lead to fragile code, disappointing performance and incorrect results.

1.2 The Proposed Solution

The operators on a parallel and distributed dataflow infrastructure are performed by both the infrastructure and the user programs, which we refer to as their **template behavior** and **dynamic behavior**. The template behavior of a stream operator depends on its meta-properties and its running pattern. For example, a map-reduce application is performed by the Hadoop infrastructure as well as the user-coded map

function and reduce function. Our streaming platform is more flexible and elastic than Hadoop in handling dynamically parallelized operations in a general graph structured dataflow topology, and our focus is placed on supporting the template behavior, or operation patterns, **automatically** and **systematically**.

Unlike applying an operator to data, stream processing is characterized by the flowing of data through a *stationed* operator. We introduce the notion of **open-station** as the container of a stream operator. The stream operators with certain common meta-properties can be executed by the class of open-stations specific to these operators. Open-stations are classified into a station hierarchy. Each class provides an **open-executor** as well as related system utilities. In the OO programming context, the open-executor is coded by invoking certain abstract functions (methods) to be implemented by users based on their application logic. In this way the station provides designated system support, while *open* for the application logic to be plugged-in. In this work we use the proposed architecture to solve several typical stream processing problems.

The key to ensure safe parallelization is to handle data flow group-wise - for each vertex representing a logical operator in the dataflow graph; the operation parallelization with multiple instances comes with input data partition (grouping) which is consistent with the data buffering at each operation instance. This ensures that in the presence of multiple execution instances of an operator, O, every stream tuple is processed *once and only once* by one of the execution instances of O; the historical data processing states of every group of the partitioned data are buffered with *one and only one execution instance* of O. Our solution to this problem is based on the open station architecture.

The key to ensure the granule semantics is to handle dataflow chunk wise by punctuating and buffering data consistently. Our solution to this problem is also based on the open station architecture.

As a generalization of these solutions, we show how to use the open station architecture to provide system support for handling parallel sliding window based stream processing.

In general, the proposed canonical operation framework allows us to *standardize* various operational patterns of stream operators, and have these patterns supported systematically and automatically. Our experience shows its power in real-time, continuous, elastic data-parallel and topological stream analytics.

The rest of this paper is organized as follows: section 2 describes the notions of open-station and open-executor; then based on these notions section 3 discusses how to guarantee the correctness of data-parallel execution of stateful operations, and how to deal with the granular execution of stream operations; in section 4 we further show how to use the open station architecture to provide system support for handling parallel sliding window based stream processing; some experimental results are illustrated in section 5; finally section 6 compares with related work and concludes the paper.

2 Open Station and Open Executor of Stream Operator

2.1 Continuous, Parallel and Elastic Stream Analytics Platform

Fontainebleau is a real-time, continuous, parallel and elastic stream analytics platform. There are two kinds of nodes on the cluster: the *coordinator node* and the *agent nodes* with each running a corresponding daemon. A dataflow process is handled by the coordinator and the agents spread across multiple machine nodes. The coordinator is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures, in the way similar to Hadoop's job-tracker. Each agent interacts with the coordinator and executes some operator instances (as threads) of the dataflow process. The *Fontainebleau* platform is built using several open-source tools, including ZooKeeper[12], ØMQ[11], Kryo[13], Storm[14], etc. ZooKeeper coordinates distributed applications on multiple nodes elastically. ØMQ supports efficient and reliable messaging. Kryo deals with object serialization. Storm provides the basic dataflow topology support.

A stream is an unbounded sequence of tuples. A stream operator transforms a stream into a new stream based on its application-specific logic. The stream transformations are packaged into a graph-structured "topology" which is the top-level dataflow process. When an operator emits a tuple to a stream, it sends the tuple to every successor operators subscribing to that stream. A stream grouping specifies how to group and partition the tuples input to an operator. There exist a few different kinds of stream groupings such as hash-partition, replication, random-partition, etc.

To support elastic parallelism, we allow a logical operator to be executed by multiple physical instances, as threads, in parallel across the cluster; they pass messages to each other in a distributed way. Using the ØMQ library [11], message delivery is reliable; messages never pass through any sort of central router, and there are no intermediate queues.

To provide an overview, we use a simplified as well as extended Linear-Road (LR) benchmark to illustrate the notion of stream process. The LR benchmark models the traffic on 10 express ways; each express way has two directions and 100 segments. Cars may enter and exit any segment. The position of each car is read every 30 seconds and each reading constitutes an event, or stream element, for the system. A car position report has attributes *vehicle_id*, *time* (in seconds), *speed* (mph), *xway* (express way), *dir* (direction), *seg* (segment), etc. With the simplified benchmark, the traffic statistics for each highway segment, i.e. the number of active cars, their average speed per minute, and the past 5-minute moving average of vehicle speed, are computed. Based on these per-minute per-segment statistics, the application computes the tolls to be charged to a vehicle entering a segment any time during the next minute. As an extension to the LR application, the traffic statuses analyzed and reported every hour. The logical stream process for this example is given in Fig. 1.

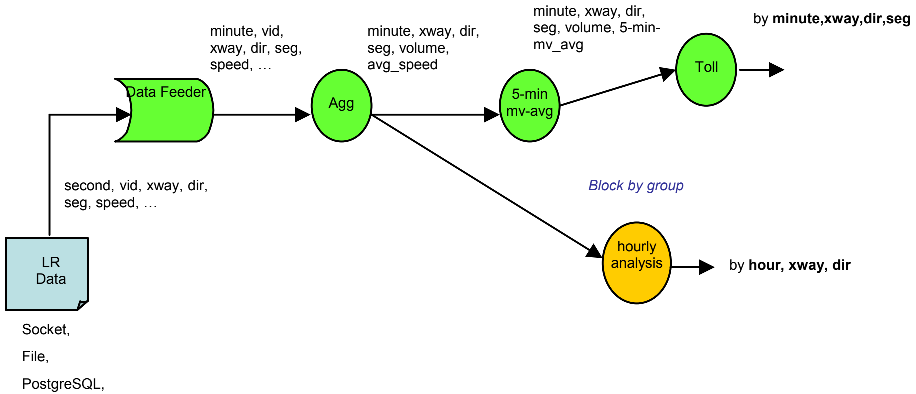


Fig. 1. The extended LR logical dataflow process with operators linked in a topology

This stream analytics process is specified by the Java program illustrated below.

```
  public class LR_Process {
…
public static void main(String[] args) throws Exception {
        ProcessBuilder builder = new ProcessBuilder();
        builder.setFeederStation("feeder", new LR_Feeder(args[0]), 1);
        builder.setStation("agg", new LR_AggStation(0, 1), 6) .hashPartition("feeder",
          new Fields("xway", "dir", "seg"));
        builder.setStation("mv", new LR_MvWindowStation(5), 4).hashPartition("agg",
          new Fields("xway", "dir", "seg"));
        builder.setStation("toll", new LR_TollStation(), 4).hashPartition("mv",
          new Fields("xway", "dir", "seg"));
        builder.setStation("hourly", new LR_BlockStation(0, 7), 2).hashPartition("agg",
          new Fields("xway", "dir"));
        Process process = builder.createProcess();
        Config conf = new Config();   conf.setXXX(...);  …
        Cluster cluster = new Cluster();
        cluster.launchProcess("linear-road", conf, process);
        …
  }
```

In the above topology specification, the hints for parallelization are given to the operators “agg” (6 instances), “mv” (5 instances), “toll” (4 instances) and “hourly” (2 instances), the platform may make adjustment based on the resource availability. Then the physical instances of these operators for data-parallel execution are illustrated in Fig 2.

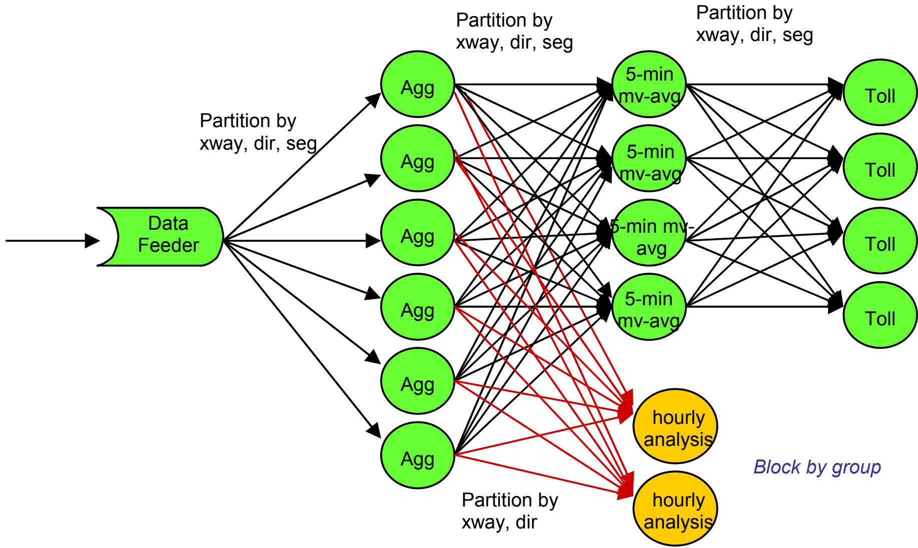


Fig. 2. The LR dataflow process instance with elastically parallelized operator instances

2.2 Meta Characteristics of Operators

Stream operators have certain characteristics in several dimensions, such as the provisioning of initial data, the granularity of event processing, memory context, invocation patterns, results grouping and shuffling, etc, which may be considered as the meta-data, or the design pattern of operators. Further, the operators for supporting a kind of applications also have certain common characteristics. Below we briefly list some characteristics.

I/O Characteristics specifies the number of input tuples and the output tuples the stream operator is designed to handle the stream data *chunk-wise*. Examples are 1:1 (one input/one output), 1:N (one input/multiple outputs), M:1(multiple inputs/ one output) and M:N (multiple inputs/ multiple outputs). Accordingly we can classify the operators into Scalar (1:1); Table Valued (TV) (1:N); Aggregate (N:1), etc, for each chunk of the input. Currently we support the following chunking criteria for punctuating the input tuples: (a) by cardinality, i.e. number of tuples; and (b) by granule as a function applied to an attribute value, e.g. *get_minute* (timestamp in second).

Blocking Characteristics tells that in the multiple input case, the operator applies to the input tuple one by one incrementally (e.g. per-chunk aggregation), or first pools the input tuples and then apply the function to all the pooled tuples. Accordingly the block mode can be *per-tuple* or *blocking*. Specifying the blocking characteristics tells the system to invoke the operator in the designated way, and save the user's effort to handle them in the application program.

Caching Characteristics is related to the 4 levels potential cache states:

- per-process state that covers the whole dataflow process with certain initial data objects;
- Per-chunk state that covers the processing of a chunk of input tuples with certain initial data objects;
- Per-input state that covers the processing of an input tuple possibly with certain initial data objects for multiple returns;
- Per-return state that covers the processing of a returned tuple.

Grouping Characteristics tells a topology how to send tuples between two operators. There's a few different kinds of stream groupings. The simplest kind of grouping is called a "random grouping" which sends the tuple to a random task. It has the effect of evenly distributing the work of processing the tuples across all of the consecutive downstream tasks. The hash grouping is to ensure the tuples with the same value of a given field go to the same task. Hash groupings are implemented using consistent hashing. There are a few other kinds of groupings.

Function Characteristics underlies the common feature of a kind of stream processing applications. Support those features systematically can ease the effort and improve the quality of application development.

2.3 Stationed Streaming Operators

Ensuring the characteristics of stream operators by user programs is often tedious and not system guaranteed. Instead, *categorizing the common classes of operation characteristics and supporting them automatically and systematically* can simplify user's effort and enhance the quality of streaming application development. This has motivated us to introduce **open-stations** for holding stream operators and encapsulating their characteristics – towards the **open station class hierarchy** (Fig 3).

Each open-station class is provided with an "open executor" as well as related system utilities for executing the corresponding kind of operators; that "open" executor invokes the abstract methods, which are defined in the station class but implemented by users with the application logic. In this way a station provides the designated system support, while open for the application logic to be plugged-in. A user defined operator captures the designated characteristics by subclass the appropriate station, and captures the application logic by implementing the abstract methods accordingly.

