Tutorial

LNBIP 324

Esteban Zimányi (Ed.)

Business Intelligence and Big Data

7th European Summer School, eBISS 2017
Bruxelles, Belgium, July 2–7, 2017
Tutorial Lectures

Springer

Esteban Zimányi (Ed.)

Business Intelligence and Big Data

7th European Summer School, eBISS 2017
Bruxelles, Belgium, July 2–7, 2017
Tutorial Lectures

Springer

Editor
Esteban Zimányi
Université Libre de Bruxelles
Brussels
Belgium

ISSN 1865-1348 ISSN 1865-1356 (electronic)
Lecture Notes in Business Information Processing
ISBN 978-3-319-96654-0 ISBN 978-3-319-96655-7 (eBook)
https://doi.org/10.1007/978-3-319-96655-7

Library of Congress Control Number: 2018948636

Preface

The 7th European Business Intelligence and Big Data Summer School (eBISS 2017[1]) took place in Brussels, Belgium, in July 2017. Tutorials were given by renowned experts and covered advanced aspects of business intelligence and big data. This volume contains the lecture notes of the summer school.

The first chapter covers data profiling, which is the process of metadata discovery. This process involves activities that range from ad hoc approaches, such as eye-balling random subsets of the data or formulating aggregation queries, to systematic inference of metadata via profiling algorithms. The chapter emphasizes the importance of data profiling as part of any data-related use-case, classifying data profiling tasks, and reviews data profiling systems and techniques. The chapter also discusses hard problems in data profiling, such as algorithms for dependency discovery and their application in data management and data analytics. It concludes with directions for future research in the area of data profiling.

The second chapter targets extract–transform–load (ETL) processes, which are used for extracting data, transforming them, and loading them into data warehouses. Most ETL tools use graphical user interfaces (GUIs), where the developer "draws" the ETL flow by connecting steps/transformations with lines. Although this gives an easy overview, it can be rather tedious and requires a lot of trivial work for simple things. This chapter proposes an alternative approach to ETL programming by writing code. It presents the Python-based framework pygrametl, which offers commonly used functionality for ETL development. By using the framework, the developer can efficiently create effective ETL solutions from which the full power of programming can be exploited. The chapter also discusses some of the lessons learned during the development of pygrametl as an open source framework.

The third chapter presents an overview of temporal data management. Despite the ubiquity of temporal data and considerable research on the processing of such data, database systems largely remain designed for processing the current state of some modeled reality. More recently, we have seen an increasing interest in the processing of temporal data. The SQL:2011 standard incorporates some temporal support, and commercial DBMSs have started to offer temporal functionality in a step-by-step manner. This chapter reviews state-of-the-art research results and technologies for storing, managing, and processing temporal data in relational database management systems. It starts by offering a historical perspective, after which it provides an overview of basic temporal database concepts. Then the chapter surveys the state of the art in temporal database research, followed by a coverage of the support for temporal data in the current SQL standard and the extent to which the temporal aspects of the standard are supported by existing systems. The chapter ends by covering a recently

[1] http://cs.ulb.ac.be/conferences/ebiss2017/

proposed framework that provides comprehensive support for processing temporal data and that has been implemented in PostgreSQL.

The fourth chapter discusses historical graphs, which capture the evolution of graphs through time. A historical graph can be modeled as a sequence of graph snapshots, where each snapshot corresponds to the state of the graph at the corresponding time instant. There is rich information in the history of the graph not present in only the current snapshot of the graph. The chapter presents logical and physical models, query types, systems, and algorithms for managing historical graphs.

The fifth chapter introduces the challenges around data streams, which refer to data that are generated at such a fast pace that it is not possible to store the complete data in a database. Processing such streams of data is very challenging. Even problems that are highly trivial in an off-line context, such as: "How many different items are there in my database?" become very hard in a streaming context. Nevertheless, in the past decades several clever algorithms were developed to deal with streaming data. This chapter covers several of these indispensable tools that should be present in every big data scientist's toolbox, including approximate frequency counting of frequent items, cardinality estimation of very large sets, and fast nearest neighbor search in huge data collections.

Finally, the sixth chapter is devoted to deep learning, one of the fastest growing areas of machine learning and a hot topic in both academia and industry. Deep learning constitutes a novel methodology to train very large neural networks (in terms of number of parameters), composed of a large number of specialized layers that are able to represent data in an optimal way to perform regression or classification tasks. The chapter reviews what is a neural network, describes how we can learn its parameters by using observational data, and explains some of the most common architectures and optimizations that have been developed during the past few years.

In addition to the lectures corresponding to the chapters described here, eBISS 2017 had an additional lecture:

- Christoph Quix from Fraunhofer Institute for Applied Information Technology, Germany: "Data Quality for Big Data Applications"

This lecture has no associated chapter in this volume.

As with the previous editions, eBISS joined forces with the Erasmus Mundus IT4BI-DC consortium and hosted its doctoral colloquium aiming at community building and promoting a corporate spirit among PhD candidates, advisors, and researchers of different organizations. The corresponding two sessions, each organized in two parallel tracks, included the following presentations:

- Isam Mashhour Aljawarneh, "QoS-Aware Big Geospatial Data Processing"
- Ayman Al-Serafi, "The Information Profiling Approach for Data Lakes"
- Katerina Cernjeka, "Data Vault-Based System Catalog for NoSQL Store Integration in the Enterprise Data Warehouse"
- Daria Glushkova, "MapReduce Performance Models for Hadoop 2.x"
- Muhammad Idris, "Active Business Intelligence Through Compact and Efficient Query Processing Under Updates"
- Anam Haq, "Comprehensive Framework for Clinical Data Fusion"

- Hiba Khalid, “Meta-X: Discovering Metadata Using Deep Learning”
- Elvis Koci, “From Partially Structured Documents to Relations”
- Rohit Kumar, “Mining Simple Cycles in Temporal Network”
- Jose Miguel Mota Macias, “VEDILS: A Toolkit for Developing Android Mobile Apps Supporting Mobile Analytics”
- Rana Faisal Munir, “A Cost-Based Format Selector for Intermediate Results”
- Sergi Nadal, “An Integration-Oriented Ontology to Govern Evolution in Big Data Ecosystems”
- Dmitriy Pochitaev, “Partial Data Materialization Techniques for Virtual Data Integration”
- Ivan Ruiz-Rube, “A BI Platform for Analyzing Mobile App Development Process Based on Visual Languages”

We would like to thank the attendees of the summer school for their active participation, as well as the speakers and their co-authors for the high quality of their contribution in a constantly evolving and highly competitive domain. Finally, we would like to thank the external reviewers for their careful evaluation of the chapters.

May 2018 Esteban Zimányi

Organization

The 7th European Business Intelligence and Big Data Summer School (eBISS 2017) was organized by the Department of Computer and Decision Engineering (CoDE) of the Université Libre de Bruxelles, Belgium.

Program Committee

Alberto Abelló	Universitat Politècnica de Catalunya, BarcelonaTech, Spain
Nacéra Bennacer	Centrale-Supélec, France
Ralf-Detlef Kutsche	Technische Universität Berlin, Germany
Patrick Marcel	Université François Rabelais de Tours, France
Esteban Zimányi	Université Libre de Bruxelles, Belgium

Additional Reviewers

Christoph Quix	Fraunhofer Institute for Applied Information Technology, Germany
Oscar Romero	Universitat Politècnica de Catalunya, BarcelonaTech, Spain
Alejandro Vaisman	Instituto Tecnológica de Buenos Aires, Argentina
Stijn Vansummeren	Université libre de Bruxelles, Belgium
Panos Vassiliadis	University of Ioannina, Greece
Hannes Vogt	Technische Universität Dresden, Germany
Robert Wrembel	Poznan University of Technology, Poland

Sponsorship and Support

Education, Audiovisual and Culture Executive Agency (EACEA)

Contents

An Introduction to Data Profiling

Ziawasch Abedjan(✉)

TU Berlin, Berlin, Germany
abedjan@tu-berlin.de

Abstract. One of the crucial requirements before consuming datasets for any application is to understand the dataset at hand and its metadata. The process of metadata discovery is known as data profiling. Profiling activities range from ad-hoc approaches, such as eye-balling random subsets of the data or formulating aggregation queries, to systematic inference of metadata via profiling algorithms. In this course, we will discuss the importance of data profiling as part of any data-related use-case, and shed light on the area of data profiling by classifying data profiling tasks and reviewing the state-of-the-art data profiling systems and techniques. In particular, we discuss hard problems in data profiling, such as algorithms for dependency discovery and their application in data management and data analytics. We conclude with directions for future research in the area of data profiling.

1 Introduction

Recent studies show that data preparation is one of the most time-consuming tasks of researchers and data scientists[1]. A core task for preparing datasets is profiling a dataset. Data profiling is the set of activities and processes to determine the metadata about a given dataset [1]. Most readers probably have engaged in the activity of data profiling, at least by eye-balling spreadsheets, database tables, XML files, etc. Possibly more advanced techniques were used, such as keyword-searching in datasets, writing structured queries, or even using dedicated analytics tools.

According to Naumann [46], data profiling encompasses a vast array of methods to examine datasets and produce metadata. Among the simpler results are statistics, such as the number of null values and distinct values in a column, its data type, or the most frequent patterns of its data values. Metadata that are more difficult to compute involve multiple columns, such as inclusion dependencies or functional dependencies. Also of practical interest are approximate versions of these dependencies, in particular because they are typically more efficient to compute. This chapter will strongly align with the survey published in 2015 on profiling relational data [1] and will focus mainly on exact methods. Note that all discussed examples are also taken from this survey.

[1] https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#2c61c6c56f63.

E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 1–20, 2018.
https://doi.org/10.1007/978-3-319-96655-7_1

Apart from managing the input data, data profiling faces two significant challenges: *(i)* performing the computation, and *(ii)* managing the output. The first challenge is the main focus of this chapter and that of most research in the area of data profiling: The computational complexity of data profiling algorithms depends on the number or rows and on the number of columns. Often, there is an exponential complexity in the number of columns and subquadratic complexity in the number of rows. The second challenge, namely meaningfully interpreting the data profiling has yet to be addressed. Profiling algorithms generate meta-data and often the amount of meta-data itself is impractical requiring a meta-profiling step, i.e., interpretation, which is usually performed by database and domain experts.

1.1 Use Cases for Data Profiling

Statistics about data and dependencies are have always been useful in query optimization [34,41,52]. Furthermore, data profiling plays an important role in use cases, such as data exploration, data integration, and data analytics.

Data Exploration. Users are often confronted with new datasets, about which they know nothing. Examples include data files downloaded from the Web, old database dumps, or newly gained access to some DBMS. In many cases, such data have no known schema, no or old documentation, etc. Even if a formal schema is specified, it might be incomplete, for instance specifying only the primary keys but no foreign keys. A natural first step is to identify the basic structure and high-level content of a dataset. Thus, automated data profiling is needed to provide a basis for further analysis. Morton et al. recognize that a key challenge is overcoming the current assumption of data exploration tools that data is "clean and in a well-structured relational format" [45].

Data Integration and Data Cleaning. There are crucial questions to be answered before different data sources can be integrated: In particular, the dimensions of a dataset, its data types and formats are important to recognize before automated integration routines can be applied. Similarly, profiling can help to detect data quality problems, such as inconsistent formatting within a column, missing values, or outliers. Profiling results can also be used to measure and monitor the general quality of a dataset, for instance by determining the number of records that do not conform to previously established constraints [35]. Generated constraints and dependencies also allow for rule-based data imputation.

Big Data Analytics. Fetching, storing, querying, and integrating big data is expensive, despite many modern technologies. Before using any sort of Big Data simple assessment of the dataset structure is necessary. In this context, leading researchers have noted "*If we just have a bunch of datasets in a repository, it is unlikely anyone will ever be able to find, let alone reuse, any of this data. With adequate metadata, there is some hope, but even so, challenges will remain*[...]." [4]

1.2 Chapter Overview

The goal of this chapter is to provide an overview on existing algorithms and open challenges in data profiling. The remainder of this chapter is organized as follows. In Sect. 2, we outline and define data profiling based on a new taxonomy of profiling tasks and briefly survey the state of the art of the two main research areas in data profiling: analysis of single and multiple columns. We dedicate Sect. 3 on the detection of dependencies between columns. In Sect. 4 we shed some light on data profiling tools from research and industry and we conclude this chapter in Sect. 5.

2 Classification of Profiling Tasks

This section presents a classification of data profiling tasks according to the aforementioned survey [1]. Figure 1 shows the classification, which distinguishes single-column tasks, multi-column tasks, and dependency detection. While dependency detection falls under multi-column profiling, we chose to assign a separate profiling class to this large, complex, and important set of tasks.

2.1 Single Column Profiling

Typically, the generated metadata from single columns comprises various counts, such as the number of values, the number of unique values, and the number of non-null values. These metadata are often part of the basic statistics gathered by the DBMS. In addition, the maximum and minimum values are discovered and the data type is derived (usually restricted to string vs. numeric vs. date). More advanced techniques create histograms of value distributions and identify typical patterns in the data values in the form of regular expressions [54]. Table 1 lists the possible and typical metadata as a result of single-column data profiling. In the following, we point out some of the interesting tasks.

From the number of distinct values the *uniqueness* can be calculated, which is typically defined as the number of unique values divided by the number of rows. Apart from determining the exact number of distinct values, query optimization is a strong incentive to *estimate* those counts in order to predict query execution plan costs without actually reading the entire data. Because approximate profiling is not the focus of this survey, we give only two exemplary pointers. Haas et al. base their estimation on data samples. They describe and empirically compare various estimators from the literature [27]. Another approach is to scan the entire data but use only a small amount of memory to hash the values and estimate the number of distinct values [6].

The *constancy* of a column is defined as the ratio of the frequency of the most frequent value (possibly a pre-defined default value) and the overall number of values. It thus represents the proportion of some constant value compared to the entire column.

A particularly interesting distribution is the first digit distribution for numeric values. Benford's law states that in naturally occurring numbers the

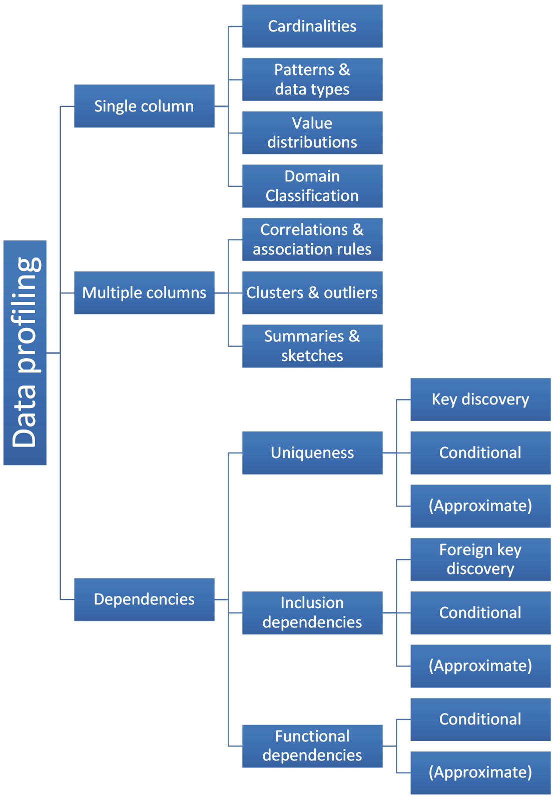


Fig. 1. A classification of traditional data profiling tasks according to [1].

distribution of the first digit d of a number approximately follows $P(d) = \log_{10}(1 + \frac{1}{d})$ [8]. Thus, the 1 is expected to be the most frequent leading digit, followed by 2, etc. Benford's law has been used to uncover accounting fraud and other fraudulently created numbers.

2.2 Multi-column Profiling

The second class of profiling tasks covers multiple columns simultaneously. In general, one can apply most of the tasks from Table 1 also on multiple columns in a bundle. For example, one could count the number of distinct value combinations in a set of columns. Additionally, multi-column profiling identifies inter-value dependencies and column similarities. Furthermore, clustering and outlier detection approaches that consume values of multiple columns and generating summaries and sketches of large datasets relates to profiling values across multiple columns. Multi-column profiling tasks generate meta-data on horizontal partitions of the data, such as values and records, instead of vertical partitions, such as columns and column groups. Although the discovery of column dependencies,

Table 1. Overview of selected single-column profiling tasks [1]

Category	Task	Description
Cardinalities	num-rows	Number of rows
	value length	Measurements of value lengths (minimum, maximum, median, and average)
	null values	Number or percentage of null values
	distinct	Number of distinct values; sometimes called “cardinality”
	uniqueness	Number of distinct values divided by the number of rows
Value distributions	histogram	Frequency histograms (equi-width, equi-depth, etc.)
	constancy	Frequency of most frequent value divided by number of rows
	quartiles	Three points that divide the (numeric) values into four equal groups
	first digit	Distribution of first digit in numeric values; to check Benford’s law
Patterns	basic type	Generic data type, such as numeric, alphabetic, alphanumeric, date, time
data types	data type	Concrete DBMS-specific data type, such as varchar, timestamp, etc.
and domains	size	Maximum number of digits in numeric values
	decimals	Maximum number of decimals in numeric values
	patterns	Histogram of value patterns (Aa9...)
	data class	Semantic, generic data type, such as code, indicator, text, date/time, quantity, identifier
	domain	Classification of semantic domain, such as credit card, first name, city, phenotype

such as key or functional dependency discovery, also relates to multi-column profiling, we dedicate a separate section to dependency discovery as described in the next section.

Correlations and Association Rules. Correlation analysis reveals related numeric columns, e.g., in an Employees table, age and salary may be correlated. A straightforward way to do this is to compute pairwise correlations among all pairs of columns. In addition to column-level correlations, value-level *associations* may provide useful data profiling information.

Traditionally, a common application of association rules has been to find items that tend to be purchased together based on point-of-sale transaction data. In these datasets, each row is a list of items purchased in a given transaction. An association rule {`bread`} → {`butter`}, for example, states that if a transaction includes bread, it is also likely to include butter, i.e., customers who buy bread also buy butter. A set of items is referred to as an *itemset*, and an association rule specifies an itemset on the left-hand-side and another itemset on the right-hand-side.

Most algorithms for generating association rules from data decompose the problem into two steps [5]:

1. Discover all frequent itemsets, i.e., those whose frequencies in the dataset (i.e., their *support*) exceed some threshold. For instance, the itemset {`bread`, `butter`} may appear in 800 out of a total of 50,000 transactions for a support of 1.6%.
2. For each frequent itemset a, generate association rules of the form $l \rightarrow a - l$ with $l \subset a$, whose *confidence* exceeds some threshold. Confidence is defined as the frequency of a divided by the frequency of l, i.e., the conditional probability of l given $a - l$. For example, if the frequency of {`bread`, `butter`} is 800 and the frequency of {`bread`} alone is 1000 then the confidence of the association rule {`bread`} → {`butter`} is 0.8.

The first step is the bottleneck of association rule discovery due to the large number of possible frequent itemsets (or patterns of values) [31]. Popular algorithms for efficiently discovering frequent patterns include Apriori [5], Eclat [62], and FP-Growth [28]. Negative correlation rules, i.e., those that identify attribute values that *do not* co-occur with other attribute values, may also be useful in data profiling to find anomalies and outliers [11]. However, discovering negative association rules is more difficult, because *infrequent* itemsets cannot be pruned the same way as frequent itemsets.

Clustering and Outlier Detection. Another useful profiling task is to identify homogeneous groups of records as clusters or to identify outlying records that do not fit into any cluster. For example, Dasu et al. cluster numeric columns and identify outliers in the data [17]. Furthermore, based on the assumption that data glitches occur across attributes and not in isolation [9], statistical inference has been applied to measure glitch recovery in [20].

Summaries and Sketches. Besides clustering, another way to describe data is to create summaries or sketches [13]. This can be done by sampling or hashing data values to a smaller domain. Sketches have been widely applied to answering approximate queries, data stream processing and estimating join sizes [18,23]. Cormode et al. give an overview of sketching and sampling for approximate query processing [15].

Another interesting task is to assess the similarity of two columns, which can be done using multi-column hashing techniques. The *Jaccard similarity* of two columns A and B is $|A \cap B|/|A \cup B|$, i.e., the number of distinct values they have in common divided by the total number of distinct values appearing in them. This gives the relative number of values that appear in both A and B. If the distinct value sets of columns A and B are not available, we can estimate the Jaccard similarity using their *MinHash signatures* [19].

2.3 Dependencies

Dependencies are metadata that describe relationships among columns. In contrast to multi-column profiling, the goal is to identify meta-data that describe relationships among column combinations and not the value combinations within the columns.

One of the common goals of data profiling is to identify suitable keys for a given table. Thus, the discovery of *unique column combinations*, i.e., sets of columns whose values uniquely identify rows, is an important data profiling task [29]. A unique that was explicitly chosen to be the unique record identifier while designing the table schema is called *primary key*. Since the discovered uniqueness constraints are only valid for a relational instance at a specific point of time, we refer to uniques and non-uniques instead of keys and non-keys. A further distinction can be made in terms of possible keys and certain keys when dealing with uncertain data and NULL values [37].

Another frequent real-world use-case of dependency discovery is the discovery of foreign keys [40] with the help of inclusion dependencies [7,42]. An inclusion dependency states that all values or value combinations from one set of columns also appear in the other set of columns – a prerequisite for a foreign key. Finally, functional dependencies (FDs) are relevant for many data quality applications. A functional dependency states that values in one set of columns functionally determine the value of another column. Again, much research has been performed to automatically detect FDs [32]. Section 3 surveys dependency discovery algorithms in detail.

2.4 Conditional, Partial, and Approximate Solutions

Real datasets usually contain exceptions to rules. To account for this, dependencies and other constraints detected by data profiling can be be relaxed. Typically, relaxations in terms of partial and conditional dependencies have been the focus of research [12]. Within the scope of this chapter, we will only discuss the approaches for discovering the exact set of dependencies.

Partial dependencies are dependencies that hold for only a subset of the records, for instance, for 95% of the records or for all but 5 records. Such dependencies are especially valuable in cleaning dirty datasets where some records might be broken and impede the detection of a dependency. Violating records can be extracted and cleansed [57].

Conditional dependencies can specify condition for partial dependencies. For instance, a conditional unique column combination might state that the column `city` is unique for all records with `Country` $\neq$ `'USA'`. Conditional inclusion dependencies (CINDs) were proposed by Bravo et al. for data cleaning and contextual schema matching [10]. Conditional functional dependencies (CFDs) were introduced in [21], also for data cleaning.

Approximate dependencies are not guaranteed to hold for the entire relation. Such dependencies are discovered using approximate solutions, such as sampling [33] or other summarization techniques [15]. Approximate dependencies can be used as input to the more rigorous task of detecting true dependencies. This survey does not discuss such approximation techniques.

2.5 Data Profiling vs. Data Mining

Generally, it is apparent that some data mining techniques can be used for data profiling. Rahm and Do distinguish data profiling from data mining by the number of columns that are examined: "Data profiling focusses on the instance analysis of individual attributes. [...] Data mining helps discover specific data patterns in large datasets, e.g., relationships holding between several attributes" [53]. While this distinction is well-defined, we believe several tasks, such as IND or FD detection, belong to data profiling, even if they discover relationships between multiple columns.

The profiling survey [1] rather adheres to the following way of differentiation: Data profiling gathers technical metadata (information about columns) to support data management; data mining and data analytics discovers non-obvious results (information about the content) to support business management with new insights. With this distinction, we concentrate on data profiling and put aside the broad area of data mining, which has already received unifying treatment in numerous textbooks and surveys [58].

3 Dependency Detection

Before discussing actual algorithms for dependency discovery, we will present a set of formalism that are used in the following. Then, we will present algorithms for the discovery of the prominent dependencies unique column combinations (Sect. 3.1), functional dependencies (Sect. 3.2), and inclusion dependencies (Sect. 3.3). Apart from algorithms for these traditional dependencies, newer types of dependencies have also been the focus of research. We refer the reader to the survey on relaxed dependencies for further reading [12].

Notation. R and S denote relational schemata, with r and s denoting the instances of R and S, respectively. We refer to tuples of r and s as r_i and s_j, respectively. Subsets of columns are denoted by upper-case X, Y, Z (with $|X|$ denoting the number of columns in X) and individual columns by upper-case A, B, C. Furthermore, we define $\pi_X(r)$ and $\pi_A(r)$ as the projection of r on the attribute set X or attribute A, respectively; thus, $|\pi_X(r)|$ denotes the count of distinct combinations of the values of X appearing in r. Accordingly, $r_i[A]$ indicates the value of the attribute A of tuple r_i and $r_i[X] = \pi_X(r_i)$. We refer to an attribute value of a tuple as a *cell*.

3.1 Unique Column Combinations and Keys

Given a relation R with instance r, a *unique column combination* (a "unique") is a set of columns $X \subseteq R$ whose projection on r contains only unique value combinations.

Definition 1 (Unique/Non-Unique). *A column combination $X \subseteq R$ is a* unique*, iff $\forall r_i, r_j \in r, i \neq j : \ r_i[X] \neq r_j[X]$. Analogously, a set of columns $X \subseteq R$ is a* non-unique column combination *(a "non-unique"), iff its projection on r contains at least one duplicate value combination.*

Each superset of a unique is also unique while each subset of a non-unique is also a non-unique. Therefore, discovering all uniques and non-uniques can be reduced to the discovery of minimal uniques and maximal non-uniques:

Definition 2 (Minimal Unique/Maximal Non-Unique). *A column combination $X \subseteq R$ is a* minimal unique*, iff $\forall X' \subset X : X'$ is a non-unique. Accordingly, a column combination $X \subseteq R$ is a* maximal non-unique*, iff $\forall X' \supset X : X'$ is a unique.*

To discover all minimal uniques and maximal non-uniques of a relational instance, in the worst case, one has to visit all subsets of the given relation. Thus, the discovery of all minimal uniques and maximal non-uniques of a relational instance is an NP-hard problem and even the solution set can be exponential [26]. Given $|R|$, there can be $\binom{|R|}{\frac{|R|}{2}} \geq 2^{\frac{|R|}{2}}$ minimal uniques in the worst case, i.e., as all combinations of size $\frac{|R|}{2}$.

Gordian – Row-Based Discovery. Row-based algorithms require multiple runs over all column combinations as more and more rows are considered. They benefit from the intuition that non-uniques can be detected without verifying every row in the dataset. GORDIAN [56] is an algorithm that works this way in a recursive manner. The algorithm consists of three stages: *(i)* Organize the data in form of a prefix tree, *(ii)* Discover maximal non-uniques by traversing the prefix tree, *(iii)* Generate minimal uniques from maximal non-uniques.

Each level of the prefix tree represents one column of the table whereas each branch stands for one distinct tuple. Tuples that have the same values in their prefix share the corresponding branches. E.g., all tuples that have the same value in the first column share the same node cells. The time to create the prefix tree depends on the number of rows, therefore this can be a bottleneck for very large datasets. However because of the tree structure, the memory footprint will be smaller than the original dataset. By a depth-first traversal of the tree for discovering maximum repeated branches, which constitute maximal non-uniques, maximal non-uniques will be discovered.

After discovering all maximal non-uniques, GORDIAN computes all minimal uniques by a complementation step. For each non-unique, the updated set of minimal uniques must be *simplified* by removing redundant uniques. This simplification requires quadratic runtime in the number of uniques.

The generation of minimal uniques from maximal non-uniques can be a bottleneck if there are many maximal non-uniques. Experiments showed that in most cases the unique generation dominates the runtime [2].

Column-Based Traversal of the Column Lattice. In the spirit of the well-known Apriori approach, minimal unique discovery working bottom-up as well as top-down can follow the same approach as for frequent itemset mining [5]. With regard to the powerset lattice of a relational schema, the Apriori algorithms generate all relevant column combinations of a certain size and verify those at once. Figure 2 illustrates the powerset lattice for the running example in Table 2. The effectiveness and theoretical background of those algorithms is discussed by Giannela and Wyss [24]. They presented three breadth-first traversal strategies: a bottom-up, a top-down, and a hybrid traversal strategy.

Table 2. Example dataset.

Tuple id	First	Last	Age	Phone
1	Max	Payne	32	1234
2	Eve	Smith	24	5432
3	Eve	Payne	24	3333
4	Max	Payne	24	3333

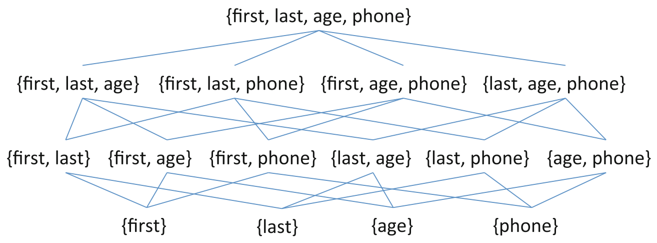


Fig. 2. Powerset lattice for the example Table 2 [1].

Bottom-up unique discovery traverses the powerset lattice of the schema R from the bottom, beginning with all 1-*combinations* toward the top of the lattice, which is the $|R|$-*combination*. The prefixed number k of k-*combination* indicates the size of the combination. The same notation applies for k-*candidates*, k-*uniques*, and k-*non-uniques*. To generate the set of *2-candidates*, we generate all pairs of *1-non-uniques*. k-*candidates* with $k > 2$ are generated by extending the $(k-1)$-*non-uniques* by another non-unique column. After the candidate generation, each candidate is checked for uniqueness. If it is identified as a non-unique, the k-*candidate* is added to the list of k-*non-uniques*.

If the candidate is verified as unique, its minimality has to be checked. The algorithm terminates when $k = |1\text{-}non\text{-}uniques|$. A disadvantage of this candidate generation technique is that redundant uniques and duplicate candidates are generated and tested.

The Apriori idea can also be applied to the top-down approach. Having the set of identified *k-uniques*, one has to verify whether the uniques are minimal. Therefore, for each *k-unique*, all possible $(k-1)$-*subsets* have to be generated and verified. Experiments have shown that in most datasets, uniques usually occur in the lower levels of the lattice, which favours bottom-up traversal [2]. HCA is an improved version of the bottom-up Apriori technique [2], which optimizes the candidate generation step, applies statistical pruning, and considers functional dependencies that have been inferred on the fly.

DUCC – Traversing the Lattice via Random Walk. While the breadth-first approach for discovering minimal uniques gives the most pruning, a depth-first approach might work well if there are relatively few minimal uniques that are scattered on different levels of the powerset lattice. Depth-first detection of unique column combinations resembles the problem of identifying the most promising paths through the lattice to discover existing minimal uniques and avoid unnecessary uniqueness checks. DUCC is a depth-first approach that traverses the lattice back and forth based on the uniqueness of combinations [29]. Following a random walk principle by randomly adding columns to non-uniques and removing columns from uniques, DUCC traverses the lattice in a manner that resembles the border between uniques and non-uniques in the powerset lattice of the schema.

DUCC starts with a seed set of *2-non-uniques* and picks a seed at random. Each k-combination is checked using the superset/subset relations and pruned if any of them subsumes the current combination. If no previously identified combination subsumes the current combination DUCC performs uniqueness verification. Depending on the verification, DUCC proceeds with an unchecked $(k-1)$-subset or $(k-1)$-superset of the current k-combination. If no seeds are available, it checks whether the set of discovered minimal uniques and maximal non-uniques correctly complement each other. If so, DUCC terminates; otherwise, a new seed set is generated by complementation.

DUCC also optimizes the verification of minimal uniques by using a position list index (PLI) representation of values of a column combination. In this index, each position list contains the tuple IDs that correspond to the same value combination. Position lists with only one tuple ID can be discarded, so that the position list index of a unique contains no position lists. To obtain the PLI of a column combination, the position lists in PLIs of all contained columns have to be cross-intersected. In fact, DUCC intersects two PLIs similar to how a hash join operator would join two relations. As a result of using PLIs, DUCC can also apply row-based pruning, because the total number of positions decreases with the size of column combinations. Since DUCC combines row-based and column-based pruning, it performs significantly better than its predecessors [29].

3.2 Functional Dependencies

A *functional dependency* (FD) over R is an expression of the form $X \rightarrow A$, indicating that $\forall r_i, r_j \in r$ if $r_i[X] = r_j[X]$ then $r_i[A] = r_j[A]$. That is, any two tuples that agree on X must also agree on A. We refer to X as the left-hand-side (LHS) and A as the right-hand-side (RHS). Given r, we are interested in finding all nontrivial and minimal FDs $X \rightarrow A$ that hold on r, with non-trivial meaning $A \cap X = \emptyset$ and minimal meaning that there must not be any FD $Y \rightarrow A$ for any $Y \subset X$. A naive solution to the FD discovery problem is to verify for each possible RHS and LHS combination whether there exist two tuples that violate the FD. This is prohibitively expensive: for each of the $|R|$ possibilities for the RHS, it tests $2^{(|R|-1)}$ possibilities for the LHS, each time having to scan r multiple times to compare all pairs of tuples. However, notice that for $X \rightarrow A$ to hold, the number of distinct values of X must be the same as the number of distinct values of XA – otherwise at least one combination of values of X that is associated with more than one value of A, thereby breaking the FD [32]. Thus, if we pre-compute the number of distinct values of each combination of one or more columns, the algorithm simplifies to:

Recall Table 2. We have $|\pi_{\mathsf{phone}}(r)| = |\pi_{\mathsf{age,phone}}(r)| = |\pi_{\mathsf{last,phone}}(r)|$. Thus, $\mathsf{phone} \rightarrow \mathsf{age}$ and $\mathsf{phone} \rightarrow \mathsf{last}$ hold. Furthermore, $|\pi_{\mathsf{last,age}}(r)| = |\pi_{\mathsf{last,age,phone}}(r)|$, implying $\{\mathsf{last,age}\} \rightarrow \mathsf{phone}$.

This approach still requires to compute the distinct value counts for all possible column combinations. Similar to unique discovery, FD discovery algorithms employ row-based (bottom-up) and column-based (top-down) optimizations, as discussed below.

Column-Based Algorithms. As was the case with uniques, Apriori-like approaches can help prune the space of FDs that need to be examined, thereby optimizing the first two lines of the above straightforward algorithms. TANE [32], FUN [47], and FD_Mine [61] are three algorithms that follow this strategy, with FUN and FD_Mine introducing additional pruning rules beyond TANE's based on the properties of FDs. They start with sets of single columns in the LHS and work their way up the powerset lattice in a *level-wise* manner. Since only minimal FDs need to be returned, it is not necessary to test possible FDs whose LHS is a superset of an already-found FD with the same RHS. For instance, in Table 2, once we find that *phone* $\rightarrow$ *age* holds, we do not need to consider $\{\mathsf{first,phone}\} \rightarrow \mathsf{age}$, $\{\mathsf{last,phone}\} \rightarrow \mathsf{age}$, etc.

Additional pruning rules may be formulated from Armstrong's axioms, i.e., we can prune from consideration those FDs that are logically implied by those we have found so far. For instance, if we find that $A \rightarrow B$ and $B \rightarrow A$, then we can prune all LHS column sets including B, because A and B are equivalent [61]. Another pruning strategy is to ignore columns sets that have the same number of distinct values as their subsets [47]. Returning to Table 2, observe that $\mathsf{phone} \rightarrow \mathsf{first}$ does not hold. Since $|\pi_{\mathsf{phone}}(r)| = |\pi_{\mathsf{last,phone}}(r)| = |\pi_{\mathsf{age,phone}}(r)| = |\pi_{\mathsf{last,age,phone}}(r)|$, we know that adding last and/or age to the LHS cannot lead to a valid FD with first on the RHS. To determine these cardinalities the approaches use the PLIs as discussed in Sect. 3.1.

Row-Based Algorithms. Row-based algorithms examine pairs of tuples to determine LHS candidates. Dep-Miner [39] and FastFDs [59] are two examples; the FDEP algorithm [22] is also row-based, but the way it ultimately finds FDs that hold is different.

The idea behind row-based algorithms is to compute the so-called difference sets for each pair of tuples that are the columns on which the two tuples differ. Table 3 enumerates the difference sets in the data from Table 2. Next, we can find candidate LHS's from the difference sets as follows. Pick a candidate RHS, say, phone. The difference sets that include phone, with phone removed, are: {first,last,age}, {first,age}, {age}, {last} and {first,last}. This means that there exist pairs of tuples with different values of phone and also with different values of these five difference sets. Next, we find minimal subsets of columns that have a non-empty intersection with each of these difference sets. Such subsets are exactly the LHSs of minimal FDs with phone as the RHS: if two tuples have different values of phone, they are guaranteed to have different values of the columns in the above minimal subsets, and therefore they do not cause FD violations. Here, there is only one such minimal subset, {last,age}, giving {last,age} $\rightarrow$ phone. If we repeat this process for each possible RHS, and compute minimal subsets corresponding to the LHS's, we obtain the set of minimal FDs.

Table 3. Difference sets computed from Table 2.[1]

Tuple ID pair	Difference set
(1,2)	first, last, age, phone
(1,3)	first, age, phone
(1,4)	age, phone
(2,3)	last, phone
(2,4)	first, last, phone
(3,4)	first

Experiments confirm that row-based approaches work well on high-dimensional tables with a relatively small number of tuples, while column-based approaches perform better on low-dimensional tables with a large number of rows [49]. Recent approaches to FD discovery are DFD [3] and HyFD [51]. DFD decomposes the attribute lattice into $|R|$ lattices, considering each attribute as a possible RHS of an FD. On each lattice, DFD applies a random walk approach by pruning supersets of FD LHS's and subsets of non-FD LHS's. HyFD is a hybrid solution that optimally switches between a row-based and a column-based strategy.

3.3 Inclusion Dependencies

An *inclusion dependency* (IND) $R.A \subseteq S.B$ asserts that each value of column A from relation R appears in column B from relation S; or $A \subseteq B$ when the relations are clear from context. Similarly, for two sets of columns X and Y, we write $R.X \subseteq S.Y$, or $X \subseteq Y$, when each distinct value combination in X appears in Y. We refer to $R.A$ or $R.X$ as the LHS and $S.B$ or $S.Y$ as the RHS. INDs with a single-column LHS and RHS are referred to as *unary* and those with multiple columns in the LHS and RHS are called *n-ary.* A naive solution to IND discovery in relation instances r and s is to try to match each possible LHS X with each possible RHS Y. For any considered X and Y, we can stop as soon as we find a value combination of X that does not appear in Y. This is not an efficient approach as it repeatedly scans r and s when testing the possible LHS and RHS combinations.

Generating Unary Inclusion Dependencies. To discover unary INDs, a common approach is to pre-process the data to speed up the subsequent IND discovery [7,43]. The SPIDER algorithm [7] pre-processes the data by sorting the values of each column and writing them to disk. Next, each sorted stream, corresponding to the values of one particular attribute, is consumed in parallel in a cursor-like manner, and an IND $A \subseteq B$ can be discarded as soon as a value in A is read that is not present in B.

Generating n-ary Inclusion Dependencies. After discovering unary INDs, a level-wise algorithm, similar to the TANE algorithm for FD discovery can be used to discover n-ary INDs. De Marchi et al. [43] propose an algorithm that constructs INDs with i columns from those with $i-1$ columns. Additionally, hybrid algorithms have been proposed in [38,44] that combine bottom-up and top-down traversal for additional pruning. Recently, the BINDER algorithm, which uses divide and conquer principles to handle larger datasets, has been introduced [50]. In the divide step, it splits the input dataset horizontally into partitions and vertically into buckets with the goal to fit each partition into main memory. In the conquer step, BINDER then validates the set of all possible inclusion dependency candidates. Processing one partition after another, the validation constructs two indexes on each partition, a dense index and an inverted index, and uses them to efficiently prune invalid candidates from the result set.

Generating Foreign Keys. IND discovery is the initial step for foreign key detection: a foreign key must satisfy the corresponding inclusion dependency but not all INDs are foreign keys. For example, multiple tables may contain auto-increment columns that serve as keys, and while inclusion dependencies among them may exist, they are not foreign keys. Once INDs have been discovered, additional heuristics have been proposed, which essentially rank the discovered INDs according to their likelihood of being foreign keys [40,55]. A very simple heuristic may be the similarity of the column names.

4 Profiling Tools

To allow a more powerful and integrated approach to data profiling, software companies have developed data profiling techniques, mostly to profile data residing in relational databases. Most profiling tools so far are part of a larger software suite, either for data integration or for data cleansing. We first give an overview of tools that were created in the context of a research project (see Table 4 for a listing). Then we give a brief glimpse of the vast set of commercial tools with profiling capabilities.

Table 4. Research tools with data profiling capabilities [1].

Tool	Main goal	Profiling capabilities
Bellman [19]	Data quality browser	Column statistics, column similarity, candidate key discovery
Potters Wheel [54]	Data quality, ETL	Column statistics (including value patterns)
Data Auditor [25]	Rule discovery	CFD and CIND discovery
RuleMiner [14]	Rule discovery	Denial constraint discovery
MADLib [30]	Machine learning	Simple column statistics

Research Prototypes. Data profiling tools are often embedded in data cleaning systems. For example, the Bellman [19] data quality browser supports column analysis (counting the number of rows, distinct values, and NULL values, finding the most frequently occurring values, etc.), and key detection (up to four columns). It also provides a column similarity functionality that finds columns whose value or n-gram distributions are similar; this is helpful for discovering potential foreign keys and join paths. Furthermore, it is possible to profile the evolution of a database using value distributions and correlations [18]: which tables change over time and in what ways (insertions, deletions, modifications), and which groups of tables tend to change in the same way. The Potters Wheel tool [54] also supports column analysis, in particular, detecting data types and syntactic structures/patterns. The MADLib toolkit for scalable in-database analytics [30] includes column statistics, such as count, count distinct, minimum and maximum values, quantiles, and the k most frequently occurring values.

Recent data quality tools are dependency-driven: dependencies, such as FDs and INDs, as well as their conditional extensions, may be used to express the intended data semantics, and dependency violations may indicate possible data quality problems. Most research prototypes, such as GDR [60], Nadeef [16], and StreamClean [36], require users to supply data quality rules and dependencies. However, data quality rules are not always known apriori in unfamiliar and undocumented datasets, in which case data profiling, and dependency discovery in particular, is an important pre-requisite to data cleaning.

There are at least three research prototypes that perform rule discovery to some degree: Data Auditor [25], RuleMiner [14], and Metanome [48]. Data Auditor requires an FD as input and generates corresponding CFDs from the data. Additionally, Data Auditor considers FDs similar to the one that is provided by the user and generates corresponding CFDs. The idea is to see if a slightly modified FD can generate a more suitable CFD for the given relation instance. On the other hand, RuleMiner does not require any rules as input and instead it is designed to generate all reasonable rules from a given dataset. RuleMiner expresses the discovered rules as *denial constraints*, which are universally-quantified first order logic formulas that subsume FDs, CFDs, INDs and many others. Metanome is a recent profiling system that embeds various profiling techniques with a unified interface. So far, it is the most comprehensive tool in this regard, covering FDs, CFDs, INDs, basic statistics etc.

Commercial Tools. Generally, every database management system collects and maintains base statistics about the tables it manages. However, they do not readily expose those metadata, the metadata are not always up-to-date and sometimes based only on samples, and their scope is usually limited to simple counts and cardinalities. Furthermore, commercial data quality or data cleansing tools often support a limited set of profiling tasks. In addition, most Extract-Transform-Load tools have some profiling capabilities. Prominent examples of current commercial tools include software from vendors, such as IBM, SAP, Attacama, or Informatica. Most commercial tools focus on the so-called easy to solve profiling tasks. Only some of them, such as the IBM InfoSphere Information Analyzer, support inter-column dependency discovery and that only up to certain dependency size.

5 Conclusions and Outlook

In this chapter, we provided an introduction into the state-of-the-art in data profiling: the set of activities and processes to determine metadata about a given database. We briefly discussed single-column profiling tasks and multi-column tasks and provided algorithmic details on the challenging aspects of dependency discovery. While many data profiling algorithms have been proposed and implemented in research prototypes and commercial tools, further work is needed, especially in the context of profiling new types of data and interpreting and visualizing data profiling results. As mentioned in the introduction, the interpretation of profiling results remains an open challenge. To facilitate interpretation of metadata, effectively visualizing and ranking profiling results is of utmost importance.