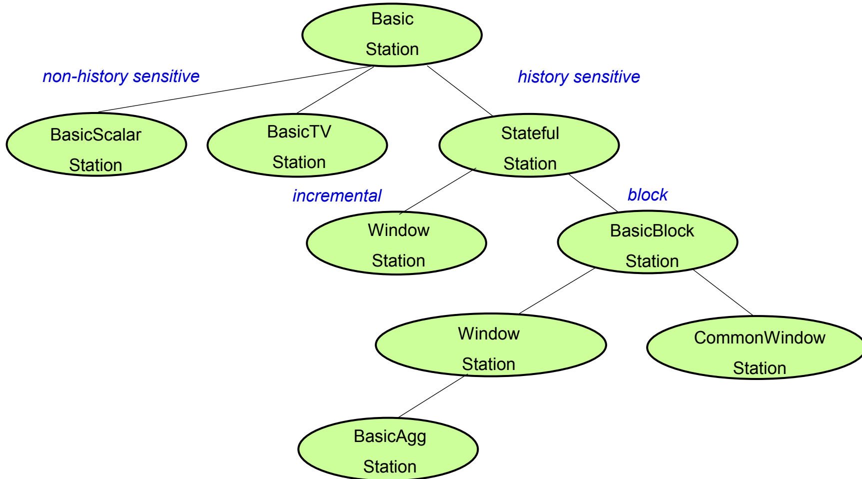


Fig. 3. Station Hierarchy Example

2.4 Open Executor

Specified in a station class, there are two kinds of pre-prepared methods: the system defined ones and the user defined ones.

- The system defined methods include the **open-executor** and other utilities which is open to plugging-in application logic, in the sense that they invoke the abstract methods to be implemented by users according to the application logic.
- The abstract methods to be implemented by the user based on the application logic.

For example, the WindowStation that extends BasicBlockStation, is used to support chunk-wise stream processing, where the framework provided functions, hidden from user programs, include

```
public boolean nextChunk(Tuple, tuple) {// group specific …}
public void execute(Tuple tuple, BasicOutputCollector collector) {
    boolean new_chunk = nextChunk(tuple);
    String grp = getGroupKey(tuple);
    GroupMeasures gm = null;
    if (new_chunk) {
        gm = getGKV().dump(grp);
    }
    updateState(getGKV(), tuple, grp);
    if (new_chunk) { //emit last chunk
        processChunkByGroup(gm, collector);
    }
}
```

The three functions marked Station are to be implemented based on the application logic; the others are system defined for encapsulating the chunk-wise stream processing semantics.

In addition to offering the dataflow operation "executor" abstraction, introducing open station also aims to provide canonical mechanisms to **parallelize stateful and granule dataflow process**. The core is to handle data flow chunk-wise and group-wise - for each vertex representing a logical operator in the dataflow graph; the operation parallelization (launching multiple instances) comes with input data partition (grouping) which is consistent with the data buffering at each operation instance. These are discussed in the following sections.

3 Support Parallelized and Granulized Stream Processing Patterns

3.1 Data-Parallel Execution of Operators

Under our approach, logically, the dataflow elements, i.e. tuples, either originated from a data-source or derived by a logical operator, say *A*, are sent to one or more receiving logical operators, say *B*. Since each logical operator may have multiple execution instances, the dataflows from *A* to *B* actually form multi-to-multi messaging channels.

To handle data-parallel operations, an operator property: parallelism hint, can be specified, that is the number (default to 1) of station threads for running the operator. The number of actual threads will be judged by the infrastructure and load-balanced over multiple available machines.

For the sake of correct parallelism, the stream from *A*'s instances to *B*'s instances are sent in a *partitioned* way (e.g. hash-partition) such that the data sent from any instance of *A* to the instances of *B* are partitioned in the same way. This is similar to the data shuffling from a Map node to a Reduce node, but in more general dataflow topology.

Although our platform offers the flexibility of dataflow grouping with options hash-partition, random-partition, range-partition, replicate, etc, the platform enforces the use of hash partition for the parallelized operators. In case an operator is specified to have parallel instances in the user's dataflow process specification, the input stream to that operator must be defined as hash-partitioned; otherwise the process specification would be invalidated.

Further, there can be multiple logical operators, B_1, B_2, …, B_n, for receiving the output stream of *A*, but each with different data partition criterion, called *inflow-grouping-attributes* (a la SQL group by). The tuples falling in the same partition, i.e. grouped together, have the same "***inflow-group-keys***". For example, the tuples representing the traffic status of an express way (*xway*), direction (*dir*) and segment (*seg*), are partitioned, thus grouped by attributes <xway, dir, seg>; tuples of each group has the same inflow-group-key derived from the values of xway, dir and seg. An operation instance may receive multiple groups of data. The abstract method, ***getGroupKey***(tuple), must be implemented, which is invoked by the corresponding open-executor.

3.2 Parallelize Stateful Streaming Operators Group-Wise

A stateful operator caches its state for future computation, and therefore is history sensitive. When a logical stateful operator has multiple instances, their input data must be partitioned, and the data partition must be consistent with the data buffering.

For example, given the logical operation, *O*, for calculating moving-average and with the input stream data partitioned by <xway, dir, seg>, the data buffers of its execution instances are also partitioned by <xway, dir, seg>, which is prepared and enforced by the system.

For history-sensitive data-parallel computation, an operation instance keeps a state computed from its input tuples (other static states may be incorporated but not the focus of this discussion). We generally provide this state as a KV store where keys, referred to as *cachhing-group-keys*, are Objects (e.g. String) extracted from the input tuples, and values are Objects derived from the past and present tuples such as numerical objects (e.g. sum, count), list objects (certain values derived from each tuple), etc. the multiple instances of a logical operation can run in data-parallel provided that the inflow-group-keys are used as the caching group-keys. In this sense we refer to the KV store as Group-wise KV store (GKV). APIs for accessing the GKV are provided as well. As illustrated in the last section, an important abstract method, `updateState()`, is defined and to be implemented by users.

With the above mechanisms, in the presence of multiple execution instances of an operator, every stream tuple is processed **once and only once** by one of the execution instances; the data processing states of every group of the partitioned input data (e.g. the tuples belonging to the same segment of the an express-way in a direction) are buffered in the function closure of **one and only one execution instance** of that operator. These properties are common to a class of tasks thus we support them in the corresponding station class, that, substantially, is subclassifiable.

3.3 Window Based Stream Analytics

Although a data stream is unbounded, very often applications require those infinite data to be analyzed granularly. Particularly, when the stream operation involves the aggregation of multiple events, for semantic reason the input data must be punctuated into bounded chunks. This has motivated us to execute such operation *window by window* to process the stream data *chunk by chunk.*

For example, in the previous car traffic example, the operation "agg" aims to deliver the average speed in each express-way's segment per minute. Then the execution of this operation on an infinite stream is made in a sequence of *windows*, one on each stream chunks. To allow this operation to apply to the stream data one chunk at a time, and to return a sequence of chunk-wise aggregation results, the input stream, is cut into 1 minute (60 seconds) based chunks, say S_0, S_1, ...S_i, ... such that the execution semantics of "agg" is defined as a sequence of one-time aggregate operation on the data stream input minute by minute.

In general, given an operator, *O*, over an infinite stream of relation tuples *S* with a criterion ϑ for cutting *S* into an unbounded sequence of chunks, e.g. by every

1-minute time window, $<S_0, S_1, \ldots, S_i, \ldots>$ where S_i denotes the *i-th* "chunk" of the stream according to the chunking-criterion ϑ. The semantics of applying O to the unbounded stream S lies in

$$Q(S) \rightarrow < Q(S_0), \ldots Q(S_i), \ldots >$$

which continuously generates an unbounded sequence of results, one on each *chunk* of the stream data.

Punctuating input stream into chunks and applying operation *window by window* to process the stream data *chunk by chunk*, is a template behavior common to many stream operations, thus we consider it as a kind of meta-property of a class of stream operations and support it automatically and systematically by our operation framework. In general, we host such operations on the **window station** (or the ones subclassing it) and provide system support in the following aspects (please refer to the window station example given previously).

- A window station hosts a stateful operation that is data-parallelizable, and therefore the input stream must be hash-partitioned which is consistent with the buffering of data chunks as described in the last section.
- Several types of stream punctuation criteria are specifiable, including punctuation by cardinality, by time-stamps and by system-time period, which are covered by the system function

  ```
  public boolean nextChunk(Tuple, tuple)
  ```

  to determine whether the current tuple belongs to the next chunk or not.
- If the current tuple belongs to the new chunk, the present data chunk is dumped from the chunk buffer for aggregation/group-by in terms of the user-implemented abstract method *processChunkByGroup*().
- Every input tuple (or derivation) is buffered, either into the present or the new chunk.

By specifying additional meta properties and by subclassing the window station, more concrete system support can be introduced. For example, an aggregate of a chunk of stream data can be made once by end of the chunk, or tupe-wise incrementally. In the latter case an abstract method for per-tuple updating the partial aggregate is provided and implemented by the user.

The *paces of dataflow* wrt timestamps can be different at different operators; for instance, the "agg" operator is applied to the input data minute by minute, so are some downstream operators of it; however the "hourly analysis" operator is applied to the input stream minute by minute, but generates output stream elements hour by hour.

The combination of group-wise and chunk-wise stream analytics provides a generalized abstraction for parallelizing and granulizing the continuous and incremental dataflow analytics.

4 Support Parallel Sliding Window Stream Processing Patterns

In this section we extend our discussion to Parallel Sliding Window (PSW) based stream analysis and illustrate the benefits of open stations in dealing with PSW. PSW based stream processing has certain meta-properties in punctuating and grouping input data, in retaining and shifting intermediate results, and in synchronizing parallel chunking. Generalizing and categorizing these operators and their running patterns allows us to provide automatic support accordingly, to ensure the operators to be executed optimally and consistently, as well as ease user's effort for dealing with them.

We build abstract stations to support the common PSW related features such as handling punctuation and parallelism where having the application specific semantics left as abstract methods for users to implement. The abstract stations form a hierarchy; they provide the mechanisms for managing the data granules, the slide and window boundaries, for punctuating input data stream for switching slides and windows, as well as for retaining and intermediate results. In general, they provide system support for synchronizing the slide and window switching wrt multiple, parallel input streams.

4.1 Sliding Window Based Stream Analytics

In stream processing, the tuples transferred between tasks can be granule based on timestamps or so; and we introduce three levels of boundaries for grouping data in the context of PSW: ***granule***, ***slide*** and ***window***; a granule is the basic unit for grouping data, it could be, for example, a chunk of N tuples or the tuples with timestamps falling in one minute; a slide is defined as a given number or range of granules, for example a slide of 10 minute is composed by 10 one-minute granules; a window is also defined as a given number or range of granules, but the size of a window is at least the size of a slide.

A sliding window based operation keeps the following variables for dealing with sliding window semantics.

- ***window_size*** – the number of *granules* per window;
- ***slide_size,*** *or* ***delta*** - the number of *granules* per slide;
- ***current*** – the current granule number;
- ***ceiling*** – the ceiling of the current slide by granule number, after the window_size is reached, it is the ceiling of current window;
- ***window*** – the number (ID) of the current window.

As usual, for each operation we provide two major system abstract methods***:*** ***initialize()*** and ***execute().*** The *initialize()* method is invoked before the per-tuple processing for instantiating the settings and gathering the topology information, e.g. the input channels. The *execute()* method is invoked upon receipt each input tuple which provides the functions of processing a tuple, or, if a slide or window boundary is reached, calculating the slide or window based summaries and outputing the data mining results.

For example, a sliding window based stream analytics operation with window based summarization but without slide based stepwise summarization, has the following operation logic.

```
1.      current = resolveGranule(tuple);
2.      if (current >= ceiling) {
3.         if (current >= window_size) {
4.            summarize_window();
5.            window++;
6.         }
7.         ceiling = (current/delta + 1)*delta;
8.         process_held_tuples(ceiling);
9.      }
10.   if (getGranule(tuple) >= ceiling) {
11.      held_tuples.add(tuple);
12.   } else {
13.      process_tuple(tuple);
14.   }
```

The above logic can be described as below.

1. resolve the least granule# from all input channels;
2. if the next slide has been reached (this tuple belong to the next slide according to its granule#);
3. if the window boundary also reached (the usual case after completing the first window, if the sliding is defined as the shift of one slide);
4. make window based data mining and output the results;
5. advance the window#;
6. .
7. update the ceiling to the upper bound of the next slide;
8. process the held tuples falling in the next slide;
9. .
10. if the overall slide operation does not advance even if this tuple is beyond the boundary of the current slide
11. hold this tuple
12. .
13. otherwise
14. process this tuple

In general, upon receipt of a tuple, the system first resolve the current granule by taking into account all input channels; if the input tuple belongs to the current slide or window it gets processed, otherwise it is held to be processed in the next or even further windows where it fits in.

The window based data mining takes place at the boundary of two consecutive windows. Since we deal with sliding window, the partial results must be retained and shift – i.e. sliding.

4.2 Parallelize Sliding Window Based Stream Analytic Operation

When a sliding window based stream analytics task has multiple parallel input channels, their punctuation must be synchronized. For example, assume a task *T* has 4 input channels and currently working in window #3, after *T* receives a tuple belonging to window #4 it may or may not be able to "conclude" window #3 depending on whether all the input channels have started to supply tuples belonging to window #4 or beyond; if not, concluding window #3 would yield inaccurate result.

We assume for each input channel the tuples are delivered in the order of granules. The granule boundary of data processing by the current task is determined by taking into account all the input channels based on the following mechanism.

- The current granule number of each input channel if maintained in the ***granuleTable***.
- Upon receipt a new input, the ***granuleTable*** is updated, and the current granule number is resolved as the minimal granule number of all input channels.
- If the granule number of the current input is larger than the resolved one, this tuple is to be held without processing; it will be processed later in the next or a future window instead.
- Once a window boundary is reached by referring to all the input channels, the data analytics results for the current window is generated and finalized, and the data analytics process enters the next window boundary, starting with processing those held tuples that fall into the new window boundary; the tuples falling in future windows will continue being held.

4.3 The Generalized Framework

We provide generalized algorithms to support PSW based incremental stream analysis performed on the per-granule, per-slide and per-window basis. We coded these algorithms as open-executors held by open-stations.

The Top Level Abstract Station. This station provides the generalized algorithm for sliding window based incremental stream analysis which covers he per-granule, per-slide and per-window based incremental stream processing. The flowchart is shown in Fig 4, where several abstract methods are provided which are to be implemented by user based on their application logic.

```
public void execute(Tuple tuple) {
   //resolve the granule the task is working on
long resolved = resolveGranule(tuple);
// If granule is advanced,
//   summarize the current granule
//   sliding the list of partial results and
//   process the held tuples in the next granule boundary
   if (this.scope == SumScope.GRANULE) {
      if (resolved > current) {
         partialResult = partial_summarize();
```

```
            this.partialResultList.add(partialResult);
            if (partialResultList.size() > window_size) {
               this.partialResultList.removeFirst();
            }
            process_held_tuples(resolved+1);
         }
      }
      current = resolved;

      // if slide is advanced
//    summarize the current slide
//    if window is advanced, get window summarization results
//    sliding the list of partial results and
//    process the held tuples in the next granule boundary
      if (current >= ceiling) {
         if (this.scope == SumScope.SLIDE) {
            partialResult = partial_summarize();
            this.partialResultList.add(partialResult);
            if (partialResultList.size() > window_size / delta) {
               this.partialResultList.removeFirst();
            }
         }
         if (current >= window_size) {
            summarize_window();
            window++;
         }
         //handle next slide
         ceiling = (current/delta + 1)*delta;
         if (this.scope == SumScope.WINDOW || this.scope == SumScope.SLIDE) {
            process_held_tuples(ceiling);
         }
}
// if tuple falls in the current scope, process it, if beyond the current scope, hold it
      long upper = (this.scope == SumScope.GRANULE)? current:ceiling;
        if (getGranule(tuple) >= upper) {
            held_tuples.add(tuple);
        else {
            //normal call
            process_tuple(tuple);
        }
 }
 public abstract long getGranule(Tuple tuple);
 public abstract void summarize_window();
 public abstract Object partial_summarize();
 public abstract void process_tuple(Tuple tuple);
```

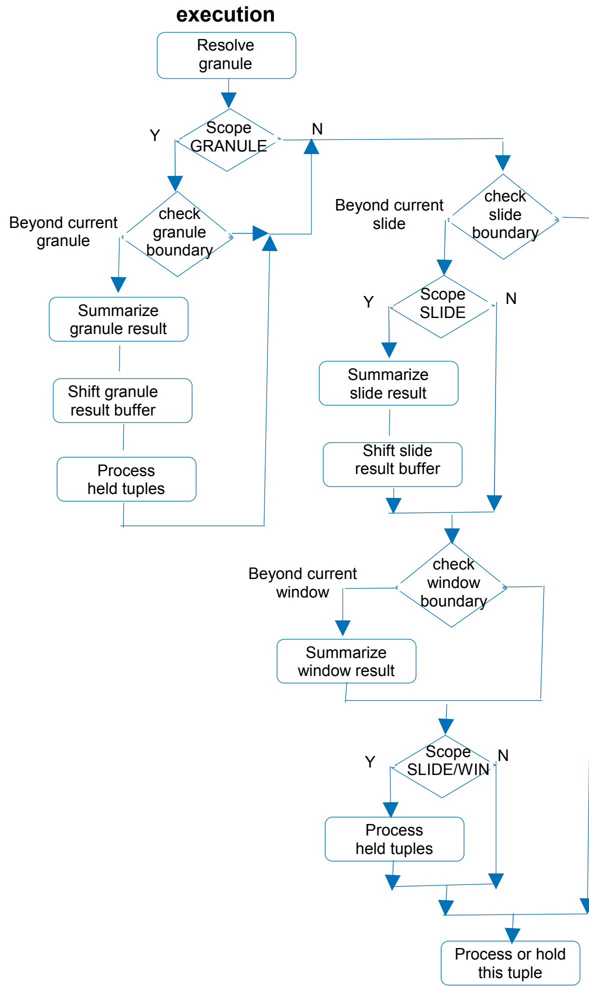


Fig. 4. Generalized PSW Framework

Abstract Station Supporting Window Oriented Summarization. This abstract station subclass the above generalized abstract PSW station; it provides the abstract algorithm with window oriented summarization as illustrated in Fig 5. It subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions. In fact, this abstract station class is implemented by only one method:

```
public Object partial_summarize() {
        return null;
}
```

The rest methods are inherited.

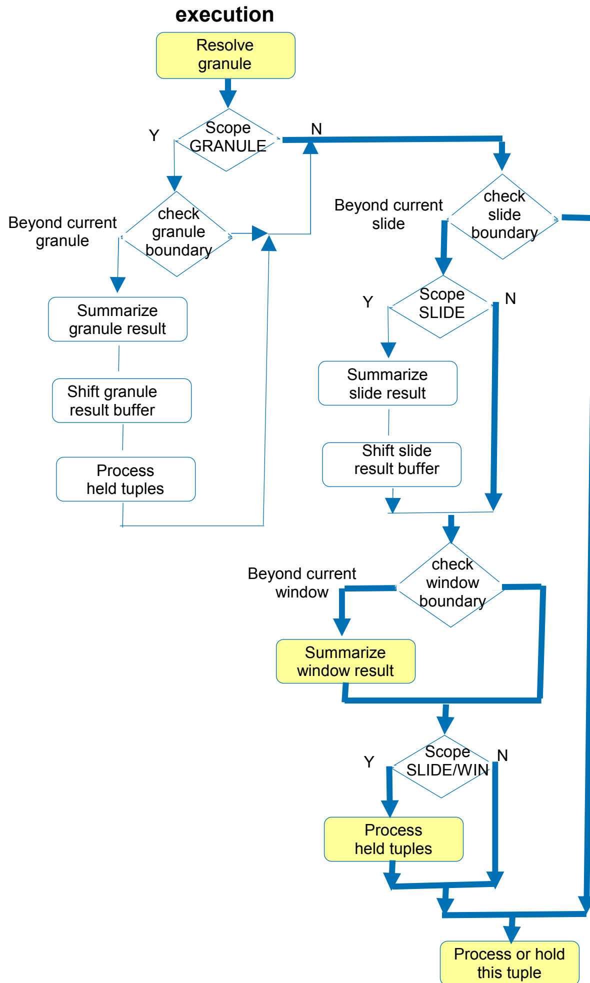


Fig. 5. Abstract Execution Framework for Window Oriented Summarization

Abstract Station Supporting Slide Oriented Summarization. The flow-chat for the abstract algorithm with slide oriented summarization is illustrated in Fig 6. The station supporting the corresponding execution pattern subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions.

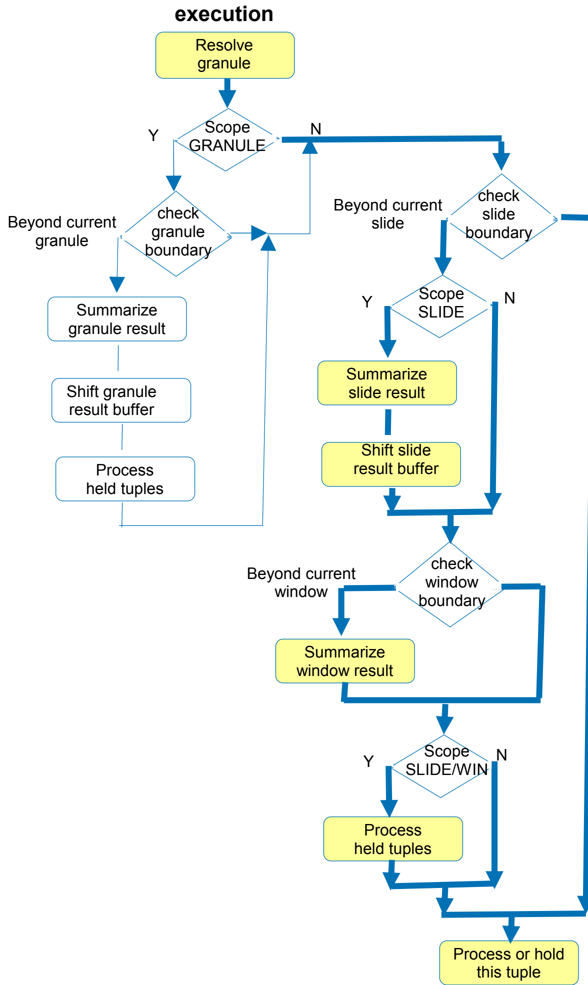


Fig. 6. Abstract Execution Framework for Slide Oriented Summarization

Abstract Station Supporting Granule Oriented Summarization. The flow-chat for the abstract algorithm with granule oriented summarization is illustrated in Fig. 7. The station supporting the corresponding operation pattern subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions.

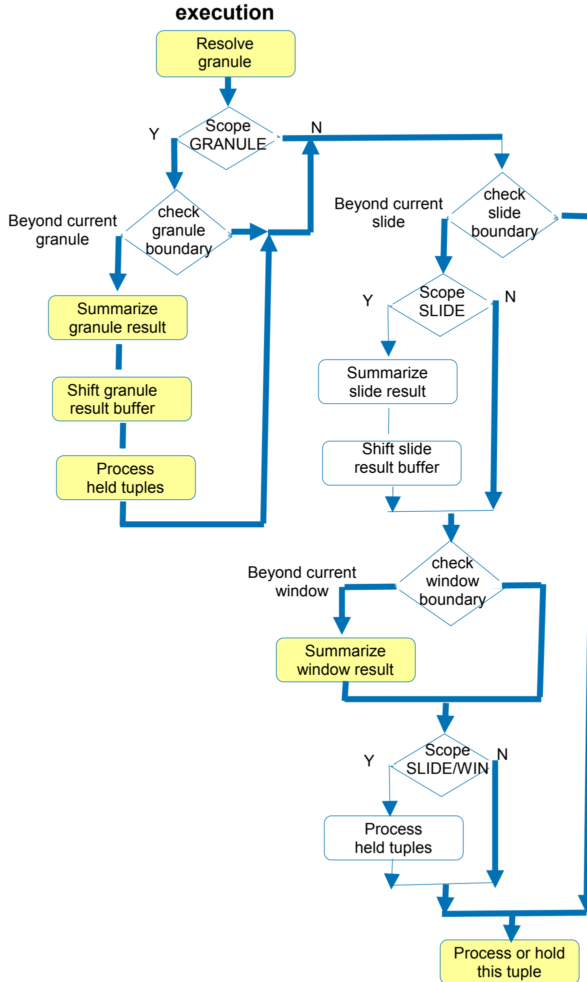


Fig. 7. Abstract Execution Framework for Granule Oriented Summarization

4.4 A PSW Stream Process Topology Example

Below we show a stream processing topology example for frequent pattern mining, we do not discuss the application here, only illustrate the role of parallel sliding window operation oriented stations in a stream analytics dataflow process. The topology specification is illustrated below

```
TopologyBuilder builder=new TopologyBuilder();
builder.setSpout("spout",
    new FileChunkItemsetSpout(filename, chunk_size));
builder.setStation("pre",
```

```
        new FPItemsetSortStation(ItemSequence), N).shuffleGrouping("spout");
    builder.setStation("mining",
        new FPSlidingWindowStation(window_size, slide_size, chunk_size,
        ItemSequence), N).fieldsGrouping("pre", new Fields("leader"));
    builder.setStation("combine",
        new FPWindowCombineStation(1, threshold), 1)
        .fieldsGrouping("mining", new Fields("itemset"));
    builder.setStation("output",
        new FPPrintStation(), 1)
        .fieldsGrouping("combine", new Fields("itemset"));
```

The process contains the following sequential building blocks (operations) but each of them can have multiple instances, and the data partition between them is defined to make the parallel processing correct. These operations are

- ***spout*****:** generates stream tuples with fields "granule", plus other fields.
- ***pre*****:** pre-processing the input data on the per-tuple basis, such as filtering or sorting, these tasks are not necessarily sliding window based.
- ***mining*****:** the major operator for playing data mining, that is coded using the parallel sliding window framework.
- ***combine*****:** combining the output of multiple ***mining*** tasks, that also follows the parallel sliding window framework.
- ***output*****:** send out the combined data mining results, these tasks are not necessarily sliding window based as far as their upstream tasks are.