References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015)
2. Abedjan, Z., Naumann, F.: Advancing the discovery of unique column combinations. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 1565–1570 (2011)
3. Abedjan, Z., Schulze, P., Naumann, F.: DFD: efficient functional dependency discovery. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 949–958 (2014)
4. Agrawal, D., Bernstein, P., Bertino, E., Davidson, S., Dayal, U., Franklin, M., Gehrke, J., Haas, L., Halevy, A., Han, J., Jagadish, H.V., Labrinidis, A., Madden, S., Papakonstantinou, Y., Patel, J.M., Ramakrishnan, R., Ross, K., Shahabi, C., Suciu, D., Vaithyanathan, S., Widom, J.: Challenges and opportunities with Big Data. Technical report, Computing Community Consortium (2012). http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf
5. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 487–499 (1994)
6. Astrahan, M.M., Schkolnick, M., Kyu-Young, W.: Approximating the number of unique values of an attribute without sorting. Inf. Syst. **12**(1), 11–15 (1987)
7. Bauckmann, J., Leser, U., Naumann, F., Tietz, V.: Efficiently detecting inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1448–1450 (2007)
8. Benford, F.: The law of anomalous numbers. Proc. Am. Philos. Soc. **78**(4), 551–572 (1938)
9. Berti-Equille, L., Dasu, T., Srivastava, D.: Discovery of complex glitch patterns: a novel approach to quantitative data cleaning. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 733–744 (2011)
10. Bravo, L., Fan, W., Ma, S.: Extending dependencies with conditions. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 243–254 (2007)
11. Brin, S., Motwani, R., Silverstein, C.: Beyond market baskets: generalizing association rules to correlations. SIGMOD Rec. **26**(2), 265–276 (1997)
12. Caruccio, L., Deufemia, V., Polese, G.: Relaxed functional dependencies - a survey of approaches. IEEE Trans. Knowl. Data Eng. (TKDE) **28**(1), 147–165 (2016)
13. Chandola, V., Kumar, V.: Summarization - compressing data into an informative representation. Knowl. Inf. Syst. **12**(3), 355–378 (2007)
14. Chu, X., Ilyas, I., Papotti, P., Ye, Y.: RuleMiner: data quality rules discovery. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1222–1225 (2014)
15. Cormode, G., Garofalakis, M., Haas, P.J., Jermaine, C.: Synopses for massive data: samples, histograms, wavelets, sketches. Found. Trends Databases **4**(1–3), 1–294 (2011)
16. Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A., Ilyas, I.F., Ouzzani, M., Tang, N.: NADEEF: a commodity data cleaning system. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 541–552 (2013)
17. Dasu, T., Johnson, T.: Hunting of the snark: finding data glitches using data mining methods. In: Proceedings of the International Conference on Information Quality (IQ), pp. 89–98 (1999)

18. Dasu, T., Johnson, T., Marathe, A.: Database exploration using database dynamics. IEEE Data Eng. Bull. **29**(2), 43–59 (2006)
19. Dasu, T., Johnson, T., Muthukrishnan, S., Shkapenyuk, V.: Mining database structure; or, how to build a data quality browser. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 240–251 (2002)
20. Dasu, T., Loh, J.M.: Statistical distortion: consequences of data cleaning. Proc. VLDB Endowment (PVLDB) **5**(11), 1674–1683 (2012)
21. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. (TODS) **33**(2), 1–48 (2008)
22. Flach, P.A., Savnik, I.: Database dependency discovery: a machine learning approach. AI Commun. **12**(3), 139–160 (1999)
23. Garofalakis, M., Keren, D., Samoladas, V.: Sketch-based geometric monitoring of distributed stream queries. Proc. VLDB Endowment (PVLDB) **6**(10) (2013)
24. Giannella, C., Wyss, C.: Finding minimal keys in a relation instance (1999). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.41.7086
25. Golab, L., Karloff, H., Korn, F., Srivastava, D.: Data auditor: exploring data quality and semantics using pattern tableaux. Proc. VLDB Endowment (PVLDB) **3**(1–2), 1641–1644 (2010)
26. Gunopulos, D., Khardon, R., Mannila, H., Sharma, R.S.: Discovering all most specific sentences. ACM Trans. Database Syst. (TODS) **28**, 140–174 (2003)
27. Haas, P.J., Naughton, J.F., Seshadri, S., Stokes, L.: Sampling-based estimation of the number of distinct values of an attribute. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 311–322 (1995)
28. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. **29**(2), 1–12 (2000)
29. Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. Proc. VLDB Endowment (PVLDB) **7**(4), 301–312 (2013)
30. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The MADlib analytics library or MAD skills, the SQL. Proc. VLDB Endowment (PVLDB) **5**(12), 1700–1711 (2012)
31. Hipp, J., Güntzer, U., Nakhaeizadeh, G.: Algorithms for association rule mining - a general survey and comparison. SIGKDD Explor. **2**(1), 58–64 (2000)
32. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
33. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 647–658 (2004)
34. Kache, H., Han, W.-S., Markl, V., Raman, V., Ewen, S.: POP/FED: progressive query optimization for federated queries in DB2. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 1175–1178 (2006)
35. Kandel, S., Parikh, R., Paepcke, A., Hellerstein, J., Heer, J.: Profiler: integrated statistical analysis and visualization for data quality assessment. In: Proceedings of Advanced Visual Interfaces (AVI), pp. 547–554 (2012)
36. Khoussainova, N., Balazinska, M., Suciu, D.: Towards correcting input data errors probabilistically using integrity constraints. In: Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), pp. 43–50 (2006)

37. Koehler, H., Leck, U., Link, S., Prade, H.: Logical foundations of possibilistic keys. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS (LNAI), vol. 8761, pp. 181–195. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11558-0_13
38. Koeller, A., Rundensteiner, E.A.: Heuristic strategies for the discovery of inclusion dependencies and other patterns. In: Spaccapietra, S., Atzeni, P., Chu, W.W., Catarci, T., Sycara, K.P. (eds.) Journal on Data Semantics V. LNCS, vol. 3870, pp. 185–210. Springer, Heidelberg (2006). https://doi.org/10.1007/11617808_7
39. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and armstrong relations. In: Zaniolo, C., Lockemann, P.C., Scholl, M.H., Grust, T. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 350–364. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46439-5_24
40. Lopes, S., Petit, J.-M., Toumani, F.: Discovering interesting inclusion dependencies: application to logical database tuning. Inf. Syst. **27**(1), 1–19 (2002)
41. Mannino, M.V., Chu, P., Sager, T.: Statistical profile estimation in database systems. ACM Comput. Surv. **20**(3), 191–221 (1988)
42. De Marchi, F., Lopes, S., Petit, J.-M.: Efficient algorithms for mining inclusion dependencies. In: Jensen, C.S., et al. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 464–476. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45876-X_30
43. De Marchi, F., Lopes, S., Petit, J.-M.: Unary and n-ary inclusion dependency discovery in relational databases. J. Intell. Inf. Syst. **32**, 53–73 (2009)
44. De Marchi, F., Petit, J.-M.: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: Proceedings of the IEEE International Conference on Data Mining (ICDM), pp. 27–34 (2003)
45. Morton, K., Balazinska, M., Grossman, D., Mackinlay, J.: Support the data enthusiast: challenges for next-generation data-analysis systems. Proc. VLDB Endowment (PVLDB) **7**(6), 453–456 (2014)
46. Naumann, F.: Data profiling revisited. SIGMOD Rec. **42**(4), 40–49 (2013)
47. Novelli, N., Cicchetti, R.: FUN: an efficient algorithm for mining functional and embedded dependencies. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 189–203. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44503-X_13
48. Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F.: Data profiling with metanome. Proc. VLDB Endowment (PVLDB) **8**(12), 1860–1871 (2015)
49. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.-P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: an experimental evaluation of seven algorithms. Proc. VLDB Endowment (PVLDB) **8**(10) (2015)
50. Papenbrock, T., Kruse, S., Quiané-Ruiz, J.-A., Naumann, F.: Divide & conquer-based inclusion dependency discovery. Proc. VLDB Endowment (PVLDB) **8**(7) (2015)
51. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 821–833 (2016)
52. Poosala, V., Haas, P.J., Ioannidis, Y.E., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 294–305 (1996)
53. Rahm, E., Do, H.-H.: Data cleaning: problems and current approaches. IEEE Data Eng. Bull. **23**(4), 3–13 (2000)
54. Raman, V., Hellerstein, J.M.: Potters Wheel: an interactive data cleaning system. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 381–390 (2001)

55. Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U.: A machine learning approach to foreign key discovery. In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB) (2009)
56. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: GORDIAN: efficient and scalable discovery of composite keys. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 691–702 (2006)
57. Stonebraker, M., Bruckner, D., Ilyas, I.F., Beskales, G., Cherniack, M., Zdonik, S., Pagan, A., Xu, S.: Data curation at scale: the Data Tamer system. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR) (2013)
58. Chen, M.S., Hun, J., Yu, P.S.: Data mining: an overview from a database perspective. IEEE Trans. Knowl. Data Eng. (TKDE) **8**, 866–883 (1996)
59. Wyss, C., Giannella, C., Robertson, E.: FastFDs: a heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 101–110. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44801-2_11
60. Yakout, M., Elmagarmid, A.K., Neville, J., Ouzzani, M.: GDR: a system for guided data repair. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1223–1226 (2010)
61. Yao, H., Hamilton, H.J.: Mining functional dependencies from data. Data Min. Knowl. Disc. **16**(2), 197–219 (2008)
62. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. Knowl. Data Eng. (TKDE) **12**(3), 372–390 (2000)

Programmatic ETL

Christian Thomsen(✉), Ove Andersen, Søren Kejser Jensen,
and Torben Bach Pedersen

Department of Computer Science, Aalborg University, Aalborg, Denmark
{chr,xcalibur,skj,tbp}@cs.aau.dk

Abstract. Extract-Transform-Load (ETL) processes are used for extracting data, transforming it and loading it into data warehouses (DWs). The dominating ETL tools use graphical user interfaces (GUIs) such that the developer "draws" the ETL flow by connecting steps/transformations with lines. This gives an easy overview, but can also be rather tedious and require much trivial work for simple things. We therefore challenge this approach and propose to do ETL programming by writing code. To make the programming easy, we present the Python-based framework `pygrametl` which offers commonly used functionality for ETL development. By using the framework, the developer can efficiently create effective ETL solutions from which the full power of programming can be exploited. In this chapter, we present our work on `pygrametl` and related activities. Further, we consider some of the lessons learned during the development of `pygrametl` as an open source framework.

1 Introduction

The Extract–Transform–Load (ETL) process is a crucial part for a data warehouse (DW) project. The task of the ETL process is to extract data from possibly heterogeneous source systems, do transformations (e.g., conversions and cleansing of data) and finally load the transformed data into the target DW. It is well-known in the DW community that it is both time-consuming and difficult to get the ETL right due to its high complexity. It is often estimated that up to 80% of the time in a DW project is spent on the ETL.

Many commercial and open source tools supporting the ETL developers exist [1,22]. The leading ETL tools provide graphical user interfaces (GUIs) in which the developers define the flow of data visually. While this is easy to use and easily gives an overview of the ETL process, there are also disadvantages connected with this sort of graphical programming of ETL programs. For some problems, it is difficult to express their solutions with the standard components available in the graphical editor. It is then time consuming to construct a solution that is based on (complicated) combinations of the provided components or integration of custom-coded components into the ETL program. For other problems, it can also be much faster to express the desired operations in some lines of code instead of drawing flows and setting properties in dialog boxes.

E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 21–50, 2018.
https://doi.org/10.1007/978-3-319-96655-7_2

The productivity does not become high just by using a graphical tool. In fact, in personal communication with employees from a Danish company with a revenue larger than one billion US Dollars and hundreds of thousands of customers, we have learned that they gained no change in productivity after switching from hand-coding ETL programs in C to using one of the leading graphical ETL tools. Actually, the company experienced a decrease during the first project with the tool. In later projects, the company only gained the same productivity as when hand-coding the ETL programs. The main benefits were that the graphical ETL program provided standardization and self-documenting ETL specifications such that new team members easily could be integrated.

Trained specialists are often using textual interfaces efficiently while non-specialists use GUIs. In an ETL project, non-technical staff members often are involved as advisors, decision makers, etc. but the core development is (in our experience) done by dedicated and skilled ETL *developers* that are specialists. Therefore it is attractive to consider alternatives to GUI-based ETL programs. In relation to this, one can recall the high expectations to Computer Aided Software Engineering (CASE) systems in the eighties. It was expected that non-programmers could take part in software development by specifying (not programming) characteristics in a CASE system that should generate the code. Needless to say, the expectations were not fulfilled. It might be argued that forcing all ETL development into GUIs is a step back to the CASE idea.

We acknowledge that graphical ETL programs are useful in some circumstances but we also claim that for many ETL projects, a code-based solution is the right choice. However, many parts of such code-based programs are redundant if each ETL program is coded from scratch. To remedy this, a framework with common functionality is needed.

In this chapter, we present *pygrametl* which is a programming framework for ETL programmers. The framework offers functionality for ETL development and while it is easy to get an overview of and to start using, it is still very powerful. pygrametl offers a novel approach to ETL programming by providing a framework which abstracts the access to the underlying DW tables and allows the developer to use the full power of the host programming language. For example, the use of snowflaked dimensions is easy as the developer only operates on *one* "dimension object" for the entire snowflake while pygrametl handles the different DW tables in the snowflake. It is also very easy to insert data into dimension and fact tables while only iterating the source data once and to create new (relational or non-relational) data sources. Our experiments show that pygrametl indeed is effective in terms of development time and efficient in terms of performance when compared to a leading open source GUI-based tool.

We started the work on pygrametl back in 2009 [23]. Back then, we had in a project with industrial partners been building a DW where a real-life dataset was loaded into a snowflake schema by means of a GUI-based ETL tool. It was apparent to us that the used tool required a lot of clicking and tedious work to be able to load the dataset. In an earlier project [21], we had not been able to find an ETL tool that fitted the requirements and source data. Instead we had created our ETL flow in Python code, but not in reusable, general way. Based on these experiences, we were convinced that the programmatic approach clearly was advantageous in many cases. On the other hand, it was also clear that the functionality for programmatic ETL should be generalized and isolated in a library to allow for easy reuse. Due to the ease of programming (we elaborate in Sect. 3) and the rich libraries, we chose to make a library in Python. The result was `pygrametl`. Since 2009 `pygrametl` has been developed further and made available as open source such that it now is used in proof-of-concepts and production systems from a variety of domains. In this chapter we describe the at the time of writing current version of `pygrametl` (version 2.5). The chapter is an updated and extended version of [23].

`pygrametl` is a framework where the developer makes the ETL program by coding it. `pygrametl` applies both functional and object-oriented programming to make the ETL development easy and provides often needed functionality. In this sense, `pygrametl` is related to other special-purpose frameworks where the user does coding but avoids repetitive and trivial parts by means of libraries that provide abstractions. This is, for example, the case for the web frameworks Django [3] and Ruby on Rails [17] where development is done in Python and Ruby code, respectively.

Many commercial ETL and data integration tools exist [1]. Among the vendors of the most popular products, we find big players like IBM, Informatica, Microsoft, Oracle, and SAP [5,6,10,11,18]. These vendors and many other provide powerful tools supporting integration between different kinds of sources and targets based on graphical design of the processes. Due to their wide field of functionality, the commercial tools often have steep learning curves and as mentioned above, the user's productivity does not necessarily get high(er) from using a graphical tool. Many of the commercial tools also have high licensing costs.

Open source ETL tools are also available [22]. In most of the open source ETL tools, the developer specifies the ETL process either by means of a GUI or by means of XML. Scriptella [19] is an example of a tool where the ETL process is specified in XML. This XML can, however, contain embedded code written in Java or a scripting language. `pygrametl` goes further than Scriptella and does not use XML around the code. Further, `pygrametl` offers DW-specialized functionality such as direct support for slowly changing dimensions and snowflake schemas, to name a few.

The academic community has also been attracted to ETL. Vassiliadis presents a survey of the research [28]. Most of the academic approaches, e.g., [20,26], use UML or graphs to model an ETL workflow. We challenge the idea that graphical programming of ETL is always easier than text-based programming. Grönniger et al. [4] have previously argued why text-based modeling is better than graphical modeling. Among other things, they point out that writing text is more efficient than drawing models, that it is easier to grasp details from text, and that the creative development can be hampered when definitions must be added to the graphical model. As graphical ETL tools often are model-driven such that the graphical model is turned into the executable code, these concerns are, in our opinion, also related to ETL development. Also, Petre [13] has previously argued against the widespread idea that graphical notation and programming always lead to more accessible and comprehensible results than what is achieved from text-based notation and programming. In her studies [13], she found that text overall was faster to use than graphics.

The rest of this chapter is structured as follows: Sect. 2 presents an example of an ETL scenario which is used as a running example. Section 3 gives an overview of `pygrametl`. Sections 4–7 present the functionality and classes provided by `pygrametl` to support *data sources*, *dimensions*, *fact tables*, and *flows*, respectively. Section 8 describes some other useful functions provided by `pygrametl`. Section 9 evaluates `pygrametl` on the running example. Section 10 presents support for parallelism in `pygrametl` and another `pygrametl`-based framework for MapReduce. Section 11 presents a case-study of a company using `pygrametl`. Section 12 contains a description of our experiences with making `pygrametl` available as open source. Section 13 concludes and points to future work. Appendix A offers readers who are not familiar with the subject a short introduction to data warehouse concepts.

2 Example Scenario

In this section, we describe an ETL scenario which we use as a running example. The example considers a DW where test results for tests of web pages are stored. This is inspired by work we did in the European Internet Accessibility Observatory (EIAO) project [21] but has been simplified here for the sake of brevity.

In the system, there is a web crawler that downloads web pages from different web sites. Each downloaded web page is stored in a local file. The crawler stores data about the downloaded files in a download log which is a tab-separated file. The fields of that file are shown in Table 1(a).

When the crawler has downloaded a set of pages, another program performs a number of different tests on the pages. These tests could, e.g., test if the pages are *accessible* (i.e., usable for disabled people) or conform to certain standards. Each test is applied to all pages and for each page, the test outputs the number of errors detected. The results of the tests are also written to a tab-separated file. The fields of this latter file are shown in Table 1(b).

After all tests are performed, the data from the two files is loaded into a DW by an ETL program. The schema of the DW is shown in Fig. 1. The DW schema has three dimensions: The test dimension holds information about each of the tests that are applied. This dimension is static and prefilled (and not changed by the ETL). The date dimension holds information about dates and is filled by the ETL on-demand. The page dimension is *snowflaked* and spans several tables. It holds information about the individual downloaded web pages including both static aspects (the URL and domain) and dynamic aspects (size,

Table 1. The source data format for the running example

Field	**Explanation**
localfile	Name of local file where the page was stored
url	URL from which the page was downloaded
server	HTTP header's Server field
size	Byte size of the page
downloaddate	When the page was downloaded
lastmoddate	When the page was modified

(a) DownloadLog.csv

Field	**Explanation**
localfile	Name of local file where the page was stored
test	Name of the test that was applied to the page
errors	Number of errors found by the test on the page

(b) TestResults.csv

server, etc.) that may change between two downloads. The page dimension is also filled on-demand by the ETL. The page dimension is a *type 2 slowly changing dimension* [8] where information about different *versions* of a given web page is stored.

Each dimension has a surrogate key (with a name ending in "id") and one or more attributes. The individual attributes have self-explanatory names and will not be described in further details here. There is one fact table which has a foreign key to each of the dimensions and a single measure holding the number of errors found for a certain test on a certain page on a certain date.

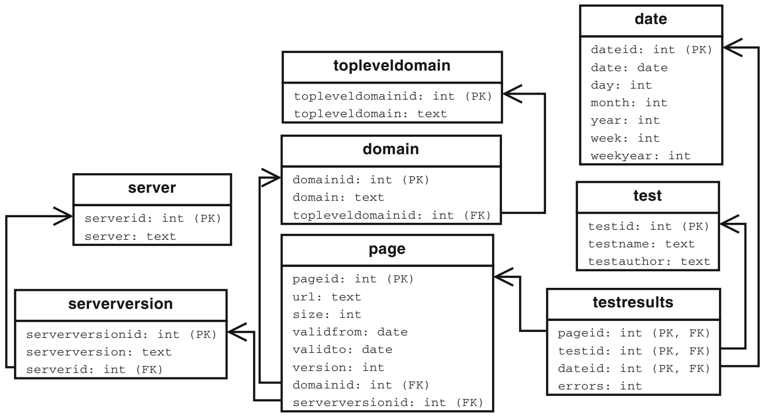


Fig. 1. The schema for the running example.

3 Overview of the Framework

Unlike many commercial ETL tools which can move data from sources to a variety of targets, the purpose of `pygrametl` is only to make it easy to load data into dimensional DWs [8] managed by relational database managements systems (RDBMSs). Focusing on RDBMSs as the targets for `pygrametl` keeps the design simple as it allows us to make assumptions and go for the good solutions specialized for this domain instead of thinking in very general "integration terms". The data sources do *not* have to be relational.

When using `pygrametl`, the programmer makes code that controls the flow, the extraction (the E in ETL) from source systems, the transformations (the T in ETL) of the source data, and the load (the L in ETL) of the transformed data. For the flow control, extraction, and load, `pygrametl` offers components that support the developer and it is easy for the developer to create more of these components. For the transformations, the programmer benefits from having access to the full-fledged Python programming language.

The loading of data into the target DW is particularly easy with `pygrametl`. The general idea is that the programmer creates *objects* for each *fact table* and *dimension* (different kinds are directly supported) in the DW. An object representing a dimension offers convenient methods like `insert`, `lookup`, etc. that hide details of caching, key assignment, SQL insertion, etc. In particular it should be noted that a snowflaked dimension also is treated in this way such that a single object can represent the entire dimension although the data is inserted into several tables in the underlying database.

The dimension object's methods take *rows* as arguments. A row in `pygrametl` is simply a mapping from names to values. Based on our personal experiences with other tools, we found it important that `pygrametl` does not try to validate that all data rows given to a dimension object have the same attributes or the same attribute types. If the programmer wants such checks, (s)he should make code for that. It is then, e.g., possible for the programmer to leave an attribute that was used as temporary value holder in a row or on purpose to leave out certain attributes. Only when attributes needed for `pygrametl`'s operations are missing, `pygrametl` complains. Attribute values that should be inserted into the target DW must exist when the insertion is done as `pygrametl` does not try to guess missing values. However, `pygrametl` has functionality for setting default values and/or on-demand call-back of user-defined functions that provide the missing values. Some other existing tools are strict about enforcing uniformity of rows. In `pygrametl`, it should be easy for the programmer to do what (s)he wants – not what the tool *thinks* (s)he wants.

`pygrametl` is implemented as a module in Python [16]. Many other programming languages could obviously have been used. We chose Python due to its design to support programmer productivity and its comprehensive standard libraries. Further, Python is both dynamically typed (the programmer does not have to declare the type a variable takes) and strongly typed (if a variable holds an integer, the programmer cannot treat it like a string). Consider, for example, this `pygrametl` function:

```
def getfloat(value, default=None):
    try:
        return float(value)
    except Exception:
        return default
```

This function converts its input to a float or – if the conversion fails – to another value which defaults to `None`, Python's null value. Note that no types are specified for the input variables in the function declaration. It is possible to call the function with different types as in the following:

```
f1 = getfloat(10)
f2 = getfloat('1e1')
f3 = getfloat('A string', 10.0)
f4 = getfloat(['A', 'list'], 'Not a float!')
```

After this, `f1`, `f2`, and `f3` all equal 10.0 while `f4` holds the string 'Not a float!'. The expression `f1 + f2` will thus succeed, while `f3 + f4` will fail since a float and a string cannot be added.

Python is object-oriented but to some degree it also supports functional programming, e.g., such that functions or lambda expressions can be used as arguments. This makes it very easy to customize behavior. `pygrametl`, for example, exploits this to support calculation of missing values on-demand (see Sect. 5). As Python also supports default arguments, `pygrametl` provides reasonable defaults for most arguments to spare the developer for unnecessary typing.

4 Data Source Support

In this and the following sections, we describe the functionality provided by `pygrametl`. As explained in Sect. 3, data is moved around in *rows* in `pygrametl`. Instead of implementing our own row class, we use Python's built-in dictionaries that provide efficient mappings between keys (i.e., attribute names) and values (i.e., attribute values). The data sources in `pygrametl` pass data on in such dictionaries. Apart from that, the only requirement to a data source is that it is iterable (i.e., its class must define the `__iter__` method) such that code as the following is possible: `for row in datasrc:...`. Thus, it does not require a lot of programming to create new sources (apart from the code that does the real extraction which might be simple or not depending on the source format). For typical use, `pygrametl` provides a few, basic data sources described below.

`SQLSource` is a data source returning the rows of an SQL query. The query, the database connection to use and optionally new names for the result columns and "initializing" SQL are given when the data source is initialized.

`CSVSource` is a data source returning the lines of a delimiter separated file turned into dictionaries. This class is in fact just implemented in `pygrametl` as a reference to the class `csv.DictReader` in Python's standard library. Consider again the running example. There we have two tab-separated files and an instance of `CSVSource` should be created for each of them to load the data. For TestResults.csv, this is done as in

```
testresults = CSVSource(open('TestResults.csv', 'r'),
                        delimiter='\t')
```

Again, we emphasize the flexibility of using a language like Python for the `pygrametl` framework. Much more configuration can be done during the instantiation than what is shown but default values are used in this example. The input could also easily be changed to come from another source than a file, e.g., a web resource or a string in memory.

`MergeJoiningSource` is a data source that equijoins rows from two other data sources. It is given two data sources (which must deliver rows in sorted order) and information about which attributes to join on. It then merges the rows from the two sources and outputs the combination of the rows.

In the running example, we consider data originating from two data sources. Both the data sources have the field `localfile` and this is how we relate information from the two files:

```
inputdata = MergeJoiningSource(testresults, 'localfile',
                               downloadlog, 'localfile')
```

where `testresults` and `downloadlog` are `CSVSource`s.

`HashJoiningSource` is also a data source that equijoins rows from two other data sources. It does this by using a hash map. Thus, the input data sources do not have to be sorted.

The data sources described above were all available in the first public release of `pygrametl` in 2009. Since then, more sources have been added including the following.

`TypedCSVSource` is like a CSV source, but will perform type casts on (textual) values coming from the input file:

```
testresults = TypedCSVSource(open('TestResults.csv', 'r'),
                             casts={'size':int},
                             delimiter='\t')
```

While `TypedCSVSource` always overwrites an attribute value with the result of a function (a cast to an int in the example above), **`TransformingSource`** allows any transformation to be applied to a row, also transformations that add new attributes. For these two sources, Python's support for functional programming is used and functions are passed as arguments.

`CrossTabbingSource` can pivot data from another source. Data from other sources can also be filtered or unioned by **`FilteringSource`** and **`UnionSource`**, respectively. A **`DynamicForEachSource`** will for each given argument create a new source which is iterated by the `DynamicForEach-Source` instance. This is, for example, useful for a directory with many CSV files to iterate. The user must provide a function that when called with a single argument returns a new source to iterate as exemplified below:

```
srcs = DynamicForEachSource([... sequence of the names of the files ...],
                            lambda f: CSVReader(open(f, 'r')))
for row in srcs: # will iterate over all rows from all the files;
   ...           # do something with the row data here
```

5 Dimension Support

In this section, we describe the classes representing dimensions in the DW to load. This is the area where the flexibility and easy use of `pygrametl` are most apparent. Figure 2 shows the class hierarchy for the dimension supporting classes (classes for parallel load of dimensions are not shown for brevity). Methods only used internally in the classes and attributes are not shown. Only required arguments are shown, not those that take default values when not given. Note that `SnowflakedDimension` actually does not inherit from `Dimension` but offers the same interface and can be used as if it were a `Dimension` due to Python's dynamic typing.

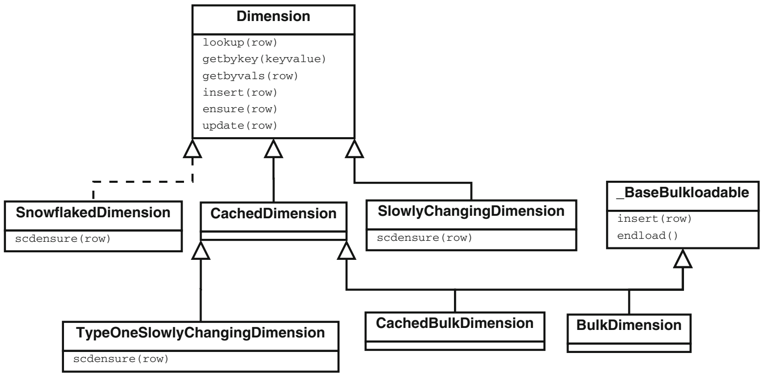


Fig. 2. Class hierarchy for the dimension supporting classes.

5.1 Basic Dimension Support

`Dimension` is the most basic class for representing a DW dimension in `pygram-etl`. It is used for a dimension that has exactly one table in the DW. When an instance is created, the name of the represented dimension (i.e., the name of the table in DW), the name of the key column[1], and a list of attributes (the underlying table may have more attributes but `pygrametl` will then not use them) must be given. Further, a number of optional settings can be given as described in the following. A list of *lookup attributes* can be given. These attributes are used when looking up the key value. Consider again the running example. The test dimension has the surrogate key testid but when data is inserted from the CSV files, the test in question is identified from its name (testname). The ETL application then needs to find the value of the surrogate key based on the test name. That means that the attribute testname is a lookup attribute. If no lookup attributes are given by the user, the full list of attributes (apart from the key) is used.

When the dimension object is given a row to insert into the underlying DW table (explained below), the row does not need to have a value for the dimension's key. If the key is not present in the row, a method (called `idfinder`) is called with the row as an argument. Thus, when creating a `Dimension` instance, the

[1] We assume that a dimension has a non-composite key.

`idfinder` method can also be set. If not set explicitly, it defaults to a method that assumes that the key is numeric and returns the current maximum value for the key incremented by one.

A default key value for unfound dimension members can also be set. If a lookup does not succeed, this default key value is returned. This is used if new members should not be inserted into the dimension but facts still should be recorded. By using a default key value, the facts would then reference a prefilled member representing that information is missing. In the running example, test is a prefilled dimension that should not be changed by the ETL application. If data from the source file TestResults.csv refers to a test that is not represented in the test dimension, we do not want to disregard the data by not adding a fact. Instead, we set the default ID value for the test dimension to be -1 which is the key value for a preloaded dimension member with the value "Unknown test" for the testname attribute. This can be done as in the following code.

```
testdim = Dimension(name='test',
                    key='testid',
                    defaultidvalue=-1,
                    attributes=['testname', 'testauthor'],
                    lookupatts=['testname'])
```

Finally, it is possible for the developer to assign a function to the argument `rowexpander`. With such a function, it is in certain situations (explained below) possible to add required fields on-demand to a row before it is inserted into the dimension.

Many of the methods defined in the `Dimension` class accept an optional *name mapping* when called. This name mapping is used to map between attribute names in the rows (i.e., dictionaries) used in `pygrametl` and names in the tables in the DW. Consider again the running example where rows from the source file TestResults.csv have the attribute test but the corresponding attribute in the DW's dimension table is called testname. When the `Dimension` instance for test in `pygrametl` is given a row r to insert into the DW, it will look for the value of the testname attribute in r. However, this value does not exist since it is called test in r. A name mapping $n = \{\text{'testname'} : \text{'test'}\}$ can then be set up such that when `pygrametl` code looks for the attribute testname in r, test is actually used instead.

`Dimension` offers the method `lookup` which based on the lookup attributes for the dimension returns the key for the dimension member. As arguments it takes a row (which at least must contain the lookup attributes) and optionally a name mapping. `Dimension` also offers the method `getbykey`. This method is the opposite of `lookup`: As argument it takes a key value and it returns a row with all attributes for the dimension member with the given key value. Another method for looking up dimension members is offered by `Dimension`'s `getbyvals` method. This method takes a row holding a subset of the dimension's attributes and optionally a name mapping. Based on the subset of attributes, it finds the dimension members that have equal values for the subset of attributes and returns those (full) rows. For adding a new member to a dimension, `Dimension` offers the method `insert`. This method takes a row and optionally a name mapping as arguments. The row is added to the DW's dimension table. All attributes of the dimension must be present in the pygrametl row. The only exception to this is the key. If the key is missing, the idfinder method is applied to find the key value. The method update takes a row which must contain the key and one or more of the other attributes. The member with the given key value is updated to have the same values as the given attributes.

Dimension also offers a combination of lookup and insert: ensure. This method first tries to use lookup to find the key value for a member. If the member does not exist and no default key value has been set, ensure proceeds to use insert to create the member. In any case, ensure returns the key value of the member to the caller. If the rowexpander has been set (as described above), that function is called by ensure before insert is called. This makes it possible to add calculated fields before an insertion to the DW's dimension table is done. In the running example, the date dimension has several fields that can be calculated from the full date string (which is the only date information in the source data). However, it is expensive to do the calculations repeatedly for the same date. By setting rowexpander to a function that calculates them from the date string, the dependent fields are only calculated the first time ensure is invoked for certain date.

CachedDimension has the same public interface as Dimension and the same semantics. However, it internally uses memory caching of dimension members to speed up lookup operations. The caching can be complete such that the entire dimension is held in memory or partial such that only the most recently used members are held in memory. A CachedDimension can also cache new members as they are being added.

When an instance of CachedDimension is created, it is possible to set the same settings as for Dimension. Further, optional settings can decide the size of the cache, whether the cache should be prefilled with rows from the DW or be filled on-the-fly as rows are used, whether full rows should be cached or only keys and lookup attributes, and finally whether newly inserted rows should be put in the cache. In the running example, a CachedDimension for the test dimension can be made as in the following code.

```
testdim = CachedDimension(name='test',
                          key='testid',
                          defaultidvalue=-1,
                          attributes=['testname', 'testauthor'],
                          lookupatts=['testname'],
                          cachesize=500,
                          prefill=True,
                          cachefullrows=True)
```

5.2 Advanced Dimension Support

SlowlyChangingDimension provides support for type 2 changes in slowly changing dimensions [8] and in addition to type 2 changes, type 1 changes can also be supported for a subset of the dimension's attributes. When an instance of SlowlyChangingDimension is created, it can be configured in the same way as a `Dimension` instance. Further, the name of the attribute that holds versioning information for type 2 changes in the DW's dimension table can be set. If it is set, the row version with the greatest value for this attribute is considered the newest row and `pygrametl` will automatically maintain the version number for newly added versions. If there is no such attribute with version information, the user can specify another attribute to be used for the ordering of row versions. A number of other things can optionally be configured. It is possible to set which attribute holds the "from date" telling from when the dimension member is valid. Likewise it is possible to set which attribute holds the "to date" telling when a member becomes replaced. A default value for the "to date" for a new member can also be set as well as a default value for the "from date" for the first version of a new member. Further, functions that, based on data in new rows to add, calculate the "to date" and "from date" can be given but if they are not set, `pygrametl` defaults to use a function that returns the current date. `pygrametl` offers some convenient functions for this functionality. It is possible not to set any of these date related attributes such that no validity date information is stored for the different versions. It is also possible to list a number of attributes that should have type 1 changes (overwrites) applied. `SlowlyChangingDimension` has built-in cache support and its details can be configured. Finally, it is possible to configure if versions should be sorted by the DBMS such that `pygrametl` uses SQL's ORDER BY or if `pygrametl` instead should fix all versions of a given member and do a sort in Python. It depends on the used DBMS what performs the best, but for at least one popular commercial DBMS, it is significantly faster to let `pygrametl` perform the sorting.

`SlowlyChangingDimension` offers the same functions as `Dimension` (which it inherits from) and the semantics of the functions are basically unchanged. `lookup` is, however, modified to return the key value for the *newest* version. To handle versioning, `SlowlyChangingDimension` offers the method `scdensure`. This method is given a row (and optionally a name mapping). It is similar to `ensure` in the sense that it first sees if the member is present in the dimension and, if not, inserts it. However, it does not only do a lookup. It also detects if any changes have occurred. If changes have occurred for attributes where type 1 changes should be used, it updates the existing versions of the member. If changes have also occurred for other attributes, it creates a new version of the member and adds the new version to the dimension. As opposed to the previously described methods, `scdensure` has side-effects on its given row: It sets the key and versioning values in its given row such that the programmer does not have to query for this information afterwards.

When a page is downloaded in the running example, it might have been updated compared to last time it was downloaded. To be able to record this history, we let the page dimension be a slowly changing dimension. We add a new version when the page has been changed and reuse the previous version when the page is unchanged. We lookup the page by means of the URL and detect changes by considering the other attributes. We create the `SlowlyChangingDimension` object as in the following.

```
pagedim = SlowlyChangingDimension(name='page',
                                  key='pageid',
                                  attributes=['url', 'size', 'validfrom',
                                              'validto', 'version', 'domainid',
                                              'serverversionid'],
                                  lookupatts=['url'],
                                  fromatt='validfrom',
                                  fromfinder=pygrametl.datereader('lastmoddate'),
                                  toatt='validto',
                                  versionatt='version')
```

In the shown code, the `fromfinder` argument is a method that extracts a "from date" from the source data when creating a new member. It is also possible to give a `tofinder` argument to find the "to date" for a version to be replaced. If not given, this defaults to the `fromfinder`. If another approach is wished, (e.g., such that the to date is set to the day before the new member's from date), `tofinder` can be set to a function which performs the necessary calculations.

`SnowflakedDimension` supports filling a dimension in a snowflake schema [8]. A snowflaked dimension is spread over more tables such that there is one table for each level in the dimension hierarchy. The fact table references one of these tables that itself references tables that may reference other tables etc. A dimension member is thus not only represented in a single table as each table in the snowflaked dimension represents a part of the member. The complete member is found by joining the tables in the snowflake.

Normally, it can be a tedious task to create ETL logic for filling a snowflaked dimension. First, a lookup can be made on the *root table* which is the table referenced by the fact table. If the member is represented there, it is also represented in the dimension tables further away from the fact table (otherwise the root table could not reference these and thus not represent the member at the lowest level). If the member is not represented in the root table, it must be inserted but it is then necessary to make sure that the member is represented in the next level of tables such that the key values can be used in references. This process continues for all the levels until the leaves[2]. While this is not difficult as such, it takes a lot of tedious coding and makes the risk of errors bigger. This is remedied with `pygrametl`'s `SnowflakedDimension` which takes care of the repeated `ensure`s such that data is inserted where needed in the snowflaked dimension but such that the developer only has to make one method call to add/find the member.

An instance of `SnowflakedDimension` is constructed from other `Dimension` instances. The programmer creates a `Dimension` instance for each table participating in the snowflaked dimension and passes those instances when creating the `SnowflakedDimension` instance. In the running example, the page dimension is snowflaked. We can create a `SnowflakedDimension` instance for

[2] It is also possible to do the lookups and insertions from the leaves towards the root but when going towards the leaves, it is possible to stop the search earlier if a part of the member is already present.

the page dimension as shown in the following code (where different `Dimension` instances are created before).

```
pagesf = SnowflakedDimension([
              (pagedim, [serverversiondim, domaindim]),
              (serverversiondim, serverdim),
              (domaindim, topleveldim) ])
```

The argument is a list of pairs where a pair shows that its first element references each of the dimensions in the second element (the second element may be a list). For example, it can be seen that pagedim references serverversiondim and domaindim. We require that if t's key is named k, then an attribute referencing t from another table must also be named k. This requirement could be removed but having it makes the specification of relationships between tables much easier. We also require that the tables in a snowflaked dimension form a tree (where the table closest to the fact table is the root) when we consider tables as nodes and foreign keys as edges. Again, we could avoid this requirement but this would complicate the ETL developer's specifications and the requirement does not limit the developer. If the snowflake does not form a tree, the developer can make `SnowflakedDimension` consider a subgraph that is a tree and use the individual `Dimension` instances to handle the parts not handled by the `SnowflakedDimension`. Consider, for example, a snowflaked date dimension with the levels day, week, month, and year. A day belongs both to a certain week and a certain month but the week and the month may belong to different years (a week has a week number between 1 and 53 which belongs to a year). In this case, the developer could ignore the edge between week and year when creating the `SnowflakedDimension` and instead use a single method call to ensure that the week's year is represented:

```
# Represent the week's year. Read the year from weekyear
row['weekyearid'] = yeardim.ensure(row,{'year':'weekyear'})
# Now let SnowflakedDimension take care of the rest
row['dateid'] = datesnowflake.ensure(row)
```

`SnowflakedDimension`'s `lookup` method calls the `lookup` method on the `Dimension` object for the root of the tree of tables. It is assumed that the lookup attributes belong to the table that is closest to the fact table. If this is not the case, the programmer can use `lookup` or `ensure` on a `Dimension` further away from the root and use the returned key value(s) as lookup attributes for the `SnowflakedDimension`. The method `getbykey` takes an optional argument that decides if the full dimension member should be returned (i.e., a join between the tables of the snowflaked dimension is done) or only the part from the root. This also holds for `getbyvals`. `ensure` and `insert` work on the entire snowflaked dimension starting from the root and moving outwards as much as needed. The two latter methods actually use the same code. The only difference is that `insert`, to be consistent with the other classes, raises an exception if nothing is inserted (i.e., if all parts were already there). Algorithm 1 shows how the code *conceptually* works but we do not show details like use of name mappings and how to keep track of if an insertion did happen. The algorithm is recursive and both `ensure` and `insert` first invoke it with the

Algorithm 1. `ensure_helper`(*dimension*, *row*)

```
1: $keyval \leftarrow dimension.lookup(row)$
2: if found then
3:     $row[dimension.key] \leftarrow keyval$
4:     return $keyval$
5: for each table $t$ that is referenced by $dimension$ do
6:     $keyval \leftarrow ensure\_helper(t, row)$
7: if $dimension$ uses the key of a referenced table as a lookup attribute then
8:     $keyval \leftarrow dimension.lookup(row)$
9:     if not found then
10:        $keyval \leftarrow dimension.insert(row)$
11: else
12:     $keyval \leftarrow dimension.insert(row)$
13: $row[dimension.key] \leftarrow keyval$
14: return $keyval$
```

table *dimension* set to the table closest to the fact table. On line 1, a normal `lookup` is performed on the table. If the key value is found, it is set in the row and returned (lines 2–4). If not, the algorithm is applied recursively on each of the tables that are referenced from the current table (lines 5–6). As side-effects of the recursive calls, key values are set for all referenced tables (line 3). If the key of one of the referenced tables is used as a lookup attribute for *dimension*, it might just have had its value changed in one of the recursive calls and a new attempt is made to look up the key in *dimension* (lines 7–8). If this attempt fails, we insert (part of) *row* into *dimension* (line 10). We can proceed directly to this insertion if no key of a referenced table is used as a lookup attribute in *dimension* (lines 11–12).

`SnowflakedDimension` also offers an `scdensure` method. This method can be used when the root is a `SlowlyChangingDimension`. In the running example, we previously created `pagedim` as an instance of `SlowlyChangingDimension`. When `pagedim` is used as the root as in the definition of `pagesf` above, we can use the slowly changing dimension support on a snowflake. With a single call of `scdensure`, a full dimension member can be added such that the relevant parts are added to the five different tables in the page dimension.

When using the graphical ETL tools such as SQL Server Integration Services (SSIS) or the open source Pentaho Data Integration (PDI), use of snowflakes requires the developer to use several lookup/update steps. It is then not easily possible to start looking up/inserting from the root as foreign key values might be missing. Instead, the developer has to start from the leaves and go towards the root. In `pygrametl`, the developer only has to use the `SnowflakedDimension` instance once. The `pygrametl` code considers the root first (and may save lookups) and only if needed moves on to the other levels.

The previously described Dimension classes were all present in the first public release of `pygrametl` [23]. Since then more classes have been added.

TypeOneSlowlyChangingDimension is similar to SlowlyChaningDimension apart from that it *only* support type 1 changes where dimension members are updated (not versioned) to reflect changes [8]. **BulkDimension** is used in scenarios where much data must be inserted into a dimension table and it becomes too time-consuming to use traditional SQL INSERTs (as the previously described Dimension classes do). BulkDimension instead writes new dimension values to a temporary file which can be bulk loaded.

The exact way to bulkload varies from DBMS to DBMS. Therefore, we rely on Python's functional programming support and require the developer to pass a function when creating an instance of BulkDimension. This function is invoked by pygrametl when the bulkload should take place. When using the database driver psycopg2 [15] and the DBMS PostgreSQL [14], the function can be defined as below.

```
def pgbulkloader(name, attributes, fieldsep, rowsep,
                 nullval, filehandle):
    global connection # Opened outside this method
    cursor = connection.cursor()
    cursor.copy_from(file=filehandle, table=name, sep=fieldsep,
                     null=nullval, columns=attributes)
```

The developer can optionally also define which separator and line-ending to use and which file the data is written to before the bulkload. A string value used to represent nulls can also be defined.

To enable efficient lookups, BulkDimension caches all data of the dimension in memory. This is viable for most dimensions as modern computers have big amounts of memory. In some scenarios it can, however, be infeasible to cache all data of a dimension in memory. If that is the case and efficient bulk loading still is desired, the **CachedBulkDimension** can be used. Like the CachedDimension, the size of its cache can be configured, but in addition it supports bulk loading. To avoid code duplication, code supporting bulk loading has been placed in the class **_BaseBulkloadable** which BulkDimension and CachedBulkDimension then inherit from (as does BulkFactTable described below).

6 Fact Table Support

pygrametl also offers three classes to represent fact tables. In this section, we describe these classes. It is assumed that a fact table has a number of key attributes and that each of these is referencing a dimension table. Further, the fact tables may have a number of measure attributes.

FactTable provides a basic representation of a fact table. When an instance is created, the programmer gives information about the name fact table, names of key attributes and optionally names of measure attributes.

FactTable offers the method insert which takes a row (and optionally a name mapping) and inserts a fact into the DW's table. This is obviously the most used functionality. It also offers a method lookup which takes a row that holds values for the key attributes and returns a row with values for both key and measure attributes. Finally, it offers a method `ensure` which first tries to use `lookup`. If a match is found on the key values, it compares the measure values between the fact in the DW and the given row. It raises an error if there are differences. If no match is found, it invokes `insert`. All the methods support name mappings.

`BatchFactTable` inherits `FactTable` and has the same methods. However, it does not insert rows immediately when `insert` is called but instead keeps them in memory and waits until a user-configurable number of rows are available. This can lead to a high performance improvement.

`BulkFactTable` provides a write-optimized representation of a fact table. It does offer the `insert` method but not `lookup` or `ensure`. When `insert` is called, the data is not inserted directly into the DW but instead written to a file. When a user-configurable number of rows have been added to the file (and at the end of the load), the content of the file is *bulkloaded* into the fact table.

`BulkFactTable` inherits from `_BaseBulkloadable` which provides support for bulk loading. As for `BulkDimension` and `CachedBulkDimension`, the user has to provide a function that invokes the bulk-loading method of her particular DB driver. For the running example, a `BulkFactTable` instance can be created for the fact table as shown below.

```
facttbl = BulkFactTable(name='testresults',
                        measures=['errors'],
                        keyrefs=['pageid', 'testid', 'dateid'],
                        bulkloader=pgbulkloader,
                        bulksize=5000000)
```

7 Flow Support

A good aspect from GUI-based ETL tools, is that it is easy to keep different aspects separated and thus to get an overview of what happens in a sub-task. To make it possible to create small components with encapsulated functionality and easily connect such components, `pygrametl` offers support for *steps* and flow of data between them. The developer can, for example, create a step for extracting data, a step for cleansing, a step for logging, and a step for insertion into the DW's tables. Each of the steps can be coded individually and finally the data flow between them can be defined.

`Step` is the basic class for flow support. It can be used directly or as a base class for other step-supporting classes. The programmer can for a given `Step` set a *worker function* which is applied on each row passing through the `Step`. If not set by the programmer, the function `defaultworker` (which does not do anything) is used. Thus, `defaultworker` is the function inheriting classes override. The programmer can also determine to which `Step` rows by default should be sent after the current. That means that when the worker function finishes its work, the row is passed on to the next `Step` unless the programmer specifies otherwise. So if no default `Step` is set or if the programmer wants to send the row to a non-default `Step` (e.g., for error handling), there is the function `_redirect` which the programmer can use to explicitly direct the row to a specific `Step`.

There is also a method `_inject` for injecting a new row into the flow before the current row is passed on. The new row can be injected without an explicit target in which case the new row is passed on the `Step` that rows by default are sent to. The new row can also be injected and sent to a specified target. This gives the programmer a large degree of flexibility.

The worker function can have side-effects on the rows it is given. This is, for example, used in the class **`DimensionStep`** which calls `ensure` on a certain `Dimension` instance for each row it sees and adds the returned key to the row. Another example is **`MappingStep`** which applies functions to attributes in each row. A typical use is to set up a `MappingStep` applying `pygrametl`'s type conversion functions to each row. A similar class is **`ValueMappingStep`** which performs mappings from one value set to another. Thus, it is easy to perform a mapping from, e.g., country codes like 'DK' and 'DE' to country names like 'Denmark' and 'Germany'. To enable conditional flow control, the class **`ConditionalStep`** is provided. A `ConditionalStep` is given a condition (which is a function or a lambda expression). For each row, the condition is applied to the row and if the condition evaluates to a `True` value, the row is passed on to the next default `Step`. In addition, another `Step` can optionally be given and if the condition then evaluates to a `False` value for a given row, the row is passed on to that `Step`. Otherwise, the row is silently discarded. This is very easy to use. The programmer only has to pass on a lambda expression or function. Also, to define new step functionality is very easy. The programmer just writes a single function that accepts a row as input and gives this function as an argument when creating a `Step`.

`Steps` can also be used for doing aggregations. The base class for aggregating steps, **`AggregatingStep`**, inherits `Step`. Like an ordinary `Step`, it has a `defaultworker`. This method is called for each row given to the `AggregatingStep` and must maintain the necessary data for the computation of the average. Further, there is a method `defaultfinalizer` that is given a row and writes the result of the aggregation to the row.

The functionality described above can of course also be implemented by the developer without `Steps`. However, the `Step` functionality was included in the first public release of `pygrametl` to support a developers who prefer to think in terms of connected steps (as typically done in GUI-based ETL programs). In hindsight, this seems to be a non-wanted functionality. While we have received comments, questions, and bug reports about most other areas of `pygrametl`, we have virtually never received anything about `Steps`. It seems that developers who choose to define their ETL flows with code, in fact prefer not to think in the terms used by GUI-based tools.

8 Further Functionality

Apart from the classes described in the previous sections, `pygrametl` also offers some convenient methods often needed for ETL. These include functions that operate on rows (e.g., to copy, rename, project attributes or set default values) and functions that convert types, but return a user-configurable default value if the conversion cannot be done (like `getfloat` shown in Sect. 3).

In particular for use with `SlowlyChangingDimension` and its support for time stamps on versions, `pygrametl` provides a number of functions for parsing strings to create date and time objects. Some of these functions apply functional programming such that they dynamically create new functions based on their arguments. In this way specialized functions for extracting time information can be created. For an example, refer to `pagedim` we defined in Sect. 5. There we set `fromfinder` to a (dynamically generated) function that reads the attribute lastmoddate from each row and transforms the read text into a date object.

While this set of provided `pygrametl` functions is relatively small, it is important to remember that with a framework like `pygrametl`, the programmer also has access to the full standard library of the host language (in this case Python). Further, it is easy for the programmer to build up private libraries with the most used functionality.

9 Evaluation

To evaluate `pygrametl` and compare the development efforts for visual and code-based programming, the full paper about `pygrametl` [24] presented an evaluation where the running example was implemented in both `pygrametl` and the graphical ETL tool Pentaho Data Integration (PDI) [12], a popular Java-based open source ETL tool. Ideally, the comparison should have included commercial ETL tools, but the license agreements of these tools (at least the ones we have read) explicitly forbid publishing any evaluation/performance results without the consent of the provider, so this was not possible. In this section, we present the findings of the evaluation. `pygrametl`, the case ETL program, and the data generator are publicly available from pygrametl.org.

9.1 Development Time

It is obviously difficult to make a comparison of two such tools, and a full-scale test would require several teams of fully trained developers, which is beyond our resources. We obviously know `pygrametl` well, but also have solid experience with PDI from earlier projects. Each tool was used twice to create identical solutions. In the first use, we worked slower as we also had to find a strategy. In the latter use, we found the "interaction time" spent on typing and clicking.

The `pygrametl`-based program was very easy to develop. It took a little less than an hour in the first use, and 24 min in the second. The program consists of ~140 short lines, e.g., only one argument per line when creating `Dimension` objects. This strongly supports that it is easy to develop ETL programs using `pygrametl`. The main method of the ETL is shown below.

```
def main():
    for row in inputdata:
        extractdomaininfo(row)
        extractserverinfo(row)
        row['size'] = pygrametl.getint(row['size'])
        # Add the data to the dimension and fact tables
        row['pageid'] = pagesf.scdensure(row)
        row['dateid'] = datedim.ensure(row,
                                  {'date':'downloaddate'})
        row['testid'] = testdim.lookup(row,
                                  {'testname':'test'})
        facttbl.insert(row)
    connection.commit()
```

The methods `extractdomaininfo` and `extractserverinfo` have four lines of code to extract domain, top-level domain, and server name. Note that the page dimension is an SCD, where `scdensure` is a very easy way to fill a both snowflaked and slowly changing dimension. The date dimension is filled using a `rowexpander` for the `datedim` object to (on demand) calculate the attribute values so it is enough to use `ensure` to find or insert a member. The test dimension is preloaded and we only do lookups.

In comparison, the PDI-based solution took us a little more than two hours in the first use, and 28 min in the second. The flow is shown in Fig. 3. We emulate the `rowexpander` feature of `pygrametl` by first looking up a date and calculating the remaining date attributes in case there is no match. Note how we must fill the page snowflake from the leaves towards the root.

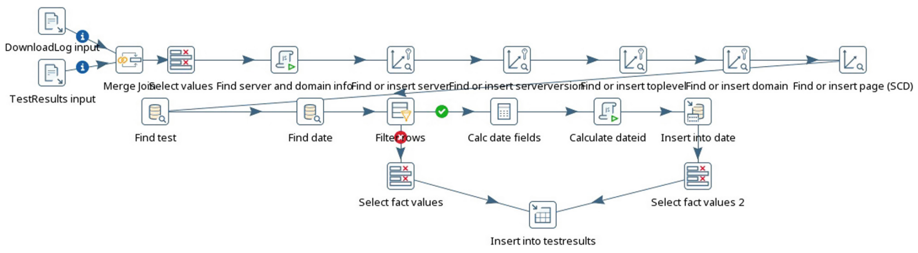


Fig. 3. Data flow in PDI-based solution.

To summarize, `pygrametl` was faster to use than PDI. The first solution was much faster to create in `pygrametl` and we believe that the strategy is far simpler to work out in `pygrametl` (compare the shown main method and Fig. 3). And although trained PDI users also may be able to generate a first solution in PDI quickly, we still believe that the `pygrametl` approach is simple and easy to use which is also seen from the second uses of the tools where it was as fast (actually a little faster) to type code as to click around in the GUI to (re)create the prepared solutions.

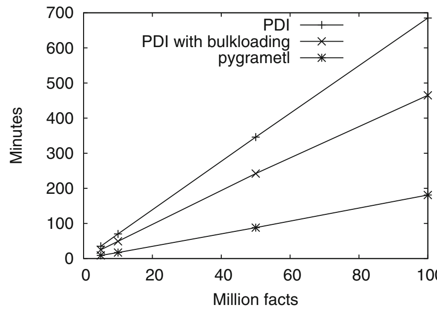
9.2 Performance

The original full paper [24] also presented performance results for both PDI and `pygrametl` on the running example. In the current chapter, we provide new results for the same example, but with the newer versions of both PDI (version 7.1) and `pygrametl` (version 2.5) and executed on newer hardware.

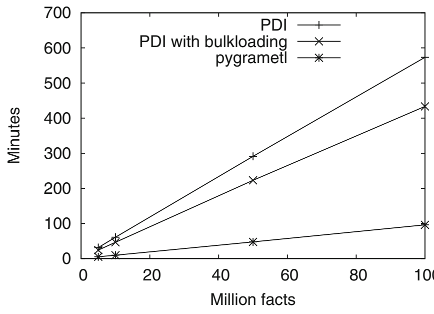
To test the performance of the solutions, we generated data. The generator was configured to create results for 2,000 different domains each having 100 pages. Five tests were applied to each page. Thus, data for one month gave 1 million facts. To test the SCD support, a page could remain unchanged between two months with probability 0.5. For the first month, there were thus 200,000 page versions and for each following month, there were ∼100,000 new page versions. We did the tests for 5, 10, 50, and 100 months, i.e., on data sets of realistic sizes. The solutions were tested on a single[3] virtual machine with three virtual processors, and 16 GB of RAM (the CPUs were never completely used during the experiments and the amount of RAM was big enough to allow both `pygrametl` and PDI to cache all dimension data). The virtual machine ran openSUSE Leap 42.2 Linux, pygrametl 2.5 on Python 3.6, PDI 7.1 on OpenJDK 8, and PostgreSQL 9.4. The virtual machine was running under VirtualBox 5.1 on Windows 10 on a host machine with 32 GB of RAM, SSD disk, and a 2.70 GHz Intel i7 CPU with 4 cores and hyperthreading.

We tested the tools on a DW where the primary key constraints were declared but the foreign key constraints were not. The DW had an index on `page(url, version)`.

PDI was tested in two modes. One with a single connection to the DW such that the ETL is transactionally *safe* and one which uses a special component for bulkloading the facts into PostgreSQL. This special component makes its own connection to the DW. This makes the load faster but transactionally *unsafe* as a crash can leave the DW loaded with inconsistent data. The `pygrametl`-based solution uses bulkloading of facts (by means of `BulkFactTable`) but is always running in a *safe* transactional mode with a single connection to the DW. The solutions were configured to use caches without size limits. When PDI was tested, the maximal Java heap size was set to 12 GB.



(a) Elapsed time



(b) CPU time

Fig. 4. Performance results.

[3] We did not test PDI's support for distributed execution.

Figure 4(a) shows the elapsed wall-clock time for the loads and Fig. 4(b) shows the spent CPU time. It can be seen that the amounts of time grow linearly for both PDI and `pygrametl`.

`pygrametl` is significantly faster than PDI in this experiment. When loading 100 million facts, the `pygrametl`-based solution handles 9208 facts/s. PDI with a single connection handles 2433 and PDI with two connections handles 3584 facts/s.

Servers may have many CPUs/cores but it is still desirable if the ETL uses little CPU time. More CPU time is then available for other purposes like processing queries. This is in particular relevant if the ETL is running on a virtualized server with contention for the CPUs. From Fig. 4(b), it can be seen that `pygrametl` also uses much less CPU time than PDI. For example, when loading the data set with 100 million facts, `pygrametl`'s CPU utilization is 53%. PDI's CPU utilization is 83% with one connection and 93% with two. It is clearly seen that it in terms of resource consumption, it is beneficial to code a specialized light-weight program instead of using a general feature-rich but heavy-weight ETL application.

10 Parallelism

ETL flows often have to handle large data volumes and parallelization is a natural way to achieve good performance. `pygrametl` has support for both task parallelism and data parallism [25]. The support was made to keep the simplicity of `pygrametl` programs For example, extraction and transformation of data can happen in another process using a **`ProcessSource`**:

```
rawdata = CSVSource(open('myfile.csv', 'r'))
transformeddata = TransformingSource(rawdata, transformation1, transformation2)
inputdata = ProcessSource(transformeddata)
```

The ProcessSource will here spawn another process in which the rows of `transformeddata` are found and then delivered to the `inputdata` object in the main process. In a similar way, `Dimension` instances can also do their work in another process, by means of **`DecoupledDimension`** which has an interface identical to `Dimension`, but just pushes all work to a `Dimension` instance in another process. This other instance is given to `DecoupledDimension` when it is created as exemplified below:

```
pagedim = DecoupledDimension(SlowlyChangingDimension(name='page', ...))
```

Work on a `FactTable` instance can also happen in another process by means of **`DecoupledFactTable`**. Decoupled instances can also be defined to *consume* data from each other such that, e.g., lookup operations don't become blocking but rather return a placeholder value that a consuming class later will get without involvement from the user (or main process). For details, see [25].

```
facttbl = DecoupledFactTable(BulkFactTable(name='testresults', ...),
                             consumes=pagedim,
                             returnvalues=False)
```

Finally, `pygrametl` has support for making user-defined functions run in parallel with other tasks by means of using a **`splitpoint`** annotation.

```
@splitpoint(instances=2)
def myfunction(*args)
    # Do some (possibly expensive) transformations here
```

In [25], it is concluded that the simple constructs for parallelism give good improvements and that "a small increase in used CPU time gives a large increase in performance. The available CPUs are used more intensively, but for a shorter amount of (wall-clock) time". In the `pygrametl` implementation, new processes are spawned when `pygrametl` is executed on CPython, the reference implementation for Python. The reason is that Python threads cannot execute Python bytecode in parallel on CPython due to the global interpreter lock (GIL). Use of threads on CPython can thus result in poor performance. When `pygrametl` is running on Jython, the implementation of Python in Java, threads are used instead since their performance is good on the JVM. It is, however, possible to tune the performance of both processes and threads by setting defining sizes for *batches* of row and the *queues* they are placed in when transferred to another process/thread. We have attempted to pick reasonable default values, but have also experienced that other values may increase or decrease the performance significantly. Unfortunately, it is not easy to pick good values as sizes that work well on one machine can be unappropriate for another machine. Automatic and dynamic selection of good values for these sizes is thus an interesting future task.

The functionality described above enables parallelism on a single computer. However, it becomes infeasible to scale up to very large amounts of data and instead it is necessary to scale out. This is exactly what the MapReduce [2] framework does. It is thus interesting to apply MapReduce to ETL programming. However, while MapReduce offers high flexibility and scalability, it is a generic programming model and has no native support of ETL-specific constructs such as star schemas, snowflake schemas, and slowly changing dimensions. Implementing a parallel ETL procedure on MapReuce is therefore complex, costly, and error-prone and leads to low programmer productivity. As a remedy, Liu et al. [9] presented ETLMR which is a dimensional ETL framework for MapReduce with direct support for high-level ETL constructs. The general idea is that ETLMR leverages the functionality of MapReduce, but also hides the complexity. To achieve this, the user only specifies transformations and declarations of sources and targets which only requires few lines of code. The implementation of ETLMR is based on `pygrametl` but some parts have been extended or modified to support MapReduce.

To exploit the strengths of MapReduce, the user has a little less freedom with ETLMR than with `pygrametl`. With ETLMR, an ETL flow always consists of dimension processing first followed by fact processing in another MapReduce job. In a file, config.py, the user has to declare sources and targets as well as a mapping called dims which tells which attributes are needed for each dimension and which transformations (i.e., Python functions) to apply to dimension data first. An example for our running example is shown below.

```
from odottables import *  # Different dimension processing schemes are supported
fileurls = ['dfs://.../TestResults0.csv' ,'dfs://.../TestResults1.csv', ...]

datedim = CachedDimension(...)  # Similar to declarations in pygrametl
pagedim = SlowlyChangingDimension(...)
testdim = CachedDimension(...)

dims = {pagedim:{'srcfields':('url', 'serverversion', 'domain',
                              'size', 'lastmoddate'),
                 'rowhandlers':(UDF_extractdomain, UDF_extractserver)},
         datedim:...,
         testdim:...
        }
```

There are different dimension processing schemes available in ETLMR. The first one is called *One Dimension, One Task (ODOT)*. In this scheme, mappers will project attributes needed for the different dimensions and emit pairs of the form (dimension name, list of attribute values). One reducer will then process *all* data for one dimension and apply user-defined transformations, do key generation, and fill/update the dimension table. Another scheme is called *One Dimenion, All Tasks (ODAT)*. In ODAT, mappers will also project attributes, but here they for each input row emit one key/value pair with data for all dimensions, i.e., a pair of the form (rownumber, [dimension1:{...}, dimension2:{...}, ...]). These pairs are then distributed to the reducers in a round-robin fashion and one reducer thus processes data for all dimensions and one dimension is processed by all reducers. This can, however, lead to inconsistencies and therefore a final step is needed to clean up such inconsistencies [9]. A hybrid of ODAT and ODOT is also possible. In this hybrid scheme, a data-intensive dimension (such as pagedim) can be partitioned based on business key (url) and processed by all tasks which is ODAT-like while the remaining dimensions are processed in ODOT-style.

11 Case Study

In this section, we describe how and why a concrete company uses programmatic ETL and `pygrametl` for its DW solutions.

The company FlexDanmark[4] handles demand-responsive transport in Denmark. For example, elderly people are picked up at home and driven to a hospital for examinations and school children living in areas without access to public transportation are driven from home to school. The yearly revenue is around 120 million US dollars. To organize the transportation and plan the different tours effectively, FlexDanmark makes routing based on detailed speed maps built from GPS logs from more than 5,000 vehicles. Typically, 2 million GPS coordinates are delivered to FlexDanmark every night. FlexDanmark has a DW where the cleaned GPS data is represented and integrated with other data such as weather data (driving speeds are related to the weather and are, e.g., lower when there is snow). The ETL procedure is implemented in Python. Among other things,

[4] http://flexdanmark.dk. One of the authors (Ove Andersen) is employed by FlexDanmark.

the ETL procedure has to transform between different coordinate systems, do map matching to roads (GPS coordinates can be quite imprecise), do spatial matching to the closest weather station, do spatial matching to municipalities and zip code areas, and finally load the DW. The latter is done by means of `pygrametl`.

The trips handled by FlexDanmark are paid by or subsidized by public funds. To be able to analyze how money is spent, another DW at FlexDanmark therefore holds data about payments for the trips, the taxi companies providing the vehicles, customers, etc. This DW integrates data from many different source systems and has some interesting challenges since payment details (i.e., facts in the DW) about already processed trips can be updated, but the history of these updates must by tracked in a similar way to how dimension changes are tracked for a type-2 slowly changing dimension. Further, new sources and dimensions are sometimes added. The ETL procedure for the DW is also implemented in Python, but FlexDanmark has created a framework that by means of *templates can generate Python code* incl. `pygrametl` objects based on metadata parameters. Thus, FlexDanmark can easily and efficiently generate ETL code that handles parallelism, versioning of facts, etc. when new sources and/or dimensions are added.

FlexDanmark's reasons for using code-based, programmatic ETL are manifold. FlexDanmark's DWs are rather advanced since they handle GPS data and versioned facts, respectively. To implement ETL produres for these things in traditional GUI-based tools was found to be hard. FlexDanmark did in fact try to implement the map matching in a widely used commercial ETL tool, but found it hard to accomplish this task. In contrast, it was found to be quite easy to do programmatically in Python where existing libraries easily could be used and also easily replaced with others when needed. Programmatic ETL does thus give FlexDanmark bigger flexibility. Yet another aspect is pricing since FlexDanmark is publicly funded. Here programmatic ETL building on free, open source software such as Python and `pygrametl` is desirable. It was also considered to use a free, open source GUI-based ETL tool, but after comparing a few programmatic solutions with their counterparts in the GUI-based tool, it was found to be faster and more flexible to code the ETL procedures.

In most cases where `pygrametl` is used, we are not aware for what since users are not obliged to register in any way. A few `pygrametl` users have, however, told us in private communication how they use `pygrametl`. We thus know that `pygrametl` is used in production systems from a wide variety of domains including health, advertising, real estate, public administration, and sales.

Based on the feedback we have received, we have so far not been able to extract guidelines or principles for how best to design programmatic ETL processes. The programming involved can vary from very little for a small proof of concept to a significant amount for a multi-source advanced DW. The principles to apply should thus probably be those already applied in the organization. Reusing existing and known development principles also reduces the time required to learn `pygrametl` as users only have to get to know a new Python library, but not new development principles. `pygrametl` does not make any strict assumptions about how the program should be organized (for example, it is *not* required that dimension data is processed before fact data). `pygrametl` is designed as a library where the user can make objects for the dimension tables or fact tables she wishes to operate on. `pygrametl` will then do all insertions into and updates of the underlying database while the user can focus on and structure the surrounding logic as she pleases.

12 Experiences from Open-Sourcing **`pygrametl`**

In this section, we describe our experiences with open-sourcing `pygrametl`.

When the first paper about `pygrametl` was published, we also made the source code available for download from our department's web page. From logs, we could see that there were some downloads and we also received some questions and comments, but not too many. Later, the source code was moved to Google Code and the received attention increased. When Google Code was taken out of service, the code was moved to GitHub[5] where we got a lot more attention. The lesson we learned from this is that it is very important to publish source code at a well-known place where people are used to look for source code. In fact, before we went to GitHub, others had already created unofficial and unmaintained repositories with the `pygrametl` code outside our control. Anyone is of course free to take, modify, and use the code as they please, but we prefer to be in control of the repository where people get the code to ensure availability of new releases. Along the same lines, we also learned that it is important to make it easy for users to install the library. For a Python library as `pygrametl`, it thus important to be available on the Python Package Index (PyPI[6]). Again we experienced that unofficial and unmaintained copies were created before we published our official version. With `pygrametl` on PyPI, installation of `pygrametl` is as simple as using the command `pip install pygrametl`.

Another lesson learned – although not very surprising – is that good documentation is needed. By making a Beginner's Guide and examples available online[7], we have reduced the number of (repeated) questions to answer via mail, but also made it easy to get started with `pygrametl`. It is also important to describe early what the tool is intended to do. For example, we have now and then received questions about how to do general data movement from one platform to another, but that is not what `pygrametl` is intended for. Instead, the focus for `pygrametl` is to do ETL for *dimensional* DWs.

Finally, we have also received very good help from users. They have found – and improved – performance bottlenecks as well as generalized and improved functionality of `pygrametl`.

[5] http://chrthomsen.github.io/pygrametl/.
[6] http://pypi.python.org/pypi.
[7] http://chrthomsen.github.io/pygrametl/doc/index.html.

`pygrametl` is published under a BSD license. We have chosen this license since it is a very permissive license. The BSD license allows proprietary use of the licensed code and has no requirements about making derivative work publicly available. In principle, there is thus a risk that users could improve the `pygrametl` source code without sharing the improvements with us. However, we prefer to give users freedom in deciding how to use the `pygrametl` code and, as described above, we do get suggestions for code improvements from users.

13 Conclusion and Future Work

We have presented `pygrametl` which is a programming framework for ETL programming. We challenge the conviction that ETL development is always best done in a graphical tool. We propose to also let the ETL *developers* (that typically are dedicated experts) do ETL *programming* by writing code. Instead of "drawing" the entire program, the developers can concisely and precisely express the processing in code. To make this easy, `pygrametl` provides commonly used functionality such as data source access and filling of dimensions and fact tables. In particular, we emphasize how easy it is to fill snowflaked and slowly changing dimensions. A single method call will do and `pygrametl` takes care of all needed lookups and insertions.

Our experiments have shown that ETL development with `pygrametl` is indeed efficient and effective. `pygrametl`'s flexible support of fact and dimension tables makes it easy to fill the DW and the programmer can concentrate on the needed operations on the data where (s)he benefits from the power and expressiveness of a real programming language to achieve high productivity.

`pygrametl` is available as open source and is used in proofs of concepts as well as production systems from a variety of different domains. We have learned the importance of publishing code at well-known places such as GitHub and the joy of users contributing improvements. When code is added or changed, we try hard to ensure that existing code does not break. For this, a future focus area is to automate testing much more than today. For future major releases, it can also be considered to introduce a new API with fewer classes, but the same or more functionality. The current class hierarchy to some degree reflects that new functionality has been added along the way when someone needed it. The way to load rows (plain SQL INSERTs, batched INSERTs, or by bulkloading) is now defined by the individual classes for tables. A more general approach could be by composition of loader classes into the classes for handling dimensions and fact tables. It would also be interesting to investigate how to allow generation of specialized code for the task at hand by using templating where the user can select features to enable such as bulkloading. This could potentially give big performance advantages. A strength of `pygrametl` is the easy integration with other Python projects. More integration with relevant projects such as data sources for Pandas[8] would also be beneficial.

[8] http://pandas.pydata.org/.

Appendix A Data Warehouse Concepts

This appendix offers a very short introduction to concepts and terms used in the chapter. More details and explanations can be found in the literature [7,8,27]. In a *data warehouse* (DW), data from an organization's different operational systems is stored in a way that supports analysis (rather than the daily operations which are supported by the operational systems). An *Extract-Transform-Load* (ETL) process extracts data from the source systems, transforms the data (to make it fit into the DW and to cleanse it), and loads it into the DW. Data is divided into *facts* and *dimensions*. Facts represent the subjects of the desired analyses (e.g., sales) and have numerical *measures* (e.g., sales amount). Dimensions provide context and describe facts (Product, Store, and Time are, for example, relevant dimensions for sales). Dimensions are thus used for selection of data and grouping of data in analyses. Dimensions have *hierarchies* with levels (a Time dimension can, for example, have the hierarchy Day → Month → Quarter → Year). Each of the levels can also have a number of attributes.

When using a relational database to represent a DW, one can choose between two approaches for the schema design. In a *snowflake schema*, each level in a dimension is represented by a table and the tables have foreign keys to the following levels. The dimension tables are thus normalized. In a *star schema* there is only one table for each dimension. This table thus represents all levels and is denormalized. In both star schemas and snowflake schemas, the facts are represented by a *fact table* which has a foreign key for each dimension and a column for each measure. In a star schema, the foreign keys reference the dimension tables while they reference the tables for the lowest levels of the dimensions in a snowflake schema. The keys used in a dimension table should be integers not carrying any special meaning. Such keys are called *surrogate keys*.

Changes may happen in the represented world. It is thus necessary to be able to represent changes in dimensions. A dimension where changes are represented is called a *slowly changing dimension* (SCD). There are a number of different techniques for SCDs [8]. Here we will consider two of the most commonly used. For *type 1 SCDs*, changes are simply represented by overwriting old values in the dimension tables. If, for example, the size of a shop changes, the size attribute is updated. This can be problematic as old facts (e.g., facts about sales from the shop when it had the previous size) now refer to the updated dimension member such that history is not correctly represented. This problem is avoided with a *type 2 SCD* where a new *version* of the dimension member is created when there is a change. In other words, for type 2 SCDs, changes result in new rows in the dimension tables. In a type 2 SCD, there are often attributes called something like ValidFrom, ValidTo, MostCurrentVersion, and VersionNumber to provide information about the represented versions.

References

1. Beyer, M.A., Thoo, E., Selvage, M.Y., Zaidi, E.: Gartner Magic Quadrant for Data Integration Tools (2017)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the OSDI, pp. 137–150 (2004). https://doi.org/10.1145/1327452.1327492
3. Django. djangoproject.com/. Accessed 13 Oct 2017
4. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Text-based modeling. In: Proceedings of ATEM (2007)
5. IBM InfoSphere DataStage. https://www.ibm.com/ms-en/marketplace/datastage. Accessed 13 Oct 2017
6. Informatica. informatica.com. Accessed 13 Oct 2017
7. Jensen, C.S., Pedersen, T.B., Thomsen, C.: Multidimensional Databases and Data Warehousing. Morgan and Claypool, San Rafael (2010). https://doi.org/10.2200/S00299ED1V01Y201009DTM009
8. Kimball, R., Ross, M.: The Data Warehouse Toolkit, 2nd edn. Wiley, New York (2002)
9. Liu, X., Thomsen, C., Pedersen, T.B.: ETLMR: a highly scalable dimensional ETL framework based on MapReduce. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 96–111. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23544-3_8
10. Microsoft SQL Server Integration Services. https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services. Accessed 13 Oct 2017
11. Oracle Data Integrator. http://www.oracle.com/technetwork/middleware/data-integrator/overview/index.html. Accessed 13 Oct 2017
12. Pentaho Data Integration - Kettle. http://kettle.pentaho.org. Accessed 13 Oct 2017
13. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM **38**(6), 33–44 (1995). https://doi.org/10.1145/203241.203251
14. PostgreSQL. `postgresql.org`. Accessed 13 Oct 2017
15. Psycopg. http://initd.org/psycopg/. Accessed 13 Oct 2017
16. Python. `python.org`. Accessed 13 Oct 2017
17. Ruby on Rails. `rubyonrails.org/`. Accessed 13 Oct 2017
18. SAP Data Services. https://www.sap.com/products/data-services.html. Accessed 13 Oct 2017
19. Scriptella. `scriptella.org`. Accessed 13 Oct 2017
20. Simitsis, A., Vassiliadis, P., Terrovitis, M., Skiadopoulos, S.: Graph-based modeling of ETL activities with multi-level transformations and updates. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 43–52. Springer, Heidelberg (2005). https://doi.org/10.1007/11546849_5
21. Thomsen, C., Pedersen, T.B.: Building a web warehouse for accessibility data. In: Proceedings of DOLAP (2006). https://doi.org/10.1145/1183512.1183522
22. Thomsen, C., Pedersen, T.B.: A survey of open source tools for business intelligence. IJDWM **5**(3), 56–75 (2009). https://doi.org/10.4018/jdwm.2009070103
23. Thomsen, C., Pedersen, T.B.: pygrametl: a powerful programming framework for extract-transform-load programmers. In: Proceedings of DOLAP, pp. 49–56 (2009). https://doi.org/10.1145/2064676.2064684
24. Thomsen, C., Pedersen, T.B.: pygrametl: a powerful programming framework for extract-transform-load programmers. DBTR-25, Aalborg University (2009). www.cs.aau.dk/DBTR

25. Thomsen, C., Pedersen, T.B.: Easy and effective parallel programmable ETL. In: Proceedings of DOLAP, pp. 37–44 (2011)
26. Trujillo, J., Luján-Mora, S.: A UML based approach for modeling ETL processes in data warehouses. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 307–320. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39648-2_25
27. Vaisman, A., Zimanyi, E.: Data Warehouse Systems: Design and Implementation. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54655-6
28. Vassiliadis, P.: A survey of extract-transform-load technology. IJDWM **5**(3), 1–27 (2009). https://doi.org/10.4018/jdwm.2009070101

Temporal Data Management – An Overview

Michael H. Böhlen[1], Anton Dignös[2], Johann Gamper[2](✉), and Christian S. Jensen[3]

[1] University of Zurich, Zurich, Switzerland
[2] Free University of Bozen-Bolzano, Bolzano, Italy
gamper@inf.unibz.it
[3] Aalborg University, Aalborg, Denmark

Abstract. Despite the ubiquity of temporal data and considerable research on the effective and efficient processing of such data, database systems largely remain designed for processing the current state of some modeled reality. More recently, we have seen an increasing interest in the processing of temporal data that captures multiple states of reality. The SQL:2011 standard incorporates some temporal support, and commercial DBMSs have started to offer temporal functionality in a step-by-step manner, such as the representation of temporal intervals, temporal primary and foreign keys, and the support for so-called time-travel queries that enable access to past states.

This tutorial gives an overview of state-of-the-art research results and technologies for storing, managing, and processing temporal data in relational database management systems. Following an introduction that offers a historical perspective, we provide an overview of basic temporal database concepts. Then we survey the state-of-the-art in temporal database research, followed by a coverage of the support for temporal data in the current SQL standard and the extent to which the temporal aspects of the standard are supported by existing systems. The tutorial ends by covering a recently proposed framework that provides comprehensive support for processing temporal data and that has been implemented in PostgreSQL.

1 Introduction

The capture and processing of temporal data in database management systems (DBMS) has been an active research area since databases were invented. In the temporal database research history, four overlapping phases can be distinguished. First, the *concept development* phase (1956–1985) concentrated on the study of multiple kinds of time and temporal aspects of data and on temporal conceptual modeling. The following phase was dedicated to the *design of query languages* (1978–1994), including relational and object-oriented temporal query languages. Then the focus shifted to *implementation aspects* (1988–present), emphasis being on storage structures, algorithms for specific operators, and temporal indices.

© Springer International Publishing AG, part of Springer Nature 2018
E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 51–83, 2018.
https://doi.org/10.1007/978-3-319-96655-7_3

Finally, the *consolidation* phase (1993–present) produced a consensus glossary of temporal database concepts [47], a query language test suite [36], and TSQL2 [81] as an effort towards standardization of a temporal extension to SQL.

A number of events and activities involving the temporal database community have impacted the evolution of temporal database research significantly. The 1987 *IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems* [78] covered a wide range of topics, including requirements for temporal data models and information systems, temporal query languages, versioning, implementation techniques, temporal logic, constraints, and relations to natural language. The 1993 *ARPA/NSF International Workshop on an Infrastructure for Temporal Databases* [80] aimed at consolidating different temporal data models and query languages. In the same year, the collection *Temporal Databases: Theory, Design, and Implementation* [87] was published, which describes primarily a number of data models and query languages. Another influential book, *The TSQL2 Temporal Query Language* [81], was published in 1995. By leveraging many of the concepts that were proposed in previous research, TSQL2 aimed to be a consensus data model and query language. At the same time, a project with the ambition of creating a new part of the SQL standard dedicated to the support of temporal data started. Several proposals were submitted, e.g., [83], but were eventually not successful. The proposals proved controversial, and they were unable to achieve support from major database vendors. The last notable event dedicated to temporal database research was the 1997 *Dagstuhl Seminar on Temporal Databases* [37] that had as its aim to discuss future directions for temporal database management both in research as well as in system development.

The last several years have seen a renewed interest in temporal database management in both academia and industry. This interest is driven in part by the needs of new and emerging applications, such as versioning of web documents [33], social network analysis and communication networks [65,76], management of normative texts [46], air traffic monitoring and patient care [7], video surveillance [71], sales analysis [70], financial market analysis [45], and data warehousing and analytics [82], to name a few. These and other applications produce huge amounts of temporal data, including time series and streaming data, which are special forms of temporal data. It has been recognized [5] that analyses of historical data can reveal valuable information that cannot be found in only the current snapshot.

This tutorial provides an overview of temporal data management concepts and techniques, covering both research results and commercial database management systems. In Sect. 2, we summarize the most important concepts developed in temporal database research. In Sect. 3, we provide a brief overview of the state-of-the-art in temporal database research. Section 4 describes the most important temporal features of the SQL:2011 standard that introduces temporal support into SQL. In Sect. 5, we provide a brief overview of the support for the temporal SQL standard in commercial database management systems. Finally, in Sect. 6, we describe a recent framework that provides a comprehensive and native solution for processing temporal queries in relational database management systems.

2 Basic Concepts of Temporal Databases

In this section, we briefly summarize important concepts that have been developed in temporal database research.

2.1 Time Domain and Structure

The *time domain* (or ontology) specifies the basic building blocks of time [66]. It is generally modeled as a set of *time instants* (or points) with an imposed partial order, e.g., $(\mathbb{N}, <)$. Additional axioms impose more structure on the time domain, yielding more refined time domains. *Linear time* advances from past to future in a step-by-step fashion. This model of time is mainly used in the database area. In contrast, AI applications often used a *branching time* model, which has a tree-like structure, allowing for possible futures. Time is linear from the past to now, where it divides into several time lines; along any future path, additional branches may exist. This yields a tree-like structure rooted at now. *Now* marks the current time point and is constantly moving forward [32]. The time domain can be *bounded* in the past and/or in the future, i.e., a first and/or last time instant exists; otherwise, it is called *unbounded*.

The time domain can be dense, discrete, or continuous. In a *discrete* time domain, time instants are non-decomposable units of time with a positive duration, called *chronons* [31]. A chronon is the smallest duration of time that can be represented. This time model is isomorphic to the natural numbers. In contrast, in a *dense* time domain, between any two instants of time, there exists another instant; this model is isomorphic to the rational numbers. Finally, *continuous* time is dense and does not allow "gaps" between consecutive time instants. Time instants are durationless. The continuous time model is isomorphic to the real numbers.

While humans perceive time as continuous, a *discrete linear time model* is generally used in temporal databases for several practical reasons, e.g., measures of time are generally reported in terms of chronons, natural language references are compatible with chronons, and any practical implementation needs a discrete encoding of time. A limitation of a discrete time model is, for example, the inability to represent continuous phenomena [40].

A *time granularity* is a partitioning of the time domain into a finite set of segments, called *granules*, providing a particular discrete image of a (possibly continuous) timeline [9,10]. The main aim of granularities is to support user-friendly representations of time. For instance, birth dates are typically measured at the granularity of days, business appointments at the granularity of hours, and train schedules at the granularity of minutes. Multiple granularities are needed in many real-world applications.

2.2 Temporal Data Models

A *data model* is defined as $M = (DS, QL)$, where DS is a set of data structures and QL is a language for querying instances of the data structures. For instance, the relational data model is composed of relations and, e.g., SQL. Many extensions of the relational data model to support time have been proposed in past research, e.g., IXSQL [63], TSQL2 [81], ATSQL [16] and SQL/TP [92]. When designing a temporal data model [49], several aspects have to be considered, such as

– different time dimensions, or temporal aspects,
– different timestamp types, and
– different forms of timestamping.

Time Dimensions. Different temporal aspects of data are of interest. Valid time and transaction time are the two aspects that have attracted the most attention by far in database research; other temporal aspects include publication time, efficacy time, assertion time, etc.

Valid time [54] is the time when a *fact was/is/will be true in the modeled reality*, e.g., John was hired from October 1, 2014 to May 31, 2016. Valid time captures the time-varying states of the modeled reality and is provided by the application or user. All facts have a valid time by definition, and it exists independently of whether the fact is recorded in a database or not. Valid time can be bounded or unbounded.

Transaction time [53] is the time when a *fact is current/present in the database* as stored data, e.g., the fact "John was hired from October 1, 2014 to May 31, 2016" was stored in the database on October 5, 2014, and was deleted on March 31, 2015. Transaction time captures the time-varying states of the database, and it is supplied automatically by the DBMS. Transaction time has a duration from the insertion of a fact to its deletion, with multiple insertions and deletions being possible for the same fact. Deletions of facts are purely logical: the fact remains in the database, but ceases to be part of the database's current state. Transaction time is always bounded on both ends. It starts when the database is created (nothing was stored before), and it does not extend past now (it is not known which facts are current in the future). Transaction time is the basis for supporting accountability and traceability requirements, e.g., in financial, medical, or legal applications.

A data model can support none, one, two, or more time dimensions. A so-called *snapshot* data model provides no support for time dimensions and records only a single snapshot of the modeled reality. A *valid time* data model supports only valid time, a *transaction time* data model only transaction time. A *bitemporal* data model supports both valid time and transaction time.

Timestamp Types. A *timestamp* is a time value that is associated with an attribute value (*attribute (value) timestamping*) or a tuple (*tuple timestamping*) in a database and captures some temporal aspect, e.g., valid time or transaction time. It can be represented as one or more attributes or columns of a relation. Three different *types of timestamps* [50,51,62] have received particular attention:

– time points,
– time intervals, and
– temporal elements.

To illustrate different types of timestamps, we consider a car rental company, where customers, identified by a `CustID`, rent cars, identified by a `CarID`. Assume the following rentals during May 1997:

– On 3rd of May, customer `Sue` rents car `C1` for three days.
– On 5th of May, customer `Tim` rents car `C2` for 3 days.
– From 9th to 12th of May, customer `Tim` rents car `C1`.
– From 19th to 20th of May, and again from 21st to 22nd of May, customer `Tim` rents car `C2`.

These rentals are stored in a relation `Rental`, which is illustrated in Fig. 1.

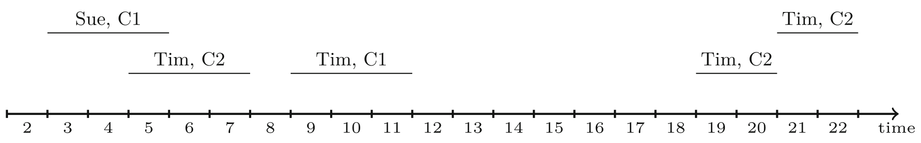


Fig. 1. Relation `Rental`.

Time Points. In a point-based data model, tuples or attribute values are timestamped with a *time point* (or instant) (cf. Fig. 2(a)). This is the most basic and simple data model. Timestamps are atomic values and can be compared easily with $=, \neq, <, >, \geq, \leq$. Multiple tuples are used if a fact is valid at several time points, e.g., four tuples for the two consecutive rentals from time 19 to time 22. Additional attributes are required to restore the original relation. In the `Rental` relation in Fig. 2(a), the `SeqNo` attribute is used to group tuples that constitute a rental. Without this attribute, it would be impossible to restore the two consecutive 2-day rentals, as they could be restored, e.g., as a single 4-day rental or four 1-day rentals. The point-based model is simple and provides an abstract view of a database, which makes it popular for theoretical studies, but inappropriate for physical implementation.

Time Intervals. In an interval-based data model, each tuple or attribute is timestamped with a *time interval, or period* (cf. Fig. 2(b)). Timestamps can be compared using Allen's 13 basic interval relationships (before, meets, during, etc.) [4], which is more convenient than comparing the endpoints of the intervals. Multiple tuples are used if a fact is valid during disjoint time intervals. The `SeqNo` attribute is not needed to distinguish among different tuples. This is the most popular model from an implementation perspective. Interval timestamps are *not closed* under set operations, e.g., subtracting the interval $[5, 7]$ from the interval $[1, 9]$ gives the set of intervals $\{[1, 4], [8, 9]\}$, not a single interval.

Rental

SeqNo	CustID	CarID	T
1	Sue	C1	3
1	Sue	C1	4
1	Sue	C1	5
2	Tim	C2	5
2	Tim	C2	6
2	Tim	C2	7
3	Tim	C1	9
3	Tim	C1	10
3	Tim	C1	11
3	Tim	C1	12
4	Tim	C2	19
4	Tim	C2	20
5	Tim	C2	21
5	Tim	C2	22

(a) Point-based model

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7]
Tim	C1	[9,12]
Tim	C2	[19,20]
Tim	C2	[21,22]

(b) Strong interval-based model

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7]
Tim	C1	[9,12]
Tim	C2	[19,22]

(c) Weak interval-based model

Fig. 2. Point- and interval-based data models for `Rental` relation.

Temporal Elements. In data models with temporal elements, each tuple or attribute is timestamped with a finite union of intervals, called a *temporal element* [38,39] (cf. Fig. 3). The full history of a fact is stored in a single tuple. For instance, the second tuple represents the fact that Tim rents car C1 from time 5 to 7 and from time 19 to 22. Temporal elements support only a point-based semantics, and an additional attribute would be necessary to distinguish between the two consecutive 2-day rentals (see also the discussion below).

Rental

CustID	CarID	T
Sue	C1	[3,5]
Tim	C2	[5,7] ∪ [19,22]
Tim	C1	[9,12]

Fig. 3. Data model with temporal elements.

Point-Based and Interval-Based Semantics. From a semantic viewpoint, two different types of models can be distinguished: models with point-based semantics and models with interval-based semantics. This distinction is orthogonal to the choice of the timestamp (i.e., time points, time intervals, or temporal elements) and focuses on the meaning of the timestamps. For instance, relation `Rental` in Fig. 2(a) uses time points as timestamps, but adopts an interval-based semantics, as information on the rental periods is preserved by using the additional `SeqNo` attribute. Similarly, a relation that uses interval timestamps may adopt either a point-based semantics or an interval-based semantics. The corresponding models are referred to as, respectively, weak interval-based model and strong interval-based model – see Figs. 2(b) and (c).

In the *weak interval-based data model*, intervals are only used as a compact and convenient representation of contiguous sets of time points. For instance, although (syntactically) different, the two relations in Figs. 2(b) and (c) are considered equivalent under point-based semantics since they are snapshot equivalent [48], i.e., they contain the same snapshots. More formally, let r and s be two temporal relations, Ω^T be the time domain, and $\tau_t(\mathsf{r})$ be the timeslice operator [52] with t being a time instant. The relations r and s are *snapshot equivalent* if and only if

$$\forall t \in \Omega^T : \tau_t(\mathsf{r}) \equiv \tau_t(\mathsf{s})$$

For the weak interval-based model, an important operation is *coalescing* [2,19,99]. Coalescing is the process of merging adjacent and overlapping interval timestamped tuples with identical nontemporal attribute values into tuples with maximal time intervals. For instance, the relation in Fig. 2(c) is the result of coalescing the relation in Fig. 2(b). Without an additional `SeqNo` attribute, the two consecutive 2-day rentals disappear: they are merged into a single 4-day rental.

In the *strong interval-based data model*, intervals are *atomic* units that carry meaning (and not just sets of time points). Thus, strong interval-based data models are more expressive. They can distinguish between a 4-day rental and two consecutive 2-day rentals without requiring an additional attribute. The relation in Fig. 2(b) is the appropriate representation of the `Rental` relation in our example since two 2-day rentals and one 4-day rental might impose different fees.

Timestamping. Timestamping denotes the association of a data element in a relation with a time value. In the above examples, we used *tuple timestamping*, which associates each tuple with a time value such as a time point, a time interval, or a temporal element.