5 Experiments

We have built the Fontainebleau prototype based on architecture and policies explained in the previous sections. In this section we briefly overview our experimental results. Our testing environment include 16 Linux servers with gcc version 4.1.2 20080704 (Red Hat 4.1.2-50), 32G RAM, 400G disk and 8 Quad-Core AMD Opteron Processor 2354 (2200.082 MHz, 512 KB cache). One server holds the coordinator daemon, 15 other servers hold the agent daemons, each agent supervises several worker processes, and each worker process handles one or more task instances. Based on the topology and the parallelism hint of each logical task, one or more instances of that task will be instantiated by the framework to process the data streams.

Below we present the experiment results of running the example topology that is similar to the Linear Road scenario; our topology modifies that scenario but we use the same test data under the stress test mode - the data are read from a file continuously without following the real-time intervals, leading to a fairly high throughput.

The performance show in Fig. 8 is based on the event rate of 1.33 million per minute with approximate 12 million (11,928,635) input events. Most of the tasks have 28 parallel instances except one having 14 parallel instances. There is no load-shedding (dropping events) observed.

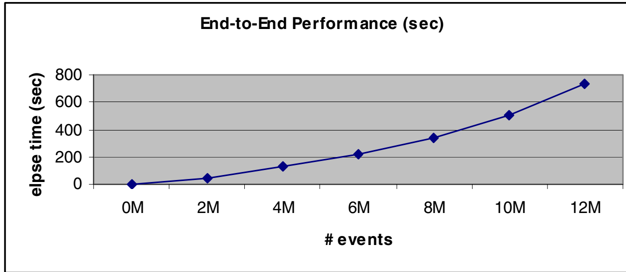


Fig. 8. The performance of data-parallel stream analytics with the LR topology

6 Related Work and Conclusions

In this paper we described our parallel and distributed stream analysis system capable of executing the real-time, continuous streaming process with general graph-structured topology. We focused on the canonical operation framework for standardizing the operational patterns of stream operators, and providing a set of open execution engines for supporting these operational patterns. We examined the power of the proposed framework by supporting the combination of group-wise and chunk-wise stream analytics which provides a generalized abstraction for parallelizing and granulizing continuous dataflow analytics, and further, the generalized support for handling parallel sliding window based stream processing..

Compared with the notable data-intensive computation systems such as DISC [3], Dryad [8], etc, our platform supports more scalable and elastic parallel computation. We share the spirit with Pig Latin [10], etc, in using multiple operations to express complex dataflows. However, unlike Pig Lain, we model the graph structured dataflow by composing multiple operations rather than decomposing a query into multiple operations; our data sources are dynamic data streams rather than static files; we partitioning stream data on the fly dynamically, rather than prepare partitioned files statically to Map-Reduce them. This work also extends the underlying tools such as Storm by elaborating it from a computation infrastructure to a state conscious computation/caching infrastructure, and from the user task oriented system to the execution engine oriented system.

Supporting truly continuous operations distinguish our platform from the current generation of stream processing systems, such as System S (IBM), STREAM (Stanford) [1], Aurora, Borealis[2], TruSQL[9], etc.

Envisaging the importance of standardizing the operational patterns of dataflow operators, we are providing a rich set of open execution engines and linking stations with existing data processors such as DBMS and Hadoop, towards the integrated dataflow cloud service.

References

[1] Arasu, A., Babu, S., Widom, J., The, C.Q.L.: Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal 15(2) (June 2006)
[2] Abadi, D.J., et al.: The Design of the Borealis Stream Processing Engine. In: CIDR (2005)
[3] Bryant, R.E.: Data-Intensive Supercomputing: The case for DISC. CMU-CS-07-128 (2007)
[4] Chen, Q., Hsu, M., Zeller, H.: Experience in Continuous analytics as a Service (CaaaS). In: EDBT 2011 (2011)
[5] Chen, Q., Hsu, M.: Experience in Extending Query Engine for Continuous Analytics. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DAWAK 2010. LNCS, vol. 6263, pp. 190–202. Springer, Heidelberg (2010)
[6] Chen, Q., Hsu, M.: Continuous mapReduce for in-DB stream analytics. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6428, pp. 16–34. Springer, Heidelberg (2010)
[7] Dean, J.: Experiences with MapReduce, an abstraction for large-scale computation. In: Int. Conf. on Parallel Architecture and Compilation Techniques. ACM (2006)
[8] Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: EuroSys 2007 (March 2007)
[9] Franklin, M.J., et al.: Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In: CIDR 2009 (2009)
[10] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: ACM SIGMOD 2008 (2008)
[11] ØMQ Lightweight Messaging Kernel, http://www.zeromq.org/
[12] Apache ZooKeeper, http://zookeeper.apache.org/
[13] Kryo - Fast, efficient Java serialization, http://code.google.com/p/kryo/
[14] Twitter's Open Source Storm Finally Hits, http://siliconangle.com/blog/2011/09/20/twitter-storm-finally-hits/

Dynamic Workload-Based Partitioning Algorithms for Continuously Growing Databases*

Miguel Liroz-Gistau[1], Reza Akbarinia[1], Esther Pacitti[2], Fabio Porto[3], and Patrick Valduriez[1]

[1] INRIA & LIRMM, Montpellier, France
{Miguel.Liroz_Gistau,Reza.Akbarinia,Patrick.Valduriez}@inria.fr
[2] University Montpellier 2, INRIA & LIRMM, Montpellier, France
Esther.Pacitti@lirmm.fr
[3] LNCC, Petropolis, Brazil
fporto@lncc.br

Abstract. Applications with very large databases, where data items are continuously appended, are becoming more and more common. Thus, the development of efficient data partitioning is one of the main requirements to yield good performance. In the case of applications that have complex access patterns, e.g. scientific applications, workload-based partitioning could be exploited. However, existing workload-based approaches, which work in a static way, cannot be applied to very large databases. In this paper, we propose *DynPart* and *DynPartGroup*, two dynamic partitioning algorithms for continuously growing databases. These algorithms efficiently adapt the data partitioning to the arrival of new data elements by taking into account the affinity of new data with queries and fragments. In contrast to existing static approaches, our approach offers constant execution time, no matter the size of the database, while obtaining very good partitioning efficiency. We validated our solution through experimentation over real-world data; the results show its effectiveness.

1 Introduction

We are witnessing the proliferation of applications that have to deal with huge amounts of data. The major software companies, such as Google, Amazon, Microsoft or Facebook have adapted their architectures in order to support the enormous quantity of information that they have to manage. Scientific applications are also struggling with those kinds of scenarios and significant research efforts are directed to deal with it [4]. An example of these applications is the management of astronomical catalogs; for instance those generated by the Dark Energy Survey (DES) [1] project with which we are collaborating. In this project, huge tables with billions of tuples and hundreds of attributes (corresponding to dimensions, mainly double precision real numbers) store the collected sky data.

* Work partially funded by the CNPq-INRIA HOSCAR project.

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 105–128, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Data are appended to the catalog database as new observations are performed and the resulting database size is estimated to reach 100TB very soon. Scientists around the globe can access the database with queries that may contain a considerable number of attributes.

The volume of data that such applications hold poses important challenges for data management. In particular, efficient solutions are needed to partition and distribute the data in multiple servers, e.g., in a cluster. An efficient partitioning scheme would try to minimize the number of fragments that are accessed in the execution of a query, thus minimizing the overhead of the distributed execution. Vertical partitioning solutions, such as column-oriented databases [18], may be useful for physical design on each node, but fail to provide an efficient distributed partitioning, in particular for applications with high dimensional queries, where joins would have to be executed by transferring data between nodes. Traditional horizontal partitioning approaches, such as hashing or range-based partitioning, are unable to capture the complex access patterns present in scientific computing applications, especially because these applications usually make use of complicated relations, including mathematical operations, over a big set of columns, and are difficult to be predefined a priori.

One solution is to use partitioning techniques based on the workload. Graph-based partitioning is an effective approach for that purpose [8]. A graph (or hypergraph) that represents the relations between queries and data elements is built and the problem is reduced to that of minimum k-way cut problem, for which several libraries are available. However, this method requires to process the entire graph in order to obtain the partitioning. This strategy works well for static applications, but scenarios where new data are inserted to the database continuously, which is the most common case for scientific computing, introduce an important problem. Each time a new set of data is appended, the partitioning should be redone from scratch, and as the size of the database grows, the execution time of such operation may become prohibitive.

In this paper, we are interested in dynamic partitioning of large databases that grow continuously. After modeling the problem of data partitioning in dynamic datasets, we propose two dynamic workload-based algorithms, called *DynPart* and *DynPartGroup*, that efficiently adapt the partitioning to the arrival of new data elements. Our algorithms are designed based on a heuristic that we developed by taking into account the affinity of new data with queries and fragments. In contrast to the static workload-based algorithms, the execution time of our algorithms do not depend on the total size of the database, but only on that of the new data and this makes them appropriate for continuously growing databases.

We validated our solutions through experimentation over real-world data sets. The results show that they obtain high performance gains in terms of partitioning execution time compared to one of the most efficient static partitioning algorithms. We also compared both algorithms and concluded that the grouping strategy of *DynPartGroup* obtains better partitioning efficiencies and performs better, specially in scenarios with high correlation between new data items and strict imbalance constraints.

This paper is a major extension of [12], which only presented the *DynPart* algorithm. Here, we propose a variation, *DynPartGroup*, which groups data items before calculating fragment affinities. This strategy adapts better for the situations where there is high correlation on the new data items and the imbalance constraints (maximum allowed imbalance) are strict, and offers an improved performance. We also extend the imbalance constraint by adding the possibility of considering the load imbalance between fragments in addition to the size imbalance. Moreover, we deal with data deletions and updates in addition to insertions. Finally, we include an extended set of experimental results for the new contributions.

The remainder of this paper is organized as follows. In Section 2, we describe our assumptions and define formally the problem we address. In Section 3, we propose our basic solution for dynamic data partitioning, that we extend in Section 4 by grouping similar data items. Section 5 reports on the results of our experimental validation. Section 6 discusses related work, and Section 7 concludes.

2 Problem Definition

In this section, we state the problem we are addressing and specify our assumptions. We start by defining the problem of static partitioning, and then extend it for a dynamic situation where the database can evolve over time.

2.1 Static Partitioning

The static partitioning is done over a set of *data items* and for a *workload*. Let $D = \{d_1, ..., d_n\}$ be the set of data items. The workload consists of a set of queries $W = \{q_1, ..., q_m\}$. We use $q(D) \subseteq D$ to denote the set of data items that a query q accesses when applied to the data set D. Given a data item $d \in D$, we say that it is *compatible* with a query q, denoted as $comp(q, d)$, if $d \in q(D)$. Queries are associated with a relative frequency $f : W \to [0, 1]$, such that $\sum_{q \in W} f(q) = 1$.

Partitioning of a data set is defined as follows.

Definition 1. *Partitioning of a data set D consists of dividing the data of D into a set of fragments, $\pi(D) = \{F_1, ..., F_p\}$, such that there is no intersection between the fragments, $\forall i \neq j : F_i \cap F_j = \emptyset$, and the union of all fragments is equal to D, i.e., $\bigcup_{i=1}^{p} F_i = D$.*

Let $q(F)$ denote the set of data items in fragment F that are compatible with q. Given a partitioning $\pi(D)$, the set of *relevant fragments* of a query q, denoted as $rel(q, \pi(D))$, is the set of fragments that contain some data accessed by q, i.e., $rel(q, \pi(D)) = \{F \in \pi(D) : q(F) \neq \emptyset\}$.

To avoid a high imbalance on the size of the fragments, we use an *imbalance factor*, denoted by ϵ_s. The size of the fragments at each time should satisfy the following condition: $|F| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil$.

In this paper, we are interested in minimizing the number of query accesses to fragments. Note that the minimum number of relevant fragments of a query q is $minfr(q, \pi(D)) = \left\lceil \frac{|q(D)|}{(|D| \,/\, |\pi(D)|)(1+\epsilon_s)} \right\rceil$. We define the *efficiency of a partitioning* for a workload based on its efficiency for queries. Intuitively, the *efficiency of a partitioning for a query* represents the ratio between the minimum number of relevant fragments of q and the number of fragments that are actually accessed under the given partitioning:

Definition 2. *Given a query q, then the efficiency of a partitioning $\pi(D)$ for q, denoted as $eff(q, \pi(D))$ is computed as:*

$$eff(q, \pi(D)) = \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|} \tag{1}$$

When the number of accessed fragments is equal to the minimum possible, i.e., $minfr(q, \pi(D))$, the efficiency is 1.

Using $eff(q, \pi(D))$, we define the efficiency of a partitioning $\pi(D)$ for a workload W as follows.