In *attribute (value) timestamping*, each attribute value in a relation is associated with a timestamp (cf. Fig. 4). Relations are *grouped* by an attribute, and *all information* about that attribute (or real-world object) is captured in a single tuple. Information about other objects is spread across several tuples. In Fig. 4(a), all information about a customer is in one tuple, while the information about cars is spread across several tuples. A single tuple may record multiple facts. For instance, the second tuple records four different rentals involving customer `Tim` and the cars `C1` and `C2`. Different groupings of the information into tuples are possible. Figure 4(b) shows the same relation grouped on `CarID`. The two relations are snapshot-equivalent. Data models using attribute value timestamping are non-first-normal-form data models.

2.3 Query Language Semantics

The querying capabilities of temporal DBMSs can be partitioned into three modes [16,82,84]: nonsequenced, current, and sequenced semantics.

Rental

SeqNo	CustID	CarID
[3,5] 1	[3,5] Sue	[3,5] C1
[5,7] 2	[5,7] ∪ [9,12] ∪ [19,22] Tim	[5,7] ∪ [19,22] C2
[9,12] 3		[9,12] C1
[19,20] 4		
[21,22] 5		

(a) Grouped by `CustID`

Rental

SeqNo	CustID	CarID
[3,5] 1	[3,5] Sue	[3,5] ∪ [9,12] C1
[9,12] 3	[9,12] Tim	
[5,7] 2	[5,7] ∪ [19,22] Tim	[5,7] ∪ [19,22] C2
[19,20] 4		
[21,22] 5		

(b) Grouped by `CarID`

Fig. 4. Attribute value timestamping.

The *nonsequenced semantics* [18] is time agnostic, that is, the DBMS does not enforce any specific meaning on the timestamps, and applications must explicitly specify how to process the temporal information. The support for the nonsequenced semantics in DBMSs is limited to extending SQL with new data types, predicates, and functions. Predicates such as `OVERLAPS`, `BEFORE`, and `CONTAINS` are part of the SQL:2011 standard. Another approach to specify temporal relationships are to use the operators of temporal logic, which target the reasoning across different database states [23]. Nonsequenced semantics is the most flexible and expressive semantics since applications handle timestamps like all other attributes without any implicit meaning being enforced.

The *current semantics* [6,17] performs query processing on the database snapshot at the current time and can be realized by restricting the data to the current time. Current semantics is present in the SQL:2011 standard, where standard SQL queries over transaction time tables (in SQL:2011 called system-versioned tables) are evaluated on the current snapshot [58]. As a simple extension to current semantics, so-called time travel queries allow to specify any snapshot of interest. The integration of current semantics into a database engine is usually done with the help of selection operations.

The *sequenced semantics* [15,44] of a temporal query is defined by viewing a temporal database as a sequence of snapshot databases and evaluating the query at each of these snapshots. This concept is known as *snapshot reducibility* [63,86]. More formally, let $\mathsf{r}_1, \ldots, \mathsf{r}_n$ be temporal relations, ψ^T be an n-ary temporal operator, ψ be the corresponding nontemporal operator, Ω^T be the time domain, and $\tau_t(\mathsf{r})$ be the timeslice operator [52] with t being a time instant. Operator ψ^T is *snapshot reducible* to ψ if and only if

$$\forall t \in \Omega^T : \tau_t(\psi^T(\mathsf{r}_1, \ldots, \mathsf{r}_n)) \equiv \psi(\tau_t(\mathsf{r}_1), \ldots, \tau_t(\mathsf{r}_n))$$

Snapshot reducibility provides a minimum requirement for sequenced semantics by constraining the result of a temporal query to be consistent with the snapshots that are obtained by computing the corresponding nontemporal query on each snapshot of the temporal database. This provides a clear semantics for theoretical studies, but a practical implementation needs additional constraints.

First, snapshot reducibility does not constrain the coalescing of consecutive tuples with identical nontemporal attribute values. For instance, the two relations in Figs. 2(b) and (c) are snapshot equivalent, yet they store different information. *Change preservation* [27,30] is a way to determine the time intervals of the result tuples, and thus control the coalescing of tuples. A new time interval is created when the argument tuples that contribute to a result tuple change (i.e., have different lineage or provenance) [20,24], yielding maximal time intervals for the result tuples over which the argument relations are constant.

Second, snapshot reducibility does not allow temporal operators to reference the timestamps of the argument relations since the intervals are removed by the timeslice operator. For example, computing the average duration of projects at each point is not possible. This problem can be tackled by propagating the original timestamp as additional attribute to relational algebra operators, yielding a concept known as *extended snapshot reducibility* [15].

Finally, sometimes attribute values need to be changed when the timestamp intervals of tuples change. For instance, if a project budget is $100,000$ for a period of two years, then the corresponding budget for one year should be $50,000$ (assuming a uniform distribution). This concept is called *scaling* of attribute values [11,28].

3 State-of-the-Art

In this section, we discuss the state-of-the-art in temporal database research, focusing on data models, SQL-based query languages, and evaluation algorithms for query processing.

3.1 Data Models and SQL-Based Query Languages

To make the formulation of temporal queries more convenient, various temporal query languages [14,87] have been proposed. The earliest and simplest approach to add temporal support to SQL-based query languages was to introduce new data types with associated predicates and functions that were strongly influenced by Allen's interval relationships [4]. While this approach facilitates the formulation of some temporal queries, it falls short in the extent to which it makes it easier to formulate temporal queries. Therefore, new constructs were added to SQL with the goal of expressing temporal queries more easily. A representative query language following this approach is TSQL2 [81], which uses so-called syntactic defaults to facilitate query formulation. Challenges with this type of approach include to be "complete" in enabling easy formulation of temporal queries and to avoid unintentional interactions between the extensions.

A more systematic approach was adopted in IXSQL [25,63], which normalizes interval timestamped tuples for query processing and works as follows: (i) a function *unfold* transforms an interval timestamped relation into a point timestamped relation by splitting each tuple into a set of point timestamped tuples; (ii) the corresponding nontemporal operation is applied to the normalized relation; (iii) a function *fold* collapses value-equivalent tuples over consecutive time points into interval timestamped tuples over maximal time intervals. The approach is conceptually simple, but timestamp normalization does not respect lineage, and no efficient implementation exists.

SQL/TP [92,93] is an approach that is based on a point-based data model: a temporal relation is modeled as a sequence of nontemporal relations (or snapshots). To evaluate a temporal query, the corresponding nontemporal query is evaluated at each snapshot. For an efficient evaluation, an interval encoding of point timestamped relations was proposed together with a *normalization* function. The normalization splits overlapping value-equivalent input tuples into tuples with equal or disjoint timestamps, on which the corresponding nontemporal SQL statements are executed. SQL/TP considers neither lineage nor extended snapshot reducibility, which are not relevant for point timestamped relations. Moreover, the normalization function is not applicable to joins, outer joins, and anti joins.

Agesen et al. [1] extend normalization to bitemporal relations by means of a *split* operator. This operator splits input tuples that are value-equivalent over nontemporal attributes into tuples over smaller, yet maximal timestamps such that the new timestamps are either equal or disjoint. The split operator supports temporal aggregation and difference in now-relative bitemporal databases.

ATSQL [16] offers a systematic way to construct temporal SQL queries from nontemporal SQL queries. The main idea is to first formulate the nontemporal query and then prepend to this query a so-called *statement modifier* that specifies the intended semantics of the query evaluation, such as sequenced or nonsequenced semantics.

The *temporal alignment* approach [27,30] is a solution for computing temporal queries over interval timestamped relations using sequenced semantics. The key idea is to first adjust the timestamps of the input tuples and then to execute the corresponding nontemporal operator to obtain the intended result. While the adjustment of timestamps is similar to the normalization in SQL/TP [92], the temporal alignment approach is comprehensive and offers snapshot reducibility, extended snapshot reducibility, and attribute value scaling for all operators of a relational algebra. This approach provides a native database implementation for temporal query languages with sequenced semantics, such as ATSQL. More details are provided in Sect. 6.

The scaling of attribute values in response to the adjustment of interval timestamps has received little attention. Böhlen et al. [11] propose three different attribute characteristics: *constant* attributes that never change value during query processing, *malleable* attributes that require adjustment of the value when the timestamp changes, and *atomic* attributes that become undefined (invalid) when the timestamp changes. For malleable attributes, an adjustment function is proposed. Terenziani and Snodgrass [91] distinguish between *atelic* facts that are valid for each point in time and *telic* facts that are only valid for one specific interval. That work focuses on the semantics of facts recorded in a database and proposes a three-sorted relational model (atelic, telic, nontemporal). Dignös et al. [28] show that scaling of attributes values is possible during query processing.

The focus of Dyreson et al. [34,35] is to provide a uniform framework for the evaluation of queries under different temporal semantics, including the two extremes of sequenced and nonsequenced semantics. Additional semantics can be realized in this framework, such as context, periodic, and preceding semantics. The framework uses lineage to track tuples through operations. The work is primarily at the conceptual level, the main goals being to unify and reconcile different temporal semantics.

3.2 Query Processing Algorithms

In terms of query processing, various query algorithms for selected operators have been studied, primarily for temporal aggregations (e.g., [13,57,67,95,97]) and temporal joins (e.g., [42,85,98]) over interval timestamped relations, which are arguably the most important and expensive operations.

Processing Temporal Aggregations. Aggregate functions enable the summarization of large volumes of data, and they were introduced in early relational DBMSs such as System R and INGRES. Various forms of temporal aggregation have been proposed since then. They differ in how the data is grouped along the time dimension [41]. In *instantaneous temporal aggregation*, an aggregate function is conceptually computed at each time point, followed by a subsequent coalescing step to merge contiguous tuples with the same aggregate value into a single interval timestamped tuple. *Moving-window temporal aggregation*, also termed cumulative temporal aggregation, works similarly, except that an aggregate at a time point is computed over all tuples that occur within a user-specified window. Finally, in *span temporal aggregation*, the aggregates are computed over sets of tuples that overlap with fixed time intervals specified by the user.

The earliest proposal aimed at the efficient processing of instantaneous temporal aggregates is by Tuma [94]. Following Tuma's work, research focused on the development of efficient main-memory algorithms for the evaluation of instantaneous temporal aggregates as the most important form of temporal aggregation. Key works in this direction include the aggregation tree algorithm [57] and the balanced tree algorithm [67].

With the diffusion of data warehouses and OLAP, disk-based index structures for incremental computation and maintenance of temporal aggregates were investigated. Notable works include the SB-tree by Yang and Widom [95], which was extended to the MVSB-tree by Zhang et al. [97] to include nontemporal range predicates. The high memory requirements of the MVSB-tree were addressed by Tao et al. [88], proposing two approximate solutions for temporal aggregation.

Vega Lopez et al. [61] formalized temporal aggregation in a unified framework that enables the comparison of the different forms of temporal aggregation based on various mechanisms for defining aggregation groups. In a similar vein, Böhlen et al. [13] propose a framework that generalizes existing forms of temporal aggregation by decoupling the partitioning of the time line from the specification of the aggregation groups.

The development of efficient temporal aggregation algorithms has recently received renewed interest. Kaufmann et al. [55,56] propose the *timeline index* to efficiently support query processing, including instantaneous temporal aggregation, in the main memory DBMS HANA. The timeline index is a general data structure that instead of intervals uses start and end points of the intervals. Query processing is performed by scanning sorted lists of endpoints. Piatov and Helmer [74] present a family of plane-sweeping algorithms that adopt the timeline index for other forms of temporal aggregation, such as aggregation over fixed intervals, sliding window aggregates, and MIN/MAX aggregates.

Temporal aggregation has been studied for different query languages and data models. Böhlen et al. [12] investigate how temporal aggregation is supported in different types of temporal extensions to SQL. Selected temporal aggregations are also found in non-relational query languages, such as XML, e.g., τXQuery [43].

Processing Temporal Joins. The overall efficiency of a query processor depends highly on its ability to evaluate joins efficiently, as joins occur frequently. Two classes of join algorithms can be distinguished: solutions that rely on indexing or secondary access paths, and solutions for ad-hoc join operations that operate on the original tables, but might take advantage of sorting the data.

Gao et al. [42] present a comprehensive and systematic study of join operations in temporal databases as of 2005, covering both semantics and implementation aspects. In addition to providing formal definitions of various join operations, the paper classifies existing evaluation algorithms along the following dimensions: nested-loop, partitioning, sort-merge, and index-based. The work includes also an experimental performance evaluation of 19 join algorithms.

Recently, a number of new studies on the efficient evaluation of temporal joins have been published. The *timeline index* by Kaufmann et al. [55,56] is a main memory index structure that supports also temporal joins where matching tuples must be overlapping.

The *overlap interval partition* join algorithm by Dignös et al. [29] partitions the input relations in such a way that the percentage of matching tuples in corresponding partitions is maximized. This yields a robust join algorithm that is not affected by the distribution of the data. The proposed partitioning works both in disk-based and main-memory settings.

The *lazy endpoint-based interval* join algorithm by Piatov et al. [75] adopts the timeline index. After creating a timeline index of the input relations, the two index structures are scanned in an interleaved fashion. Thereby, active tuples are managed by an in-memory hash map, called a *gapless hash map*, that is optimized for sequential reads of the entire map. Additionally, a lazy evaluation technique is used to minimize the number of scans of the active tuple map.

The *disjoint interval partitioning* join algorithm by Cafagna and Böhlen [22] first creates so-called *disjoint* partitions for each relation, where all tuples in a partition are temporally disjoint. To compute a temporal join, all outer partitions are then sort-merge-joined with each inner partition to produce the final result. Since tuples within a partition are disjoint, the algorithm is able to avoid expensive backtracking.

Bouros and Mamoulis [21] implement a forward-scan based *plane sweep algorithm* for temporal joins and provide two optimizations. The first optimization groups consecutive tuples such that join results can be produced in batches in order to avoid redundant comparisons. The second optimization extends the grouping with a bucket index to further reduce the number of comparisons. A major contribution of this work is a parallel evaluation strategy based on a domain-based partitioning of the input relations.

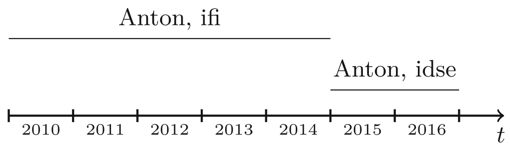
4 Temporal Support in the SQL:2011 Standard

This section summarizes the most important temporal features of the SQL:2011 standard, which is the first SQL standard with support for time.

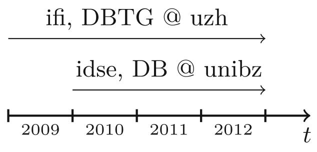
The ability to create and manipulate tables whose rows are associated with one or two temporal periods, representing valid and transaction time, is the key extension in SQL:2011 [58,96]. A core concept of this extension is the *specification of time periods* associated with tables, bundled with support for updates, deletions, and integrity constraints. The support for querying temporal relations is limited to simple range restrictions and predicates.

In the following discussion, we use two valid time relations, which are illustrated in Fig. 5 and initially contain the following data:

- an employee relation, named `Emp`, records that Anton was working in the `ifi` department from 2010 to 2014 and in the `idse` department from 2015 to 2016;
- a department relation, named `Dept`, contains descriptions of the `ifi` and `idse` departments.



(a) `Emp` relation



(b) `Dept` relation

Fig. 5. Graphical illustration of the `Emp` and `Dept` relations.

4.1 Creation of Tables with Time Periods

SQL:2011 adopts an interval-based data model with tuple timestamping. Rather than introducing a new data type for a time interval, a *time period specification* is added as metadata to the table schema. A period specification combines a physical *start time* attribute and an *end time* attribute to form a period. Periods are specified with the `PERIOD FOR` clause:

```
PERIOD FOR <PeriodName> (<StartTime>, <EndTime>)
```

Here, `<StartTime>` and `<EndTime>` are of type `DATE` or `TIMESTAMP` and together form a time period, which can be referred to by the name `<PeriodName>`. A period is by default a *closed-open* interval, where `StartTime` is included and `EndTime` is excluded[1].

By designing periods as metadata, a minimally invasive approach was chosen that achieves *backward compatibility*, i.e., old schemas, queries, and tools still work. If start and end time points are already stored in a table (which is often the case), a time period can be added without the need to modify the physical table schema. A new data type would require more substantial changes across the DBMS.

SQL:2011 supports valid time and transaction time, which are called, respectively, *application time* and *system time*. At most one application time and one system time can be specified for a table. Tables that have an application time period are called *application time period tables*, and tables with a system time period are called *system time period tables*. Tables that support both time dimensions are usually called bitemporal tables, although SQL:2011 does not provide an explicit name.

To facilitate reading and to be consistent with the research literature, we will use the widely adopted terms valid time for application time and transaction time for system time.

Valid Time Tables. Valid time tables store for each tuple the time interval when the tuple is true in the modeled reality. In our example, the schema of the `Emp` relation with a valid time attribute can be created as follows:

```
CREATE TABLE Emp (
  EName VARCHAR,
  EDept VARCHAR,
  EStart DATE,
  EEnd DATE,
  PERIOD FOR EPeriod (EStart, EEnd)
);
```

The `PERIOD FOR` clause specifies a valid time interval named `EPeriod`, which is built from the physical attributes `EStart` and `EEnd`. Notice that `EPeriod` is not

[1] To comply with the SQL:2011 standard, in this section we use closed-open intervals, whereas in the other sections we use closed-closed intervals.

a physical column of the table, but it is stored as metadata and can be used to refer to the interval. A table with this schema is shown in Fig. 6(a).

When *inserting* tuples, the user has to specify the valid time period in addition to the nontemporal attribute values of the tuples. For instance, the following statements insert two tuples indicating that Anton was working in the `ifi` department from 2010 to 2014 and in the `idse` department from 2015 to 2016:[2]

```
INSERT INTO Emp VALUES (Anton, ifi, 2010, 2015);
INSERT INTO Emp VALUES (Anton, idse, 2015, 2017);
```

Emp

EName	**EDept**	**EStart**	**EEnd**
Anton	ifi	2010	2015
Anton	idse	2015	2017

(a) Valid time table

Dept

DName	**DDesc**	**DStart**	**DEnd**
ifi	DBTG @ uzh	2009	9999
idse	DB @ unibz	2010	9999

(b) Transaction time table

Fig. 6. Tables with period timestamps.

Transaction Time Tables. In a transaction time table, each tuple stores an interval that records when the tuple was current in the database. Different from the valid time, the transaction time is set by the DBMS when a tuple is created, updated, or deleted; the user is not allowed to change the transaction time values. The following statement creates a transaction time table for the department relation:

```
CREATE TABLE Dept (
  DName VARCHAR,
  DDesc TEXT,
  DStart DATE GENERATED ALWAYS AS ROW START,
  DEnd DATE GENERATED ALWAYS AS ROW END,
  PERIOD FOR SYSTEM_TIME (DStart, DEnd)
) WITH SYSTEM VERSIONING;
```

The `GENERATED ALWAYS` clause specifies that the two attributes `DStart` and `DEnd` are generated by the system when a tuple is, respectively, inserted, deleted, or modified. The `PERIOD FOR` clause in combination with the `WITH SYSTEM VERSIONING` clause define a system-versioned table. While the attribute names `DStart` and `DEnd` are user-specified, the name of the transaction time attribute must be `SYSTEM_TIME`.

The following SQL statements insert two tuples in the system-versioned department relation as shown in Fig. 6(b):

```
INSERT INTO DEPT (DName, DDesc) VALUES (‘ifi’, ‘DBTG @ uzh’);
INSERT INTO DEPT (DName, DDesc) VALUES (‘idse’, ‘DB @ unibz’);
```

[2] To keep the examples simple, we use only the year, not complete dates or timestamps.

The user specifies only the nontemporal attributes, whereas the transaction time is added automatically by the system. The value of `DStart` is set to the current transaction time when the tuple/row is created. Hence, the first tuple was inserted in 2009 and the second in 2010. The value `9999` of the `DEnd` attribute is the highest possible timestamp value and indicates that the tuple is current in the database.

A transaction time table conceptually distinguishes between current tuples and historical tuples. A tuple is considered a *current tuple* if its timestamp contains the current time (aka *now*). All other tuples are called *historical tuples.* Historical tuples are never modified and form *immutable snapshots* of the past.

4.2 Modification of Tables

The SQL:2011 standard specifies the behavior of temporal tables in the case of updates and deletions, which is different for valid time tables and transaction time tables.

Valid Time Tables. Conventional update and delete operations work in the same way as for nontemporal tables. That is, both nontemporal and temporal attributes can be modified using the known SQL syntax. In addition, there is enhanced support for modifying tuples over *parts* of the associated time periods by using the `FOR PORTION OF` clause. In this case, overlapping tuples are automatically split or cut. Consider the following statement:

```
DELETE Emp
FOR PORTION OF EPeriod FROM DATE ‘2011’ TO DATE ‘2013’
WHERE EName = ‘Anton’;
```

This statement deletes a portion of the first tuple in the `Emp` relation in Fig. 7(a). As a consequence, the tuple is automatically split in two: one stating that Anton was employed at ifi in 2010, and the other that he was employed from 2013 to 2014, as shown in Fig. 7(b). Non-overlapping tuples are not affected.

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2015
Anton	idse	2015	2017

(a) Before DELETE statement

Emp

EName	EDept	EStart	EEnd
Anton	ifi	2010	2011
Anton	ifi	2013	2015
Anton	idse	2015	2017

(b) After DELETE statement

Fig. 7. Modifying a valid time table.

The behavior of the UPDATE operation is similar. For instance, the following statement would split the first tuple in Fig. 7(a) into three tuples:

```
UPDATE Emp
FOR PORTION OF EPeriod FROM DATE ‘2011’ TO DATE ‘2013’
SET EDept = ‘ai’
WHERE EName = ‘Anton’;
```

Transaction Time Tables. Any modification of a transaction time table operates only on the current tuples, and the user can only modify nontemporal attributes, not the timestamp attribute. The transaction time is automatically modified when nontemporal attributes of current tuples are modified. That is, if a current tuple is modified, a copy of that tuple is created with the end timestamp set to the current time. The tuple ceases to be current in the database and becomes a historical tuple. Then, the start time of the tuple is updated to the current timestamp, and the nontemporal attributes are changed accordingly. A DELETE statement creates only the historical tuple with end time equal to the current time. The following UPDATE statement changes the description of the IDSE department as of 2016 (cf. Fig. 8):

```
UPDATE Dept
SET DDesc = ‘DBS @ unibz’
WHERE DName = ‘idse’;
```

This creates a historical tuple `(idse, 2010, 2016, DB @ unibz)`, which records the name of the `idse` department until `2015` (the gray tuple in Fig. 8). At the same time, the start time of the current tuple for the `idse` department is set to `2016` and the description is set to `DBS @ unibz`.

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DB @ unibz	2010	9999

(a) Before UPDATE statement

Dept

DName	DDesc	DStart	DEnd
ifi	DBTG @ uzh	2009	9999
idse	DBS @ unibz	2016	9999
idse	DB @ unibz	2010	2016

(b) After UPDATE statement

Fig. 8. Modifying a transaction time table.

4.3 Integrity Constraints

Valid Time Tables. Primary keys enforce uniqueness of attribute values in a table. In a valid time table, the natural interpretation of a *primary key* is to require *uniqueness of attribute values at each time point.* To achieve this, the primary key specification includes, in addition to the nontemporal key attributes, also the valid time period together with the `WITHOUT OVERLAPS` constraint.

This ensures that only *one value at a time* exists for nontemporal key attributes (that is, the same values for nontemporal key attributes require *disjoint* periods). For instance, we can use the following primary key constraint to enforce that an employee is never in two different departments at the same time:

```
ALTER TABLE Emp
ADD PRIMARY KEY (EName, EPeriod WITHOUT OVERLAPS);
```

This primary key constraint would reject the table in Fig. 9 since the valid times of both tuples with EName equal to Anton include year 2014. Without the WITHOUT OVERLAPS clause, but including EPeriod, we would obtain a conventional primary key, which is satisfied by the table in Fig. 9 since the tuples have syntactically different values for these two attributes.

Emp

EName	**EDept**	**EStart**	**EEnd**
Anton	ifi	2010	2015
Anton	idse	2014	2017

Fig. 9. Primary key constraint is violated.

Valid time tables support also foreign keys to enforce the existence of certain tuples. A *foreign key constraint* in a valid time table guarantees that, at each point in time, for each tuple in the child table there exists a corresponding tuple in the referenced parent table. Consider Fig. 10 and assume that both Emp and Dept are valid time tables with valid time EPeriod and DPeriod, respectively. The following foreign key constraint achieves that, at any time, the department, in which an employee works, exists:

Emp

EName	**EDept**	**EStart**	**EEnd**
Anton	ifi	2010	2015
Anton	idse	2015	2017

Dept

DName	**DDesc**	**DStart**	**DEnd**
ifi	DBTG @ uzh	2009	9999
idse	DBS @ unibz	2016	9999
idse	DB @ unibz	2010	2016

Fig. 10. Valid time foreign keys.

```
ALTER TABLE Emp
ADD FOREIGN KEY (EDept, PERIOD EPeriod)
REFERENCES (DName, PERIOD DPeriod);
```

It is not required that a single matching tuple exists in the referenced parent table that entirely covers the tuple in the child table. It is sufficient that the union of the timestamps of matching tuples in the parent table covers the timestamp of the corresponding tuple in the child table. The tables in Fig. 10 satisfy the above constraint. The first tuple in the `Emp` table is covered by a single tuple in the `Dept` table, while the second tuple is covered by the union of the second and third tuples in `Dept`.

Transaction Time Tables. The enforcement of primary and foreign key constraints in transaction time tables is much simpler since only current tuples need to be considered. Historical data continue to satisfy the constraints as they are never changed. Therefore, the time periods need not to be included in the definition of the key constraints.

A *primary key* on an attribute in a transaction time table enforces that *at most one current tuple* exists with a given value for that attribute. Note that there might be several historical tuples with the same key attribute value. Consider now that `Emp` and `Dept` are transaction time tables. Then, the following primary key constraint ensures that there exists at most one current tuple with a given `DName` value:

```
ALTER TABLE Dept
ADD PRIMARY KEY (DName);
```

In a similar way, also *foreign keys* need only be verified among the current tuples of the two tables. That is, for each *current tuple* in the child table, there exists a matching *current tuple* in the parent table. For instance, the following constraint enforces that for each current `Emp` tuple there exists *now* a tuple in the `Dept` table with `DName = EDept`:

```
ALTER TABLE Dept
ADD FOREIGN KEY (EDept) REFERENCES (DName);
```

4.4 Querying Temporal Tables

Valid Time Tables. SQL:2011 provides limited support for querying temporal tables, in particular for valid time tables. The usual SQL syntax can be used to specify constraints on the period end points. For instance, the following query retrieves all departments that existed in 2012:

```
SELECT DName, DDesc
FROM Emp
WHERE DStart <= ‘2012’ AND DEnd > ‘2012’;
```

To facilitate the formulation of queries, so-called *period predicates* are introduced, such as `OVERLAPS`, `BEFORE`, `AFTER`, etc. Although similar, they do not correspond exactly to Allen's interval relations [4]. With these predicates, the selection predicate in the above statement can be specified as `WHERE EPeriod CONTAINS DATE ‘2011’`.

The temporal predicates can also be used in the `FROM` clause. For instance, the `OVERLAPS` predicate allows to formulate a temporal join, which requires that matching result tuples are temporally overlapping. The following query is a temporal join on the department name of the `Emp` and `Dept` tables:

```
SELECT *
FROM Emp
JOIN Dept ON EDept = DName AND EPeriod OVERLAPS DPeriod;
```

Transaction Time Tables. To facilitate the retrieval of data from transaction time tables, three new SQL extensions are provided. First, the `FOR SYSTEM_TIME AS OF` extension retrieves tuples *as of a given time point*, i.e., tuples with start time less than or equal to and end time larger than a user-specified time point. The following statement retrieves all employee tuples that were current in the database in 2010:

```
SELECT *
FROM Emp FOR SYSTEM_TIME AS OF DATE '2010';
```

The second extension, `FOR SYSTEM_TIME FROM TO`, retrieves tuples *between any two time points*, where the start time is included and the end time is excluded, corresponding to a closed-open interval model. The following statement retrieves all tuples that were current from 2011 (including) up to 2013 (excluding):

```
SELECT *
FROM Dept FOR SYSTEM_TIME FROM DATE '2011' TO DATE '2013';
```

The third extension is `FOR SYSTEM_TIME BETWEEN` and is similar to the previous one, except that the end time point is also included, corresponding to a closed-closed interval model:

```
SELECT *
FROM Dept FOR SYSTEM_TIME BETWEEN DATE '2011' AND DATE '2012';
```

If none of the above extensions are specified in the `FROM` clause, only the *current* tuples are considered. This corresponds to `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP`. This feature facilitates the migration to system-versioned tables, as old queries would continue to produce correct results by considering only current tuples.

5 Temporal Data Support in Commercial DBMSs

Since the introduction of the temporal features in the SQL:2011 standard [58,96], major database vendors have started to implement temporal support in their database management systems [59,72,73]. Some companies realized the need for supporting temporal data earlier, and they extended their database systems with basic temporal features, such as data types, functions, and time travel queries, that make past states of a database available for querying.

IBM was the first vendor to integrate the temporal features from SQL:2011 into their DB2 database system, which occurred in version 10 [79]. DB2 supports both valid time and transaction time tables, which are called business time and system time tables, respectively. Transaction time tables are implemented by means of two distinct tables: a current table and a history table. The current table stores the current snapshot of the data, i.e., all tuples whose timestamp contains the current time (now). The history table stores all previously current data, i.e., all tuples that were modified or deleted in the past. Queries on transaction time tables are automatically rewritten into queries over one or both of these two tables.

The Oracle DBMS supports temporal features from SQL:2011 as of version 12c. The temporal features are implemented using the Oracle flashback technology [69]. The syntax employed differs slightly from that of the SQL standard, e.g., `AS OF PERIOD FOR` is used instead of `FOR` to retrieve data in a certain time period. Earlier versions of Oracle offered similar support for temporal data through the Oracle Workspace Manager [68]. The workspace manager (DBMS_WM package) offered a PERIOD data type with associated predicates and functions as well as additional support for valid and transaction time. Querying temporal relations was possible at a specific time point (snapshot) or for a specific period.

PostgreSQL originally provided an external module [26] that introduced a PERIOD data type for anchored time intervals together with Boolean predicates and functions, such as intersection, union, and minus. Most of the functionality of this module was subsequently integrated into the core of PostgreSQL version 9.2 using range types [77]. Unlike the period specification that is metadata in the SQL standard, a range type in PostgreSQL is a new data type in the query language that was introduced to represent generic intervals, and it comes with associated predicates, functions, and indices. Indices on range types are based on the extendible index infrastructure GiST (Generalized Search Tree) and SP-GiST (space-partitioned Generalized Search Tree). These indices support efficient querying when predicates involve range types as well as support efficient constraint implementation, such as uniqueness for overlapping intervals in a table.

The Teradata DBMS as of version 15.00, supports derived periods and temporal features from the SQL:2011 standard. Version 13.10 already integrated temporal features, including a PERIOD data type with associated predicates and functions as well as support for valid and transaction time [3,89]. The querying of valid and transaction time tables is achieved by means of so-called temporal statement modifiers such as `SEQUENCED` and `NONSEQUENCED` [16]. The implementation of the sequenced semantics is based on query rewriting, where a temporal query is rewritten into a standard SQL query [2,3]. The support for temporal features in Teradata has been enhanced gradually. As of version 14.10 [90], support for sequenced aggregation and coalescing (using the syntax `NORMALIZE ON`) was added. Sequenced outer joins and set operations are not yet supported.

Since 2016, Microsoft's SQL Server [64] has supported temporal features from SQL:2011. The support is limited to transaction time tables, called system time tables. To achieve general temporal support for querying, users have to write user-defined functions [8] that implement the fold and unfold functions of IXSQL.

6 Native Support for Managing Temporal Data in RDBMSs

In this section, we describe a recent approach [30] to extending a relational database engine to achieve full-fledged, industrial-strength, and comprehensive support for sequenced temporal queries. The key idea is to reduce temporal queries to nontemporal queries by first adjusting the timestamps of the input tuples, which produces intermediate relations on which the corresponding nontemporal operators are applied. This solution provides comprehensive support for temporal queries with sequenced semantics without limiting the use of queries with nonsequenced semantics. The approach is systematic and separates interval adjustment from the evaluation of the operators. This strategy renders it possible to fully leverage the query optimization and evaluation engine of a DBMS for sequenced temporal query processing.

6.1 Requirements for Sequenced Temporal Queries

The evaluation of sequenced temporal queries over a temporal database has to satisfy four properties: *snapshot reducibility*, *change preservation*, *extended snapshot reducibility*, and *scaling* (cf. Sect. 2.3). Two important ingredients are needed for the query execution in order to achieve these properties:

– timestamps must be *adjusted* for the result, and
– some values might have to be *scaled* to the adjusted timestamps.

This is illustrated in the example in Fig. 11, which computes the budget for each department in the `Dept` relation. In a nontemporal context, the result would be 380 K for the DB department and 150 K for the AI department. In a temporal context, we want to obtain the time-varying budget shown in Fig. 11(b). We observe that there are two result tuples for the `DB` department. Result tuple z_1 is over the time period [Feb, Apr], where only one project is running. Result tuple z_2 is over the time period [May, Jul] with two contributing input tuples, namely r_1 and r_2. A second observation is that the total budget of 200 K of the input tuple r_1 is distributed over (or *scaled* to) the two sub-periods [Feb, Apr] and [May, Sep], i.e., 100 K is assigned to each of the two periods.

A major limitation of SQL that renders it difficult to process interval timestamped data, such as in the above example, is that periods are considered as atomic units. Comparing interval timestamped tuples in SQL yields the following results:

Dept

	Name	Dept	Budget	Time
r_1	Sue	DB	200K	[Feb, Jul]
r_2	Tim	DB	180K	[May, Jul]
r_3	Joe	AI	150K	[Apr, Aug]

(a) Department relation

Result

	Dept	SUM	Time
z_1	DB	100K	[Feb, Apr]
z_2	DB	280K	[May, Jul]
z_3	AI	150K	[Apr, Aug]

(b) Budget per department

Fig. 11. Compute budget for each department.

- (DB, [May, Jul]) = (DB, [May, Jul]) $\rightarrow$ *true*
- (DB, [Feb, Apr]) = (DB, [May, Jul]) $\rightarrow$ *false*
- (DB, [Feb, Jul]) = (DB, [May, Jul]) $\rightarrow$ *false*

The first two comparisons are ok, since the two tuples are, respectively, identical in the first case and have disjoint timestamps (and hence are syntactically different) in the second case. The result of the last comparison is problematic in a temporal context. Since the two timestamps are overlapping, the two tuples are equal over the common part of the timestamps.

6.2 Reducing Temporal Operators to Nontemporal Operators via Temporal Alignment

Based on the above requirements and observations, the core of the temporal alignment approach [30] is to adjust the timestamps of input tuples such that all tuples that contribute to a single result tuple have identical timestamps. The adjusted timestamps can then be treated as atomic units, and the corresponding nontemporal operator with SQL's notion of equality produces the expected result. Additionally, for some queries, the original timestamp needs to be preserved, and the attribute values need to be scaled. This reduction of a temporal operator ψ^T to the corresponding nontemporal operator ψ is a four-step process (cf. Fig. 12):

1. *Timestamp propagation* replicates the original timestamps in the argument relations as additional attributes. This step is optional and is only executed if the original timestamps are needed, either to scale attribute values in step 3 or to evaluate a predicate or a function that references the original timestamps, in step 4.
2. *Interval adjustment* splits the overlapping timestamps of the input tuples such that they are aligned. This yields an intermediate relation, where all tuples that (in step 4) are processed together to produce a result tuple have the same timestamp. This intermediate relation can conceptually be considered as a sequence of snapshots, each of which lasts for one or more time points. Two interval adjustment primitives are needed: a *temporal normalizer*, $\mathcal{N}$, for the operators π, ϑ, $-$, $\cap$, and $\cup$, where for each time point, one input tuple can contribute to at most one result tuple; and a *temporal aligner*, ϕ, for the operators $\times$, $\bowtie$, $⟖$, $⟕$, $⟗$, and $\triangleright$, where for each time point, one input tuple can contribute to more than one result tuple.

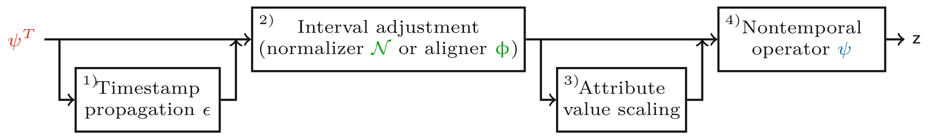

ψ^T
1)Timestamp propagation ϵ
2) Interval adjustment (normalizer $\mathcal{N}$ or aligner ϕ)
3)Attribute value scaling
4)Nontemporal operator ψ
z

Fig. 12. Reduction of a temporal operator ψ^T to the corresponding nontemporal operator ψ using interval adjustment, timestamp propagation, and attribute value scaling.

3. *Attribute value scaling* is optional and scales, if required, the attribute values of intermediate tuples to the adjusted timestamps. For this, the original and new timestamps in addition to the original value of the attribute to be scaled are needed. As part of this step, the propagated timestamps are removed if they are no longer needed by subsequent operators.
4. The *nontemporal operator evaluation* applies the corresponding nontemporal operator ψ to the intermediate relations. An additional equality constraint over the adjusted timestamps (e.g., as a grouping attribute for aggregation or an equality predicate in joins) guarantees that all tuples that produce a single result tuple are processed together. For join operations, a post-processing step α is required to remove non-maximal duplicates.

The interval adjustment (step 2) and the evaluation of the corresponding nontemporal operator (step 4) form the core of the temporal alignment approach and guarantee snapshot reducibility and change preservation. In addition, the propagation of the timestamp intervals (step 1) enables attribute value scaling (step 3) and the access to the original timestamp in step 4 (needed for extended snapshot reducibility).

Table 1 provides a summary of the reduction rules, following the above strategy, for all operators of the relational algebra. Here, r and s are temporal relations, $\mathbf{A}$ and $\mathbf{B}$ are sets of attributes, T is the timestamp attribute, θ is a condition, and α is a post-processing operator that removes duplicates. For instance, the temporal aggregation operator ${}_{\mathbf{B}}\vartheta^T_F(\mathsf{r}) = {}_{\mathbf{B},T}\ \vartheta_F(\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r}))$ with grouping attributes $\mathbf{B}$ can be computed as follows: First, input relation r is aligned by calling the temporal normalizer $\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r})$, which yields an intermediate relation of aligned tuples. Then, nontemporal aggregation is applied to the intermediate result. By adding the timestamp attribute T as an (additional) grouping attribute, the intermediate tuples are grouped by snapshot, and the nontemporal aggregation operator is applied to each snapshot.

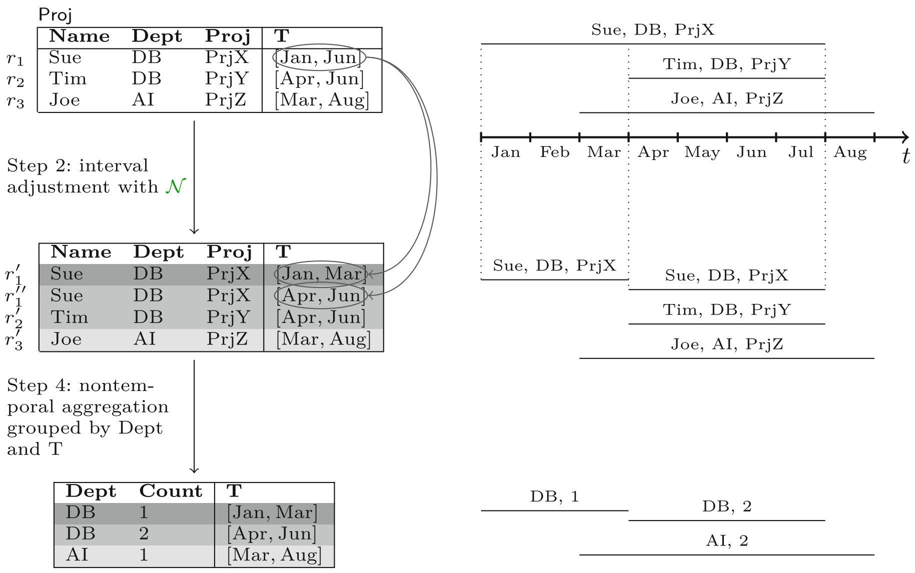
Figure 13 illustrates the temporal alignment approach by using a temporal aggregation query to compute the number of projects for each department: ${}_{Dept}\vartheta^T{}_{Count(Proj)}(\mathsf{Proj})$. Since timestamp propagation and attribute value scaling are not needed for this query, steps 1 and 3 are skipped. During the adjustment of the timestamps using the temporal normalizer (step 2), the first input tuple r_1 is split into tuples r_1' and r_1''. The split point is determined by tuple r_2, which belongs to the same department. There is no need to split tuples r_2 and r_3, yielding an intermediate relation with four tuples. Then, the intermediate

Table 1. Reduction rules $\psi^T \longrightarrow \{\mathcal{N},\phi\} + \psi$ (from [27,30])

Operator			Reduction
Selection	$\sigma^T_\theta(\mathsf{r})$	$=$	$\sigma_\theta(\mathsf{r})$
Projection	$\pi^T_{\mathbf{B}}(\mathsf{r})$	$=$	$\pi_{\mathbf{B},T}(\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r}))$
Aggregation	${}_{\mathbf{B}}\vartheta^T_F(\mathsf{r})$	$=$	${}_{\mathbf{B},T}\vartheta_F(\mathcal{N}_{\mathbf{B}}(\mathsf{r},\mathsf{r}))$
Difference	$\mathsf{r} -^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathbf{A}}(\mathsf{r},\mathsf{s}) - \mathcal{N}_{\mathbf{A}}(\mathsf{s},\mathsf{r})$
Union	$\mathsf{r} \cup^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathbf{A}}(\mathsf{r},\mathsf{s}) \cup \mathcal{N}_{\mathbf{A}}(\mathsf{s},\mathsf{r})$
Intersection	$\mathsf{r} \cap^T \mathsf{s}$	$=$	$\mathcal{N}_{\mathbf{A}}(\mathsf{r},\mathsf{s}) \cap \mathcal{N}_{\mathbf{A}}(\mathsf{s},\mathsf{r})$
Cartesian Product	$\mathsf{r} \times^T \mathsf{s}$	$=$	$\alpha(\phi_\top(\mathsf{r},\mathsf{s}) \bowtie_{\mathsf{r}.T=\mathsf{s}.T} \phi_\top(\mathsf{s},\mathsf{r}))$
Inner Join	$\mathsf{r} \bowtie^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) \bowtie_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Left Outer Join	$\mathsf{r} \text{⟕}^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) \text{⟕}_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Right Outer Join	$\mathsf{r} \text{⟖}^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) \text{⟖}_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Full Outer Join	$\mathsf{r} \text{⟗}^T_\theta \mathsf{s}$	$=$	$\alpha(\phi_\theta(\mathsf{r},\mathsf{s}) \text{⟗}_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r}))$
Anti Join	$\mathsf{r} \triangleright^T_\theta \mathsf{s}$		$\phi_\theta(\mathsf{r},\mathsf{s}) \triangleright_{\theta\wedge\mathsf{r}.T=\mathsf{s}.T} \phi_\theta(\mathsf{s},\mathsf{r})$

relation is aggregated (step 4) by using the nontemporal aggregation, where the timestamp attribute T is added to the grouping attributes. Hence, all tuples with the same adjusted timestamp and the same department are processed together.

Figure 14 illustrates a temporal left outer join using the temporal aligner primitive. Given a manager relation Mgr and a project relation Proj, we want



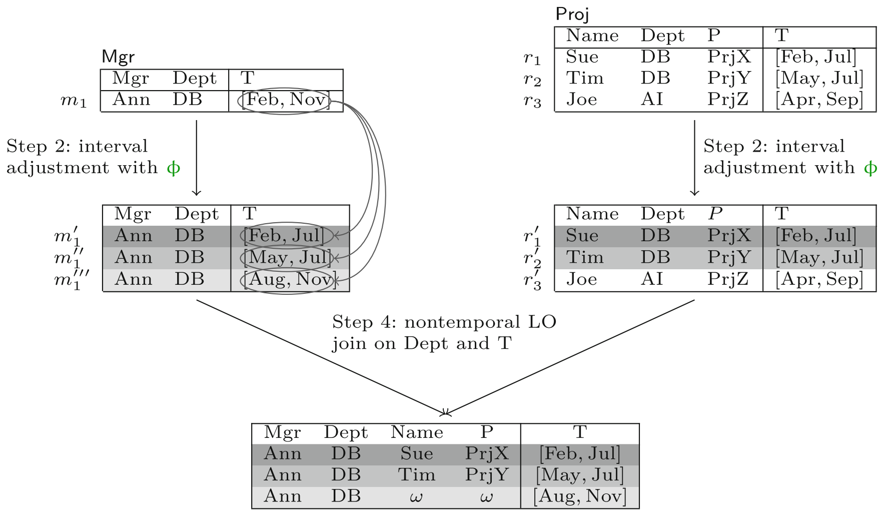
Proj

	Name	Dept	Proj	T
r_1	Sue	DB	PrjX	[Jan, Jun]
r_2	Tim	DB	PrjY	[Apr, Jun]
r_3	Joe	AI	PrjZ	[Mar, Aug]

	Name	Dept	Proj	T
r_1'	Sue	DB	PrjX	[Jan, Mar]
r_1''	Sue	DB	PrjX	[Apr, Jun]
r_2'	Tim	DB	PrjY	[Apr, Jun]
r_3'	Joe	AI	PrjZ	[Mar, Aug]

Dept	Count	T
DB	1	[Jan, Mar]
DB	2	[Apr, Jun]
AI	1	[Mar, Aug]

Fig. 13. Illustration of temporal normalizer for a temporal aggregation query.


Mgr
Proj
Step 2: interval adjustment with ϕ
Step 2: interval adjustment with ϕ
Step 4: nontemporal LO join on Dept and T
m_1
m_1'
m_1''
m_1'''
r_1
r_2
r_3
r_1'
r_2'
r_3'

Mgr

	Mgr	Dept	T
m_1	Ann	DB	[Feb, Nov]

Proj

	Name	Dept	P	T
r_1	Sue	DB	PrjX	[Feb, Jul]
r_2	Tim	DB	PrjY	[May, Jul]
r_3	Joe	AI	PrjZ	[Apr, Sep]

	Mgr	Dept	T
m_1'	Ann	DB	[Feb, Jul]
m_1''	Ann	DB	[May, Jul]
m_1'''	Ann	DB	[Aug, Nov]

	Name	Dept	P	T
r_1'	Sue	DB	PrjX	[Feb, Jul]
r_2'	Tim	DB	PrjY	[May, Jul]
r_3'	Joe	AI	PrjZ	[Apr, Sep]

Mgr	Dept	Name	P	T
Ann	DB	Sue	PrjX	[Feb, Jul]
Ann	DB	Tim	PrjY	[May, Jul]
Ann	DB	ω	ω	[Aug, Nov]

Fig. 14. Illustration of temporal aligner for a temporal left-outer join query.

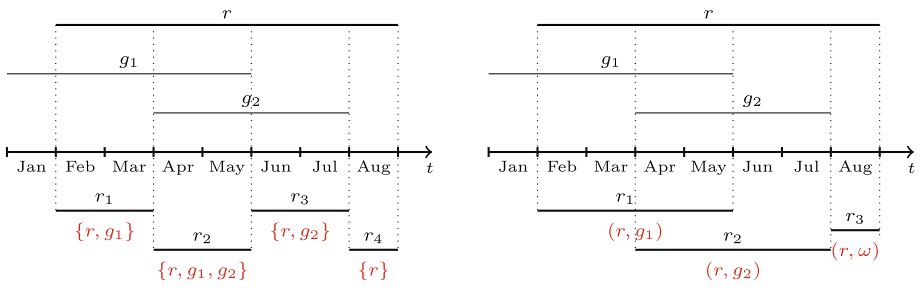
to determine a manager's budget: $\mathsf{Mgr} \;⟕^{T}_{\mathsf{Mgr}.Dept=\mathsf{Proj}.Dept}\; \mathsf{Proj}$. Again, to keep the example simple, timestamp propagation and attribute value scaling are not involved. Hence, we first align the timestamps of the two input relations. In the manager relation, the only tuple m_1 is split into three intermediate tuples. The first two, m_1' and m_1'', are generated from the intersection of m_1's timestamp and the timestamp of the joining tuples r_1 and r_2 of Proj, respectively. The third intermediate tuple, m_1''', covers the part of m_1's timestamp not covered by any matching tuple in the project relation; this tuple is needed for the outer join. Similarly, the tuples in the project relation are adjusted. The first two tuples are completely covered by matching tuple m_1, so no split is required. The third tuple need not to be split since it has no matching tuple in Mgr. After adjusting the timestamps of the two input relations, the two intermediate tables are joined by using the nontemporal left outer join, where the timestamp attribute T is added to the join condition in order to join only tuples that have identical timestamps.

6.3 Temporal Primitives

The temporal alignment approach requires two new temporal primitives, a temporal normalizer and a temporal aligner, to break the timestamps of the input tuples into aligned pieces.

The *temporal normalizer* is used for operators, $\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)$, for which more than one tuple of each argument relation r_i can contribute to a result tuple z (i.e., the lineage set of z can contain more than one tuple from each input relation). This holds for the following operators: aggregation (ϑ), projection (π), difference ($-$), intersection ($\cap$), and union ($\cup$). The temporal normalizer splits each input tuple into temporally *disjoint* pieces, where groups of matching tuples define the split points. This is illustrated in Fig. 15(a) for an input tuple r and two other input tuples g_1 and g_2 in the same group. Tuple r is split whenever another tuple in the same group starts or finishes, producing r_1, r_2 and r_3. Moreover, all parts of r that are not covered by another tuple in the group are reported, i.e., r_4. Notice that the intermediate tuples are disjoint.

The *temporal aligner* is used for operators, $\psi(\mathsf{r}_1, \ldots, \mathsf{r}_n)$, for which at most one input tuple from each argument relation r_i can contribute to a result tuple z (i.e., the lineage set of z contains at most one tuple from each input relation). This holds for the following operators: Cartesian product ($\times$) and all forms of joins (⋈, ⟗, ⟕, ⟖, $\triangleright$). The temporal aligner considers pairs of matching tuples and determines the *intersections* of their timestamps; the resulting intermediate relation might contain temporally overlapping tuples. Figure 15(b) illustrates the temporal aligner for an input tuple r and two other matching input tuples g_1 and g_2. Tuple r produces three intermediate tuples: one as the intersection with tuple g_1, one as the intersection with tuple g_2, and one for the part of the timestamp that is not covered by any matching tuple (r_3).



(a) Temporal normalizer (b) Temporal aligner

Fig. 15. Temporal normalizer vs. aligner (from [27,30]).

6.4 Implementation

The temporal alignment approach to transform temporal queries to the corresponding nontemporal queries with the help of two adjustment primitives requires minimal extensions of an existing DBMS. Moreover, this strategy renders it possible to fully leverage the query optimization and evaluation engine of a DBMS for sequenced temporal query processing, and it does not affect the use of nonsequenced queries. The key extension is the integration of the normalizer $\mathcal{N}$ and aligner ϕ operators into the DBMS kernel. Timestamp propagation (ϵ) and attribute value scaling can be achieved, respectively, by means of generalized projections and user defined functions.

The temporal alignment approach has been implemented in the kernel of the PostgreSQL database system and is available at tpg.inf.unibz.it [27,30].

7 Conclusion

In this tutorial, we provided an overview of temporal data management, covering both research results and commercial database management systems. Following a brief summary of important concepts that have been developed and used in temporal database research, we discussed the state-of-the-art in temporal database research, focusing on query languages and evaluation algorithms. We then described the most important temporal features in SQL:2011, which is the first SQL standard to introduce temporal support in SQL. Next, we briefly discussed the degree to which temporal features of the SQL:2011 standard have been adopted by commercial database management systems. The tutorial ends with a description of a recent framework that provides a comprehensive and native solution to the processing of so-called sequenced temporal queries in relational database management systems.

Future work in temporal databases points in various directions. While temporal alignment provides a solid and systematic framework for implementing temporal query support in relational database systems, a number of open issues require further investigation. First, it would be interesting to extend the framework to multisets that allow duplicates, as well as to support two or more time dimensions. Second, for some operators, a significant boost in efficiency is needed to scale for very large datasets. Ideas for performance optimizations range from additional and more targeted alignment primitives over more precise cost estimates to specialized query algorithms and equivalence rules. Third, support for user-friendly formulation of complex temporal queries is needed, including a SQL-based temporal query language.

References

1. Agesen, M., Böhlen, M.H., Poulsen, L., Torp, K.: A split operator for now-relative bitemporal databases. In: Proceedings of the 17th International Conference on Data Engineering, ICDE 2001, pp. 41–50 (2001)
2. Al-Kateb, M., Ghazal, A., Crolotte, A.: An efficient SQL rewrite approach for temporal coalescing in the teradata RDBMS. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012. LNCS, vol. 7447, pp. 375–383. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32597-7_32
3. Al-Kateb, M., Ghazal, A., Crolotte, A., Bhashyam, R., Chimanchode, J., Pakala, S.P.: Temporal query processing in teradata. In: Proceedings of the 16th International Conference on Extending Database Technology, EDBT 2013, pp. 573–578 (2013)
4. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
5. Arbesman, S.: Stop hyping big data and start paying attention to 'long data'. Wired.com (2013). https://www.wired.com/2013/01/forget-big-data-think-long-data/
6. Bair, J., Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Notions of upward compatibility of temporal query languages. Wirtschaftsinformatik **39**(1), 25–34 (1997)

7. Behrend, A., et al.: Temporal state management for supporting the real-time analysis of clinical data. In: Bassiliades, N., et al. (eds.) New Trends in Database and Information Systems II. AISC, vol. 312, pp. 159–170. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-10518-5_13
8. Ben-Gan, I., Sarka, D., Wolter, R., Low, G., Katibah, E., Kunen, I.: Inside Microsoft SQL Server 2008 T-SQL programming, Chap. 12. In: Temporal Support in the Relational Model. Microsoft Press (2008)
9. Bettini, C., Jajodia, S., Wang, S.: Time Granularities in Databases, Data Mining, and Temporal Reasoning. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04228-1
10. Bettini, C., Sean Wang, X., Jajodia, S.: Temporal granularity. In: Liu and Özsu [60], pp. 2968–2973
11. Böhlen, M.H., Gamper, J., Jensen, C.S.: An algebraic framework for temporal attribute characteristics. Ann. Math. Artif. Intell. **46**(3), 349–374 (2006)
12. Böhlen, M.H., Gamper, J., Jensen, C.S.: How would you like to aggregate your temporal data? In: Proceedings of the 13th International Symposium on Temporal Representation and Reasoning, TIME 2006, pp. 121–136 (2006)
13. Böhlen, M., Gamper, J., Jensen, C.S.: Multi-dimensional aggregation for temporal data. In: Ioannidis, Y., et al. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 257–275. Springer, Heidelberg (2006). https://doi.org/10.1007/11687238_18
14. Böhlen, M.H., Jensen, C.S.: Temporal data model and query language concepts. In: Encyclopedia of Information Systems, pp. 437–453. Elsevier (2003)
15. Böhlen, M.H., Jensen, C.S.: Sequenced semantics. In: Liu and Özsu [60], pp. 2619–2621
16. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Temporal statement modifiers. ACM Trans. Database Syst. **25**(4), 407–456 (2000)
17. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Current semantics. In: Liu and Özsu [60], pp. 544–545
18. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Nonsequenced semantics. In: Liu and Özsu [60], pp. 1913–1915
19. Böhlen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in temporal databases. In: Proceedings of 22th International Conference on Very Large Data Bases, VLDB 1996, pp. 180–191 (1996)
20. Cohen Boulakia, S., Tan, W.C.: Provenance in scientific databases. In: Liu and Özsu [60], pp. 2202–2207
21. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. PVLDB **10**(11), 1346–1357 (2017)
22. Cafagna, F., Böhlen, M.H.: Disjoint interval partitioning. VLDB J. **26**(3), 447–466 (2017)
23. Chomicki, J., Toman, D., Böhlen, M.H.: Querying ATSQL databases with temporal logic. ACM Trans. Database Syst. **26**(2), 145–178 (2001)
24. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. **25**(2), 179–227 (2000)
25. Date, C.J., Darwen, H., Lorentzos, N.A.: Temporal Data and the Relational Model. Elsevier (2002)
26. Davis, J.: Online temporal PostgreSQL reference (2009). http://temporal.projects.postgresql.org/reference.html
27. Dignös, A., Böhlen, M.H., Gamper, J.: Temporal alignment. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, pp. 433–444 (2012)

28. Dignös, A., Böhlen, M.H., Gamper, J.: Query time scaling of attribute values in interval timestamped databases. In: Proceedings of the 29th International Conference on Data Engineering, ICDE 2013, pp. 1304–1307 (2013)
29. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2014, pp. 1459–1470 (2014)
30. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S.: Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. ACM Trans. Database Syst. **41**(4), 26:1–26:46 (2016)
31. Dyreson, C.E.: Chronon. In: Liu and Özsu [60], p. 329
32. Dyreson, C.E., Jensen, C.S., Snodgrass, R.T.: Now in temporal databases. In: Liu and Özsu [60], pp. 1920–1924
33. Dyreson, C.E., Lin, H., Wang, Y.: Managing versions of web documents in a transaction-time web server. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 422–432 (2004)
34. Dyreson, C.E., Rani, V.A.: Translating temporal SQL to nested SQL. In: Proceedings of the 23rd International Symposium on Temporal Representation and Reasoning, TIME 2016, pp. 157–166 (2016)
35. Dyreson, C.E., Rani, V.A., Shatnawi, A.: Unifying sequenced and non-sequenced semantics. In: Proceedings of the 22nd International Symposium on Temporal Representation and Reasoning, TIME 2015, pp. 38–46 (2015)
36. Jensen, C.S., Clifford, J., Gadia, S.K., Grandi, F., Kalua, P.P., Kline, N., Lorentzos, N., Mitsopoulos, Y., Montanari, A., Nair, S.S., Peressi, E., Pernici, B., Robertson, E.L., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Tansel, A., Tiberio, P., Tuzhilin, A., Wuu, G.T.J.: A consensus test suite of temporal database queries. Technical report R 93–2034, Aalborg University, Department of Mathematics and Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark, November 1993
37. Etzion, O., Jajodia, S., Sripada, S. (eds.): Temporal Databases: Research and Practice. LNCS, vol. 1399. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053695
38. Gadia, S.K.: A homogeneous relational model and query languages for temporal databases. ACM Trans. Database Syst. **13**(4), 418–448 (1988)
39. Gadia, S.K., Yeung, C.-S.: A generalized model for a relational temporal database. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD 1988, pp. 251–259 (1988)
40. Galton, A.: A critical examination of Allen's theory of action and time. Artif. Intell. **42**(2–3), 159–188 (1990)
41. Gamper, J., Böhlen, M.H., Jensen, C.S.: Temporal aggregation. In: Liu and Özsu [60], pp. 2924–2929
42. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), 2–29 (2005)
43. Gao, D., Snodgrass, R.T.: Temporal slicing in the evaluation of XML queries. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, pp. 632–643 (2003)
44. Grandi, F.: Temporal databases. In: Encyclopedia of Information Science and Technology, 3rd edn., pp. 1914–1922. IGI Global (2015)
45. Grandi, F., Mandreoli, F., Martoglia, R., Penzo, W.: A relational algebra for streaming tables living in a temporal database world. In: Proceedings of the 24th International Symposium on Temporal Representation and Reasoning, TIME 2017, pp. 15:1–15:17 (2017)

46. Grandi, F., Mandreoli, F., Tiberio, P.: Temporal modelling and management of normative documents in XML format. Data Knowl. Eng. **54**(3), 327–354 (2005)
47. Jensen, C.S., Dyreson, C.E., Böhlen, M.H., Clifford, J., Elmasri, R., Gadia, S.K., Grandi, F., Hayes, P.J., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N.A., Mitsopoulos, Y.G., Montanari, A., Nonen, D.A., Peressi, E., Pernici, B., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Soo, M.D., Uz Tansel, A., Tiberio, P., Wiederhold, G.: The consensus glossary of temporal database concepts. In Temporal Databases, Dagstuhl, pp. 367–405 (1997)
48. Jensen, C.S., Snodgrass, R.T.: Snapshot equivalence. In: Liu and Özsu [60], p. 2659
49. Jensen, C.S., Snodgrass, R.T.: Temporal data models. In: Liu and Özsu [60], pp. 2952–2957
50. Jensen, C.S., Snodgrass, R.T.: Temporal element. In: Liu and Özsu [60], p. 2966
51. Jensen, C.S., Snodgrass, R.T.: Time instant. In: Liu and Özsu [60], p. 3112
52. Jensen, C.S., Snodgrass, R.T.: Timeslice operator. In: Liu and Özsu [60], pp. 3120–3121
53. Jensen, C.S., Snodgrass, R.T.: Transaction time. In: Liu and Özsu [60], pp. 3162–3163
54. Jensen, C.S., Snodgrass, R.T.: Valid time. In: Liu and Özsu [60], pp. 3253–3254
55. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1173–1184 (2013)
56. Kaufmann, M., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F.: Comprehensive and interactive temporal query processing with SAP HANA. PVLDB **6**(12), 1210–1213 (2013)
57. Kline, N., Snodgrass, R.T.: Computing temporal aggregates. In: Proceedings of the 11th International Conference on Data Engineering, ICDE 1995, pp. 222–231 (1995)
58. Kulkarni, K.G., Michels, J.-E.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
59. Künzner, F., Petković, D.: A comparison of different forms of temporal data management. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015. CCIS, vol. 521, pp. 92–106. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18422-7_8
60. Liu, L., Tamer Özsu, M. (eds.): Encyclopedia of Database Systems. Springer, Boston (2009)
61. López, I.F.V., Snodgrass, R.T., Moon, B.: Spatiotemporal aggregate computation: a survey. IEEE Trans. Knowl. Data Eng. **17**(2), 271–286 (2005)
62. Lorentzos, N.A.: Time period. In: Liu and Özsu [60], p. 3113
63. Lorentzos, N.A., Mitsopoulos, Y.G.: SQL extension for interval data. IEEE Trans. Knowl. Data Eng. **9**(3), 480–499 (1997)
64. Microsoft. SQL Server 2016 - temporal tables (2016). https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables
65. Moffitt, V.Z., Stoyanovich, J.: Towards sequenced semantics for evolving graphs. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, pp. 446–449 (2017)
66. Montanari, A., Chomicki, J.: Time domain. In: Liu and Özsu [60], pp. 3103–3107
67. Moon, B., López, I.F.V., Immanuel, V.: Efficient algorithms for large-scale temporal aggregation. IEEE Trans. Knowl. Data Eng. **15**(3), 744–759 (2003)
68. Murray, C.: Oracle database workspace manager developer's guide (2008). http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28396.pdf

69. Oracle. Database development guide - temporal validity support (2016). https://docs.oracle.com/database/121/ADFNS/adfns_design.htm#ADFNS967
70. Papaioannou, K., Böhlen, M.H.: TemProRA: top-k temporal-probabilistic results analysis. In: Proceedings of the 32nd IEEE International Conference on Data Engineering, ICDE 2016, pp. 1382–1385 (2016)
71. Persia, F., Bettini, F., Helmer, S.: An interactive framework for video surveillance event detection and modeling. In: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, pp. 2515–2518 (2017)
72. Petković, D.: Modern temporal data models: strengths and weaknesses. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015. CCIS, vol. 521, pp. 136–146. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18422-7_12
73. Petkovic, Dušan: Temporal data in relational database systems: a comparison. In: Rocha, Á., Correia, A.M., Adeli, H., Teixeira, M.M., Reis, L.P. (eds.) New Advances in Information Systems and Technologies. AISC, vol. 444, pp. 13–23. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31232-3_2
74. Piatov, D., Helmer, S.: Sweeping-based temporal aggregation. In: Gertz, M., et al. (eds.) SSTD 2017. LNCS, vol. 10411, pp. 125–144. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64367-0_7
75. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: Proceedings of the 32nd International Conference on Data Engineering, ICDE 2016, pp. 1098–1109 (2016)
76. Pitoura, E.: Historical graphs: models, storage, processing. In: Zimányi, E. (ed.) eBISS 2017. LNBIP, vol. 324, pp. 84–111. Springer, Cham (2017)
77. PostgreSQL Global Development Group. Documentation manual PostgreSQL - range types (2012). http://www.postgresql.org/docs/9.2/static/rangetypes.html
78. Rolland, C., Bodart, F., Léonard, M. (eds.) Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems (1988)
79. Saracco, C., Nicola, M., Gandhi, L.: A matter of time: Temporal data management in DB2 10 (2012). http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf
80. Snodgrass, R.T. (ed.): Proceedings of the International Workshop on an Infrastructure for Temporal Databases (1993)
81. Snodgrass, R.T. (ed.): The TSQL2 Temporal Query Language. Kluwer (1995)
82. Snodgrass, R.T. (ed.): A Case Study of Temporal Data. Teradata Corporation (2010)
83. Snodgrass, R.T., Böhlen, M.H., Jensen, C.S., Steiner, A.: Adding valid time to SQL/temporal. Technical report ANSI-96-501r2, October 1996
84. Snodgrass, R.T., Böhlen, M.H., Jensen, C.S., Steiner, A.: Transitioning temporal support in TSQL2 to SQL3. In: Temporal Databases, Dagstuhl, pp. 150–194 (1997)
85. Son, D., Elmasri, R.: Efficient temporal join processing using time index. In: Proceedings of the 8th International Conference on Scientific and Statistical Database Management, SSDBM 1996, pp. 252–261 (1996)
86. Soo, M.D., Jensen, C.S., Snodgrass, R.T.: An algebra for TSQL2. In: The TSQL2 Temporal Query Language, Chap. 27, pp. 501–544. Kluwer (1995)
87. Uz Tansel, A., Clifford, J., Gadia, S.K., Jajodia, S., Segev, A., Snodgrass, R.T. (eds.): Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings (1993)
88. Tao, Y., Papadias, D., Faloutsos, C.: Approximate temporal aggregation. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, pp. 190–201 (2004)

89. Teradata. Teradata database 13.10 - temporal table support (2010). http://www.info.teradata.com/download.cfm?ItemID=1005295
90. Teradata. Teradata database 14.10 - temporal table support (2014). http://www.info.teradata.com/eDownload.cfm?itemid=131540028
91. Terenziani, P., Snodgrass, R.T.: Reconciling point-based and interval-based semantics in temporal relational databases: a treatment of the telic/atelic distinction. IEEE Trans. Knowl. Data Eng. **16**(5), 540–551 (2004)
92. Toman, D.: Point vs. interval-based query languages for temporal databases. In: Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996, pp. 58–67 (1996)
93. Toman, D.: Point-based temporal extensions of SQL and their efficient implementation. In: Etzion, O., Jajodia, S., Sripada, S. (eds.) Temporal Databases: Research and Practice. LNCS, vol. 1399, pp. 211–237. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053704
94. Tuma, P.A.: Implementing Historical Aggregates in TempIS. Ph.D. thesis, Wayne State University (1992)
95. Yang, J., Widom, J.: Incremental computation and maintenance of temporal aggregates. VLDB J. **12**(3), 262–283 (2003)
96. Zemke, F.: Whats new in SQL: 2011. SIGMOD Rec. **41**(1), 67–73 (2012)
97. Zhang, D., Markowetz, A., Tsotras, V.J., Gunopulos, D., Seeger, B.: Efficient computation of temporal aggregates with range predicates. In: Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2001 (2001)
98. Zhang, D., Tsotras, V.J., Seeger, B.: Efficient temporal join processing using indices. In: Proceedings of the 18th International Conference on Data Engineering, ICDE 2002, pp. 103–113 (2002)
99. Zhou, X., Wang, F., Zaniolo, C.: Efficient temporal coalescing query support in relational database systems. In: Bressan, S., Küng, J., Wagner, R. (eds.) DEXA 2006. LNCS, vol. 4080, pp. 676–686. Springer, Heidelberg (2006). https://doi.org/10.1007/11827405_66

Historical Graphs: Models, Storage, Processing

Evaggelia Pitoura[✉]

Computer Science and Engineering Department, University of Ioannina, Ioannina, Greece
pitoura@cs.uoi.gr

Abstract. Historical graphs capture the evolution of graphs through time. A historical graph can be modeled as a sequence of graph snapshots, where each snapshot corresponds to the state of the graph at the corresponding time instant. There is rich information in the history of the graph not present in just the current snapshot of the graph. In this chapter, we present logical and physical models, query types, systems and algorithms for managing historical graphs. We also highlight promising directions for future work.

Keywords: Data graph · Graph query · Evolving graph Temporal graph

1 Introduction

Graphs offer a natural model for the interactions and relationships between entities. There are numerous applications of graphs. In collaborative networks, graph edges capture the cooperation between actors in movies, authors of scientific articles, or co-workers in a team. In social networks, graphs edges express the relationship (e.g., friend, follower) as well as the interactions and reactions (e.g., retweets, likes) between users. In communication networks, edges indicate email and phone exchanges between people, in transportation networks, edges represent roads and flights between cities, while in biological networks, edges may model interactions between proteins.

Most graphs that model such real networks evolve with time. New interactions and relationships are created, while existing ones may no longer be valid. In addition, new entities join the network, while old ones leave the network. Furthermore, content associated with the graph structure, such as labels on vertices or weights on edges, is also updated. We use the term *historical graph* to refer to the sequence $\mathcal{G} = \{G_1, G_2, ...\}$ of graph snapshots, where each snapshot G_t in the sequence captures the state of the graph (i.e., vertices, edges and content) at time instant t.

There is rich information in a historical graph not to be found in just the current snapshot. As a very simple example, there is no way to differentiate between a vertex whose centrality (e.g., degree, or Pagerank) keeps rising and a vertex whose centrality keeps falling, if both vertices have the same current centrality value. In general, looking at the whole history of the graph helps us identify interesting patterns in its evolution and may be useful in predicting its future states.

In this chapter, we present recent work in processing historical graphs. Since this is an active area of research, the goal is both to survey recent research results and also highlight promising directions for future research.

The remainder of this chapter is organized as follows. In Sect. 2, we start with a formal definition of a historical graph and logical models for its representation. A distinction is also made between historical graphs and other types of graphs with temporal information. Then, in Sect. 3, we present a general overview of research in graph management to serve as background for the work on historical graph management. An overview and a taxonomy of the variety of interesting graph queries that are possible in the case of historical graphs is presented in Sect. 4. In Sect. 5, we discuss various issues regarding the physical representation and storage of historical graphs, while in Sect. 4, we introduce a taxonomy of the approaches to processing historical graphs queries. Section 7 concludes the chapter.

2 Historical Graphs

A *graph* is typically represented as an ordered pair $G = (V, E)$ of a set V of *vertices* and a set $E \subseteq V \times V$ of *edges*. Graphs can be *undirected* or *directed*. In undirected graphs, an edge is a 2-multiset of vertices, that is edge (u, v) is identical with edge (v, u). In directed graphs, an edge (u, v) is an ordered pair of vertices where order indicates a direction, from source vertex u to destination vertex v. *Multigraphs* are graphs that allow multiple edges between two vertices.

In some cases, *data* or *content* is attached to the graph structure. The simplest form is that of values or *labels* associated with vertices, edges or both. For example, a *vertex labeled graph* is a triplet $G = (V, E, L)$ where $L\colon V \rightarrow \Sigma$ is a labeling function that assigns to each vertex in V a label from a set of labels Σ. A special case of a labeled graph is a *weighted* graph where labels take numerical values. Often, labels are used to associate semantics with vertices and edges, for example, to give vertices and edges a *type*.

In the following definitions, for notational simplicity, we will consider vertex labeled graphs, but the definitions easily extend to include graphs with other type of content associated with them. We will also use the term *graph element* to denote a vertex, edge or label.

Graphs change over time. Graph updates may involve structural updates, that is, the addition or deletion of vertices and edges. The content associated with vertices and edges may also be updated. A *historical graph* captures the evolution of the graph over time by maintaining all edges, vertices and content.

We assume a linearly ordered, discrete time domain Ω^t and use successive integers to denote successive time instants in Ω^t. A time interval is a contiguous set of time instants where $[t_i, t_j]$, with $t_j \geq t_i$, denotes the time instants from t_i to t_j.

Let $G_t = (V_t, E_t, L_t)$ denote the *graph snapshot* at time instant t, that is, the set of vertices, edges and the labeling function that exist at time instant t.

Definition 1 (Historical Graph). *A historical graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.*

An example of a historical graph is shown in Fig. 1. Time instants may, for example, correspond to milliseconds, seconds, minutes or days. Defining various levels of time granularity are also possible by grouping sets of time instants.

We call *lifespan* of a graph element the set of time intervals during which the element exists. To model the case where the same element is deleted and then re-inserted, lifespans are sets of time intervals, also known as *temporal elements*. For example, the lifespan of edge (1, 2) in the historical graph in Fig. 1 is $\{[1, 2], [4, 5]\}$. We assume that lifespans consist of non-overlapping and non-adjacent time intervals.

Note that, if there are no element re-inserts, a lifespan is just a time interval instead of a set of intervals. Finally, if no element is ever deleted, we can use simple time instants to represent lifespans, i.e., the time instant when the element was created.

We now define useful aggregation graphs of the historical graph, namely, the union, intersection and version graph.

Definition 2 (Union Graph). *Given a historical graph $\mathcal{G}$, we call union graph of $\mathcal{G}$, the graph $G_\cup = (V_\cup, E_\cup, L_\cup)$ where $V_\cup = \bigcup_{G_t \in \mathcal{G}} V_t$, $E_\cup = \bigcup_{G_t \in \mathcal{G}} E_t$ and $L_\cup(u) = \bigcup_{G_t \in \mathcal{G}} L_t(u)$.*

An example is shown in Fig. 2(a). The union graph includes all elements independently of their lifespan. Any time information is lost.

Definition 3 (Intersection Graph). *Given a historical graph $\mathcal{G}$, we call intersection graph of $\mathcal{G}$, the graph $G_\cap = (V_\cap, E_\cap, L_\cap)$ where $V_\cap = \bigcap_{G_t \in \mathcal{G}} V_t$, $E_\cap = \bigcap_{G_t \in \mathcal{G}} E_t$ and $L_\cap(u) = \bigcap_{G_t \in \mathcal{G}} L_t(u)$.*

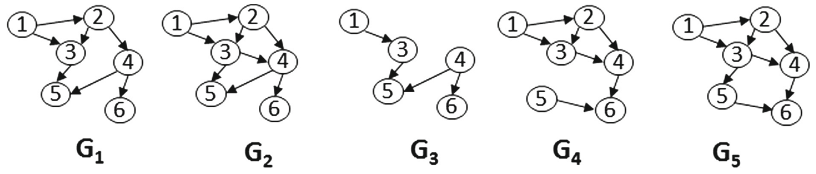


Fig. 1. A historical graph consisting of five snapshots. Vertex labels are not shown.

An example is shown in Fig. 2(b). In the intersection graph, transient elements are lost. The intersection graph includes only the elements that exist in all time instants.

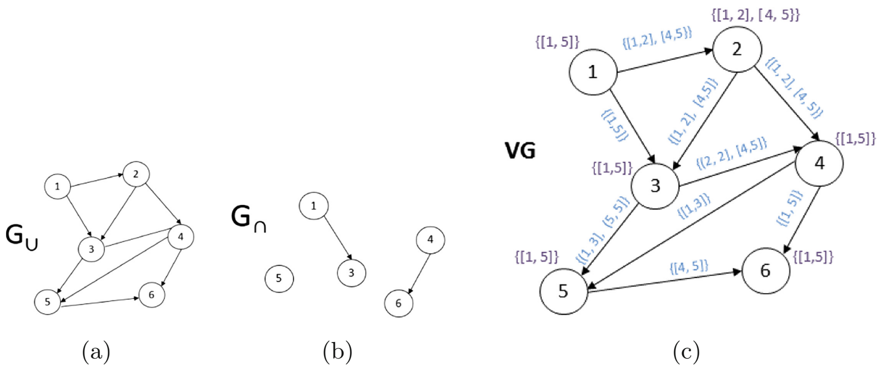


Fig. 2. The (a) union, (b) intersection, and (c) version graph of the historical graph in Fig. 1.

Definition 4 (Version Graph). *Given a historical graph $\mathcal{G}$, the version graph of $\mathcal{G}$, is the union graph of $\mathcal{G}$ where each element is annotated with its lifespan.*

An example is shown in Fig. 2(c). The version graph offers a concise representation of a historical graph preserving all the information in the graph sequence. The lifespans associated with the elements in the version graph can be seen as *timestamps*.

Note that in temporal database research, there are two notions of time, namely valid and transaction time, associated with the facts recorded in a database (e.g., see [19], and [6] in this collection). The *valid time* of a fact corresponds to the time period over which the fact is true in the real world, while the *transaction time* of a fact corresponds to the time when the fact is recorded in the database. *Bitemporal* databases provide support for both valid and transaction time. In our case, lifespans report valid times, since the lifespan of a graph element indicates the time periods during which the graph element exists in the real world modeled by the graph.

We also adopt *point-based* semantics. An alternative approach is an *interval-based* view where time intervals are not just containers of time points (i.e., instants). Instead, time-intervals are atomic values that can not be split or merged without altering the meaning of data [5].

Other Graphs with Temporal Information. Another line of research that studies graphs over time focuses on *dynamic*, *time-evolving*, or, *evolutionary* graphs (e.g., see [1]). As opposed to research in historical graphs, research in dynamic graphs looks at just a single snapshot of the graph, namely, the current one. For example, centrality computation (e.g., Pagerank) in dynamic graphs would output the current centrality value of a vertex. In contrast, when the graph is seen as a historical one, centrality computations involve many previous snapshots and may reflect fluctuations and trends in the centrality values of the vertices over time.

Research in dynamic graphs aims at timely graph analysis and query evaluation so that the current state of the graph is reflected in the results. Often, the focus is on efficient incremental updates of any indexes and auxiliary data structures so as to avoid reconstructing them from scratch each time the graph is updated.

Graph streams are a special type of dynamic graphs (see e.g., [33] for a survey). In the case of graph streams, graph updates arrive in a streaming fashion and queries are continuously evaluated against this stream of updates in real time. An additional constraint in this model is limited storage and the impossibility of storing the whole graph in memory or disk. A key challenge is dealing with the high rate at which updates are generated. A popular model of processing infinite data streams in small space is a *sliding-window model* where the goal is to evaluate a query using the data that has arrived in the last window of time.

Another type of dynamic graphs are *online* dynamic graphs (e.g., [3]). Research in online dynamic graphs also considers graphs that evolve over time but it assumes that these updates are not known. Instead, to get information about the current state of the graph, one needs to explicitly pose requests, or *probes* to the graph. There is a clear trade-off between the number and the frequency of probes and the quality of the analysis: the larger the number of probes, the more accurate the results of the analysis. This model finds applications for example in third-party computations on the Twitter graph, where although the graph constantly evolves, access to the graph is made possible only through a rate-limited API.

Finally, another form of graphs with time information are *temporal graphs* (e.g., [49]). Temporal graphs are often used to model interaction networks, or transportation networks, and in general, networks for which there is time duration or delay associated with their edges.

Definition 5 (Temporal graph). *A temporal graph $G = (V, E)$ is a graph where each edge $e \in E$ is a quadruple (u, v, t_s, δ) where u, $v \in V$, t_s is the starting time of e and δ is the delay or duration of e.*

For example, when a temporal graph is used to model a phone call network, t_s is the time instant when the call between user u and user v started and δ is the duration of the call. Analogously, for a temporal graph representing a road network, vertices may correspond to places and edges to routes between them. In this case, t_s is the time instant of arrival at a place u and δ the traversal time to get from place u to place v. Since there may be multiple interactions between vertices, temporal graphs are often multigraphs.

3 Overview of Graph Processing and Graph Systems

Graph management is a very active area of research. Research on data graphs can be roughly classified into research that focus on developing algorithms and indexes for specific graph queries and research that focus on developing general purpose graph management systems. In this section, we provide a concise review of related research to serve as background for the following sections.

3.1 Graph Processing

We can distinguish graph processing into two general categories:

– graph queries, and
– graph analytics.

There are two generic types of *graph queries*, namely *navigational queries* and *graph pattern queries* [4]. The basic form of a navigational query is a *path query*, Q_{path}: $u \xrightarrow{C} v$, where u, v denote the starting and ending points of the paths and C conditions on the paths. The query Q_{path} on a graph G retains all paths of G from u to v that satisfy C. The starting and ending points can be specific vertices or vertices with specific content, or a mix of both. C expresses constraints on the content (e.g., labels) that the vertices and the edges of the path must satisfy. Two special cases of navigational queries that have received considerable attention are (a) *shortest path* queries that return the shortest among the qualifying paths, and (b) *reachability* queries that return true if there is at least one qualifying path and false otherwise.

A graph pattern query specifies as input a graph pattern $Q_{pattern}$ which is usually a labeled subgraph. The query returns all *matches* of $Q_{pattern}$ in the graph G. Most commonly, a match is defined as a subgraph of G that is isomorphic to $Q_{pattern}$, but various other semantics have also been considered.

There are many types of online analysis in the case of graphs. Most common forms of analysis performed on graphs include PageRank and other types of centrality computations, finding all pairs shortest paths, identifying connected components and triangle counting. The authors of [22] list 29 different types of graph analysis tasks along with their frequency of use for evaluating graph processing systems.

Finally, another way to distinguish graph processing is based on the extent of the graph that is explored. In this respect, graph processing can be:

– *local*, when only specific parts of the graph are considered, an example is an *egonetwork* query that looks only in the neighborhood of a vertex, or
– *global*, when the whole graph is explored as is the case with most navigational queries.

3.2 Systems

Numerous systems have been proposed for managing graphs (see, e.g., [21] and [51] for recent surveys and additional references). We can distinguish graph systems into two general categories, namely, graph databases and graph processing systems. Graph databases provide out of core storage of graphs, offer OLTP queries and transactions. Graph processing systems are tailored to OLAP type of graph analysis.

Graph Databases. Graph databases are based on a graph data model and provide support for both navigational queries and graph pattern matching. The data models most commonly supported by graph databases are the *resource* *description framework (RDF)* and the *property graph model (PGM)*. Some graph databases come with their own custom graph data model.

RDF is the standard model for exchanging information on the web consisting of (subject, predicate, object) triples. If we consider subjects and objects as vertices and predicates as edges, a dataset consisting of RDF triples forms a directed labeled multigraph. For the RDF graph databases, the related query language is called SPARQL.

A property graph is a directed multigraph where an arbitrary set of key-value pairs, called *properties*, can be attached to vertices and edges. In addition, property graphs support labels that specify vertex and edge types. An example is shown in Fig. 3. In its basic form, PGM is schema-free, in that, there is no dependency between a type label and the allowed property keys. Although not a standard, the Gremlin graph traversal language, part of the TinkerPop graph processing framework, is the language often supported by property graph databases [48]. Another graph query language for property graphs that is gaining popularity is the query language of the Neo4j database, called Cypher [46].

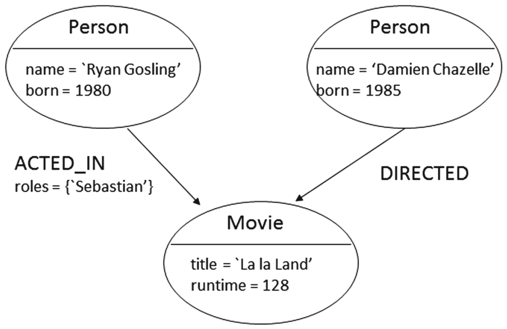


Fig. 3. An example of a property graph: there are two different types of vertices (Person and Movie) and two different types of edges (DIRECTED and ACTED_IN); properties are also associated with vertices and edges, for example, there are two properties associated with Person, namely name and born.

We can distinguish graph databases into native and non-native ones. *Native graph databases* use a storage model tailored to graphs, for example, adjacency lists. *Non-native graph databases* are built on top of databases supporting alternative data models, for example, on top of relational, or document databases. The most common way to store a graph in a relational database is by maintaining a vertex and an edge table. Edges are commonly stored as vertex pairs.

Graph Processing Systems. Graph processing systems focus on analytical tasks that need to iteratively process the graph. We can distinguish graph processing systems into two broad categories, namely *specialized* graph processing frameworks such as Pregel and its derivatives and graph systems built *on top* of general purpose parallel programming frameworks. The overall architecture uses a master node for coordination and a set of worker nodes for the actual distributed processing. The input graph is partitioned among the worker nodes.

Pregel advocates the *think-like-a-vertex* programming model [32]. The user provides a vertex compute function that proceeds in three steps: read all incoming messages, update the internal state of the vertex and send information to its neighbors. These functions are executed in synchronized supersteps. At each superstep, the worker applies this function to all its vertices. This model of execution is also known as bulk synchronous parallel (BSP) mode. Giraph provides an open source implementation of the Pregel model [12].

Many systems propose performance optimizations and additional features, for example graph mutations or combiners and aggregators for reducing the computation and communication cost. Powergraph [13] uses a *gather-apply-scatter (GAS)* programming model, where the user provides three functions: a gather function that aggregates all messages addressing the same vertex on the sending worker node, an apply function that updates the state of each vertex based on the incoming messages, and a scatter function that uses the vertex state to create the outgoing messages. There are also systems such as GraphLab [30] and GraphChi [30] that allow for asynchronous execution.

In addition to the think-like-a-vertex approaches that operate on the scope of a single vertex, there are also programming models that operate on the subgraph level. In the *graph-partition* centric model, the user provides a compute function to be applied to all vertices managed by a worker node. In the *neighborhood* centric model the compute function operates on custom subgraphs of the input graph explicitly built around vertices and their multihop neighborhoods.

The *filter-process* programming model, also known as *think like an embedding*, operates on embeddings, where an embedding is a subgraph instance of the input graph that matches a user-specified pattern. The user provides a filter function that examines whether a given embedding is eligible and a process function that performs some computation on the embedding and may produce output. The filter-process model differs from the previous two models in that embeddings are dynamically generated during execution, whereas partitions and subraphs are generated once, at the beginning of computation as a pre-processing step. The *filter-process* model comes closer to pattern matching queries seen in graph databases.

There are also systems that use *linear algebra* primitives and operations for expressing graph algorithms. An example is Pegasus [23], an open source library that implements various graph analytics based on an iterated matrix-vector multiplication primitive.

Besides specialized graph processing systems, there are also graph systems built on top of existing distributed programming frameworks. This allows treating graph analysis as part of a larger data analysis pipeline. Two well-known graph systems built on top of distributed in memory dataflow systems are GraphX [14] built on top of Spark and Gelly [11] built on top of Flink. Both adopt variants of the vertex centric programming model.

Finally, we can further distinguish graph processing systems in those that require the full graph to be in memory and those that do not (i.e., out-of-core systems). Further, there are single machine processing systems as well as distributed ones.

3.3 Storage Layout

The simplest way to represent a graph is as a list of edges, that is, as a list of vertex pairs. A very common representation is also an *adjacency list* representation where edges are stored in per-vertex edge arrays. Each vertex u points to an array containing its neighboring vertices. Most commonly the array contains the destination vertices of the outgoing edges of u. In some cases, there is also an additional array containing the source vertices of the incoming edges of u.

A commonly used in-memory representation for graphs is the *compressed sparse row representation* (CSR) format. CSR keeps vertices and edges in separate arrays. The vertex array is indexed by vertex id. It stores offsets into the edge array; each position points to the first neighbor of the corresponding vertex in the edge array. The CSR representation of the first snapshot G_1 in Fig. 1 is shown in Fig. 4. CSR uses minimum $O(n, m)$ storage, where n is the number of vertices and m the number of edges. Note that the CSR format is not very appropriate for dynamic graphs, since modifying the neighbor of a vertex affects the pointers and neighbors of all vertices that follow it.

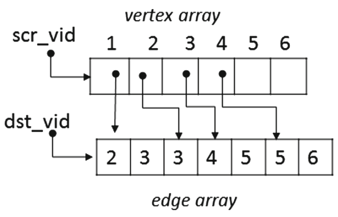


Fig. 4. CSR representation of graph G_1 in Fig. 1.

4 Historical Graph Queries

In this section, we present a taxonomy of graph queries for historical graphs. Although, we focus on graph queries, the presented query types translate easily to graph analytics as well.

Past-Snapshot Historical Graph Queries. The first type of historical graph queries refers to typical graph queries applied to past snapshots. Let us call them *past-snasphot* historical graph queries. Specifically, each graph query Q is associated with a temporal element (i.e., a set of time intervals) I_Q and Q is applied at all time instants in I_Q. The simplest type of a past-snapshot query is a *time-point* query where I_Q is just a single time instant t. In this case, Q is just applied at the corresponding graph snapshot G_t. When I_Q is a single time interval, past-snapshot queries are also known as *time-slice* queries. In general,

I_Q may involve an arbitrary number of time intervals to allow the application of Q at various time instants of interest, for example, for identifying temporal patterns, such as, periodic behavior.

When I_Q includes more than one time instant, we can get various different types of past-snapshot queries by considering alternative ways of aggregating the results of applying Q in the time instants in I_Q. Take for example a path query Q_{path}: $u \xrightarrow{C} v$ from u to v to be applied in all graph snapshots corresponding to the time instants in I_Q. Which of the paths from u to v that satisfy C should be retained? At one extreme, we could retain all paths that exist in at least one graph snapshot in I_Q. At the other extreme, we could retain only the paths that exist in all graph snapshots in I_Q. In general, we may ask that a path appears in at least $L > 1$ graph snapshots, for some user defined L value.

A variety of additional possible interpretations exist in the case of distance path queries that return the length of the shortest path between u and v. For example, are we looking for the distance of the shortest among the paths that exist in all graph snapshots? for the average of the path distances of the shortest path in each graph snapshot? and so on. Similar considerations are applicable to graph pattern queries.

Different aggregation semantics are appropriate for different applications. They also result in processing algorithms with different complexities. Although there has been some initial work in the topic (e.g., [17,42,45]), there are many issues that need further exploration.

Persistence or Durability Historical Graph Queries. Persistence or durability historical graph queries return the most *persistent*, or, *durable* result of a query Q in I_Q. There are two different interpretations of durable results: contiguous and non-contiguous. With *contiguous semantics*, we ask for the results that exist for the longest time interval, while with *non contiguous semantics* for the results that exist for the largest number of time instants. We may also ask not just for the most durable result, but for the top-k most durable results.

A straightforward way to process a durable query is to apply the query at each graph snapshot and then output the most durable among the results. However since the number of results at each graph snapshot may be large, an interesting approach is to "estimate" the duration of the most durable match and avoid unnecessary computations. Although, there has been some work for graph pattern queries [41,44], there are still many directions for future work. A promising direction is considering historical graph pattern queries with relaxed pattern matching semantics.