Definition 3. *The efficiency of a partitioning $\pi(D)$ for a workload W, denoted as $eff(W, \pi(D))$, is equal to the sum of the efficiencies of partitioning $\pi(D)$ for all queries in W multiplied by their relative frequencies. In other words,*

$$eff(W, \pi(D)) = \sum_{q \in W} f(q) \times eff(q, \pi(D)) \tag{2}$$

Given a set of data items D and a workload W, the goal of static partitioning is to find a partitioning $\pi(D)$ such that $eff(W, \pi(D))$ is maximized.

2.2 Dynamic Partitioning

Let us assume now that the data set D grows over time. For a given time t, we denote the set of data items of D at t as $D(t)$[1].

During the application execution, there are some events, namely *data insertions*, by which new data items are inserted into D. These events in the model correspond to the appending of the tuples corresponding to new observations in the DES catalog. No changes in the schema are involved. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to those events. Note that between two consecutive time points t_i, t_{i+1}, D remains constant. In this paper, we assume that the workload is stable and neither the queries nor their frequencies change. However, the queries may access new data items as the data set grows.

Let us now define the problem of dynamic partitioning as follows. Let $T_{\text{ev}} = (t_1, \ldots, t_m)$ be the sequence of time points corresponding to data insertion events; $D(t_1), \ldots, D(t_m)$ be the set of data items at $t_1, \ldots, t_m$ respectively; and

[1] We confine this formulation to this subsection for the sake of simplicity, so that, in the next sections, when we use D we mean $D(t_i)$.

W be a given workload. Note that, as we only consider data insertions, if $t_i < t_j$ then $D(t_i) \subset D(t_j)$ $\forall t_i, t_j \in T_{ev}$.

The goal is to find a set of partitionings $\pi(D(t_1)), ..., \pi(D(t_m))$ for data sets $D(t_1), \ldots, D(t_m)$ respectively, such that the sum of the efficiencies of the partitionings for W are maximized. In other words, our objective is as follows:

Objective: Maximize $\left(\sum_{q \in W} \left(f(q) \times \mathit{eff}(q, \pi(D(t)))\right)\right) \forall t \in T_{ev}$.

3 Affinity Based Dynamic Partitioning

In this section, we propose an algorithm, called *DynPart*, that deals with dynamic partitioning of data sets. It is based on a principle that we developed using the partitioning efficiency measure described in the previous section.

3.1 System Overview

In this paper, our proposal mainly focuses on how the data is partitioned in fragments. Here, we provide an overview of a system architecture taking advantage of our partitioning approach. The components of this architecture are as following (see Figure 1):

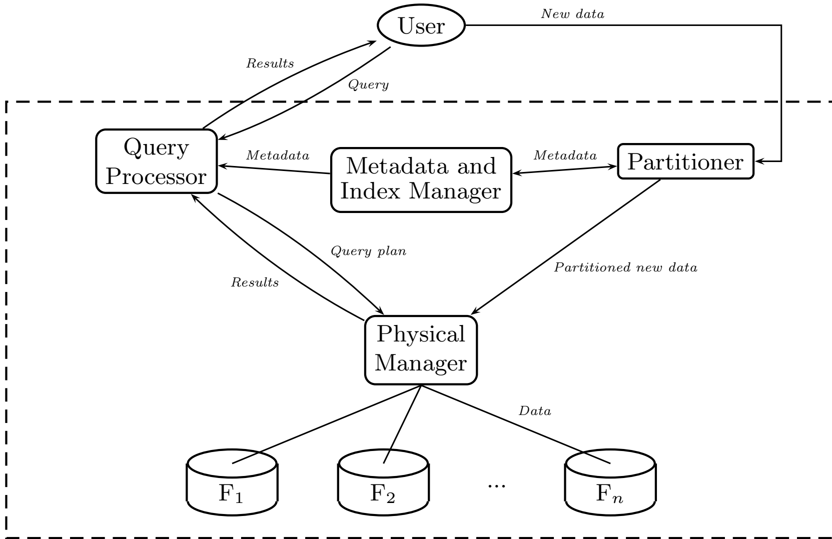


Fig. 1. System architecture

– **Query processor:** It parses the user queries, accesses the metadata and index manager, prepares an optimized execution plan and sends it to the physical manager to retrieve the data from fragments.

- **Metadata and Index Manager:** Stores metadata about the partitioning, and also indexes the location of the data items in the fragments.
- **Physical Manager:** It is in charge of storing/retrieving data to/from fragments.
- **Partitioner:** It holds the data items until a given number of items is inserted. Then, it obtains the necessary metadata and executes the partitioning algorithm. Finally, it transfers the data items to the corresponding fragments and informs the metadata and index manager about the modifications in the fragments. This component may also be contacted to include in the query results the corresponding data items in new added data.

We assume a shared nothing architecture composed of data nodes containing a physical data manager that stores one or several fragments at each node, and dedicated nodes for other components. We used a shared nothing architecture as it is the most common one since it is cheaper and can be scaled easily when required. The query processor and the metadata and index manager are preferred to be executed in the same node (nodes) to avoid communication overhead, as the query processor always has to access the index.

3.2 Principle

Let d be a new inserted data item. We can express the efficiency of the new dataset as:

$$eff(W, \pi(D \cup \{d\})) = eff(W, \pi(D)) + \Delta \tag{3}$$

Let assume that F is the fragment selected to insert d. The efficiency will remain the same for all queries but those which now have to access F in order to retrieve d but did not before. Hence, we can calculate Δ as[2]:

$$\begin{aligned}
\Delta &\approx \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(eff(q, \pi(D \cup \{d\})) - eff(q, \pi(D))\right) && (4)\\
&= \sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\left(\frac{minfr(q, \pi(D))}{|rel(q, \pi(D))| + 1} - \frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|}\right) && (5)\\
&= -\sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|\left(|rel(q, \pi(D))| + 1\right)} && (6)
\end{aligned}$$

where $q : q(F) = \emptyset \wedge comp(q, d)$ is the set of queries that will read d but no other data items in F.

Based on this idea, we define *the affinity between the data d and fragment F*:

$$aff(d, F) = -\sum_{q:q(F)=\emptyset \wedge comp(q,d)} f(q)\frac{minfr(q, \pi(D))}{|rel(q, \pi(D))|\left(|rel(q, \pi(D))| + 1\right)} \tag{7}$$

[2] Note that this approximation is an equality in all cases except when the increment in $|q(D)|$ makes $minfr(q, \pi(D))$ to be increased by 1, which happens very rarely.

Using (7), we develop a heuristic algorithm that places the new data items in the fragments based on the maximization of the affinity between the data items and the fragments.

3.3 Algorithm

Our *DynPart* algorithm takes a set of new data items D' as input and selects the best fragments to place them. For each new data item $d \in D'$, it proceeds as follows (see the pseudo-code in Algorithm 1). First, it finds the set of queries that are compatible with the data item. This can be done by executing the queries of W on D' or by comparing their predicates with every new data item. Then, for each compatible query q, *DynPart* finds the relevant fragments of q, and increases the fragments affinity by using the expression in (7). Initially the affinity of fragments is set to zero.

Algorithm 1. Algorithm *DynPart*

```
procedure DynPart(D')
    for each d ∈ D' do
        for each q : comp(q, d) do
            for each F ∉ rel(q, π(D)) do
                if feasible(F) then
                    //aff(F) is initialized to 0
                    aff(F) ← aff(F) − f(q) minfr(q,π(D)) / (|rel(q,π(D))|(|rel(q,π(D))|+1))
                end if
            end for
        end for
        if ∃F ∈ π(D) : aff(F) > 0 then
            dests ← arg max_{F∈π(D)} aff(F)
        else
            dests ← {F ∈ π(D) : feasible(F)}
        end if
        F_dest ← select from arg min_{F∈dests} |F|
        move d to F_dest
        update metadata
    end for
end procedure
```

After computing the affinity of the relevant fragments, *DynPart* has to choose the best fragment for d. Not all of the fragments satisfy the imbalance constraints, thus we must only consider those that do meet the restrictions. We define the function $feasible(F)$ to determine whether a fragment can hold more data items or not. Accordingly, *DynPart* selects from the set of feasible fragments the one with the highest affinity. If there are multiple fragments that have the highest affinity, then the smallest fragment is selected, in order to keep the partitioning as balanced as possible.

DynPart works over a set of new data items D', instead of a single data item. This allows the system to perform bulk operations over a set of n data items instead of executing n times the same operations, which is in general more costly. Moreover, it gives the algorithm more flexibility in the application of the imbalance constraints and groups data insertions in each of the fragments.

Let $comp_{\text{avg}}$ be the average number of compatible queries per data item, and rel_{avg} be the average number of relevant fragments per query. Then, the average execution time of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |D'|)$, where $|D'|$ is the number of new data items to be appended to the fragments. The complexity can be $O(|W| \times |\pi(D)| \times |D'|)$ in the worst case, e.g. when all queries are compatible to all new data and the partitioning has not been done well. However, in practice, the averages are usually much smaller than the worst case values. The reason is that the queries usually access a small portion of the data (not the whole set), thus the average number of compatible queries per data item is low. In any case, in order to reduce the number of queries, we may use a threshold on the frequency, so that only queries above that threshold are considered. In addition, the partitioning efficiency of our approach is good (see experimental results in the next section), so the average number of relevant fragments per query is low.

3.4 Example

Figure 2 illustrates the execution of the *DynPart* algorithm. Before its execution, the system is partitioned into 4 fragments, whose sizes are shown in the figure. The workload consists of 5 queries, which are represented inside the fragments they access. There are 16 new data items, $d_1, ..., d_{16}$, that should be distributed over the fragments. The imbalance factor is $\epsilon_s = 0.05$, so resulting maximum size (taking into account new data items) is 42. We show the execution of the algorithm for some of the steps.

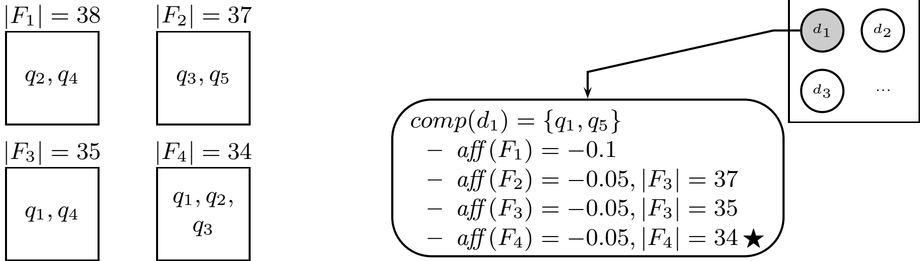
In Step 1 we show the insertion of data item d_1. The set of compatible queries is indicated in $comp(d_1)$. For each of these queries, the affinity of the relevant fragments is increased by the corresponding expression. As a consequence, F_1 has a total affinity of -0.1, resulting from the affinity expression applied to q_1 and q_5; and F_1, F_2 and F_3 have an affinity of -0.05, resulting from the expression applied to q_1 for F_2 and q_5 for F_3 and F_4. The three fragments have the highest affinity, but F_4 is selected since it is the smallest fragment.

In Step 2, the processing for data item d_2 is depicted. Note that the information has been updated as a consequence of last move: the size of F_4 has been incremented by 1 and the accessing queries now include q_5, provided that d_1 is accessed by it. In this case, the highest affinity is that of fragment F_4, so it is selected and d_2 is moved to it.

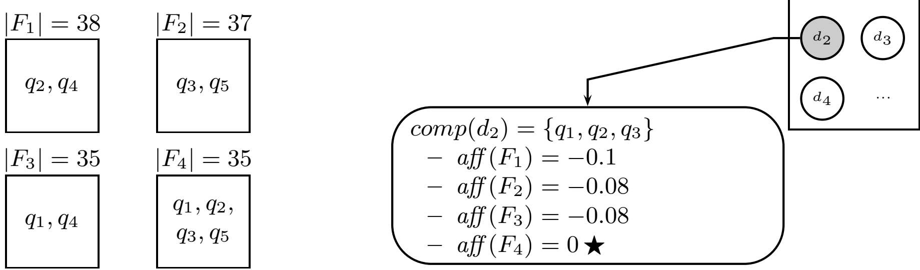
The algorithm continues to execute as before until Step 14. In that case, the fragment with the highest affinity is F_4, but it can not be selected, as it would violate the imbalance constraint. As a consequence, the next fragment in terms of affinity is selected and data item d_{14} is placed in fragment F_3.

$$D = \{d_1, ..., d_{16}\},\ \epsilon_s = 0.05,\ W = \{q_1, q_2, q_3, q_4, q_5\},$$

$$f(q_1) = 0.3,\ \ q_1(D) = \{d_1, d_2, d_3, d_4, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$$
$$f(q_2) = 0.2,\ \ q_2(D) = \{d_2, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$$
$$f(q_3) = 0.3,\ \ q_3(D) = \{d_2, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\}$$
$$f(q_4) = 0.1,\ \ q_4(D) = \{d_9, d_{10}\}$$
$$f(q_5) = 0.1,\ \ q_5(D) = \{d_1, d_3, d_4\}$$

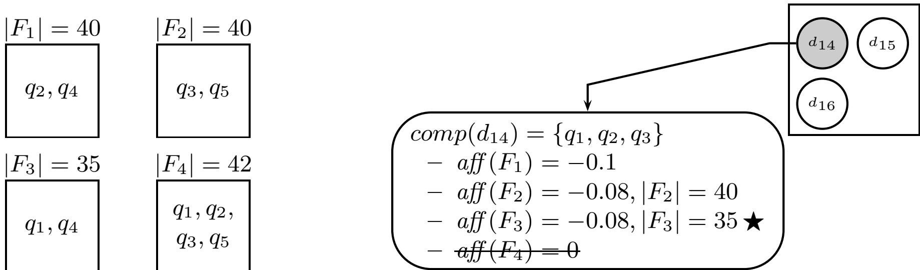


Step 1



Step 2

...