"When" Historical Graph Queries. An interesting type of historical graph queries that has not been sufficiently explored yet are queries that focus on the time instants that a result (e.g., a path or a graph pattern) appeared. An example would be a query that asks for the *first time* that a path between u and v appeared, or, for the frequency of the appearance of a graph pattern.

Evolution Historical Graph Queries. Finally, a novel type of historical graph queries are queries that look into the evolution of the graph. An instance of such queries can be queries whose input is not a graph pattern or a path but instead an *evolution pattern*. It is an open question how to express an evolution pattern. As a simple example consider looking for vertices (or, subgraphs) that appear and disappear periodically. Another instance of evolution queries is looking for the results that have changed the most, for example, for the pairs of vertices with the largest change in their shortest path distance.

Table 1 depicts examples of different types of historical shortest path queries.

Table 1. Example of different types of historical queries, where the query Q is a shortest path query from vertex u to vertex v and $I_Q = [1, 5]$.

Query type	Example
Past-snapshot	Find the shortest path from vertex u to vertex v during $[1, 5]$
Persistence	Find the top-3 most durable shortest paths from vertex u to vertex v during $[1, 5]$
When	Find the time instant when the shortest path from vertex u to vertex v become smaller than $d = 2$ for the first time during $[1, 5]$
Evolution	Find pairs of vertices such that their shortest paths decreased and then increased again during $[1, 5]$

5 Physical Representation of Historical Graphs

In this section, we focus on the physical representation of historical graphs. We start by presenting general models and techniques and show how they have been exploited and applied in various historical graph management systems. We also briefly discuss how continuous updates are treated.

5.1 Models and Techniques

As with any versioned database, there are two basic strawman approaches to storing a sequence of graph snapshots [40]:

- a COPY approach, where we store each snapshot in the sequence as a separate graph, and
- a LOG approach, where we store only the initial snapshot G_1 and a *delta* log that includes the differences of each snapshot from the one preceding it in the sequence. To get any graph snapshot, we apply the appropriate part of delta to G_1.

The two approaches present a clear tradeoff between storage and performance. The LOG approach takes advantage of the commonalities among snapshots and stores only the different parts of the graph snapshots, thus avoiding the storage redundancy of the COPY approach. However, the LOG approach introduces extra overhead at query processing time for constructing the required snapshots, whereas, in the COPY approach, any snapshot is readily available.

A *hybrid* COPY+LOG approach has also been proposed (e.g., [24], [27]) where instead of storing just a single snapshot, a small set of selected snapshots $\mathcal{G}_{\mathcal{S}} \subset \mathcal{G}$ is stored along with the deltas that relate them. To construct a snapshot that is not in $\mathcal{G}_{\mathcal{S}}$, one of the snapshots in $\mathcal{G}_{\mathcal{S}}$ is selected and the corresponding deltas are applied to it. Deltas can be applied to an old snapshot G_i to create a more recent one G_j, $j > i$. It is also possible to *reverse* a delta and apply it to a new snapshot G_j to get an older one, G_i, $i < j$.

Another approach is a VERSIONING approach (e.g., [45]) where we use just a single graph, the version graph, to represent the sequence. Recall that the version graph is the union graph where each graph element is annotated with its lifespan (see Fig. 2(c)).

In most cases, the graph structure (edges and vertices) is stored separately from any data content, such as properties, weights, or labels, associated with edges and vertices. A COPY, LOG, or VERSIONING approach can be used for representing data content as well. It is also possible to use one approach for storing structural updates and a different one for storing content updates.

In laying-out a historical graph, two types of locality can be exploited [15,34]:

- *time locality*, where the states of a vertex (or an edge) at two consecutive graph snapshots are laid out consecutively, and
- *structure locality*, where the states of two neighboring vertices at the same graph snasphot are laid out close to each other.

While time locality can be achieved perfectly since time progresses linearly, structure locality can only be approximate because it is challenging to project a graph structure into a linear space.

Indexes have also been proposed for historical graphs. Some of these indexes aim at improving the efficiency of constructing specific snapshots. Other indexes focus on the efficiency of reconstructing the history (e.g., all snapshots) of a given vertex or subgraph. There are also indexes tailored to specific types of historical graph queries.

Finally, there are different ways to implement a historical graph management system:

- as a *native* historical graph database (e.g., G* [29]), or as a native historical graph processing system (e.g., Chronos [15]); that is, to build a new system tailored to historical graphs;
- *on top* of an existing graph database or graph processing system (e.g., [37, 43]), that is, to use the model of an existing system to store the graph and to implement historical queries by translating them to the query language supported by the system;
- as an *extension* of an existing graph database or graph processing system (e.g., [20]); that is, to modify an existing system appropriately so as to support historical graphs.

Many representations have been proposed for graphs, deltas and lifespans for both in memory and external storage. In the following, we present representative approaches.

5.2 Historical Graph Systems

Depending on the type of the graph system, many alternative physical representation of a historical graph have been proposed.

A common way to represent a delta log is as *an operational log*, that is, as a sequence of graph operations, i.e., a sequence of vertex and edge insertions and deletions. For example, an operational delta log from G_1 to G_3 in Fig. 1 would look like $\Delta_1 = \{(\textit{insert-edge}, (3,4))\}$, $\Delta_2 = \{(\textit{delete-vertex}, 2)$, $(\textit{delete-edge}, (1,2))$, $(\textit{delete-edge}, (2,3))$, $(\textit{delete-edge}, (2,4))$, $(\textit{delete-edge}, (3,4))\}$.

Alternatively, one could *co-design graphs and deltas* to reduce the cost of re-constructing snapshots. This is the approach taken by most graph processing systems. Examples are LLAMA [31] and Version Traveler (VT) [20] that augment a CSR-based in memory graph representations to store multi-snapshot graphs.

LLAMA and VT In-Memory Graph Layout. LLAMA [31] is a single machine graph processing system optimized for incremental updates that can also provide analytics on multiple snapshots. LLAMA separates the modified neighbors related to a snapshot into a dedicated consecutive area in the neighbor array of the CSR, thus avoiding copying unmodified neighbors. In particular, in LLAMA, a historical graph is represented by a single vertex table and multiple edge tables, one per graph snapshot. The vertex table is organized as a multi-versioned array (LAMA) using a data structure that avoids unnecessary copying of unmodified pages across snapshots. Each vertex in the vertex table maintains the necessary information to retrieve its adjacency list from the edge tables. An edge table for a snapshot G_i contains adjacency list fragments stored consecutively where each fragment contains the edges of a vertex u that were added in G_i and continuation records pointing to any edges of u added in previous snapshots. Deletion of an edge of u is handled by either writing the entire neighborhood of u with no continuation or by deletion vectors encoding when an edge was deleted.

Version Traveler (VT) [20] is a system designed for fast snapshot switching in in-memory graph processing systems. VT further refines the use of CSR for historical graphs with *sharing* and *chaining*. Sharing reduces memory consumption by merging the delta entries of a vertex that span multiple snapshots into a single shared entry. Chaining refers to the representation of the neighbors of a vertex with a chain of vectors, each containing a subset of neighbors and capturing the difference between the snapshots associated with it and the first snapshot.

G* Native Graph Database. G* [29] is a distributed native graph database with a custom nested data model. The goal of G* is storing and managing multiple graphs and graph snapshots efficiently by exploring commonalities among the graphs. G* assigns each vertex and its outgoing edges to the same server and at the same logical disk block. A new version of a vertex is created only when the attributes or the outgoing edges of a vertex change, else the same vertex is shared among snapshots. Each G* server maintains a compact graph index to quickly find the disk location of a vertex and its edges. The index stores only a single (vertex id, disk location) pair for each version of the vertex for the combination of snapshots that contain this version.

The approaches taken by LLAMA, VT, and G* where deltas are co-designed with the graph snapshots can also be seen as a partial COPY approach. In this respect, when a new snapshot G_{i+1} is created, instead of storing the entire G_{i+1} as a separate graph, we store only the parts (subgraphs) of G_i that have been modified. The subgraphs are in the granularity of the neighborhood of each updated vertex.

DeltaGraph Graph Store. An excellent example of a hybrid COPY+LOG representation for storing the historical graph is the approach taken by the *DeltaGraph system* [24]. DeltaGraph focus on the compact storage and the efficient retrieval of graph snapshots. Each event (such as the addition and deletion of a vertex or an edge) is annotated with the time point when the event happened. These events are maintained in event lists. Graph snapshots are not explicitly stored.

There are two main components, namely the *DeltaGraph* hierarchical index and the *GraphPool* in memory data structure. The leaves of the DeltaGraph index correspond to (not explicitly stored) graph snapshots. These snapshots correspond to time points equally-spaced in time. They are connected with each other through bidirectional event lists. The internal nodes of the index correspond to graphs constructed by combining the lower level graphs. These graphs do not necessarily correspond to any actual graph snapshot. The edges are annotated with event lists, termed *event deltas* that maintain information for constructing the parent node from the corresponding child node. An example is shown in Fig. 5. In this example, internal nodes are constructed as the intersection of their children. Other functions, such as the union, could also be used.

To construct the snapshot G_t corresponding to time instant t, first the eventlist L that contains t is located. Any path from the root of the DeltaGraph index to the leaves (snapshots) adjacent to L is a valid solution. For

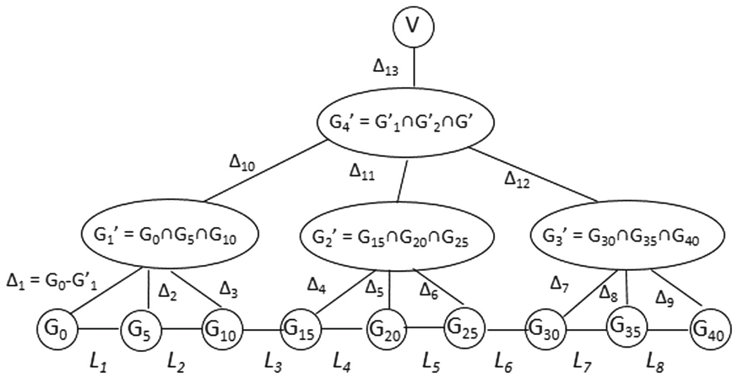


Fig. 5. DeltaGraph for snapshots in time period [0, 45]. Leaves are connected with eventlists (L) and edges are annotated with event deltas (Δ).

instance, to construct snapshot G_{12} in the example of Fig. 5, candidates are the paths to leaves G_{10} and G_{15}. The optimal solution corresponds to the path with the lowest weight, where the weight of each edge captures the cost of reading the associated event delta and applying it to the graph constructed so far. The approach generalized to materializing more than one graph snapshot by considering the lowest-weight Steiner tree that includes the leaves involved.

The GraphPool maintains the union of (a) the current state of the graph, (b) any graph snapshots constructed during the processing of previous queries and (c) graphs and deltas corresponding to internal nodes and edges of the DeltaGraph materialized during previous queries. The union graph is stored in the form of a version graph.

The *Historical Graph Store (HGI)* extends DeltaGraph in two basic ways [25]. First, HGI supports a more refined partitioning of the historical graph than the original equally-spaced partitioning. Second, HGI maintains *version chains* for all vertices in the graph. The version chain for a vertex is a chronologically sorted list of pointers to all references of this vertex in the event lists.

Chronos. Chronos is a parallel graph processing system that supports time-range graph analytics [15]. The in-memory layout of Chronos leverages time locality and uses a variation of the CSR approach. Graph snapshots are stored in a vertex and an edge array. In the vertex array, data content is grouped by the vertices. The content of a vertex in consecutive graph snapshots is stored together. In the edge array, edges are grouped by the source vertices. Each edge stores its target vertex id along with its lifespan. In a sense, this is a VERSIONING approach.

Chronos exploits time locality also for storing historical graphs on disk. For on disk storage, it uses a COPY+LOG approach. Snapshots are partitioned into groups, where a group for interval $[t_i, t_j]$ contains snapshot G_{t_i} and an operational delta that includes all updates until t_j. Snapshot groups are stored in vertex and edge files. Each edge file starts with an index to each vertex in the snapshot group, followed by a sequence of segments, each corresponding to a vertex. The index allows Chronos to locate the starting point of a segment corresponding to a specific vertex without a sequential scan. A segment for a vertex v consists of a starting sector, which includes the edges associated with v and their properties at the start time of this snapshot group, followed by edge operations associated with v. To further speed-up processing, operations that refer to the same edge or vertex are linked together.

Lifespan Representation. An important issue in historical graph processing is the representation of lifespans. Various approaches are presented and evaluated in [41]. One approach is to represent lifespans as a ordered list of time instants. Take for example lifespan $\{[1, 3], [8, 10]\}$. With this representation, the lifespan is maintained as $\{1, 2, 3, 8, 9, 10\}$. Another representation follows the physical representation of an interval by storing an ordered list of time objects where each time object represents an interval by its start and end point.

Finally, an efficient way of representing lifespan is using bit arrays. Assume without loss of generality, that the number of graph snapshots is T. Then, each lifespan is represented by a bit array B of size T, such that $B[i] = 1$ if time instant i belongs to $\mathcal{I}$ and $B[i] = 0$ otherwise. For example, for $T = 16$, the bit array representation of the above interval is 1110000111000000. The bit array representation of lifespans provides an efficient implementation for joining lifespans. Specifically, let $\mathcal{I}$ and $\mathcal{I}'$ be two lifespans and B and B' be their bit arrays. Then, the intersection of the two lifespans (the time instants that belong to both of them) can be computed as B logical-AND B'.

Native Graph Databases. A natural way of storing a historical graph in a native graph database is by using the VERSIONING approach. In this case, a key issue is modeling lifespans. Most native graph database systems support the property model. There are two options for modeling lifespans in this case, one using types and the other using properties. These options are studied in [42,43] where a *multi-edge approach* and a *single-edge approach* are introduced.

The multi-edge (ME) approach utilizes a different edge type between two vertices u and v for each time instant in the lifespan of the edge (u, v). The multi-edge representation of the historical graph $\mathcal{G}_{[1,5]}$ of Fig. 1 is depicted in Fig. 6. Since all native graph databases provide efficient traversal of edges having a specific type, the ME approach provides an efficient way of retrieving the graph snapshot G_t corresponding to time instant t.

The single-edge approach (SE) uses a single edge between any two vertices and uses properties to model theirs lifespans. Two different approaches are considered. In the *single-edge with time points* (SEP) approach, the lifespan of an edge is modeled with one property whose value is a sorted list of the time instants in the lifespan of the edge. The representation of the historical graph $\mathcal{G}_{[1,5]}$ of Fig. 1 is shown in Fig. 7(a). In the *single-edge with time intervals* (SEI) approach, two properties Ls and Le are associated with each edge. Each of the two properties is an ordered list of time instants. Let m be the number of time intervals in the lifespan of an edge. $Ls[i]$, $1 \leq i \leq m$, denotes the start point of the

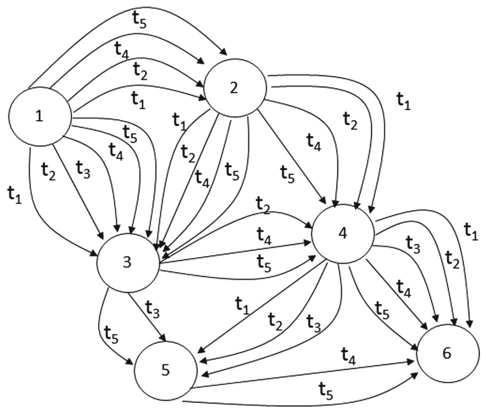


Fig. 6. Representation of the historical graph in Fig. 1 in a native graph database using the multi-edge approach. Vertex labels are not shown for clarity.

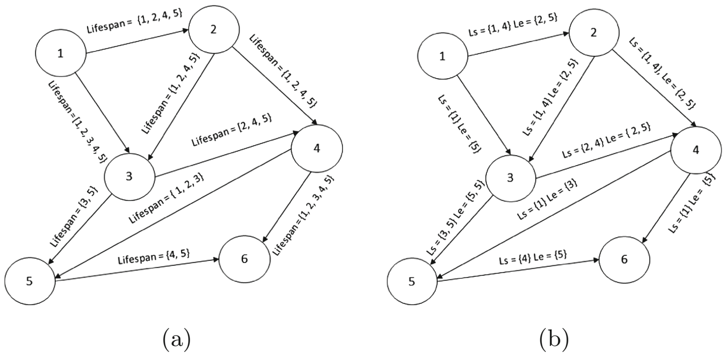


Fig. 7. Representation of the historical graph in Fig. 1 (a) with the single-edge with time points approach, and (b) with the single-edge with time intervals approach. Vertex labels are not shown for clarity.

i-th interval in the lifespan, while $Ls[i]$, $1 \leq i \leq m$, denotes its endpoint. An example is shown in Fig. 7(b). With the single-edge approaches, retrieving the graph snapshot G_t at time instant t requires further processing of the related properties.

A partial COPY approach is followed in [7], where the historical graph is stored as a sequence of graph snapshots. Each graph snapshot is represented using a *frame* vertex that has edges pointing to all graph elements active at the corresponding graph snapshot. Frames are linked together. A hierarchical temporal index is also proposed to provide different time granularity, for example, hours are aggregated to days.

For the VERSIONING approach, the authors of [9] follow a single edge with time intervals approach in which each graph element is annotated with its lifespan, called timestamp. Timestamps are simple time intervals, as opposed to set of time intervals. Combining VERSIONING with a LOG approach is also considered. Operational deltas are proposed called *backlogs*. Each operation in the backlog is represented as a an edge of a special backlog type. A special logger vertex is introduced. Each backlog edge connects this logger vertex with the graph element that the edge (i.e., corresponding operation) refers to.

Relational Graph Databases. The basic way of storing a historical graph in relational databases is to extend the vertex and edge tables with an additional lifespan attribute. For example, the authors of [36,37] use a vertex relation with schema $V(\underline{u}, \underline{p})$ that associates a vertex u with the time interval p during which the vertex is present and an edge relation with schema $E(\underline{u}, \underline{v}, \underline{p})$ connecting pairs of vertices from V during time interval p. Additional relations may be used to store any content associated with the graph structure. For example, in the case of vertex labeled graphs, we may have a relation with schema $L(\underline{u}, \underline{p}, l)$ storing the label l of u at time interval p.

Graph Partitioning. Most graph processing systems exploit some form of parallelism or distribution by assigning parts of a graph to different workers. In abstract terms, a graph partitioning algorithm must satisfy two goals: achieve load balance and minimize the communication cost among the partitions. There are two general approaches to achieving these goals, namely *edge cuts* and *vertex cuts*. Systems based on edge cuts, assign vertices to partitions, while systems based on vertex cuts assign edges to partitions. In both cases, structural locality is considered to minimize the number of neighboring nodes (or, adjacent edges) placed in different partitions. In the case of historical graphs, an additional type of locality, time locality is introduced. Time locality requires placing snapshots that are nearby in time in the same partition.

Partitioning a historical graph is a hard problem, since it depends both on the structure of the graphs in the sequence and on the type of historical graph queries. An exploitation of the tradeoffs of temporal and structural locality for local and global queries is presented in [34]. Some preliminary results regarding partitioning a historical graph in GraphX are presented in [35].

Indexes. There have been proposals for indexes for specific type of graph queries for the case of historical graphs. For path queries, a form of index commonly used is based on 2hop-covers. A 2hop-cover based index maintains for each vertex u a label consisting of a set of vertices that are reachable from u along with their distance from u. Then, to estimate the distance between two vertices u and v, the vertices in the intersection of the labels of u and v are used. The work in [2,16] extends 2hop-cover indexes and proposes appropriate pruning techniques for both dynamic and historical shortest path queries. The work in [45] also considers 2hop-cover indexes but for historical reachability queries.

The authors of [17] propose an extension of *contraction hierarchies* (CHs) for historical shortest path queries. CHs use shortcuts to bypass irrelevant vertices during search. A certain ordering of the vertices in the graph is established according to some notion of relative importance; then the CH is constructed by "contracting" one vertex at a time in increasing order. CHs were extended by adding temporal information on the shortcut edges. Finally, there has been work on building indexes appropriate for graph pattern queries in historical graphs [41,44], where path indexes are extended to compactly store time information.

5.3 Handling Continuous Updates

In the case of historical graphs, most commonly, graph computation is performed separately from graph updates. In particular, graph processing is performed in a sequence of existing snapshots, while any updates are collected to create the next snapshot to be added to the sequence.

This is the approach takes by Kineograph, an in-memory distributed graph store and engine that supports incremental graph mining for graph streams [8]. Kineograph decouples graph mining from graph updates by applying graph mining to an existing snapshot, while graph updates are used to create new snapshots. The graph store produces reliable and consistent snapshots periodically using a simple epoch commit protocol without global locking. The protocol ensures that each snapshot reflects all updates within an epoch by grouping updates in sequence of transactions and applying them in a predetermined order to achieve atomicity. GRAPHTAU considers generating window-based snapshots for the streaming dynamic graph model [18].

6 Processing Historical Graphs

In this section, we present general approaches to processing historical graphs for both the case of graph queries and the case of graph analytics. The presented approaches are tailored to past-snapshot historical graph queries, since the other types of historical graph queries introduced in Sect. 4 are less studied.

6.1 Two-Phase Approach

Let Q be a query (or an analytical computation) to be applied to a historical graph for a temporal element I_Q. A baseline approach to historical query processing is the following *two-phase* strategy. In the first phase, all graph snapshots in I_Q are constructed. Based on the representation of the historical graph, this step may involve, for example, applying deltas, or using a time-index to select the valid elements at each snapshot. In the second phase, the best known static algorithm for Q is applied at each of the snapshots in I_Q. Then, in an optional phase, the results of applying Q at each snapshot are combined. The phases of this approach are depicted in Fig. 8, where $Q(G)$ denotes the results of applying query Q on graph G.

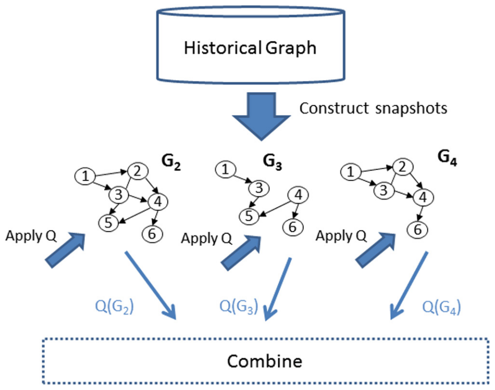


Fig. 8. Two-phase approach for $I_Q = [2, 4]$.

An example of a system that follows a two-phase approach is the *DeltaGraph system* [24]. DeltaGraph focuses on the efficient retrieval of snapshots and supports their compact in-memory representation and re-use.

Since reconstructing the whole graph snapshot is expensive, *partial reconstruction* has been proposed for reducing the cost of the two-phase approach. With partial reconstruction, only those parts of each graph snapshot that are relevant to the query are reconstructed. This approach is especially relevant for local or vertex-centric queries that need to access only a specific subgraph of the historical graph.

Partial views are introduced in [26] to capture the partial reconstruction of historical graphs. A partial view is modeled as an extended egonets. An extended $egonet(v, R, t)$ of a historical graph centered at vertex v is the subgraph of snapshot G_t induced by v and the neighbors of v lying at distance at most R from v. Local historical queries are also modeled as egonets. The view selection problem is then defined as follows. Given a workload of local historical queries and limited storage, select a set of egonets to materialize so that the total cost of evaluating the historical queries is minimized.

To support partial reconstruction, all versions of a vertex are linked together in [25] so as to efficiently reconstruct the neighbors of a vertex through time. Partial reconstruction is also facilitated by the approach taken by HiNode [28]. HiNode stores along with each vertex its complete history, thus making the reconstruction of its egonet very efficient. The resulting vertex is called *diachronic*. Specifically, a diachronic vertex u maintains an external interval tree which stores information regarding the incident edges of u and any content associated with u for the entire graph history.

For some queries, it may be also possible to avoid reconstructing the snapshots. For instance, it is possible to report the degree fluctuation of a vertex through time by just accessing an operational delta. An interesting research problem is identifying the type of historical graph queries that do not require reconstructing graph snapshots. Some initial steps to this end are taken in [27].

6.2 Grouping Snapshots Appproach

The two-phase approach applies query Q at every graph snapshot G_t, t in I_Q. However, since snapshots at consecutive time points are quite similar, some applications of Q may be redundant. At a preprocessing step, the grouping snapshots approach aggregates graph snapshots or subgraphs of these snapshots. At query processing, instead of applying the query at each individual snapshot, the grouping snapshots approach applies the query at the aggregated graphs. Then, the attained results are refined for individual graph snapshots. The basic steps of this approach are depicted in Fig. 9.

This is the approach taken by the *Find-Verify-and-Fix (FVF)* processing framework [38]. At a high level, FVF work as follows. In a preprocessing step, similar graph snapshots are clustered together. For each cluster, a number of representative graphs are constructed. In the *find* step, query Q is applied at the representative graphs of each cluster. In the *verification* step, the result of Q is verified for each graph snapshot G_t in I_Q. Finally, in the *fix* step, Q is applied to those snapshots in I_Q for which the verification step failed.

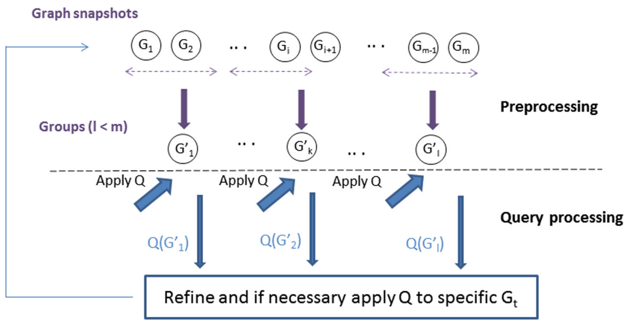


Fig. 9. Grouping snapshots approach.

Many interesting research challenges are involved in realizing FVF. One is how to cluster snapshots. Since graphs evolve gradually, consecutive graph snapshots have many edges in common. To exploit such redundancy, each clusters in [38] contains consecutive graph snapshots. Let us denote with $G^C_\cap$ and $G^C_\cup$ the intersection and the union graph of the graph snapshots in cluster C. As graph snapshots are created, they are placed in the same cluster C until the (Jaccard) distance between the edge sets of $G^C_\cap$ and $G^C_\cup$ exceeds a system defined threshold. If the inclusion of a graph snapshot G_t in the current cluster C would result in the threshold being exceeded, a new cluster is created and G_t is added to the new cluster. The procedure continues until all snapshots are clustered. This clustering algorithm has been extended to a hierarchical one in [39]. Another research question is which graphs to retain as representatives of each cluster C. The authors of [38] maintain $G^C_\cap$ and $G^C_\cup$.

The fix and verify steps depend heavily on the type of historical query. The authors of [38,39] have explored the application of the framework to shortest path queries and closeness centrality queries. The closeness centrality of a vertex u is the reciprocal of the sum of distances from u to all other vertices.

Another approach that falls in this general type of query processing is the *TimeReach* approach to historical reachability queries [45]. At a preprocessing step, TimeReach finds the strongly connected components (SCCs) in each of the graph snapshots, identifies similar SCCs at different graph snapshots and maps them to each other. The mapping of similar SCCs is done in rounds. Initially, the SCCs of G_1 are mapped with the SCCs of G_2. At the next round, the resulting SCCs are mapped with the SCCs of G_3. Mapping continues until the last graph snapshot is mapped. The mapping of the SCCs at each round is reduced to a maximum-weight bipartite matching problem. Let S_i and S_{i+1} be the two sets of SCCs to be mapped at round i. An edge-weighted graph is constructed whose vertices are the SCCs in S_i and S_{i+1}. There is an edge from a vertex representing a SCC in S_i to a vertex representing a SCC in S_{i+1}, if the two SCCs have at least one vertex in common. The weight of each edge corresponds to the number of common vertices between its incident SCCs.

At query processing, to test reachability between two nodes u and v, the identified SCCs are used. For any time point t in I_Q, for which u and v belong to the same SCC, there is no need for refinement. For the remaining graph snapshots, a refinement step is necessary. This step checks reachability between the corresponding SCCs.

Both the FVF and TimeReach approaches take advantage of commonalities between consecutive snapshots to process specific graph queries. It is an open question to formally characterize the set of queries for which the grouping snapshots strategy is applicable. It is also interesting to define appropriate groups, group representatives and general algorithms for refining the results for additional type of queries, such as graph pattern queries.

6.3 Incremental Processing

Another approach to historical query processing is an incremental one. With incremental query processing, instead of computing the results of Q on each graph snapshot G_{t+1} from scratch, the results of Q on G_t are exploited. Figure 10 shows the steps of incremental historical query processing. Incremental processing is often studied in the context of dynamic graphs and streams, where a query is continuously evaluated on the current snapshot.

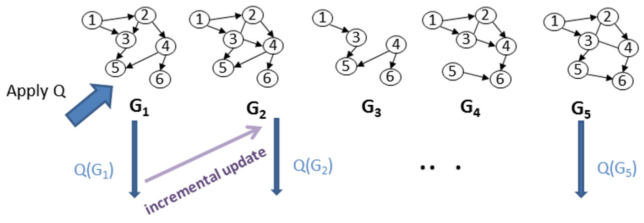


Fig. 10. Incremental approach, the results of Q on G_t are used as input for computing the results of Q on G_{t+1}.

This is the approach taken for example by Kineograph [8]. Kineograph is a dynamic graph processing system that adopts a think-like-a-vertex gather-apply-scatter (GAS) approach. Kineograph uses user-defined rules to check the vertex status compared to the previous snapshot. If the vertex has been modified, for example, if edges were added, Kineograph invokes user-specified functions to compute the new value of the query for that vertex (e.g., a new PageRank value). When the value changes significantly, Kineograph propagates the changes, or deltas, of these values to a user-defined set of vertices, usually in the neighborhood of the vertex. Kineograph supports both push and pull models for communicating updates. In the push model, vertices push updates to neighboring vertices, while in the pull model, a vertex proactively pulls data from its neighboring vertices.

It is not possible to compute all queries incrementally. Recent work studies this problem for dynamic graphs [10]. Let $S \oplus \Delta S$ denote the result of applying updates ΔS to S, where S is either a graph G or a query result $Q(G)$. The *incremental graph computation problem* is formally defined as follows: for a class $\mathcal{Q}$ of graph queries (for example for the class of shortest path queries), find an algorithm, such that, given a query $Q \in \mathcal{Q}$, a graph G, query result $Q(G)$ and updates ΔG of G as input, the incremental algorithm computes changes ΔO of query result $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$. That is, the algorithm answers Q in the updated graph by computing only the changes to the old output of Q. The authors in [10] present various results regarding the existence of incremental algorithms and their efficiency. It is an interesting problem to see how these results translate to historical graphs and to the various types of historical graph queries.

6.4 Time-Locality Aware Iterative Processing

In most approaches to historical query processing that we have presented so far, the same query processing algorithm is applied to multiple snapshots. Such redundancy may be avoided by exploiting time locality. As opposed to applying the same algorithm one snapshot at a time, with time-locality aware iterative processing, each step of the algorithm is applied to multiple snapshots at the same time.

This is the approach taken by Chronos [15] (and its descendant called *Immortal Graph* [34]), a parallel in memory graph processing system. Chronos uses an in-memory layout that exploits time locality. Chronos proposes *locality-aware batch scheduling* (LABS) that exploits this layout to extend the think-like-a-vertex model. LABS batches computation at a vertex as well as propagation to neighboring vertices across all snapshots. Thus, instead of grouping message propagation by snapshots, propagation is grouped by the source and target vertices. An example is shown in Fig. 11.

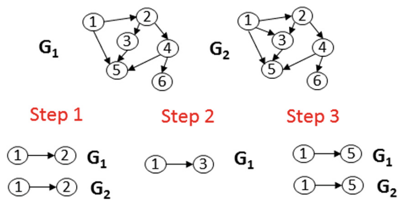


Fig. 11. First steps of an iterative approach on two graph snapshots.

A similar strategy is taken by the *single algorithm multiple snapshots* (SAMS) approach [47]. As opposed to Chronos that focuses on a vertex-centric model, SAMS advocates the automatic transformation of any graph algorithm so that all instances (i.e., evaluations on a particular snapshot) of the algorithm execute the same statement concurrently.

6.5 Recency-Based Processing

An issue with historical graph query processing is the *recency* of the results. As time progresses, the number of graph snapshots and potentially the proportion of stale data in these snapshots becomes larger. Consequently, the results of any analysis may increasingly reflect out-of-date characteristics. On the other extreme, dynamic graph query processing looks only at the current graph snapshot and may miss to detect interesting patterns in the evolution of the graph. In between historical and dynamic graph query processing, the sliding window approach considers a small fixed number of the most current snapshots, and may fail to reflect the *continuity* of results with time. To increase recency and preserve continuity, *decay*, or, *aging* schemes consider all snapshots but they weight the participation of each snapshot in the result based on the recency of the snapshot.

This is the approach taken by TIDE, a system for the analysis of interaction graphs where new interactions, i.e., edges, are continuously created [50]. TIDE proposes a *probabilistic edge decay* (PED) model to produce a static view of dynamic graphs. PED takes one or more samples of the snapshots at a given time instant. The probability $P^f(e)$ that a given edge e is considered at time instant t decays over time according to a user specified decay function, $f : \mathbb{R}_+ \rightarrow [0, 1]$, specifically, $P^f(e) = f(t - t_c(e))$, where $t_c(e)$ is the creation time of edge e. With PED, every edge has a non-zero chance of being included in the analysis (continuity), and this chance becomes increasingly small over time, so that newer edges are more likely to participate (recency).

TIDE focus on the commonly used class of *exponential* decay functions. Such functions are of the general form $f(x) = \alpha^x$, for some $0 < \alpha < 1$, where x denotes the age of an edge. It can be shown that with such functions, the lifespan of each edge in the sample graph follows a geometric distribution.

Recency-based processing introduces an interesting variation in historical graph query processing. A future research direction would be to define novel types of queries on historical graphs that exploit different notions of aging for extracting useful information from the evolution of the graph.

7 Conclusions

Most graphs evolve through time. A historical graph captures the evolution of a graph by maintaining its complete history in the form of a sequence of graph snapshots $\{G_1, G_2, ...\}$ where each graph snapshot G_t in the sequence corresponds to the state of the graph at time instant t. In this chapter, we have presented various approaches to modeling, storing and querying such historical graphs.

Historical and temporal graphs is an active area of research with many open problems. In the chapter, we have also highlighted many promising directions for future research. These directions range from defining novel query types for extracting useful information from the history of the graph to designing more efficient storage models and query processing algorithms.

References

1. Aggarwal, C.C., Subbian, K.: Evolutionary network analysis: a survey. ACM Comput. Surv. **47**(1), 1–36 (2014)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: 23rd International World Wide Web Conference, WWW 2014, Seoul, Republic of Korea, 7–11 April 2014, pp. 237–248 (2014)
3. Anagnostopoulos, A., Kumar, R., Mahdian, M., Upfal, E., Vandin, F.: Algorithms on evolving graphs. In: Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, 8–10 January 2012, pp. 149–160 (2012)
4. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017)
5. Böhlen, M.H., Busatto, R., Jensen, C.S.: Point-versus interval-based temporal data models. In: Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, 23–27 February 1998, pp. 192–200 (1998)
6. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Temporal data management: an overview. In: Business Intelligence - 7th European Summer School, eBISS 2017, Brussels, Belgium, 2–7 July 2017. Tutorial Lectures (2017)
7. Cattuto, C., Quaggiotto, M., Panisson, A., Averbuch, A.: Time-varying social networks in a graph database: a Neo4j use case. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, 24 June 2013, p. 11 (2013)
8. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys 2012, Bern, Switzerland, 10–13 April 2012, pp. 85–98 (2012)
9. Durand, G.C., Pinnecke, M., Broneske, D., Saake, G.: Backlogs and interval timestamps: building blocks for supporting temporal queries in graph databases. In: Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, 21–24 March 2017 (2017)
10. Fan, W., Hu, C., Tian, C.: Incremental graph computations: doable and undoable. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, 14–19 May 2017, pp. 155–169 (2017)
11. Gelly: Documentation (2018). https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html. Accessed Jan 2018
12. Apache Giraph: Documentation (2018). http://giraph.apache.org/literature.html. Accessed Jan 2018
13. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, 8–10 October 2012, pp. 17–30 (2012)
14. GraphX: Programming Guide (2018). https://spark.apache.org/docs/latest/graphx-programming-guide.html. Accessed Jan 2018
15. Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., Chen, E.: Chronos: a graph engine for temporal graph analysis. In: Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, 13–16 April 2014, pp. 1:1–1:14 (2014)

16. Hayashi, T., Akiba, T., Kawarabayashi, K.: Fully dynamic shortest-path distance query acceleration on massive networks. In: Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, 24–28 October 2016, pp. 1533–1542 (2016)
17. Huo, W., Tsotras, V.J.: Efficient temporal shortest path queries on evolving social graphs. In: Conference on Scientific and Statistical Database Management, SSDBM 2014, Aalborg, Denmark, 30 June–02 July 2014, pp. 38:1–38:4 (2014)
18. Padmanabha Iyer, A., Li, L.E., Das, T., Stoica, I.: Time-evolving graph processing at scale. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, 24 June 2016, p. 5 (2016)
19. Jensen, C.S., Snodgrass, R.T.: Temporal data management. IEEE Trans. Knowl. Data Eng. **11**(1), 36–44 (1999)
20. Ju, X., Williams, D., Jamjoom, H., Shin, K.G.: Version traveler: fast and memory-efficient version switching in graph processing systems. In: 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, 22–24 June 2016, pp. 523–536 (2016)
21. Junghanns, M., Petermann, A., Neumann, M., Rahm, E.: Management and analysis of big graph data: current systems and open challenges. In: Zomaya, A., Sakr, S. (eds.) Handbook of Big Data Technologies, pp. 457–505. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-49340-4_14
22. Kalavri, V., Vlassov, V., Haridi, S.: High-level programming abstractions for distributed graph processing. IEEE Trans. Knowl. Data Eng. **30**(2), 305–324 (2018)
23. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system. In: ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6–9 December 2009, pp. 229–238 (2009)
24. Khurana, U., Deshpande, A.: Efficient snapshot retrieval over historical graph data. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, 8–12 April 2013, pp. 997–1008 (2013)
25. Khurana, U., Deshpande, A.: Storing and analyzing historical graph data at scale. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, 15–16 March 2016, pp. 65–76 (2016)
26. Koloniari, G., Pitoura, E.: Partial view selection for evolving social graphs. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, 24 June 2013, p. 9 (2013)
27. Koloniari, G., Souravlias, D., Pitoura, E.: On graph deltas for historical queries. CoRR, abs/1302.5549 (2013). Proceedings of 1st Workshop on Online Social Systems (WOSS) 2012, in conjunction with VLDB 2012
28. Kosmatopoulos, A., Tsichlas, K., Gounaris, A., Sioutas, S., Pitoura, E.: HiNode: an asymptotically space-optimal storage model for historical queries on graphs. Distrib. Parallel Databases **35**(3–4), 249–285 (2017)
29. Labouseur, A.G., Birnbaum, J., Olsen, P.W., Spillane, S.R., Vijayan, J., Hwang, J.-H., Han, W.-S.: The G* graph database: efficiently managing large distributed dynamic graphs. Distrib. Parallel Databases **33**(4), 479–514 (2015)
30. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)

31. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: LLAMA: efficient graph analytics using large multiversioned arrays. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, 13–17 April 2015, pp. 363–374 (2015)
32. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 135–146 (2010)
33. McGregor, A.: Graph stream algorithms: a survey. SIGMOD Rec. **43**(1), 9–20 (2014)
34. Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., Chen, W.: ImmortalGraph: a system for storage and analysis of temporal graphs. TOS **11**(3), 14:1–14:34 (2015)
35. Moffitt, V.Z., Stoyanovich, J.: Towards a distributed infrastructure for evolving graph analytics. In: Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, 11–15 April 2016, Companion Volume, pp. 843–848 (2016)
36. Moffitt, V.Z., Stoyanovich, J.: Temporal graph algebra. In: Proceedings of the 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, 1 September 2017, pp. 10:1–10:12 (2017)
37. Moffitt, V.Z., Stoyanovich, J.: Towards sequenced semantics for evolving graphs. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, 21–24 March 2017, pp. 446–449 (2017)
38. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R.: On querying historical evolving graph sequences. PVLDB **4**(11), 726–737 (2011)
39. Ren, C., Lo, E., Kao, B., Zhu, X., Cheng, R., Cheung, D.W.: Efficient processing of shortest path queries in evolving graph sequences. Inf. Syst. **70**, 18–31 (2017)
40. Salzberg, B., Tsotras, V.J.: Comparison of access methods for time-evolving data. ACM Comput. Surv. **31**(2), 158–221 (1999)
41. Semertzidis, K., Pitoura, E.: Durable graph pattern queries on historical graphs. In: 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, 16–20 May 2016, pp. 541–552 (2016)
42. Semertzidis, K., Pitoura, E.: Time traveling in graphs using a graph database. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, 15 March 2016 (2016)
43. Semertzidis, K., Pitoura, E.: Historical traversals in native graph databases. In: Kirikova, M., Nørvåg, K., Papadopoulos, G.A. (eds.) ADBIS 2017. LNCS, vol. 10509, pp. 167–181. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66917-5_12
44. Semertzidis, K., Pitoura, E.: Top-k durable graph pattern queries on temporal graphs. IEEE Trans. Knowl. Data Eng. (2018, to appear)
45. Semertzidis, K., Pitoura, E., Lillis, K.: TimeReach: historical reachability queries on evolving graphs. In: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, 23–27 March 2015, pp. 121–132 (2015)
46. The Neo4j Team: Manual (2018). https://neo4j.com/docs/developer-manual/3.3/. Accessed Jan 2018
47. Then, M., Kersten, T., Günnemann, S., Kemper, A., Neumann, T.: Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. PVLDB **10**(8), 877–888 (2017)

48. Apache TinkerPop (2018). http://tinkerpop.apache.org/. Accessed Jan 2018
49. Huanhuan, W., Cheng, J., Huang, S., Ke, Y., Yi, L., Yanyan, X.: Path problems in temporal graphs. PVLDB **7**(9), 721–732 (2014)
50. Xie, W., Tian, Y., Sismanis, Y., Balmin, A., Haas, P.J.: Dynamic interaction graphs with probabilistic edge decay. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, 13–17 April 2015, pp. 1143–1154 (2015)
51. Yan, D., Bu, Y., Tian, Y., Deshpande, A.: Big graph analytics platforms. Found. Trends Databases **7**(1–2), 1–195 (2017)

Three Big Data Tools for a Data Scientist's Toolbox

Toon Calders[1,2](✉)

[1] Université Libre de Bruxelles, Brussels, Belgium
[2] Universiteit Antwerpen, Antwerp, Belgium
toon.calders@uantwerpen.be

Abstract. Sometimes data is generated unboundedly and at such a fast pace that it is no longer possible to store the complete data in a database. The development of techniques for handling and processing such streams of data is very challenging as the streaming context imposes severe constraints on the computation: we are often not able to store the whole data stream and making multiple passes over the data is no longer possible. As the stream is never finished we need to be able to continuously provide, upon request, up-to-date answers to analysis queries. Even problems that are highly trivial in an off-line context, such as: "How many different items are there in my database?" become very hard in a streaming context. Nevertheless, in the past decades several clever algorithms were developed to deal with streaming data. This paper covers several of these indispensable tools that should be present in every big data scientists' toolbox, including approximate frequency counting of frequent items, cardinality estimation of very large sets, and fast nearest neighbor search in huge data collections.

1 Introduction

Many data sources produce data as a never-ending stream of records. Examples include sensor networks, logs of user activities on the web, or credit card transactions. Processing these data becomes a challenge, because often there is no storage space or time to store the data for an in-depth off-line analysis. Imagine for instance a credit card fraud detection system that requires that transactions are collected over time and stored on disk for analysis later on. In such a scenario the delay between a credit card fraud and the actual detection of this fraud would be unacceptable. In this case not only the application of fraud prediction methods needs to be online and on the fly, but also the collection of several statistics and modeling parameters needs to be immediate to be able to keep the model up-to-date. Indeed, an important factor in fraud detection is learning what is the normal behavior of a person. This behavior may be changing over time, necessitating flexible and dynamic modelling of what constitutes normal behavior.

We call this type of dynamic processing of data, *stream processing* [4]. We distinguish three different types of stream processing. In the literature these terms are often lumped together while in fact their requirements are quite different.

© Springer International Publishing AG, part of Springer Nature 2018
E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 112–133, 2018.
https://doi.org/10.1007/978-3-319-96655-7_5

1. Online Stream Processing: the distribution of the stream is changing over time and we need to have, at any point in time, an up-to-date model of the current situation. An examples of this challenging processing type is monitoring web traffic for intrusion detection, where the intrusion patterns may change over time. More recent data is more important, and data loses its importance over time. For algorithms under this computational model it is very important that they scale very well with data size as in theory the streams could go on forever. Memory bounds that are logarithmic in the number of instances seen over the stream sofar are considered reasonable. Furthermore, it is important that the algorithms require processing time which is independent from the number of instances already seen as otherwise the streaming algorithms would become increasingly slower. A popular technique to deal with online stream processing is using the window-based technique which considers a conceptual window of the most recent instances in the stream only. Continuously new instances enter the window while old, outdated instances leave the window. A window-based algorithm then continuously and incrementally maintains a summary of the contents of the window that allows to quickly answer analytical queries over the data.
2. Batch Processing: new data are processed in batches. This is for instance the case when new documents arrive that need to be indexed in an information retrieval context, or predictive models need to be updated. Often it is sufficient if the new data are processed continuously, but not necessarily immediately. This setting is far less challenging than the online stream processing model and is hence preferable if the application allows. Algorithms in this category are often incremental in the sense that they are able to incrementally update an existing model with a new batch of data.
3. One-pass algorithms: sometimes datasets to be processed are extremely large and disk-based. Given the relative efficiency of sequential data processing for secondary memory as compared to random access, algorithms that can process the data in one scan are preferable. Such algorithms are often termed streaming as well, since data is streamed from disk into the algorithm for processing. The requirements, however, are different from those of online or batch stream processing as there is not necessarily a temporal aspect in the data; there is no notion of more important recent tuples nor online results that need to be maintained.

It is important to note that distributed computing facilities such as offered by Hadoop [2], Spark [3], Flink [1], can only be part of the answer to the need expressed by these three categories of stream processing. First of all, distributed computing does not address the online aspect of the stream mining algorithms, although it may actually help to increase the throughput. For most batch processing algorithms it is conceivable that multiple batches could be treated in parallel, yet this would introduce an additional delay: handling n batches in parallel implies that batch 1 is still being processed while batch n is fully received, realistically putting a limitation on the scaling factors achievable. And last but not least, distributing computations over 1000 data processors can make processing at most 1000 times faster, and usually because of communication overhead the speedup is far less. In contrast, here in this paper we will exhibit several methods that achieve exponential performance gains with respect to memory consumption, albeit at the cost of having approximate results only.

Most streaming algorithms do not provide exact results as exact results often imply unrealistic lower complexity bounds. For many applications approximations are acceptable, although guarantees on the quality are required. Approximate results without guarantee should not be trusted any more than a gambler's "educated guess" or a manager's "gut feeling". Guarantees can come in many different forms; a method that finds items exceeding a minimal popularity threshold may guarantee that no popular items are missed, although maybe some items not meeting the threshold may be returned, or a method counting frequencies of events may have a guarantee on the maximal relative or absolute error on the reported frequency. A popular generalization of these guarantees are the so-called ϵ, δ-guarantees. An approximation algorithm A for a quantity q provides an ϵ, δ-guarantee if in at most $1-\delta$ of the cases, the quantity $A(D)$ computed by the algorithm for a dataset D differs at most ϵ from the true quantity $q(D)$; i.e., $P[|A(D) - q(D)| > \epsilon] < 1 - \delta$. Notice incidentally that this guarantee requires some notion of probability over all possible datasets and hence always has to come with an assumption regarding the distribution over possible datasets, such as a uniform prior over all possible datasets.

In this paper we will see three different building blocks that were, arguable subjectively, selected on the basis that at some point in the author's scientific career they proved to be an indispensable algorithmic tool to solve a scientific problem. The content of the paper can as such be seen as a tools offered to the reader to acquire and add into his or her data scientist's toolbox. The building blocks that will be provided are the following:

1. What is hot? Tracking heavy hitters: count which items exceed a given frequency threshold in a stream. We'll see *Karp's algorithm* [9] and *Lossy Counting* [11] as prototypical examples and show an application in blocking excessive network usage.
2. Extreme Counting: estimate the cardinality of a set. *Flajolet-Martin sketches* [7] and the related *HyperLogLog sketch* [6] are discussed. These sketches offer a very compact representation of sets that allow cardinality estimation of the sets. There are many applications in telecommunication, yet we will show an example use of the HyperLogLog sketch for estimating the neighborhood function of a social network graph.
3. Anyone like me? Similarity search: last but not least, we consider the case of similarity search in huge collections. Especially for high-dimensional data, indexing is extremely challenging. We show how *Locality Sensitive Hashing* [8] can help reduce complexity of similarity search tremendously. We show an application for plagiarism detection in which the detection of near duplicates of a given document decreases in complexity from hours to execute to subsecond response times.

We do not claim that our list of techniques is exhaustive in any sense. Many other very important building blocks exist. However, we are convinced that the set provided in this paper is a nice addition to any data scientist's professional toolbox. The individual blocks should not be seen as the endpoint, but rather as a set of blocks that can be freely adapted and combined, depending on need. For additional resources we refer the reader to the excellent books by *Aggrawal* on stream processing [4] and by *Leskovec et al.* on mining massive datasets [10].

2 Efficient Methods for Finding Heavy Hitters

The first building block we consider is efficiently finding heavy hitters. We assume that the stream consists of items from a fixed but potentially infinite universe. For example, keyword sequences entered in a search engine, IP addressed that are active in a network, webpages requested, etc. Items arrive continuously and may be repeating. A *heavy hitter* is an item whose frequency in the stream observed sofar exceeds a given relative frequency threshold. That is, suppose that the stream we observed sofar consists of the sequence of items

$$\mathcal{S} \;=\; \langle i_1, \ldots, i_N \rangle.$$

The relative frequency of an item a is defined as:

$$\mathit{freq}(a, \mathcal{S}) \;:=\; \frac{|\{j \mid i_j = a\}|}{N}.$$

The heavy hitters problem can now be stated as follows:

Heavy Hitters Problem: Given a threshold θ and a stream $\mathcal{S}$, give the set of items

$$\mathit{HH}(\mathcal{S}, \theta) \;:=\; \{a \mid \mathit{freq}(a, \mathcal{S}) \geq \theta\}.$$

Before we go into the approximation algorithms, let's first see how much memory would be required by an exact solution. First of all it is important to realize that in an exact solution we need to maintain counts for all items seen sofar, because the continuation of the stream in future is unknown and even an error on the count of the frequency of 1 will result in a wrong result. As such, we need to be able to distinguish any two situations in which the count of even a single item differs. Indeed, suppose $\theta = 0.5$, we have seen $N/2 + 1$ items in the stream, and the count of item a is 1. Then, if the next $N/2 - 1$ items are all a's, a should be in the output. On the other hand, if in the first $N/2 + 1$ items there are no occurrences of a, a should not be in the answer, even if all $N/2 - 1$ items are a's. Therefore, the internal state of the algorithm has to be different for these two cases, and we need to keep counters for each item that appeared in the stream. In worst case, memory consumption increases linearly with the size of the stream. If the number of different items is huge, this memory requirement is prohibitive.

To solve this problem, we will rely on approximation algorithms with much better memory usage. We will see two prototypical algorithms for *approximating* the set of heavy hitters. Both algorithms have the property that they produce a superset of the set of heavy hitters. Hence, they do not produce any false negatives, but may produce some false positives. The first algorithm by Karp uses maximally $\frac{1}{\theta}$ counters, and may produce up to $\frac{1}{\theta}$ false positives, while the second algorithm, called *Lossy Counting*, is parameterized by ϵ and has the guarantee that it produces no items b with $\mathit{freq}(b) < \theta - \epsilon$. Hence, the only false positives are in the range $[\theta - \epsilon, \theta[$ which likely represents still acceptable results given that the threshold θ is fuzzy anyway in most cases. The algorithm realizes this guarantee using only $\mathcal{O}\left(\frac{1}{\epsilon}\log(N\epsilon)\right)$ space in worst case.

2.1 Karp's Algorithm

Karp's algorithm [9] is based on the following simple observation: suppose we have a bag with N colored balls. There may be multiple balls of the same color. Now repeat the following procedure: as long as it is possible, remove from the bag sets of exactly k balls of all different color. This procedure is illustrated in Fig. 1. When our procedure ends, it is clear that there will be balls of at most $k - 1$ colors left. Furthermore, each color that appeared more than N/k times in the original bag will still be present. That is easy to verify: suppose there are $\lfloor N/k + 1 \rfloor$ red balls. In order to remove all red balls, there need to be $\lfloor N/k + 1 \rfloor$ sets of size k of balls of different color that were removed. But this is impossible as $k\lfloor N/k + 1 \rfloor > N$. Hence, if we want to find all colors that have a relative frequency of at least θ, then we can run the algorithm with $k = \lceil 1/\theta \rceil$. In this way we are guaranteed that in the final bag we will have all θ-frequent colors left. If we want to get rid of the false positives, we can run through our original bag for a second time, counting only the at most $k - 1$ different colors that were left in the bag after our procedure.

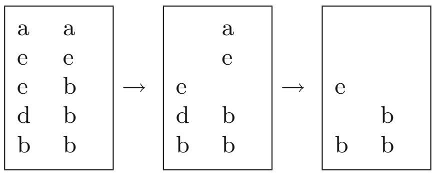


Fig. 1. Iteratively removing 3 different items from a bag; all element that had a relative frequency exceeding 1/3 will be left in the final result. In this case b and e are left. b indeed has a frequency exceeding 1/3, while e is a false positive. This procedure cannot have false negatives

The nice part of this observation is that it easily can be generalized to streaming data. Indeed, suppose we have a stream of items arriving. Each item can be considered a "color" and we need to retrieve all items that have a relative frequency exceeding θ. This can be realized by following the remove-k-different-colors procedure with $k = \lceil 1/\theta \rceil$. Because of the streaming aspect we do not have

New item	a	a	e	e	e	b	d	b	b	b
Updated counters	a:1	a:2	a:2 e:1	a:2 e:2	a:2 e:3	a:1 e:2	e:1	b:1 e:1	b:2 e:1	b:3 e:1

Fig. 2. Streaming version of the procedure in Fig. 1

any controle, however, over the order in which we need to treat the items/balls. Therefore we remember what we have seen sofar, until we reach k different items/colors. As soon as that happens, we throw out k different items. In order to remember what remains, it is easy to see that we need at most $k-1$ variables holding the items/colors we still have, plus $k-1$ counters holding how many times we still have each of them. Whenever a new ball/item arrives, we check if that color/item is already among the variables. If that is the case, we increase the associated counter. Otherwise, either we start a new counter if not all counters are in use yet, or we decrease all $k-1$ counters by 1 in order to reflect that we remove a k-tuple (one ball of each of the $k-1$ colors we already have plus the new color that just arrived). This leads to Karp's algorithm which is given in Algorithm 1 and illustrated in Fig. 2.

Notice that the update time of Algorithm 1 is $\mathcal{O}(k)$ in worst case, but in their paper, Karp et al. describe a data structure which allows processing items in constant time amortized.

Algorithm 1. Karp's algorithm

Input: Threshold ω, sequence $\mathcal{S}$ of items $i_1, \ldots, i_N$ arriving as a stream
Output: Superset of $HH(\mathcal{S}, \theta)$.

```
1: $k \leftarrow \lceil 1/\theta \rceil$
2: $L \leftarrow$ empty map
3: for each item $i$ arriving over $\mathcal{S}$ do
4:     if exists key $i$ in $L$ then
5:         $L[i] \leftarrow L[i] + 1$
6:     else
7:         if $|L| = k - 1$ then
8:             for key $k$ of $L$ do
9:                 $L[k] \leftarrow L[k] - 1$
10:                if $L[k] = 0$ then
11:                    Remove the element with key $k$ from $L$
12:                end if
13:            end for
14:        else
15:            $L[i] \leftarrow 1$
16:        end if
17:    end if
18: end for
19: return $\{k \mid k$ key in $L\}$
```

2.2 Lossy Counting

One of the disadvantages of Karp's algorithm is that it only allows for identifying a set of candidate heavy hitters, but does not provide any information regarding their frequencies. The *Lossy counting* algorithm [11] covered in this subsection, however, does allow for maintaining frequency information. Lossy counting is parameterized by ϵ. ϵ will be the bound on the maximal absolute error on the relative frequency that we guarantee. Lossy counting is based on the observation that we do not have to count every single occurrence of an item. As long as we can guarantee that the relative frequency of an item in the part of the stream in which it was not counted, does not exceed ϵ, the absolute error on the relative frequency will be at most ϵ. Indeed: suppose $\mathcal{S}$ can be divided into two disjoint sub-streams $\mathcal{S}_1$ and $\mathcal{S}_2$, and we do have the exact number of occurrences cnt_1 of item a in $\mathcal{S}_1$, and an upper bound of ϵ on the exact relative frequency $f_2 = \frac{cnt_2}{|\mathcal{S}_2|}$ of a in $\mathcal{S}_2$. Then the true relative frequency of a in $\mathcal{S}$ equals:

$$freq(a, \mathcal{S}) \;=\; \frac{cnt_1 + cnt_2}{|\mathcal{S}|} < \frac{cnt_1 + \epsilon\mathcal{S}_2}{|\mathcal{S}|} \leq \frac{cnt_1}{|\mathcal{S}|} + \epsilon.$$

This observation means that we can postpone counting any item that has a relative frequency below ϵ if we are fine with an absolute error of at most ϵ. This is exactly what Lossy Counting does: basically it counts everything, but from the moment on that it is noticed that an item's relative frequency in the window we are counting it, drops below ϵ we immediately stop counting it. If the item reappears, we start counting it again. In this way, at any point in time we can guarantee that any item that isn't counted has a relative frequency below ϵ. This principle is illustrated in Fig. 3.

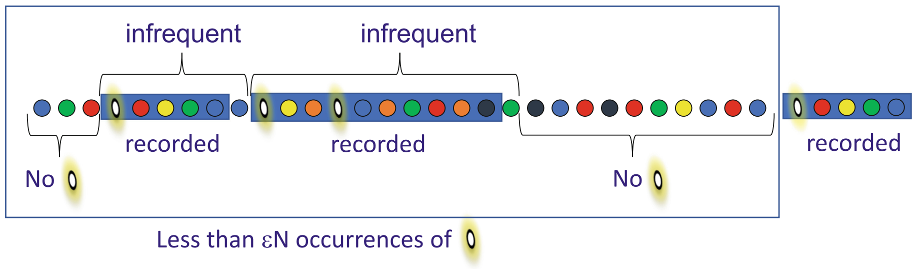


Fig. 3. Illustration of the Lossy Counting algorithm. The blue rectangles indicate periods in which the item was counted. If one takes the whole stream except for the last rectangle which is still open, then the item is ϵ-infrequent in that area (Color figure online)

The pseudo code of Lossy Counting is given in Algorithm 2. Notice that lines 10–14 constitute a potential bottleneck as we need to check after each item received from the stream, if the item is still frequent. We can, however, avoid

Algorithm 2. Lossy Counting algorithm

Input: Threshold ω, threshold ϵ, sequence $\mathcal{S}$ of items $i_1, \ldots, i_N$ arriving as a stream
Output: (i, f)-pairs such that $freq(i, \mathcal{S}) \in [f, f + \epsilon]$ and $f \geq \theta - epsilon$. The output contains a pair for each element of $HH(\mathcal{S}, \theta)$

```
 1: $Cnt \leftarrow$ empty map
 2: $Start \leftarrow$ empty map
 3: for each item $i_j$ arriving over $\mathcal{S}$ do
 4:     if exists key $i$ in $Cnt$ then
 5:         $Cnt[i_j] \leftarrow Cnt[i_j] + 1$
 6:     else
 7:         $Cnt[i_j] \leftarrow Cnt[i_j] + 1$
 8:         $Start[i_j] \leftarrow j$
 9:     end if
10:     for all keys $k$ of $Cnt$ do
11:         if $\frac{Cnt[k]}{j - Start[k] + 1} < \epsilon$ then
12:             Remove the elements with key $k$ from $Cnt$ and $Start$
13:         end if
14:     end for
15: end for
16: return $\left\{(i, f) \mid i \text{ key of } Cnt, f := \frac{Cnt[k]}{j - Start[k] + 1} > \theta - \epsilon\right\}$
```

this costly check by associating with every item an "expiration date"; that is: whenever we update the count of an item, we also compute after how many steps the item will no longer be ϵ-frequent unless it occurs again. This is easily achieved by finding the smallest number t such that:

$$\frac{Cnt[k]}{t - Start[i] + 1} < \epsilon.$$

The smallest t that satisfies this inequality is:

$$\left\lfloor \frac{Cnt[k]}{\epsilon} + start[i] \right\rfloor.$$

We can order the items for which a counter exists in a priority queue according to this number and update the number and position of the item in this queue every time the item occurs. Steps 10–14 then simply become evading all items having the current time as expiration date.

For didactic purposes, the Lossy Counting variant we explained in this subsection differs slightly from the one given by *Manku and Motwani* in [11]. The computational properties, intuitions and main ideas, however, were preserved.

Let us analyze the worst case memory consumption of the Lossy Counting algorithm. The analysis is illustrated in Fig. 4. The memory consumption is proportional to the number of items for which we are maintaining a counter. This number can be bounded by the observation that every item for which we maintain a counter, must be frequent in a suffix of the stream. To analyze how

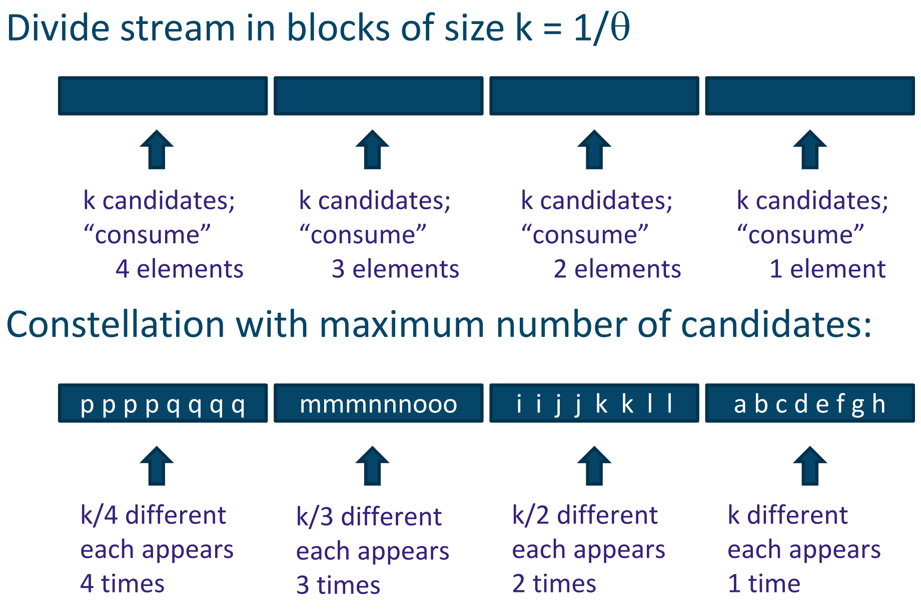

Divide stream in blocks of size k = 1/θ
k candidates; "consume" 4 elements
k candidates; "consume" 3 elements
k candidates; "consume" 2 elements
k candidates; "consume" 1 element
Constellation with maximum number of candidates:
p p p p q q q q
mmmnnnooo
i i j j k k l l
a b c d e f g h
k/4 different each appears 4 times
k/3 different each appears 3 times
k/2 different each appears 2 times
k different each appears 1 time

Fig. 4. Worst possible case w.r.t. memory consumption for the Lossy Counting algorithm

this affects the number of items being counted, we conceptually divide our stream in blocks of size $k = 1/\epsilon$. For an item to be supported, it needs to appear either:

- at least once in the last block;
- at least twice in the last two blocks;
- ...
- at least i times in the last i blocks;
- ...
- at least $N/k = N\epsilon$ times in the last N/k blocks; i.e., in the whole stream.

Let n_i denote the number of items that fall in the ith category above. The above observations translate into the following constraints:

- $n_1 \leq k$;
- $n_1 + 2n_2 \leq 2k$;
- ...
- $n_1 + 2n_2 + 3n_3 + \ldots + in_i \leq ik$;
- ...
- $n_1 + 2n_2 + 3n_3 + \ldots + N/kn_{N/k} \leq N$.

This number is maximized if $n_1 = k$, $n_2 = k/2$, ..., $n_i = k/i$, ... In that situation we obtain the following number of items for which we are maintaining a counter ($H(i)$ is the ith Harmonic number):

$$\sum_{i=1}^{N/k} k/i = kH(N/k) = \mathcal{O}(k \log(N/k)) = \mathcal{O}(1/\epsilon \log(\epsilon N)).$$

The memory requirements are hence logarithmic in the size of the stream *in worst case*. This worst-case, however, is a pathological case; in experiments with real-life data it was observed that the memory requirements are far less.

2.3 Applications of Heavy Hitters

Approximation algorithms for heavy hitters have many useful applications. Imagine for instance a network provider wanting to monitor its network for unreasonable bandwidth usage, and block or slow down the connection of any user using more than 0.1% of the bandwidth on any of its routers. To achieve this policy, the provider could install a lossy counter on each of its routers with θ set to 0.11% and ϵ to 0.01%. The lossy counter counts how many times IP-addresses participate in the traffic; for every packet the sender and receiver IP address is monitored. The lossy counters would catch all items with a frequency higher than 0.11% as well as some items with a frequency in the interval $[0.1\%, 0.11\%]$. Some of the users using between 0.1% and 0.11% of the bandwidth may remain unnoticed, but that can be acceptable. Installing such a lossy counter would require $10000 \log(N/10000))$ items to be stored in the absolute worst case. If the counters are reset every 1 billion packets, this would add up to at most 12K counters. That is quite acceptable for finding heavy hitters in up to a billion items.

3 Approximation Algorithms for Cardinality Counting over Streams

Another interesting building block in our toolbox is efficient cardinality counting. The setting is similar as in previous section; items are arriving one by one over a stream. This time, however, we are not interested in tracking the frequent items, but instead we want to know how many *different* items there are. Any exact solution must remember every item we have already seen. For large data collections this linear memory requirement may be unacceptable. Therefore, in this section we describe a sketching technique that maintains a succinct sketch of the stream that allows for accurately estimating the number of different items.

Cardinality Estimation Problem: Given a stream $\mathcal{S} = \langle i_1, \ldots, i_N \rangle$, give the cardinality of the set $\{i_1, \ldots, i_N\}$. That is, count the number of unique items in $\mathcal{S}$.

Reliable and efficient cardinality estimation has many applications such as counting the number of unique visitors to a website, or estimating the cardinality of projecting a relation that does not fit into memory onto a subset of its attributes without sorting.

3.1 Flajolet-Martin Sketches

The *Flajolet-Martin* sketch [7] is based on the observation that if we have a set of random numbers, the probability of observing a very high number increases with increasing size of the set. This observation is exploited as follows: suppose we have a randomly selected hash function h that hashes every element that can arrive over our stream to a random natural number. h is a function, so if an item a occurs repeatedly in the stream, it gets assigned the same natural number. Whenever an item a arrives over the stream, we apply the hash function to it, and record the size of the longest suffix of $h(a)$ consisting only of 0's. Let $\rho(a)$ denote this number. For example, if $h(a) = 0110010_b$, then $\rho(a) = 1$, as $h(a)$ ends with only 1 zero; for $h(b) = 01101011_b$, $\rho(b) = 0$, and for $h(c) = 1000000_b$, $\rho(c) = 6$. The more different elements we observe in the stream, the more likely it is that we have seen an element x with a high $\rho(x)$, and vice versa, the higher the highest $\rho(x)$ we observed, the more likely it is that we have seen many different elements. The Flajolet-Martin sketch is based on this principle, and records the highest number $\rho(x)$ we have observed over the stream. For this we only need to remember one number: the highest $\rho(x)$ observed sofar, and update this number whenever an element y arrives over the stream with an even higher $\rho(y)$. Let's use R to denote this highest observed $\rho(x)$.

If we have one element x, the probability that $\rho(x) = t$ for a given threshold t equals $1/2^{t+1}$. Indeed, half of the numbers ends with a 1, 1/4th with 10, 1/8th with 100 and so on. The probability that $\rho(x) < t$ equals $1/2 + 1/4 + \ldots + 1/2^t = 1 - 1/2^t$.

So, suppose we have a set S with N different items, what is the probability that R exceeds a threshold t? This equals

$$\begin{aligned} P[\max_{x\in S} \rho(x) \geq t] &= 1 - \prod_{x\in S} P[\rho(x) < t] && (1)\\ &= 1 - (1 - 1/2^t)^N && (2)\\ &= 1 - \left((1 - 1/2^t)^{2^t}\right)^{N/2^t} && (3)\\ &\approx 1 - e^{-N/2^t} && (4) \end{aligned}$$

Hence, we can conclude that if $N \gg 2^t$, the probability that $R \geq t$ is close to 0, and if $N \ll 2^t$, the probability that $R \geq t$ is close to 1. We can thus use 2^R as an estimate for the cardinality N. In the original Flajolet-Martin algorithm not the maximum number $\rho(x)$ observed is used, but instead the smallest number r such that no element a was observed with $\rho(a) = r$. Then the estimator $2^r/\phi$ where ϕ is a correction factor approximately equal to 0.77351 has to be used.

The variance of this estimation, however, can be high. Therefore we can use multiple independent hash functions to create multiple independent estimators and combine them. Averaging them, however, is very susceptible to outliers, while taking the median has the disadvantage of producing estimates which are always a power of 2. Therefore, a common solution is to group estimates, take the average for each group, and take the median of all averages. In this way we get an estimate which is less susceptible to outliers because of the median, and is not necessarily a power of 2 because of the averages.

3.2 HyperLogLog Sketch

A HyperLogLog (HLL) sketch [6] is another probabilistic data structure for approximately counting the number of distinct items in a stream. The HLL sketch approximates the cardinality with no more than $\mathcal{O}(\log(\log(N)))$ bits. The HLL sketch is an array with $\beta = 2^k$ cells $(c_1, \ldots, c_\beta)$, where k is a constant that controls the accuracy of the approximation. Initially all cells are 0. Every time an item x in the stream arrives, the HLL sketch is updated as follows: the item x is hashed deterministically to a positive number $h(x)$. The first k bits of this number determine the 0-based index of the cell in the HLL sketch that will be updated. We denote this number $\iota(x)$. For the remaining bits in $h(x)$, the position of the least significant bit that is 1 is computed. Notice that this is the $\rho(x) + 1$. If $\rho(x) + 1$ is larger than $c_{\iota(x)}$, $c_{\iota(x)}$ will be overwritten with $\rho(x) + 1$.

For example, suppose that we use a HLL sketch with $\beta = 2^2 = 4$ cells. Initially the sketch is empty:

0	0	0	0

Suppose now item a arrives with $h(a) = 1110100110010110_b$. The first 2 bits are used to determine $\iota(a) = 11_\beta = 3$. The rightmost 1 in the binary representation of $h(a)$ is in position 2, and hence c_3 becomes 2. Suppose that next items arrive in the stream with $(c_{\iota(x)}, \rho(x))$ equal to: $(c_1, 3)$, $(c_0, 7)$, $(c_2, 2)$, and $(c_1, 2)$, then the content of the sketch becomes:

7	3	2	2

Duplicate items will not change the summary. For a random element x, $P(\rho(x)+1 \geq \ell) = 2^{-\ell}$. Hence, if d different items have been hashed into cell c_ι, then $P(c_\iota \geq \ell) = 1 - (1 - 2^{-\ell})^d$. This probability depends on d, and all c_i's are independent. Based on a clever exploitation of these observations, *Flajolet et al.* [6] showed how to approximate the cardinality from the HLL sketch.

Last but not least, two HLL sketches can easily be combined into a single sketch by taking for each index the maximum of the values in that index of both sketches.

3.3 Applications: Estimating the Neighborhood Function

One application of the HLL sketch is approximating the so-called neighborhood function [5]. The algorithm we will see computes the neighborhood vector for all nodes in a graph at once. The neighborhood vector of a node is a vector $(n_1, n_2, \ldots)$ holding respectively the number of nodes at distance 1, distance 2, etc. The more densely connected a node is, the larger the numbers at the start of the vector will be. The neighborhood function is then the componentwise average of the neighborhood vector of the individual nodes; it gives the average number of neighbors at distance 1, 2, 3, etc. The neighborhood function is useful for instance to compute the effective diameter of a graph; that is: the average number of steps needed from a random node to reach a predefined fraction of the other nodes in the graph. For instance, the effective diameter for a ratio of

Algorithm 3. Neighborhood function

Input: Graph $G(V, E)$.
Output: Neighborhood function $(N_0, N_1, \ldots)$.

```
 1: for $v \in V$ do
 2:     $nv_0(v) \leftarrow \{\}$
 3: end for
 4: $N_0 \leftarrow 1$
 5: $i \leftarrow 0$
 6: while $N_i \neq 0$ do
 7:     $i \leftarrow i + 1$
 8:     for $v \in V$ do
 9:         $nv_i(v) \leftarrow nv_{i-1}(v)$
10:         for $\{v, w\} \in E$ do
11:             $nv_i(v) \leftarrow nv_i(v) \cup nv_{i-1}(w)$
12:             $nv_i(w) \leftarrow nv_i(w) \cup nv_{i-1}(v)$
13:         end for
14:     end for
15:     $N_i \leftarrow avg_{v \in V}(|nv_i(v)|) - N_{i-1}$
16: end while
17: return $(N_0, N_1, \ldots, N_{i-1})$
```

90% is the average number of steps needed from a random node to reach 90% of the other nodes. Using the neighborhood function, we can easily see from which point in the vector 90% of the other nodes are covered. For instance, if the neighborhood function is $(12, 1034, 12349, 234598, 987, 3)$, then the number of steps needed is 4, as more than 90% of the nodes are at distance 4 or less. The diameter of the graph we can get by observing the rightmost entry in the neighborhood function that is nonzero. We can see if the graph is connected by adding up all numbers and comparing it to the total number of nodes. The neighborhood function of a graph is hence a central property from which many other characteristics can be derived.

A straightforward algorithm for computing the neighborhood function is given in Algorithm 3. It is based on the observation that the nodes at distance i or less of node v can be gotten by taking the union of all nodes at distance $i - 1$ or less of its neighbors. Iteratively applying this principle gives subsequently the neighbors at distance 1, 2, 3, etc. of all nodes in the graph. This we continue as long as new nodes are being added for at least one vector.

The space complexity of this algorithm is $\mathcal{O}(|V|^2)$, as for every node we need to keep all other reachable nodes. This complexity, however, can easily be reduced using a HyperLogLog sketch to approximate the neighbors for all nodes. Instead of storing $nv_i(v)$ for each node, we store $HLL(nv_i(v))$. All operations we need in the algorithm are supported for the HLL; that is: taking unions (componentwise maximum of the two HLL sketches), and estimating the cardinality. In this way we get a much more efficient algorithm using only $\mathcal{O}(|V| b \log\log(|V|))$ space, where b is the number of buckets in the HyperLogLog sketches we keep. b depends only on the accuracy of the approximation, and not on the size of the graph.

4 Anyone Like Me? Similarity Search

In a big data context, high-dimensional similarity search is a very common problem. One of the most successful classification techniques is nearest neighbor, which requires quickly finding all closest points to a given query point. Although the setting is strictly speaking no longer a streaming setting, the *Locality Sensitive Hashing* technique [8] which we will cover in this section can usefully be applied whenever items arrive at a fast pace, and quickly need to be matched to a large database of instances to find similar items. Examples include face recognition where cameras are continuously producing a sequence of faces to be recognized in a large database, or image search where one quickly needs to produce images which are alike a given image. One example we will use to illustrate the locality sensitive hashing is that of plagiarism detection, where we assume that we have a large collection of documents and whenever a new document arrives we need to be able to quickly generate all near neighbors; that is: candidate original sources of a plagiarized document.

4.1 Similarity Measure: Jaccard

We will first introduce the locality sensitive hashing technique with the so-called *Jaccard similarity measure*. The Jaccard similarity measures distances between sets. These could be sets of words occurring in a document, sets of properties or visual clues of pictures, etc. Later we will see how the Locality Sensitive Hashing technique can be extended to other similarity measure, such as the Cosine Similarity measure for instance. Given two sets A and B, their similarity is defined as:

$$J(A,B) \ := \ \frac{|A \cap B|}{|A \cup B|}.$$

Suppose now that we order all elements of the universe from which the sets A and B are drawn. Let $r(a)$ denote the rank of element a in this order, $\min_r(A)$ is then defined as $\min\{r(a) \mid a \in A\}$. We now have the following property which will be key for the locality sensitive hashing technique we will develop in the next subsection.

Minranking Property: Let r be a random ranking function assigning a unique rank to all elements from a domain U. Let $A, B \subseteq U$. Now the following property holds:

$$P[\min_r(A) = \min_r(B)] \ = \ J(A,B).$$

The probability is assuming a uniform distribution over all ranking functions r. Indeed, every element in $A \cup B$ has the same probability of being the unique element in $A \cup B$ that has the minimal rank in $A \cup B$. Only if this element is in the intersection of A and B, $\min_r(A) = \min_r(B)$. The probability that the minimum over all elements in $A \cup B$ is reached in an element of $A \cap B$ equals $J(A, B)$.

Minrank Sketch of a Set: If we have multiple ranking functions $r_1, \ldots, r_k$, we can use these functions in order to get an estimate for $J(A, B)$ as follows: compute $\min_{r_i}(A)$ and $\min_{r_i}(B)$. Count for how many $i = 1, \ldots, k$, $\min_{r_i}(A) = \min_{r_i}(B)$. This gives us an estimate of $P[\min_r(A) = \min_r(B)] = J(A, B)$. The higher k, the more accurate our approximation will become.

There is one problem with the minrank sketch: a ranking function is very expensive to represent and store. Indeed: for a universe with n elements, there exist $n!$ rankings. Representing them requires on average $\log(n!)$ space. Therefore, instead of using a ranking function, we can use a hash function assigning numbers to the items in the range $[0, L]$ where L is significantly smaller than n. Such hash functions are usually easy to represent, and a popular choice for a hash function is $((ax + b)\ mod\ p)\ mod\ L$, where p is a prime number larger than $|U|$, and a and b are randomly drawn integers from $[1, p-1]$ and $[0, p-1]$ respectively. One problem with hash functions is that $P[\min_h(A) = \min_h(B)]$ is no longer equal to $J(A, B)$, but is slightly higher as there may be hash collisions. The probability of such a collision is, however, extremely low: let $a, b \in U$ be two different items. $P[h(a) = h(b)] = 1/L$. If L is sufficient large, this quantity can be neglected, the more since it will only cause problems if the collision happens between the smallest element in A and the smallest element in B. Unless the sets A and B are extremely large, in the order of L, we can replace ranking function by hash function in the above definitions. In this way we obtain the minhash sketch of a set A as $(\min_{h_1}(A), \ldots, \min_{h_k}(A))$. When comparing two sketches $(a_1, \ldots, a_k)$ and $(b_1, \ldots, b_k)$ we hence have the approximation $\frac{|\{i=1 \ldots k \mid a_i = b_i\}|}{k}$.

4.2 Locality-Sensitive Hash Functions

The name *Locality Sensitive Hashing* comes from the idea that in order to index high dimensional data, we need a way to hash instances into buckets in a way that is sensitive to *locality*. Locality here means that things that are similar should end up close to each other. In other words, we look for hash functions such that the probability that two instances are hashed into the same bucket, monotonically increases if their similarity increases. If we have such a family of independent hash functions at our disposition, there are principled ways to combine them into more powerful and useful hash functions. Our starting point is a family of independent hash functions $\mathcal{F}$ which is (s_1, p_1, s_2, p_2)-*sensitive* for a similarity measure *sim*. That means that for any $h \in \mathcal{F}$, $P[h(x) = h(y)]$ is non-decreasing with respect to $sim(x, y)$, $P[h(x) = h(y) | sim(x, y) < s_1] \leq p_1$, and $P[h(x) = h(y) | sim(x, y) \geq s_2] \geq p_2$. For the Jaccard-index we do have such a family of hash functions, namely the functions $\min_h(.)$ for random hash functions h. This family of functions is (s_1, s_1, s_2, s_2)-sensitive.

Suppose we have a set D of objects from universe U and we need to index them such that we can quickly answer the following (at this point still informal) near-duplicate query: given a query point q from universe U, give all objects d in D such that $sim(q,d)$ is high. If we have a family of (s_1,p_1,s_2,p_2)-sensitive hash functions, we could index the objects in D as follows: pick a hash function h from $\mathcal{F}$ and divide D into buckets according to the hash value given by h; that is: for each hash value v in $h(D)$, there is a bucket $D[v] := \{d \in D \mid h(d) = v\}$. If we now need a near duplicate of a query point $q \in U$, we will only check the elements d in the bucket $D[h(q)]$. Since h is from a (s_1,p_1,s_2,p_2)-sensitive family, we are guaranteed that if an object $d \in D$ has $sim(d,q) \geq s_2$, then $P[d \in D[h(q)]] \geq p_2$. On the other hand if the similarity $sim(q,d)$ is lower than s_1, the chance of finding d in the same bucket as q decreases to p_1. If the documents with similarity exceeding s_2 represent the ones we need to retrieve, the ones in the interval $[s_1,s_2]$ are acceptable but not necessary in the result, and the ones with similarity lower than s_1 represent documents that shouldn't be in the answer, then p_1 can be interpreted as the *False Positive Ratio*; that is, the probability that a negative example is a FP, and p_2 the probability that a positive example is correctly classified and hence a TP; i.e., the *True Positive Ratio*.

Often, however, the sensitivities (s_1,p_1,s_2,p_2) are insufficient for applications. For instance, if we use minhashing for finding plagiarized texts where documents are represented as the set of words they contain, the sensitivity we get is (s_1,s_1,s_2,s_2) for any pair of numbers $s_1 < s_2$. So, if we consider a text plagiarized if the similarity is above 90% and not plagiarized if the similarity is less than 80%, then the indexing system proposed above has a guaranteed true positive rate of only 90% and a false positive rate of up to 80%. This is clearly not acceptable. Fortunately, there exist techniques for "boosting" a family of sensitive hash functions in order to achieve much better computational properties. This boosting technique can be applied on any family of hash-functions that are (s_1,p_1,s_2,p_2)-sensitive for a given similarity function, as long as the hash functions are independent, as we will see next.

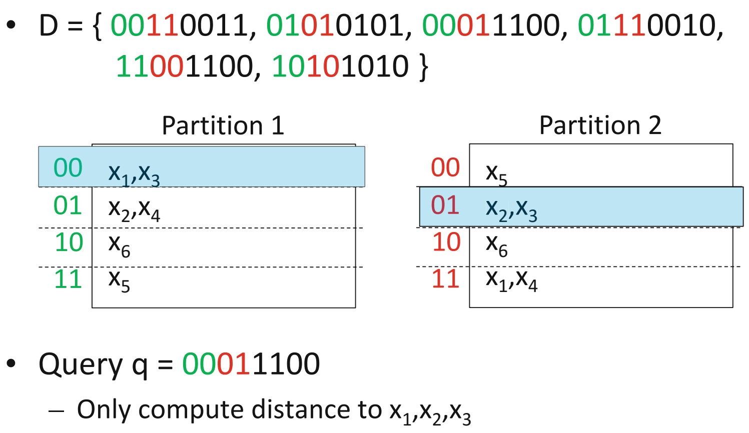
4.3 Combining Locality-Sensitive Hash Functions

Instead of creating an index based on one hash function, we can combine up to k hash functions $h_1,\ldots,h_k$ as follows: assign each document x in our collection D to a bucket $D[h_1(x),h_2(x),\ldots,h_k(x)]$ where $D[v_1,\ldots,v_k] := \{d \in D \mid h_1(d) = v_1,\ldots,h_k(d) = v_k\}$. If a query q comes, we check the similarity with the documents in bucket $D[h_1(q),\ldots,h_k(q)]$ only. We now get:

$$P[q \in D[h_1(d),\ldots,h_k(d)]] = P[\forall i = 1\ldots k : h_i(q) = h_i(d)] \tag{5}$$

$$= \prod_{i=1}^{k} P[h_i(q) = h_i(d)] \tag{6}$$

Hence, the combination of hash functions is (s_1,p_1^k,s_2,p_2^k)-sensitive. In this way we can reduce the number of false positives tremendously; for $p_1 = 80\%$ as in our example above, for $k = 10$, the false positive rate decreases from 80% to $(80\%)^{10}$, which is less than 11%!



• D = { 00110011, 01010101, 00011100, 01110010, 11001100, 10101010 }
Partition 1
00 x1,x3
01 x2,x4
10 x6
11 x5
Partition 2
00 x5
01 x2,x3
10 x6
11 x1,x4
• Query q = 00011100
– Only compute distance to x1,x2,x3

Fig. 5. LSH-Index based on a (2, 2)-scheme. For illustrative purposes a simple hamming distance between 0–1 vectors is chosen, defined as the fraction of entries on which the vectors correspond. The first index is based on the first two entries in the vector, and the second index on the next two entries. A query point is compared to all vectors in the 2 buckets in which the query point is hashed (one for the first index, one for the second)

The true positive rate, however, decreases as well: from 90% to around 35%. To counter this problem, however, we can create multiple indices for different sets of hash functions: $H^1 = (h_1^1, \ldots, h_k^1)$, ..., $H^\ell = (h_1^\ell, \ldots, h_k^\ell)$. For each $j = 1 \ldots \ell$ we create an independent index for the documents. Each document $d \in D$ gets assigned to ℓ buckets: $D^1[H^1(d)]$, ..., $D^\ell[H^\ell(d)]$, where $H^i(d)$ is shorthand for the composite tuple $(h_1^i(d), \ldots, h_k^i(d))$. If a query q comes, we will compare q to all documents in $D^1[H^1(q)] \cup \ldots \cup D^\ell[H^\ell(q)]$. This way of indexing data is illustrated in Fig. 5.

Suppose that $P[h(x) = h(y)] = p$ for al given pair of documents x, y and a random hash function from a given family of hash functions. Then the probability that x and y share at least one bucket in the ℓ indices under our (k, ℓ)-scheme equals:

$$\begin{aligned} P[x \text{ and } y \text{ share at least one bucket}] &= 1 - P[x \text{ and } y \text{ share no bucket}] && (7)\\ &= 1 - \prod_{j=1}^{\ell} P[H^j(x) \neq H^j(y)] && (8)\\ &= 1 - \prod_{j=1}^{\ell} (1 - P[H^j(x) = H^j(y)]) && (9)\\ &= 1 - \prod_{j=1}^{\ell} (1 - p^k) && (10)\\ &= 1 - (1 - p^k)^\ell && (11) \end{aligned}$$

Hence our (k, l)-scheme is $(s_1, 1-(1-p_1^k)^\ell, s_2, 1-(1-p_2^k)^\ell)$-sensitive. As long as our family of hash functions is large enough to allow for $k\ell$ hash functions, we can achieve any desired precision (s_1, P_1, s_2, P_2) for our indexing scheme by solving the following system of equations for l and k:

$$\begin{cases} P_1 = 1-(1-p_1^k)^\ell \\ P_2 = 1-(1-p_2^k)^\ell \end{cases}$$

Figure 6 plots some examples in which the similarity of two documents is plotted against the probability that they share at least one bucket, for the Jaccard similarity measure using the minhash family. Recall that $P[h(x) = h(y)] \approx J(x, y)$, which makes the relation between the similarity of two documents x and y and their probability of sharing a bucked straightforward: $P[x \text{ shares bucket with } y] = 1-(1-J(x, y)^k)^\ell$.

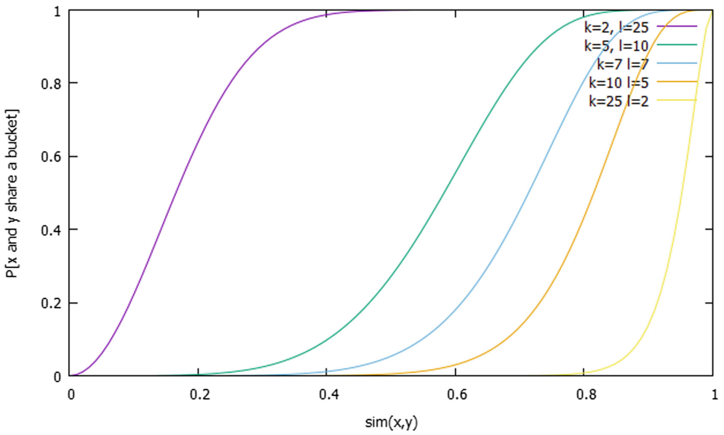


Fig. 6. Relation between the similarity and probability of being in the same bucket under different (k, ℓ)-hashing schemes (LSH for Jaccard using minhash)

4.4 LSH for Cosine Similarity

Locality-sensitive hashing works not only for the Jaccard-index; any similarity measure for which we can find an appropriate family of hash functions for, can benefit from this framework. We will illustrate this principle with one more example: the *cosine similarity measure*. The universe from which our documents and queries come are N-dimensional vectors of non-negative numbers, for instance TF.IDF-vectors for text documents. Given two vectors, $\mathbf{x} = (x_1, \ldots, x_N)$ and $\mathbf{y} = (y_1, \ldots, y_N)$, the *cosine similarity* between them is defined as $\frac{\mathbf{x}\cdot\mathbf{y}}{|\mathbf{x}||\mathbf{y}|}$, where $\cdot$ is the scalar product and $|.|$ the $l2$-norm. The cosine similarity measure thanks its name to the fact that it equals the cosine of the angle formed by the two vectors.

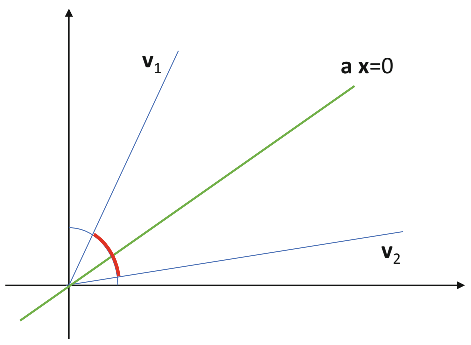


Fig. 7. Random hyperplane separating 2 2D vectors (Color figure online)

It is this property that will be exploited by the hash functions we will design: every hash function we consider is associated with a random hyperplane through the origin. All points on one side of the hyperplane get assigned 0, and all points on the other side get assigned 1. That is, if the equation of the hyperplane is $\mathbf{a} \cdot \mathbf{x} = 0$, then the hash function $h_\mathbf{a}$ we consider is defined as $h_\mathbf{a}(\mathbf{x}) = sign(\mathbf{a} \cdot \mathbf{x})$. It can be shown that if we chose the hyperplane by drawing each of the components of $\mathbf{a}$ from an independent standard normal distribution, then the probability that we separate two vectors by the random hyperplane $\mathbf{a} \cdot \mathbf{x} = 0$ is proportional to the angle between the two vectors. This situation is depicted for 2 dimensions in Fig. 7. The green line represents the separating hyperplane between two vectors. The plane, in 2D a line, separates the vectors if its slope is in $[\alpha_1, \alpha_2]$ where α_i is the angle between the horizontal axis and the vector $\mathbf{v}_i$. The probability that this happens, if all slopes are equally likely, is $\frac{\alpha}{\pi/2}$, where α is the angle between $\mathbf{v}_1$ and $\mathbf{v}_2$; i.e., $\alpha = |\alpha_2 - \alpha_1|$. Hence, we get:

$$P[h_\mathbf{a}(\mathbf{x}) = h_\mathbf{a}(\mathbf{y})] = 1 - \frac{\alpha}{\pi/2} \tag{12}$$

$$= 1 - \frac{\arccos(sim(\mathbf{x}, \mathbf{y}))}{\pi/2} \tag{13}$$

As $\arccos(x)$ is monotonically decreasing for $x \in [0, 1]$, the probability that two elements share a bucket is monotonically increasing with the cosine similarity between the two elements, which is exactly the LSH property we need to use the technique of last subsection. We can again combine the hash functions into ℓ groups of k independent hashes. In this way we get an index where two elements share at least one bucket with a probability of:

$$1 - \left(1 - \left(1 - \frac{\arccos(sim(\mathbf{x}, \mathbf{y}))}{\pi/2}\right)^k\right)^\ell$$

This probability in function of the cosine similarity between two documents is depicted in Fig. 8.