Step 14

Fig. 2. Example of operation of the *DynPart* algorithm

3.5 Data Structures

Our algorithm needs to maintain information about the relevant fragments of each query, so that we can compute the affinity efficiently. Queries are assigned a unique identifier and stored on a hash table for efficient access. For each of them, we store the set of relevant fragments as a list, as they are always accessed sequentially, i.e., no random access. Space complexity is $O(|W| \times |\pi(D)|)$ in the worst case, but, as we have pointed out, the average number of relevant fragments stays low even when the number of fragments increases. For example, in our experiments, for 1024 fragments, the average number of relevant fragments do not exceed 18 in any scenario. We also need to store the set of queries for each of the new data items. Again, as this set is accessed sequentially, we keep a list of query identifiers.

Our algorithm needs to create a data structure for each new data item to store the affinity of the possible destination fragments. For this, there are several alternatives. One option is to keep an array of size $|\pi(D)|$ initialized to zero. Note that, as the actual number of possible destinations is much lower than the total number of fragments, we would waste a lot of space with zero-affinity entries. Therefore, we keep a hash table of fragments and only compute those for which the affinity is non-zero. By using this method, access time will be maintained, while space requirements will be significantly reduced.

3.6 Dealing with Deletes and Updates

So far, we have only considered the case where data items are appended to the database. However, we could easily extend our approach to deal with deletions and updates. For a deletion, we only need to consider metadata maintenance. Whenever a data item d is deleted, the size of the fragment where it was placed should be reduced by one. We would also have to check for all queries compatible with d whether they still have to access that fragment or not, and update their set of relevant fragments if necessary. An efficient way to do this is to keep the number of data items accessed by each query on every of its relevant fragments, i.e., $|q(F)| \, \forall F \in rel(q, \pi(D))$. Then, whenever d is deleted from a fragment F, $|q(F)|$ would be reduced by 1. If the size reaches 0, then F should be deleted from the set of relevant fragments.

The case of updating a data item can be considered as a deletion followed by an insertion. However, we can benefit from previous information, and only recalculate the compatibility of queries that are affected by the changes.

4 Dealing with Imbalance

In the algorithm presented in the previous section, new data items are treated individually even if they are highly correlated. As a consequence, the destination chosen for them may differ if at a given point the selected fragment reaches the maximum size constrained by the imbalance factor. The problem might be

specially important when there are big groups of similar elements and/or the imbalance constraints are too restrictive. In this section we present a variation of the previous algorithm which tries to avoid such a situation by grouping similar elements together and taking a common decision for all the elements.

4.1 Algorithm

The extended version of our algorithm, which we call *DynPartGroup*, starts by dividing the buffer of new data items D' into a set of groups G such that all members of each group are accessed exactly by the same set of queries. Thus, the members of each group share exactly the same affinity for each given fragment. If they are allocated to different fragments, the partitioning efficiency of each of the incident queries is likely to decrease. The construction of the groups is included in Algorithm 2. A list of groups is built, where each group stores the set of composing tuples and the set of accessing queries. All items in a group are treated in the same way.

Algorithm 2. Function *CreateGroups*

```
function CreateGroups(D')
    G ← emptyList( )
    for each d ∈ D' do
        qs = {q : comp(q, d)}
        if ∃g ∈ G : g.qs = qs then
            g.ts ← g.ts ∪ {d}
        else
            g_new.ts ← {d}
            g_new.qs ← qs
            G ← insert(G, g_new)
        end if
    end for
    return G
end function
```

The algorithm (the pseudo-code is shown on Algorithm 3) first creates the groups and orders them by size in descending order, i.e., the biggest groups are considered before the smallest ones. The rationale is that, if we consider first the biggest groups, there is more free space on the fragments and the probability that all members of these groups fit on the same fragment is higher.

Once groups are ordered, an affinity value is calculated for each group, exactly in the same way it was done for individual data items in the basic algorithm. In this case, function $feasible(F, g)$ will return true if F plus the data items of the group g does not violate the imbalance factor, i,e:

$$feasible(F, g) = |F \cup g.ts| \leq \left\lceil \frac{|D|}{|\pi(D)|}(1 + \epsilon_s) \right\rceil \tag{8}$$

Algorithm 3. Algorithm *DynPartGroup*

```
procedure DynPartGroup(D')
    G ← createGroups(D')
    order G by |g.ts| in descending order
    while G ≠ ∅ do
        g ← first(G)
        G ← G − {g}
        for each q ∈ g.qs do
            for each F ∉ rel(q, π(D)) do
                if feasible(F) then
                    //aff(F) is initialized to 0
                    aff(F) ← aff(F) − f(q) minfr(q,π(D)) / (|rel(q,π(D))|(|rel(q,π(D))|+1))
                end if
            end for
        end for
        if ∃F ∈ π(D) : aff(F) > 0 then
            dests ← arg max_{F∈π(D)} aff(F)
        else
            dests ← {F ∈ π(D) : feasible(F)}
        end if
        if dests ≠ ∅ then
            F_dest ← select from arg min_{F∈dests} |F|
            move d to F_dest
            update metadata
        else
            split g into two equal sets g1 and g2
            insert g1 and g2 in G maintaining G's order
        end if
    end while
end procedure
```

If there is no feasible destination for F, the group is split into two equal halves and the resulting groups are inserted back in the list in the corresponding positions so that the order is maintained. At some point, those groups would be considered again but, in this case, individually. Note that other splitting strategies may be envisioned, e.g., assigning only the elements that fit in the fragment with the highest affinity and considering the rest as a new group. However, this will be in detriment of other big groups that might have to be subsequently split, and they will not offer any gain regarding the partitioning efficiency, as the group would be split anyway.

Let us now analyze the complexity of the algorithm. We divide the analysis in two parts; first we analyze the group creation and ordering part, and then the rest of the algorithm. Function createGroups(D') has to go over all the elements in D'. Each of them has to be compared with existing groups to check if accessing queries match, which can be done by defining a hash function over the query sets. This function has a complexity of $O(|W|)$. As a result, the total

complexity of group creation is $O(|D'| \times |W|)$. Let $|G|$ be the number of groups, then the complexity of group sorting is $O(|G| \times \log|G|)$. In the worst case, $|G| = |D'|$, but as we will see in the experimental section, the number of groups is usually much lower than that value.

The complexity of the rest of the algorithm is calculated in a similar way than in the basic algorithm. The main difference is the number of times the outer loop has to be executed. The worst case is the situation where there is a single group, the imbalance factor is near 0 and $|\pi(D)| \geq |D'|$. In that case, only one data item can be inserted on each fragment, and the group would have been split in $|D'|$ groups of size 1. This would cause $|D'| - 1$ splits and require $2 \times |D'| \in O(|D'|)$ executions of the outer loop, which would imply $O(|W| \times |\pi(D)| \times |D'|)$ affinity calculations, as in the basic algorithm.

The size of $|G|$ can vary throughout the execution, as each split increases its size by one. In the worst scenario explained above, its size will increase until reaching $|D'|$, point from which it will be consumed, as all groups would be of size 1. Assume that the ordered insertion on G is executed on $O(\log|G|)$. Then, all the sequence of insertions would need $O(\log 1) + O(\log 2) + ... + O(\log|D'|) = O(\log|D'|!) = O(|D'|\log|D'|)$. Hence, the worst case complexity is $O(|W| \times |\pi(D)| \times |D'| + |D'|\log|D'|)$

However, that worst case is very rare as usually there are a higher number of groups, and the splits are uncommon. Thus, we can say that in the average case execution complexity of this part of the algorithm is $O(comp_{\text{avg}} \times rel_{\text{avg}} \times |G|)$.

4.2 Example

Figure 3 compares the assignments performed by the basic version of the algorithm (*DynPart*), and the algorithm we described above (*DynPartGroup*), in the same scenario as in the previous section. Compatible queries for all data items are shown in previous example but can also be inferred from the groupings shown in the top of the figure, i.e., all the data items in a group have the corresponding set of compatible queries. In the basic algorithm, data items are assumed to be processed in the order indicated in the subindex, i.e., first d_1, then d_2, etc. Finally, recall that an imbalance factor of 0.05 for a fragment of size 40 means that the maximum size of the fragment at the end of the execution is 42.

Figure 3(a) shows the final assignment performed by the extended algorithm. All the groups are assigned to a single fragment and the chosen fragments have always one of the highest affinities, so the allocations are optimal. In figure 3(b) the assignments resulting from the execution of the basic algorithm are depicted. Note that, in this case, groups g_1 and g_2 have to be split into different fragments. As a consequence, q_1, q_3 and q_5 increment the number of accessed fragments by 1 and q_2 by 2, thus decreasing partitioning efficiency. This is the consequence of fragment F_4 being at its maximum size in step 14, which prevents it to be selected in further phases of the algorithm.

$$
\begin{array}{ll}
g_1.ts = \{d_2, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\} & g_1.qs = \{q_1, q_2, q_3\} \\
g_2.ts = \{d_5, d_6, d_7, d_8\} & g_2.qs = \{q_5\} \\
g_3.ts = \{d_1, d_3, d_4\} & g_3.qs = \{q_1, q_5\} \\
g_4.ts = \{d_9, d_{10}\} & g_4.qs = \{q_2, q_4\}
\end{array}
$$

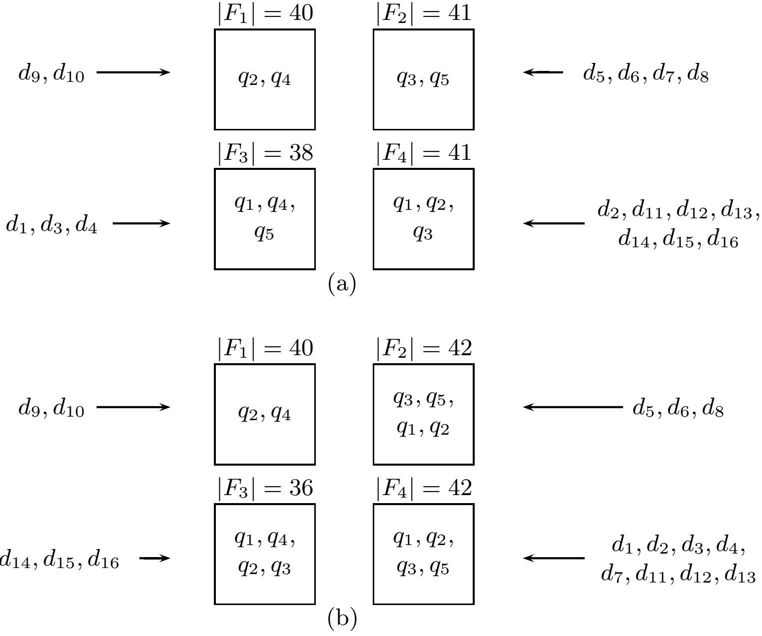


Fig. 3. Example of execution of the distribution algorithms: a) algorithm *DynPart-Group*, b) algorithm *DynPart*

4.3 Balancing Fragments Based on Load

In Section 2, we modeled the problem of data partitioning by using a size balancing constraint. Nonetheless, the problem may also be formalized if a load balancing constraint is required. Intuitively, with load we mean the number of accesses to the fragments.

Let us first define formally the load of a dataset as follows.

Definition 4. *The load of a data set D, denoted $L(D)$ is defined as the sum of the frequencies of the queries accessing its data items:*

$$L(D) = \sum_{q \in W} f(q) \times |q(D)| \tag{9}$$

Given this definition, we can reformulate the imbalance constraint in the following way: $L(F) \leq \frac{L(D)}{|\pi(D)|}(1 + \epsilon_l)$. As a result, the formula for the minimum

number of fragments that should be accessed for a given query should be modified accordingly:

$$minfr(q, \pi(D)) = \left\lceil \frac{L(q(D))}{(L(D)\,/\,|\pi(D)|)(1+\epsilon_l)} \right\rceil \tag{10}$$

Note that in the numerator we use $L(q(D))$ instead of $|q(D)|$ because we should take into account that items accessed by q are also accessed by other queries that we have to consider.

To use this new imbalance constraint, our algorithms only need some minor modifications as follows. In Algorithm 1, in case of ties in the affinity measure, the least loaded fragment should be selected instead of the smallest one. Moreover, in Algorithm 3, groups should be ordered by load instead of by size. Furthermore, function *feasible* should be redefined as follows:

$$feasible(F, g) = L(F \cup g.ts) \leq \left\lceil \frac{L(D)}{|\pi(D)|}(1+\epsilon_l) \right\rceil \tag{11}$$

5 Experimental Evaluation

To validate our dynamic partitioning algorithms, we conducted a thorough experimental evaluation over real-world data. In Section 5.1, we describe our experimental setup. In Section 5.2, we report on the execution time of our algorithms and compare them with a well known static workload-based algorithm. In Section 5.3, we study the effect of the heuristic, which we used in our algorithms, on partitioning efficiency. Finally, Section 5.4 studies how the imbalance factor and the correlation of new data affect the partitioning efficiency.

5.1 Set-Up

For our experimental evaluation we used the data from the Sloan Digital Sky Survey catalog, Data Release 8 (DR8) [2], as it is being used in LIneA in Brazil[3]. It consists of a relational database with several observations for both stars and galaxies. We obtained a workload sample from the SDSS SkyServer SQL query log data, which stores the information about the real accesses performed by users. In total, the database comprises almost 350 million tuples, that take 1.2 TB of space. The query log consists of a total of 27000 queries, some of which are similar in the SQL form but produce different results, as they use different parameters.

All queries were executed on the database and the tuple ids accessed by each of them were recorded. Only tuples accessed by at least one query were considered. We simulated the insertions on the database by selecting a subset of the tuples as the initial state and appending the rest of the tuples in groups. We varied

[3] Data from the DES project is still unavailable, so we have used data from SDSS, which is a similar, previous project.

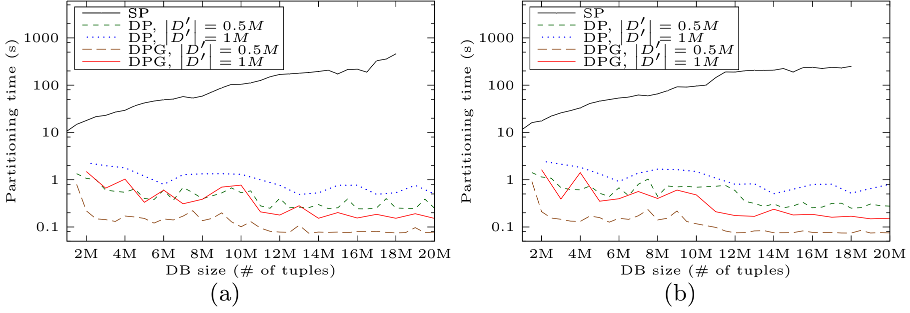


Fig. 4. Comparison of partitioning times of the dynamic and graph-based partitioning algorithms as the DB size increases ($|\pi(D)| = 16$) for a) data size balancing ($\epsilon_s = 0.15$) and b) load balancing ($\epsilon_l = 0.15$)

the following parameters: 1) the number of tuples inserted to the database on each execution of our algorithm, $|D'|$; 2) the number of fragments in which the database is partitioned, $|\pi(D)|$; 3) the imbalance factors, ϵ_s and ϵ_l; and 4) the order of data items, so as to produce datasets with higher correlation between consecutive data items. On each of the experiments, the specific numbers are detailed.

All experiments were executed in a 3.0 GHz Intel Core 2 Duo E8400, running Ubuntu 11.10 64-bit with 4GB of memory.

5.2 Partitioning Time

In this section, we study the execution time of the dynamic algorithms *DynPart* (DP in the figure) and *DynPartGroup* (DPG) and compare them with a static graph partitioning algorithm (SP). For the later, we use PaToH[4], an hypergraph partitioner. Figure 4 shows the comparison of the partitioning time for 16 fragments and for data size balancing ($\epsilon_s = 0.15$) and load balancing ($\epsilon_l = 0.15$). We executed the dynamic algorithms with two values for $|D'|$: 500000 and 1 million tuples. Similar results are obtained for different values of $|\pi(D)|$. As the difference between execution times of the static and the dynamic algorithms is significant, we use a logarithmic scale for the y-axis in order to show the results. The results are only depicted until a database size of 20 million tuples, as the memory requirements for the static partitioning are bigger than the memory of our servers. The dynamic algorithms, on the other hand, do not cause any problem as the memory footprint depends on $|D'|$, which is constant throughout the experiment.

As it can be seen, partitioning time increases for the graph partitioning algorithm as the size of the database increases, provided that the size of the graph increases accordingly. For the dynamic algorithms, on the other hand, the execution time stays at the same level, as it is always executed for the same number

[4] `http://bmi.osu.edu/~umit/software.html`

of data items. Some variation is observed since the features of the new items adapt differently to the partitioning. However the trend is constant.

In the figure, we can also observe that the execution times of the *DynPart-Group* algorithm are better that those of the basic algorithm. This is caused by the reduced number of affinity calculations, as we will show later.

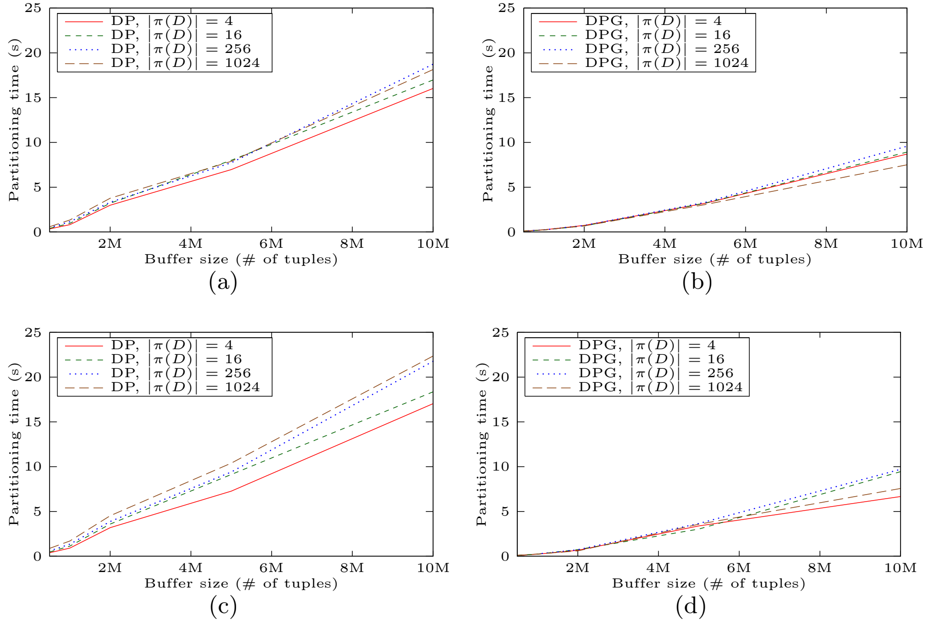


Fig. 5. Partitioning time vs. $|D'|$ for a) *DynPart* and data size balancing ($\epsilon_s = 0.15$), b) *DynPartGroup* and data size balancing ($\epsilon_s = 0.15$), c) *DynPart* and load balancing ($\epsilon_l = 0.15$) and d) *DynPartGroup* and load balancing ($\epsilon_l = 0.15$)

We compared the execution of our algorithms for different sizes of D'. Figure 5 shows the average execution time of the *DynPart* and the *DynPartGroup* algorithms as $|D'|$ increases for different number of fragments and for both balancing strategies. As expected, the execution time is linearly related to the buffer size. Also, the higher number of fragments, the higher the execution time. This increase is not linear since the number of relevant fragments does not increase at the same pace. In fact, the number of relevant fragments does not exceed 8 for $|\pi(D)| = 256$ and 16 for $|\pi(D)| = 1024$. The difference on the execution time between the *DynPart* and the *DynPartGroup* algorithms is also noticeable.

In Figure 6, we represent the average execution times for the different stages of the dynamic algorithms corresponding to the same scenario of Figure 4. Both algorithms contain the following stages: calculate affinities, select max affinity and update metadata. The extended algorithm also contains two additional stages, namely create groups and sort groups. Finally, another phase is depicted, which represents the rest of the operations executed during the distribution but not linked to a particular algorithm.

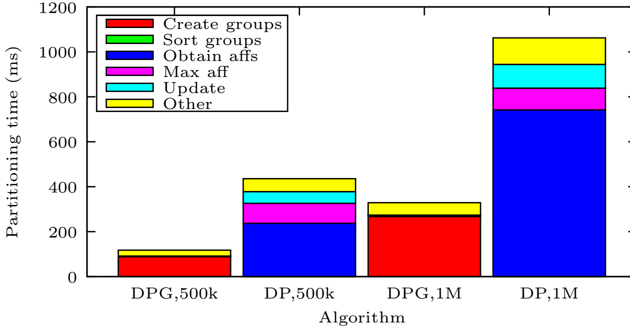


Fig. 6. Comparison of dynamic algorithms' execution times (data size balancing with $\epsilon_s = 0.15$)

As we can observe in the figure, the distribution of execution times is completely different for both algorithms. The *DynPart* algorithm spends most of the time in the calculation of the affinities, although the time spent in the rest of the phases is also significant. On the other hand, *DynPartGroup* spends almost all the time in the creation of the groups, whereas the time spent in the rest of the stages is negligible. This can be explained by considering the number of groups created in average, 664 for $|D'| = 500k$ and 1360 for $|D'| = 1M$, which represent around 0.13% of the number of tuples. As a consequence, with *DynPartGroup* the time for computing affinities, selecting the best fragment, and updating the corresponding metadata is significantly reduced.

5.3 Partitioning Efficiency

One of the important issues to consider for the dynamic algorithms is how they affect the partitioning efficiency.

We executed the algorithms as the database is fed with new data after an initial partitioning using the graph-based partitioning approach. With $|D'| = 1\,\text{M}$, Figure 7 shows how the partitioning efficiency evolves as the database grows for different number of fragments, $|\pi(D)|$. Similar results were obtained for other configurations of $|D'|$. The efficiency decreases as the database grows, as expected, but this reduction is very small. For example, in the worst case, $|\pi(D)| = 1024$ and data size balancing, the partitioning efficiency decreases 2.82×10^{-3} in average for each 10 million new tuples. The difference between *DynPart* and *DynPartGroup* is very small for small values of $|\pi(D)|$, but increases for higher values. In any case, it is below 5% for the worst case.

To evaluate the quality of our partitioning approach, in addition to the partitioning efficiency metrics, as in [8,16] we studied the percentage of single-node queries, which means the percentage of the queries that can be executed by using the data of only one fragment. Figure 8 shows the results. As seen, when the number of fragments is small, the results are similar to what we reported for partitioning efficiency metrics. However, for higher number of nodes, the number

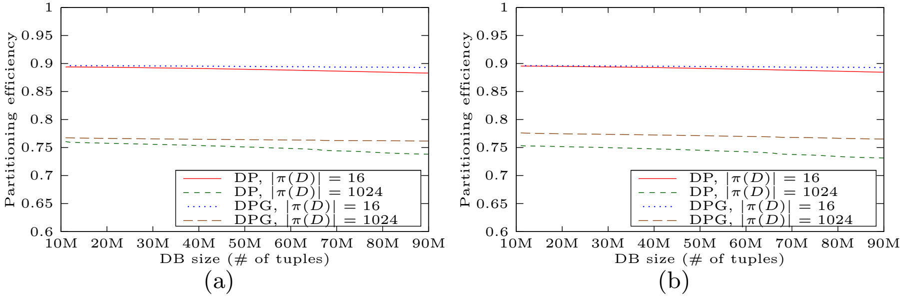


Fig. 7. Comparison of partitioning efficiency as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

of single-node queries is lower. The reason is that in these cases the partitions are smaller, so it is more difficult to confine all the results of a query in a single fragment

5.4 Effect of Imbalance Factor and Data Correlation

The imbalance factor (ϵ_s or ϵ_l) may affect the efficiency as it constraints the flexibility of the algorithm in allocating new data items. The lower the imbalance factor, the less flexibility, which may imply that some data items are not placed in the optimal fragments because they are full. Figures 9(a) and 9(c) show the average partitioning efficiency for different values of ϵ_s and ϵ_l, respectively. The efficiency decreases as the imbalance factor decreases, as expected, but it is much more noticeable for the *DynPart* algorithm.

To enrich our study, we have considered other scenarios by reordering the data so that correlated data items arrive together. In order to do that, we executed the *DynPart* algorithm over the initial data set and created the corresponding partitions. Then we reordered the data by placing on defined intervals data of only one of the fragments at a time. That way, we increase the correlation of new data (D') on each execution of the algorithm.

Figures 9(b) and 9(d) show the same configuration as before but with a new ordering created by producing 8 fragments on the original data and placing items of one of those fragment in intervals of $10M$[5]. As we see, in the case of correlated data, the impact of the imbalance factor is higher than in the previous scenario. Nevertheless, the *DynPartGroup* algorithm still shows good behavior for different values of ϵ_s and ϵ_l.

Finally, in Figure 10 we show the evolution of the partitioning efficiency as the database grows for imbalance factors of 0.001 and 0.5, which represent both extremes on the studied values of ϵ_s and ϵ_l. This confirms that higher correlations on the inserted data affect the resulting partitioning efficiency. At the beginning the efficiency is low, since all the inserted data is highly correlated and data items that should be allocated together have to be split because of imbalance

[5] We have produced different reorderings and the experiments show similar results.

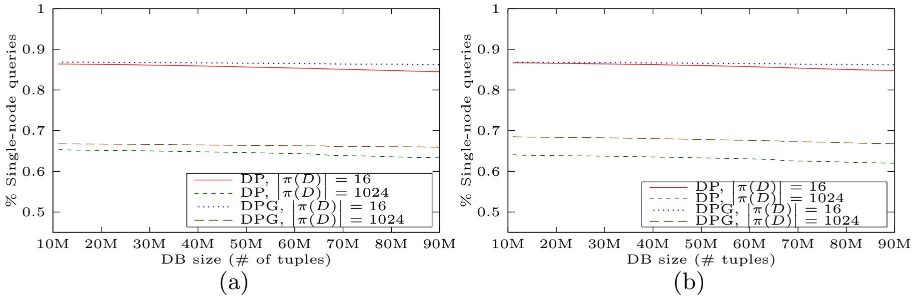


Fig. 8. Comparison of percentage of single-node queries as the size of the DB grows ($|D'| = 1M$) for a) data imbalance and b) load imbalance

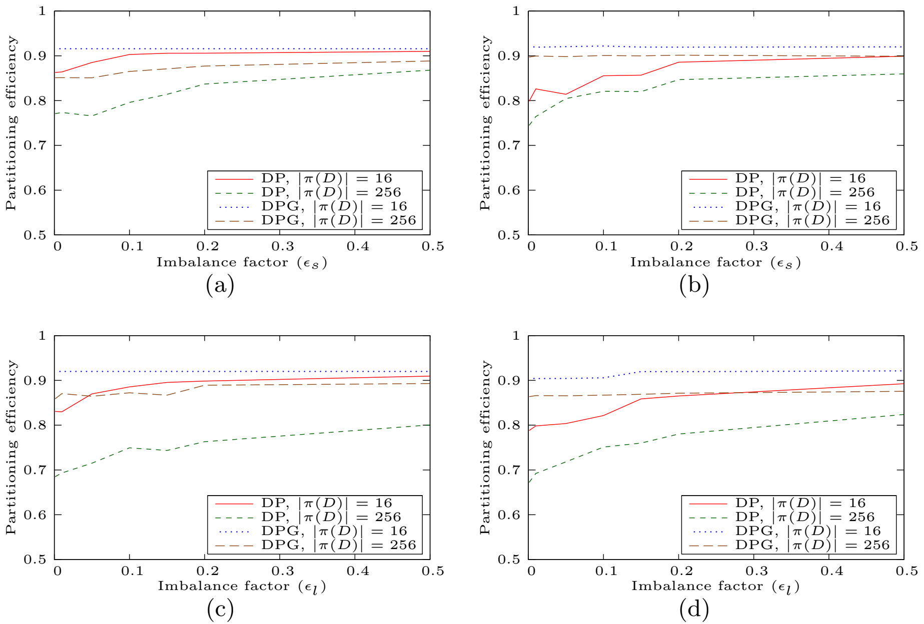


Fig. 9. Partitioning efficiency vs. imbalance factor for a) original data set and data size balancing, b) reordered data set and data size balancing, c) original data set and load balancing and d) reordered data set and load balancing

constraints. However, as new data items with different affinities are included and the imbalance is more flexible, the efficiency increases.

By comparing the behavior of both dynamic algorithms we can state that the *DynPartGroup* algorithm obtains better partitioning efficiencies consistently. The *DynPart* algorithm approaches *DynPartGroup* when the imbalance factor is high, but degrades as the imbalance constraints are stricter. This difference between the partitioning efficiency of the two algorithms is even higher for configurations with more number of fragments.

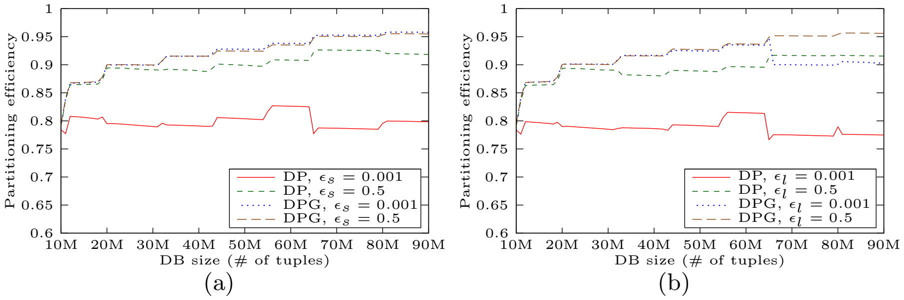


Fig. 10. Partitioning efficiency for the reordered data ($|\pi(D)| = 16$) for a) data size balancing and b) load balancing

6 Related Work

Partitioning has been used both for declustering (whose goal is to maximize parallelism) and clustering (to minimize the fragment accesses). In this paper, we are interested in the later, as we are trying to reduce the number of query accesses to the fragments.

The most popular approaches for database partitioning [10] are 1) round-robin, which consists on assigning each tuple to a different fragment; 2) hash-partitioning, which applies a hash function of a predefined set of attributes; and 3) range-based partitioning, which splits data on ranges on a given set of attributes. Recently, distributed key-value stores have been applying them. Dynamo [9] uses a modified version of hash-partitioning on the key and, as a consequence, only obtain single-site query executions when the query contains equality predicates on the key. In general, hash-based partitions are good for clustering only when the queries contain equality predicates on the partitioning attributes, which is not the case of our workload. BigTable [5] and PNUTS [7] use range-based partitioning on the keys; which still is too simple for our reference queries. In general, the complexity of scientific workloads makes it hard to design a good partitioning strategy manually, so automatic partitioning is preferred [15].

Automatic database partitioning have received significant attention from researchers and applied by some database vendors, notably Microsofts SQL Server AutoAdmin [6,3] and IBMs DB2 Database Advisor [17,19]. Many of these works have focused on partitioning (both vertical and horizontally) as an element of physical design for a single-node, along with indexing and materialized views. For instance, in [3] a set of physical design alternatives (that includes partitioning) is generated. Then, in order to limit the search space they prune the set of candidates. Similar procedures are used in other works, such as AutoPart [15], which is focused on scientific workloads. In this case only vertical and categorical partitioning are considered. After generating a set of fragment candidates from the predicates in the workload, composition of fragments is evaluated to reduce the overhead of joins. The resulting partitionings are also used for physical design in a single-node.

Some other proposals use analogous techniques to automatically generate partitions in distributed systems. The solution proposed in [17] uses a similar approach but with the goal of distributing the queries over all the nodes (data declustering). For the queries in the workload model a set of candidate partitions, which consist of applying a hash partitioning over a subset of columns, is generated. Then, they use the optimizer to estimate the costs under the new partitioning and eventually recommend some of the candidates. Automatic database partitioning for distributed databases has recently received further attention. In [14], data is partitioned automatically to optimize the execution of MPP systems. As a possible alternative they only consider hash-based partitioning over a single column. In [16], both hash and range-based partitioning on the most accessed attributes are considered for partitioning in OLTP systems. To find a near optimal solution, their approach explores a solution space by adapting the large-neighborhood search technique. However, this approach and most of the approaches mentioned above are not well suited for our underlying scientific applications that are characterized by complex workload predicates involving many attributes; and this significantly degrades the efficiency of those approaches

Graph-based approaches have been used to capture more complex relations between the workload and the data both for partitioning with the objective of declustering [13,11] and clustering [8]. They use two different models to represent data and queries: simple graph and hypergraph. In the hypergraph model [11], each query is modeled as a hyperedge (a set of vertices). In the simple graph model [13,8], queries are modeled as cliques of simple edges. Schism [8] is a recent system that partitions the data by building a graph containing the relations between queries and tuples. Data items are retrieved using an index or by means of predicate-based explanations, depending on the scenario. However, like other existing graph-based approaches, it is static and needs to redo the partitioning from scratch when the data changes. As we showed in the paper, this approach does not perform well for growing databases, and a dynamic approach is hence required. Furthermore, as new produced partitionings are not aware of previous ones, large amounts of data transfers may have to take place in order to apply the new data placements.

7 Conclusions

In this paper, we proposed a pair of dynamic algorithms for partitioning continuously growing large databases. We modeled the partitioning problem for dynamic datasets and proposed a new heuristic to efficiently distribute new arriving data, based on the affinity it has with the different fragments in the application. We designed two alternatives, *DynPart*, the basic algorithm, and *DynPartGroup*, which deals better with strict imbalance constraints.

We validated our approach through implementation, and compared its execution time with that of a static graph-based partitioning approach. The results show that as the size of the database grows, the execution time of the static algorithm increases significantly, but that of our algorithms remains stable. They also

show that, for the given dataset, our algorithms, although based on a heuristic approach, do not degrade partition efficiency considerably.

The results show that in the case of datasets in which there is a high correlation between new data items, the *DynPartGroup* algorithm maintains a very good behavior. The also show that this algorithm is not highly affected by the imbalance of fragments' sizes.

On the whole, our experiments show that our dynamic partitioning strategy is able to efficiently deal with the data of our astronomic application. But, we believe that its utilization is not limited to this application, and it can be used for data partitioning in many other applications in which the data items are appended continuously. We leave for a possible future work the scenarios with even higher data correlation where a simple eager approach, like ours, does not work and some form of data reorganization is needed.

References

1. The dark energy survey, `http://www.darkenergysurvey.org/`
2. Sloan digital sky survey, `http://www.sdss3.org`
3. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Weikum, G., König, A.C., Deßloch, S. (eds.) SIGMOD Conference, pp. 359–370. ACM (2004)
4. Ailamaki, A., Kantere, V., Dash, D.: Managing scientific data. Communications of the ACM 53(6), 68–78 (2009)
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Transactions on Computer Systems 26(2), 1–26 (2008)
6. Chaudhuri, S., Narasayya, V.R.: Autoadmin 'what-if' index analysis utility. In: Haas, L.M., Tiwary, A. (eds.) SIGMOD Conference, pp. 367–378. ACM Press (1998)
7. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
8. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1), 48–57 (2010)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, vol. 41, pp. 205–220. ACM (2007)
10. DeWitt, D.J., Gray, J.: Parallel database systems: the future of high performance database systems. Communications of the ACM 35(6), 85–98 (1992)
11. Koyutürk, M., Aykanat, C.: Iterative-improvement-based declustering heuristics for multi-disk databases. Information Systems 30, 47–70 (2005)
12. Liroz-Gistau, M., Akbarinia, R., Pacitti, E., Porto, F., Valduriez, P.: Dynamic workload based partitioning for large-scale databases. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part II. LNCS, vol. 7447, pp. 183–190. Springer, Heidelberg (2012)

13. Liu, D.R., Shekhar, S.: Partitioning similarity graphs: a framework for declustering problems. Information Systems 21(6), 475–496 (1996)
14. Nehme, R.V., Bruno, N.: Automated partitioning design in parallel database systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1137–1148 (2011)
15. Papadomanolakis, S., Ailamaki, A.: Autopart: Automating schema design for large scientific databases using data partitioning. In: SSDBM, pp. 383–392. IEEE Computer Society (2004)
16. Pavlo, A., Curino, C., Zdonik, S.B.: Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 61–72 (2012)
17. Rao, J., Zhang, C., Megiddo, N., Lohman, G.M.: Automating physical database design in a parallel database. In: Franklin, M.J., Moon, B., Ailamaki, A. (eds.) SIGMOD Conference, pp. 558–569. ACM (2002)
18. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented DBMS. In: Proceedings of the 31st international conference on Very Large Data Basesm, VLDB 2005, pp. 553–564 (2005)
19. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A.J., Garcia-Arellano, C., Fadden, S.: Db2 design advisor: Integrated automatic physical database design. In: Nascimento, M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B. (eds.) VLDB, pp. 1087–1097. Morgan Kaufmann (2004)

Author Index

Akbarinia, Reza 105
Arour, Khedija 54

Bouzeghoub, Amel 54

Chen, Qiming 83

Domps, Richard 27

Feng, Haitang 27
Fernandes, Alvaro A.A. 1

Guo, Chenjuan 1

Hacid, Mohand-Saïd 27
Hedeler, Cornelia 1
Hsu, Meichun 83

Liroz-Gistau, Miguel 105
Lumineau, Nicolas 27

Pacitti, Esther 105
Paton, Norman W. 1
Porto, Fabio 105

Valduriez, Patrick 105

Yeferny, Taoufik 54