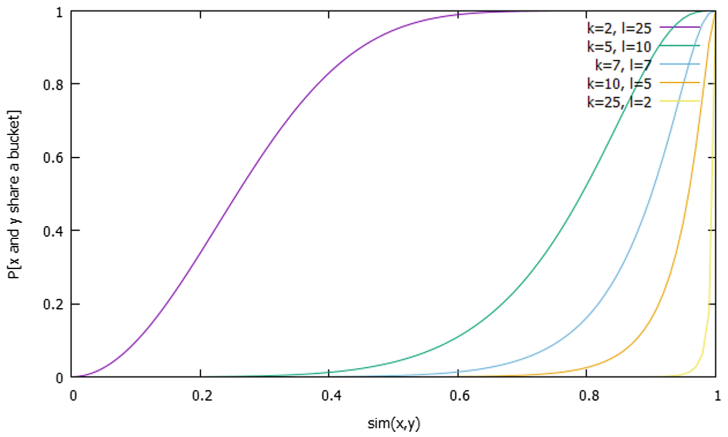


Fig. 8. Relation between the similarity and probability of being in the same bucket under different (k, ℓ)-hashing schemes (LSH for cosine similarity)

4.5 Application: Plagiarism Detection

One potential application of LSH is plagiarism detection. We will illustrate this application with a collection of 23M Wikipedia documents. Each document consists of one chapter of a Wikipedia page. The pages are preprocessed as follows: first the pages are decomposed into their 4-shingles; that is: each page is represented by the set of all 4 consecutive words in the text. For instance, if the document is "Royal Antwerp Football Club is the number 1 team in Belgium", then the representation becomes: {"Royal Antwerp Football Club", "Antwerp Football Club is", "Football Club is the", ..., "1 team in Belgium"}. Subsequently, to reduce space, all shingles are hashed into a unique number. After that, two documents are compared using the Jaccard similarity. Via minhashing we create 49 independent (s_1, s_1, s_2, s_2)-sensitive hash functions. These are combined into an LSH-index using a $(7, 7)$-scheme.

In order to get an idea of the overall distribution of the similarities between two random documents in the collection, we sampled a subset of 1000 documents. For these 1000 documents, the similarity between all pairs is measured. These numbers, extrapolated to the whole collection, are plotted as a histogram in Fig. 9. As can be seen in this histogram, the vast majority of pairs of documents has low similarity (notice incidentally that the scale on the vertical axis is logarithmic). Only about 100 document pairs have a similarity higher than 90%, and there is a gap between 70% and 90%. This indicates that, as to be expected, there is some duplicate content in Wikipedia, which scores a similarity higher than 90%, while the normal pairs score at most around 70% with a vast majority of pairs of documents having similarities around 10%, 20%, 30%. This is very good news for the application of LSH. Indeed, it indicates that any indexing scheme which is $(30\%, p_1, 90\%, p_2)$-sensitive with low p_1 and high p_2 will perform very well. The large gap between $s_1 = 30\%$ and $s_2 = 90\%$ means

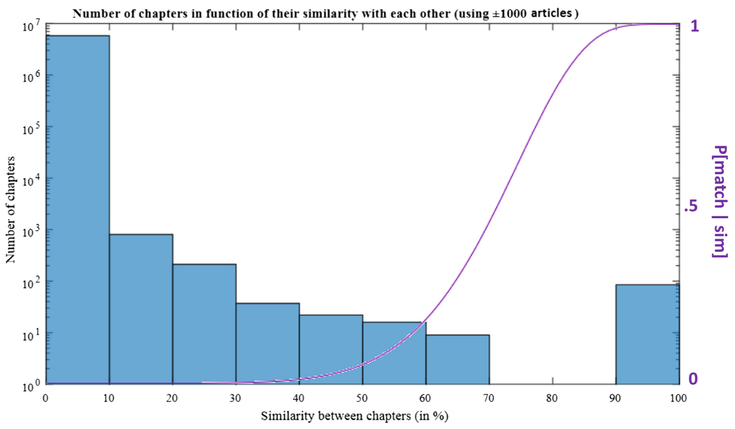


Fig. 9. Histogram representing the extrapolated numbers of pairs of documents with respect to similarity (binned per 10%; logscale). Overlayed is the probability that two documents share a bucket in a (7, 7) LSH index (normal scale)

that we will not need a lot of hash functions. In Fig. 9, the histogram has been overlayed with the probability of sharing a bucket for a (7, 7)-scheme. As can be seen, this indexing scheme should perform very well; for most of the pairs of documents the similarity is low, and at the same time the probability that those pairs end up in the same bucket is extremely low. Hence, the number of false positives will be very low relative to the total number of pairs. On the other hand, for the highly similar documents, the similarity is high and the probability of those pairs ending up in the same bucket is nearly 1. So, not too many false negatives to be expected either.

Because of the low number of candidates that will have to be tested, the time of finding duplicates in an experiment with this setup went down from over 6 hours to compare a query document to all documents in the collection, to less than a second, all on commodity hardware. The exact run times depend on the exact characteristics of the setup and the similarity distribution among the documents, but in this particular case a speedup of over 20000 times could be observed using LSH, with virtually no false negatives.

5 Conclusions

In this overview we reviewed three techniques which can come in handy when working with large amounts of data. First of all, we looked into fast and efficient algorithms for recognizing *heavy hitters*; that is: highly frequent items, in a stream. Then we went into even more efficient sketches for streaming data which allow for cardinality estimation of a stream. Last but not least, we reviewed the *Locality Sensitive Hashing* technique for similarity search in large data collections. These techniques, and combinations thereof are frequently handy when working with large data collections, and are a nice addition to a data scientists toolbox. A number of applications we gave were: finding users in an IP network using an excessively large fraction of the bandwidth, computing the neighborhood function of a graph, and plagiarism detection.

References

1. Apache flink. https://flink.apache.org/
2. Apache hadoop. http://hadoop.apache.org
3. Apache spark. https://spark.apache.org/
4. Aggarwal, C.C.: Data Streams. ADBS, vol. 31. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-47534-9
5. Boldi, P., Rosa, M., Vigna, S.: HyperANF: approximating the neighbourhood function of very large graphs on a budget. In: Proceedings of the 20th International Conference on World Wide Web, pp. 625–634. ACM (2011)
6. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: AofA: Analysis of Algorithms. Discrete Mathematics and Theoretical Computer Science, pp. 137–156 (2007)
7. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
8. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 604–613. ACM (1998)
9. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. (TODS) **28**(1), 51–55 (2003)
10. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2014)
11. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 346–357. VLDB Endowment (2002)

Let's Open the Black Box of Deep Learning!

Jordi Vitrià^(✉)

Departament de Matemàtiques i Informàtica,
Universitat de Barcelona, Barcelona, Spain
jordi.vitria@ub.edu

Abstract. Deep learning is one of the fastest growing areas of machine learning and a hot topic in both academia and industry. This tutorial tries to figure out what are the real mechanisms that make this technique a breakthrough with respect to the past. To this end, we will review what is a neural network, how we can learn its parameters by using observational data, some of the most common architectures (CNN, LSTM, etc.) and some of the tricks that have been developed during the last years.

Keywords: Deep learning · Automatic differentiation · Optimization

1 On Deep Learning and Black Boxes

Neural Networks is a biologically inspired programming paradigm which enables a computer to learn from observational data. The origins of neural networks can be traced, at least, until 1949, when Hebb proposed a simple learning principle for neurons: 'fire together, wire together' [1]. Later, in 1957, Rosenblatt invented the first learning machine: the perceptron [2]. These simple models were further developed during more than 30 years until 1988, when the backpropagation algorithm was proposed by Rumelhart et al. [3]. This algorithm was the first computational approach for training modern multilayer neural networks.

In spite of the fact that the discovery of the backpropagation algorithm started a rich research field leading to interesting models such as convolutional neural networks or recurrent neural networks, its severe practical limitations at that time provoked after a few years what is known as the neural net winter, a period of reduced funding and interest in the field. The neural net winter ended in 2012, when a neural network called AlexNet [4] competed in the ImageNet Large Scale Visual Recognition Challenge. The network achieved a top-5 error of 15.3%, more than 10.8% points ahead of the runner up. These results, that where unexpectedly good, led to the current deep learning boom.

Deep Learning is a powerful set of techniques (and tricks) for learning in neural networks with a high number of layers. Its notoriety builds upon the fact that it currently provides the best solutions to many problems in image recognition, speech recognition, and natural language processing [5], but at the

E. Zimányi (Ed.): eBISS 2017, LNBIP 324, pp. 134–153, 2018.
https://doi.org/10.1007/978-3-319-96655-7_6

same time some critics describe deep learning as a black box where data goes in and predictions come out without any kind of transparency.

The *black box* criticism can be approached from many sides, such as discussing model explainability, predictability or transparency, but in this paper we will try to open the black box by explaining some of the inner workings of deep models. The success of deep learning does not stand on a brilliant formula neither in a set of heuristic rules implemented in code, but in a carefully assembled software machinery that uses in a very smart way several strategies from different fields, from the chain rule of Calculus to dynamic graph computation or efficient matrix multiplication.

In a few words, this chapter tries to shed light on this cryptic but precise sentence: deep learning can be defined as a methodology to train large and highly complex models with deeply cascaded non-linearities by using automatic differentiation and several computational tricks. I hope that we will be able to fully appreciate what G.Hinton, one of the fathers of deep learning, said in a recent speech: 'All we have really discovered so far is that discriminative training using stochastic gradient descend works far better than any reasonable person would have expected'.

2 How to Learn from Data?

In general, *Learning from Data* is a scientific discipline that is concerned with the design and development of algorithms that allow computers to infer, from data, a model that allows a compact representation of raw data and/or good generalization abilities. In the former case we are talking about non supervised learning. In the later, supervised learning.

This is nowadays an important technology because it enables computational systems to improve their performance with experience accumulated from the observed data in real world scenarios.

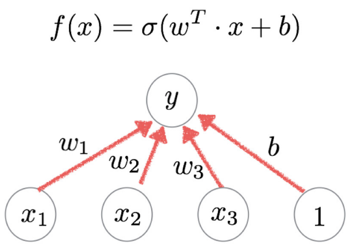
Neural nets are a specific method for learning from data, a method that is based on a very simple element, the *neuron unit.* A neuron unit is a mathematical function of this kind:

$$f(\mathbf{x}, \mathbf{w}, b) = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b) \tag{1}$$

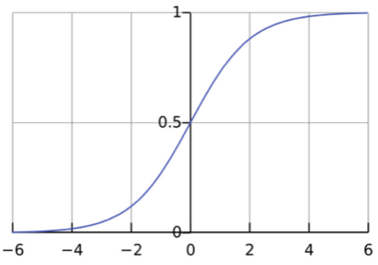
where $\mathbf{x}$ represents an input element in vector form, $\mathbf{w}$ is a vector of weights, σ is a non-linear function and b a scalar value. $(\mathbf{w}, b)$ are called the parameters of the function. The output of this function is called the *activation* of the neuron. Figure 1a shows the most common graphical representation of a neuron.

Regarding the non-linear function, historically the most common one was the Sigmoid function (see Fig. 1b), but nowadays there are several alternatives that are supposed to be better suited to learning from data, such as ReLU [6] and variants.

Simple neurons can be organized in larger structures by applying to the same data vector different sets of weights, forming what is called a *layer*, and by stacking layers one on top of the output of the other. In Fig. 2 we can see



(a) Graphical representation of a neuron unit



$\sigma(x) = \frac{1}{(1+e^{-x})}$

(b) Sigmoid function

Fig. 1. 1-layer neural network

a 2-layer neural network that gets as input 4-dimensional data and produces 2-dimensional outcomes based on the activations of its neurons. It is important to notice that a multilayer neural network can be seen as a composition of matrix products (matrices represent weights) and non-linear function activations. For the case of the network represented in Fig. 2, the outcome is:

$$\mathbf{y} = \sigma\Big(W^1 \sigma\Big(W^0 \mathbf{x} + \mathbf{b}^0\Big) + \mathbf{b}^1\Big) \tag{2}$$

where σ represents a vectorial version of the sigmoid function and W^i are the weights of each layer in matrix form.

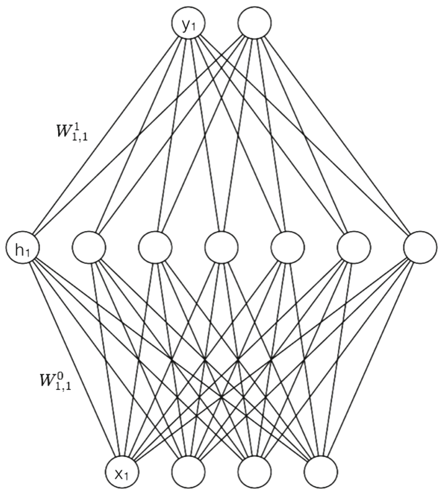


Fig. 2. Neural network

What is interesting about this kind of structures is that it has been showed that even a neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function of $\mathbf{R}^n$ [7]. This fact makes neural networks a sound candidate to implement learning from data methods. The question is then: how to find the optimal parameters, $\mathbf{w} = (W^i, \mathbf{b})$, to approximate a function that is implicitly defined by a set of samples $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$?

From a technical point of view, not only neural networks but most of the algorithms that have been proposed to infer models from large data sets are based on the iterative solution of a mathematical problem that involves data and a mathematical model. If there was an analytic solution to the problem, this should be the adopted one, but this is not the case for most of the cases. The techniques that have been designed to tackle these problems are grouped under a field that is called optimization. The most important technique for solving optimization problems is *gradient descend*.

2.1 Learning from Data

Let's consider the supervised learning problem from an optimization point of view. When learning a model from data the most common scenario is composed of the following elements:

- A dataset $(\mathbf{x}, y)$ of n examples. For example, $(\mathbf{x}, y)$ can represent:
  - $\mathbf{x}$: the behavior of a game player; y: monthly payments.
  - $\mathbf{x}$: sensor data about your car engine; y: probability of engine error.
  - $\mathbf{x}$: financial data of a bank customer; y: customer rating.

  If y is a real value, the problem we are trying to solve is called a *regression* problem. If y is binary or categorical, it is called a *classification* problem.
- A target function $f_{(\mathbf{x},y)}(\mathbf{w})$, that we want to minimize, representing the discrepancy between our data and the model we want to fit.
- A model M that is represented by a set of parameters $\mathbf{w}$.
- The gradient of the target function, $\nabla f_{(\mathbf{x},y)}(\mathbf{w})$ with respect to model parameters.

In the case of regression $f_{(\mathbf{x},y)}(\mathbf{w})$ represents the errors from a data representation model M. Fitting a model can be defined as finding the optimal parameters $\mathbf{w}$ that minimize the following expression:

$$f_{(\mathbf{x},y)}(\mathbf{w}) = \frac{1}{n} \sum_i (y_i - M(\mathbf{x}_i, \mathbf{w}))^2 \tag{3}$$

Alternative regression and classification problems can be defined by considering different formulations to measure the errors from a data representation model. These formulations are known as the *Loss Function* of the problem.

2.2 Gradient Descend

Let's suppose that we have a function $f(w) : \mathbf{R} \rightarrow \mathbf{R}$ and that our objective is to find the argument w that minimizes this function (for maximization, consider $-f(w)$). To this end, the critical concept is the *derivative.*

The derivative of f of a variable w, $f'(w)$ or $\frac{\mathrm{d}f}{\mathrm{d}w}$, is a measure of the rate at which the value of the function changes with respect to the change of the variable. It is defined as the following limit:

$$f'(w) = \lim_{h \to 0} \frac{f(w+h) - f(w)}{h}$$

The derivative specifies how to scale a small change in the input in order to obtain the corresponding change in the output. Knowing the value of $f(w)$ at a point w, this allows to predict the value of the function in a neighboring point:

$$f(w+h) \approx f(w) + hf'(w)$$

Then, by following these steps we can decrease the value of the function:

1. Start from a random w^0 value.
2. Compute the derivative $f'(w) = \lim_{h \to 0} \frac{f(w+h)-f(w)}{h}$.
3. Walk small steps in the opposite direction of the derivative, $w^{i+1} = w^i - hf'(w^i)$, because we know that $f(w - hf'(w))$ is less than $f(w)$ for small enough h, until $f'(w) \approx 0$.

The search for the minima ends when the derivative is zero because we have no more information about which direction to move. w is called a critical or stationary point of $f(w)$ if $f'(w) = 0$.

All extrema points (maxima/minima) are critical points because $f(w)$ is lower/higher than at all neighboring points. But these are not the only critical points: there is a third class of critical points called *saddle points.* Saddle points are points that have partial derivatives equal to zero but at which the function has neither a maximum nor a minimum value.

If f is a *convex function*, when the derivative is zero this should be the extremum of our function. In other cases it could be a local minimum/maximum or a saddle point.

The recipe we have proposed to find the minima of a function is based on the numerical computation of derivatives. There are two problems with the computation of derivatives by using numerical methods:

– It is approximate: it depends on a parameter h that cannot be tuned in advance to get a given precision.
– It is very slow to evaluate: there are two function evaluations at each step: $f(w+h), f(w)$. If we need to evaluate complex functions, involving millions of parameters, a lot of times, this problem becomes a real obstacle.

The classical solution to this problem is the use the of the analytic derivative $f'(w)$, which involves only one function evaluation, but this solution is only feasible to those problems where $f'(w)$ can be analytically derived.

2.3 From Derivatives to Gradient

Let's now consider a n-dimensional function $f(\mathbf{w}) : \mathbf{R}^n \to \mathbf{R}$. For example:

$$f(\mathbf{w}) = \sum_{i=1}^{n} w_i^2$$

As in the previous section, our objective is to find the argument $\mathbf{w}$ that minimizes this function.

The *gradient* of f is the vector whose components are the n partial derivatives of f.

$$\nabla f = \Big(\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_n}\Big)$$

It is thus a vector-valued function. The gradient is an interesting function because it plays the same role as the derivative in the case of scalar functions: it can be shown that it points in the direction of the greatest rate of increase of the function. Then, we can follow this steps to minimize the function:

- Start from a random $\mathbf{w}$ vector.
- Compute the gradient vector ∇f.
- Walk small steps in the opposite direction of the gradient vector.

It is important to be aware that this gradient computation is very expensive when using numerical derivatives: if $\mathbf{w}$ has dimension n, we have to evaluate f at $2 * n$ points.

2.4 Stochastic Gradient Descend

Taking for granted that we will apply an iterative method to minimize a loss function at every step, we must consider the cost of this strategy. The inspection of Eq. (3) shows that we must evaluate the discrepancy between the prediction of our model and a data element for the whole dataset $(\mathbf{x}_i, y_i)$ at every optimization step. If the dataset is large, this strategy is too costly. In this case we will use a strategy called *Stochastic Gradient Descend* (SGD).

Stochastic Gradient Descend is based on the fact that the cost function is additive: it is computed by adding single discrepancy terms between data samples and model predictions. Then, it can be shown [11] that we can compute an estimate, maybe noisy, of the gradient (and move towards the minimum) by using only one data sample (or a small data sample). Then, we can probably find a minimum of $f(\mathbf{w})$ by iterating this noisy gradient estimation over the dataset. A full iteration of over the dataset is called an *epoch*. To ensure convergence properties, during each epoch, data must be used in a random order.

If we apply this method we have some theoretical guarantees [11] to find a good minimum:

– SGD essentially uses the inaccurate gradient per iteration. What is the cost by using approximate gradient? The answer is that the convergence rate is slower than the gradient descent algorithm.
– The convergence of SGD has been analyzed using the theories of convex minimization and of stochastic approximation: it converges almost surely to a global minimum when the objective function is convex or pseudoconvex, and otherwise converges almost surely to a local minimum.

During the last years there have been proposed several improved stochastic gradient descend algorithms, such as Momentum-SGD [8], Adagrad [9] or Adam [10], but a discussion about these methods is out of the scope of this tutorial.

2.5 Training Strategies

In Python-like code, a standard Gradient Descend method that considers the whole dataset at each iteration looks like this:

```
nb_epochs = 100
for i in range(nb_epochs):
    grad = evaluate_gradient(target_f, data, w)
    w = w - learning_rate * grad
```

For a pre-defined number of epochs, we first compute the gradient vector of the target function for the whole dataset w.r.t. our parameter vector and update the parameters of the function.

In contrast, Stochastic Gradient Descent performs a parameter update for each training example and label:

```
nb_epochs = 100
for i in range(nb_epochs):
    np.random.shuffle(data)
    for sample in data:
        grad = evaluate_gradient(target_f, sample, w)
        w = w - learning_rate * grad
```

Finally, we can consider an hybrid technique, *Mini-batch Gradient Descent*, that takes the best of both worlds and performs an update for every small subset of m training examples:

```
nb_epochs = 100
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    grad = evaluate_gradient(target_f, batch, w)
    w = w - learning_rate * grad
```

Minibatch SGD has the advantage that it works with a slightly less noisy estimate of the gradient. However, as the minibatch size increases, the number of updates done per computation decreases (eventually it becomes very inefficient, like batch gradient descent).

There is an optimal trade-off (in terms of computational efficiency) that may vary depending on the data distribution and the particulars of the class of function considered, as well as how computations are implemented.

2.6 Loss Functions

To learn from data we must face the definition of the function that evaluates the fitting of our model to data, the *loss functions*. Loss functions specifically represent the price paid for inaccuracy of predictions in classification/regression problems: $L(y, M(\mathbf{x}, \mathbf{w})) = \frac{1}{n}\sum_i \ell(y_i, M(\mathbf{x}_i, \mathbf{w}))$.

In regression problems, the most common loss function is the *square loss* function:

$$L(y, M(\mathbf{x}, \mathbf{w})) = \frac{1}{n}\sum_i (y_i - M(\mathbf{x}_i, \mathbf{w}))^2$$

In classification this function could be the *zero-one loss*, that is, $\ell(y_i, M(\mathbf{x}_i, \mathbf{w}))$ is zero when $y_i = M(\mathbf{x}_i, \mathbf{w})$ and one otherwise. This function is discontinuous with flat regions and is thus impossible to optimize using gradient-based methods. For this reason it is usual to consider a proxy to the zero-one loss called a *surrogate loss function*. For computational reasons this is usually a convex function. In the following we review some of the most common surrogate loss functions.

For classification problems the *hinge loss* provides a relatively tight, convex upper bound on the *zero-one loss*:

$$L(y, M(\mathbf{x}, \mathbf{w})) = \frac{1}{n}\sum_i \max(0, 1 - y_i M(\mathbf{x}_i, \mathbf{w}))$$

Another popular alternative is the *logistic loss* (also known as *logistic regression*) function:

$$L(y, M(\mathbf{x}, \mathbf{w})) = \frac{1}{n} log(1 + exp(-y_i M(\mathbf{x}_i, \mathbf{w})))$$

This function displays a similar convergence rate to the hinge loss function, and since it is continuous, simple gradient descent methods can be utilized.

Cross-entropy is a loss function that is very used for training *multiclass problems*. In this case, our labels have this form $\mathbf{y}_i = (1.0, 0.0, 0.0)$. If our model predicts a different distribution, say $M(\mathbf{x}_i, \mathbf{w}) = (0.4, 0.1, 0.5)$, then we'd like to nudge the parameters so that $M(\mathbf{x}_i, \mathbf{w})$ gets closer to $\mathbf{y}_i$. C. Shannon showed that if you want to send a series of messages composed of symbols from an alphabet with distribution y (y_j is the probability of the j-th symbol), then to use the smallest number of bits on average, you should assign $\log(\frac{1}{y_j})$ bits to the j-th symbol. The optimal number of bits is known as *entropy*:

$$H(\mathbf{y}) = \sum_j y_j \log \frac{1}{y_j} = -\sum_j y_j \log y_j$$

Cross-entropy is the number of bits we'll need if we encode symbols by using a wrong distribution $\hat{y}$:

$$H(y, \hat{y}) = -\sum_j y_j \log \hat{y}_j$$

In our case, the real distribution is $\mathbf{y}$ and the 'wrong' one is $M(\mathbf{x}, \mathbf{w})$. So, minimizing *cross-entropy* with respect our model parameters will result in the model that best approximates our labels if considered as a probabilistic distribution.

Cross entropy is used in combination with the *Softmax* classifier. In order to classify $\mathbf{x}_i$ we could take the index corresponding to the max value of $M(\mathbf{x}_i, \mathbf{w})$, but Softmax gives a slightly more intuitive output (normalized class probabilities) and also has a probabilistic interpretation:

$$P(\mathbf{y}_i = j \mid M(\mathbf{x}_i, \mathbf{w})) = \frac{e^{M_j(\mathbf{x}_i, \mathbf{w})}}{\sum_k e^{M_k(\mathbf{x}_i, \mathbf{w})}}$$

where M_k is the k-th component of the classifier output.

3 Automatic Differentiation

Let's come back to the problem of the derivative computation and the cost it represents for Stochastic Gradient Descend methods. We have seen that in order to optimize our models we need to compute the derivative of the loss function with respect to all model parameters for a series of epochs that involve thousands or millions of data points.

In general, the computation of derivatives in computer models is addressed by four main methods:

- manually working out derivatives and coding the result;
- using numerical differentiation, also known as finite difference approximations;
- using symbolic differentiation (using expression manipulation in software);
- and automatic differentiation (AD).

When training large and deep neural networks, AD is the only practical alternative. AD works by systematically applying the chain rule of differential calculus at the elementary operator level.

Let $y = f(g(w))$ our target function. In its basic form, the chain rule states:

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial w} \tag{4}$$

or, if there is more than one variable g_i in-between y and w (f.e. if f is a two dimensional function such as $f(g_1(w), g_2(w))$), then:

$$\frac{\partial f}{\partial w} = \sum_i \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial w}$$

For example, let's consider the derivative of one-dimensional 1-layer neural network:

$$f_x(w, b) = \frac{1}{1 + e^{-(w \cdot x + b)}} \tag{5}$$

Now, let's write how to evaluate $f(w)$ via a sequence of primitive operations:

```
x = ?
f1 = w * x
f2 = f1 + b
f3 = -f2
f4 = 2.718281828459 ** f3
f5 = 1.0 + f4
f = 1.0/f5
```

The question mark indicates that x is a value that must be provided. This *program* can compute the value of f and also populate program variables.

We can evaluate $\frac{\partial f}{\partial w}$ at some x by using Eq. (4). This is called *forward-mode differentiation*. In our case:

```
def dfdx_forward(x, w, b):
    f1 = w * x
    df1 = x                               # = d(f1)/d(w)
    f2 = f1 + b
    df2 = df1 * 1.0                       # = df1 * d(f2)/d(f1)
    f3 = -f2
    df3 = df2 * -1.0                      # = df2 * d(f3)/d(f2)
    f4 = 2.718281828459 ** f3
    df4 = df3 * 2.718281828459 ** f3      # = df3 * d(f4)/d(f3)
    f5 = 1.0 + f4
    df5 = df4 * 1.0                       # = df4 * d(f5)/d(f4)
    df6 = df5 * -1.0 / f5 ** 2.0          # = df5 * d(f6)/d(f5)
    return df6
```

It is interesting to note that this *program* can be readily executed if we have access to subroutines implementing the derivatives of primitive functions (such as $\exp(x)$ or $1/x$) and all intermediate variables are computed in the right order. It is also interesting to note that AD allows the accurate evaluation of derivatives at machine precision, with only a small constant factor of overhead.

Forward differentiation is efficient for functions $f : \mathbf{R}^n \rightarrow \mathbf{R}^m$ with $n \ll m$ (only $O(n)$ sweeps are necessary). For cases $n \gg m$ a different technique is needed. To this end, we will rewrite Eq. (4) as:

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial x}\frac{\partial f}{\partial g} \tag{6}$$

to propagate derivatives backward from a given output. This is called *reverse-mode differentiation*. Reverse pass starts at the end (i.e. $\frac{\partial f}{\partial f} = 1$) and propagates backward to all dependencies.

```
def dfdx_backward(x, w, b):
    f1 = w * x
    f2 = f1 + b
    f3 = -f2
    f4 = 2.718281828459 ** f3
    f5 = 1.0 + f4
    f6 = 1.0/f5

    df6 = 1.0                              # = d(f)/d(f)
    df5 = 1.0 * -1.0 / (f5 ** 2)           # = df6 * d(f6)/d(f5)
    df4 = df5 * 1.0                        # = df5 * d(f5)/d(f4)
    df3 = df4 * log(2.718281828459)
          * 2.718281828459 ** f3           # = df4 * d(f4)/d(f3)
    df2 = df3 * -1.0                       # = df3 * d(f3)/d(f2)
    df1 = df2 * 1.0                        # = df2 * d(f2)/d(f1)
    df  = df1 * x                          # = df1 * d(f1)/d(w)
    return df
```

In practice, reverse-mode differentiation is a two-stage process. In the first stage the original function code is run forward, populating f_i variables. In the second stage, derivatives are calculated by propagating in reverse, from the outputs to the inputs.

The most important property of reverse-mode differentiation is that it is cheaper than forward-mode differentiation for functions with a high number of input variables. In our case, $f : \mathbf{R}^n \rightarrow \mathbf{R}$, only one application of the reverse mode is sufficient to compute the full gradient of the function $\nabla f = \left(\frac{\partial f}{\partial w_1}, \dots, \frac{\partial f}{\partial w_n}\right)$. This is the case of deep learning, where the number of parameters to optimize is very high.

As we have seen, AD relies on the fact that all numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known. For this reason, given a library of derivatives of all elementary functions in a deep neural network, we are able of computing the derivatives of the network with respect to all parameters at machine precision and applying stochastic gradient methods to its training. Without this automation process the design and debugging of optimization processes for complex neural networks with millions of parameters would be impossible.

4 Architectures

Up to now we have used classical neural network layers: those that can be represented by a simple weight matrix multiplication plus the application of a non linear activation function. But automatic differentiation paves the way to consider different kinds of layers without pain.

4.1 Convolutional Neural Networks

Convolutional Neural Networks have been some of the most influential innovations in the field of computer vision in the last years. When considering the analysis of an image with a computer we must define a computational representation of an image. To this end, images are represented as $n \times m \times 3$ array of numbers, called *pixels*. The 3 refers to RGB values and n, m refers to the height and width of the image in pixels. Each number in this array is given a value from 0 to 255 which describes the pixel intensity at that point. These numbers are the only inputs available to the computer.

What is to classify an image? The idea is that you give the computer this array of numbers and it must output numbers that describe the probability of the image being a certain class.

Of course, this kind of image representation is suited to be classified by a classical neural network composed of dense layers, but this approach has several limitations.

The first one is that large images with a high number of pixels will need from extremely large networks to be analyzed. If an image has $256 \times 256 = 65,536$ pixels, the first layer of a classical neural network needs to have $65,536 \times 65,536 = 4,294,967,296$ different weights to consider all pixel interactions. Even in the case that this number of weights could be stored in the available memory, learning these weights would be very time consuming. But there is a better alternative.

Natural images are not a random combination of values in a 256×256 array, but present strong correlations at different levels. At the most basic, it is evident that the value of a pixel is not independent of the values of its neighboring pixels. Moreover, natural images present another interesting property: location invariance. That means that visual structures, such as a *cat* or a *dog*, can be present on any place of the image at any scale. Image location is not important, what is important for attaching a meaning to an image are the relative positions of geometric and photometric structures.

All this considerations leaded, partially inspired by biological models, to the proposal of a very special kind of layers: those based on *convolutions*. A convolution is a mathematical operation that combines two input images to form a third one. One of the input images is the image we want to process. The other one, that is smaller, is called the kernel. Let's suppose that our kernel is this one:

$$Kernel = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{7}$$

The output of image convolution is calculated as follows:

1. Flip the kernel both horizontally and vertically. As our selected kernel is symmetric, the flipped kernel is equal to the original.
2. Put the first element of the kernel at every element of the image matrix. Multiply each element of the kernel with its corresponding element of the image matrix (the one which is overlapped with it).
3. Sum up all product outputs and put the result at the same position in the output matrix as the center of kernel in image matrix.

Mathematically, given a convolution kernel K represented by a $(M \times N)$ array, the convolution of an image I with K is:

$$output(x, y) = (I \otimes K)(x, y) = \sum_{m=0}^{M-1} \sum_{n=1}^{N-1} K(m, n) I(x - n, y - m)$$

The output of image convolution is another image that might represent some kind of information that was present in the image in a very subtle way. For example, the kernel we have used is called an *edge detector* because it highlights the edges of visual structures and attenuates smooth regions. Figure 3 depicts the result of two different convolution operations.

In convolutional neural networks the values of the kernel matrix are free parameters that must be learned to perform the optimal information extraction in order to classify the image.

Convolutions are linear operators and because of this the application of successive convolutions can always be represented by a single convolution. But if we apply a non linear activation function after each convolution the application of successive convolution operators makes sense and results in a powerful image feature structure.

In fact, after a convolutional layer there are two kinds of non linear functions that are usually applied: non-linear activation functions such as sigmoids or ReLU and *pooling*. Pooling layers are used with the purpose to progressively reduce the spatial size of the image to achieve scale invariance. The most common layer is the *maxpool* layer. Basically a maxpool of 2×2 causes a filter of 2 by 2 to traverse over the entire input array and pick the largest element from the window to be included in the next representation map. Pooling can also be implemented by using other criteria, such as averaging instead of taking the max element.

A convolutional neural network is a neural network that is build by using several convolutional layers, each one formed by the concatenation of three different operators: convolutions, non-linear activation and pooling. This kind of networks are able of extracting powerful image descriptors when applied in sequence. The power of the method has been credited to the fact that these descriptors can be seen as hierarchical features that are suited to optimally represent visual structures in natural images. The last layers of a convolutional neural network are classical dense layers, which are connected to a classification or regression loss function.

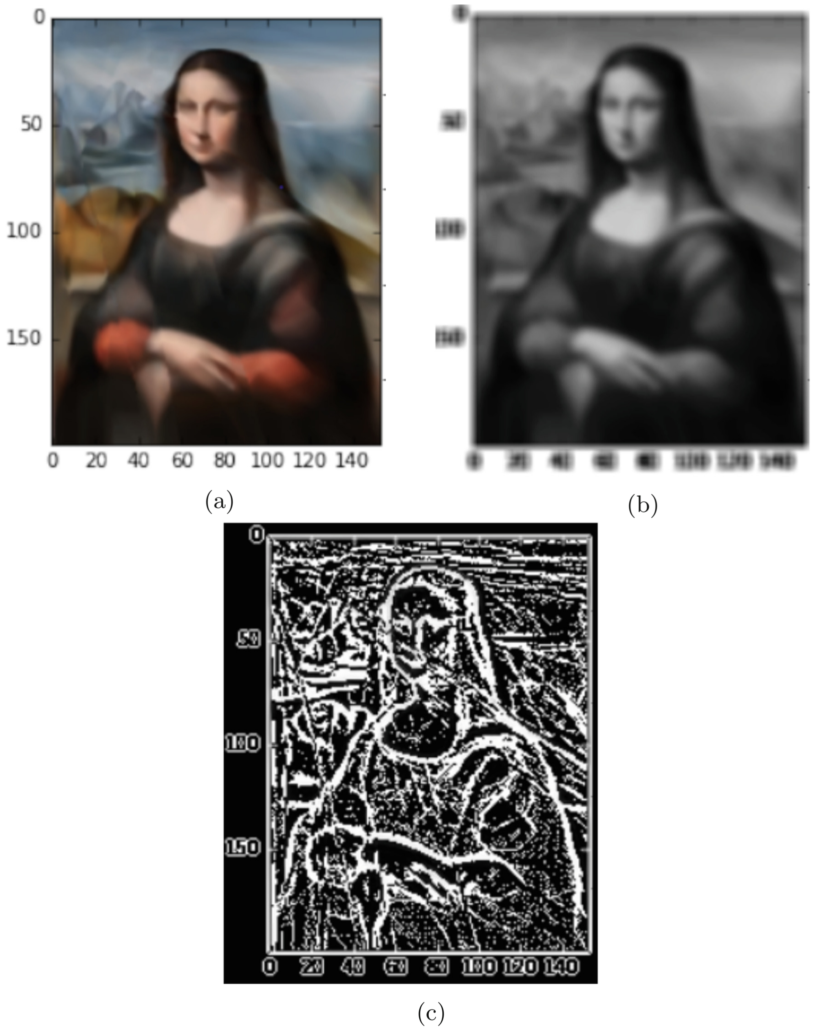


Fig. 3. (a) Original image. (b) Result of the convolution of the original image with a $3 \times 3 \times 3$ kernel where all elements have $1/27$ value. This kernel is a smoother. (c) Result of the convolution of image (b) with the kernel defined in Eq. (7). This kernel is an edge detector.

Finally, it is interesting to point out that convolutional layers are much lighter, in terms of number of weights, than fully connected layers, but more computationally demanding[1]. In some sense, convolutional layers trade weights for computation when extracting information.

[1] The number of weights we must learn for a $(M \times N)$ convolution kernel is only $(M \times N)$, which is independent of the size of the image.

4.2 Recurrent Neural Networks

Classical neural networks, including convolutional ones, suffer from two severe limitations:

– They only accept a fixed-sized vector as input and produce a fixed-sized vector as output.
– They do not consider the sequential nature of some data (language, video frames, time series, etc.)

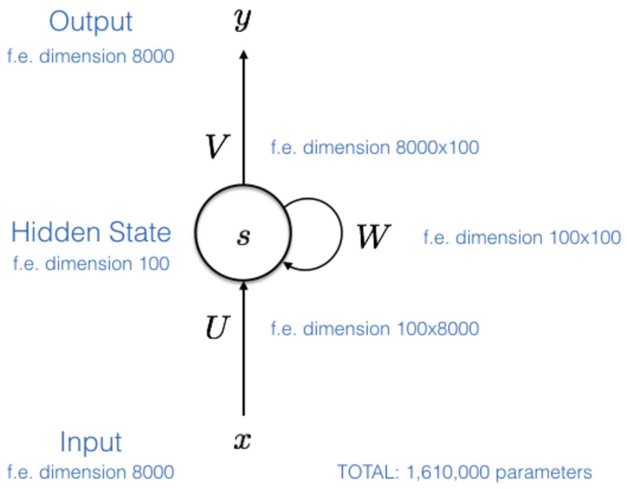


Fig. 4. Recurrent Neural Network. The figure shows the number of parameters that must be trained if the input vector dimension is 8000 and the hidden state is defined to be a 100-dimensional vector.

Recurrent neural networks (RNN) overcome these limitations by allowing to operate over sequences of vectors (in the input, in the output, or both). RNNs are called recurrent because they perform the same task for every element of the sequence, with the output depending on the previous computations (see Fig. 4). The basic formulas of a simple RNN are:

$$s_t = f_1(Ux_t + Ws_{t-1})$$
$$y_t = f_2(Vs_t)$$

These equations basically say that the current network state, commonly known as hidden state, s_t is a function f_1 of the previous hidden state s_{t-1} and the current input x_t. U, V, W matrices are the parameters of the function.

Given an input sequence, we apply RNN formulas in a recurrent way until we process all input elements. The RNN shares the parameters U, V, W across all recurrent steps. We can think of the hidden state as a memory of the network that captures information about the previous steps.

The computational layer implementing this very basic recurrent structure is this:

```
def rnn_layer(x, s):
    h = np.tanh(np.dot(W, s) + np.dot(U, x))
    y = np.dot(V, s)
    return y
```

where `np.tanh` represents the non-linear tanh function and `np.dot` represents matrix multiplication.

The novelty of this type of network is that we have encoded in the very architecture of the network a sequence modeling scheme that has been in used in the past to predict time series as well as to model language. In contrast to the precedent architectures we have introduced, now the hidden layers are indexed by both 'spatial' and 'temporal' index.

These layers can also be stacked one on top of the other for building deep RNNs:

```
y1 = rnn_layer(x)
y2 = rnn_layer(y1)
```

Training a RNN is similar to training a traditional neural network, but with some modifications. The main reason is that parameters are shared by all time steps: in order to compute the gradient at $t = 4$, we need to propagate 3 steps and sum up the gradients. This is called Backpropagation through time (BPTT) [12].

The inputs of a recurrent network are always vectors, but we can process sequences of symbols/words by representing these symbols by numerical vectors.

Let's suppose we want to classify a phrase or a series of words. Let $x^1, ..., x^C$ the word vectors corresponding to a corpus with C symbols[2]. Then, the relationship to compute the hidden layer output features at each time-step t is $h_t = \sigma(W s_{t-1} + U x_t)$, where:

- $x_t \in \mathbf{R}^d$ is input word vector at time t.
- $U \in \mathbf{R}^{D_h \times d}$ is the weights matrix of the input word vector, x_t.
- $W \in \mathbf{R}^{D_h \times D_h}$ is the weights matrix of the output of the previous time-step, $t - 1$.
- $s_{t-1} \in \mathbf{R}^{D_h}$ is the output of the non-linear function at the previous time-step, $t - 1$.
- $\sigma()$ is the non-linearity function (normally, "tanh").

The output of this network is $\hat{y}_t = softmax(V h_t)$, that represents the output probability distribution over the vocabulary at each time-step t.

[2] The computation of useful vectors for words is out of the scope of this tutorial, but the most common method is *word embedding*, an unsupervised method that is based on shallow neural networks.

Essentially, $\hat{y}_t$ is the next predicted word given the document context score so far (i.e. h_{t-1}) and the last observed word vector $x^{(t)}$.

The loss function used in RNNs is often the cross entropy error:

$$L^{(t)}(W) = -\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j})$$

The cross entropy error over a corpus of size C is:

$$L = \frac{1}{C}\sum_{c=1}^{C} L^{(c)}(W) = -\frac{1}{C}\sum_{c=1}^{C}\sum_{j=1}^{|V|} y_{c,j} \times log(\hat{y}_{c,j})$$

These simple RNN architectures have been shown to be too prone to forget information when sequences are long and they are also very unstable when trained. For this reason several alternative architectures have been proposed. These alternatives are based on the presence of *gated units*. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The two most important alternative RNN are Long Short Term Memories (LSTM) [13] and Gated Recurrent Units (GRU) networks [14].

Let us see how a LSTM uses h_{t-1}, C_{t-1} and x_t to generate the next hidden states C_t, h_t:

$$\begin{gathered}
f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) \text{ (Forget gate)} \\
i_t = \sigma(W_i \cdot [h_{t-1}, x_t]) \text{ (Input gate)} \\
\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t]) \\
C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \text{ (Update gate)} \\
o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) \\
h_t = o_t * \tanh(C_t) \text{(Output gate)}
\end{gathered}$$

GRU are a simpler architecture that has been shown to perform at almost the same level as LSTM but using less parameters:

$$\begin{gathered}
z_t = \sigma(W_z \cdot [x_t, h_{t-1}]) \text{ (Update gate)} \\
r_t = \sigma(W_r \cdot [x_t, h_{t-1}]) \text{ (Reset gate)} \\
\tilde{h}_t = \tanh(r_t \cdot [x_t, r_t \circ h_{t-1}]) \text{ (New memory)} \\
h_t = (1 - z_t) \circ \tilde{h}_{t-1} + z_t \circ h_t \text{ (Hidden state)}
\end{gathered}$$

Recurrent neural networks have shown success in areas such as language modeling and generation, machine translation, speech recognition, image description or captioning, question answering, etc.

5 Conclusions

Deep learning constitutes a novel methodology to train very large neural networks (in terms of number of parameters), composed of a large number of specialized layers that are able of representing data in an optimal way to perform regression or classification tasks.

Nowadays, training of deep learning models is performed with the aid of large software environments [15,16] that hide some of the complexities of the task. This allows the practitioner to focus in designing the best architecture and tuning hyper-parameters, but this comes at a cost: seeing these models as black boxes that learn in an almost magical way.

To fully appreciate this fact, here we show a full model specification, training procedure and model evaluation in Keras [17], for a convolutional neural network:

```
model = Sequential()
model.add(Convolution2D(32, 3, 3,
                        activation='relu',
                        input_shape=(1,28,28)))
model.add(Convolution2D(32, 3, 3,
                        activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128,
          activation='relu'))
model.add(Dense(10,
          activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='SGD',
              metrics=['accuracy'])
model.fit(X_train, Y_train,
          batch_size=32,
          nb_epoch=10)
score = model.evaluate(X_test, Y_test)
```

It is not difficult to see in this program some of the elements we have discussed in this paper: SGD, minibatch training, epochs, pooling, convolutional layers, etc.

But to fully understand this model, it is necessary to understand everyone of the parameters and options. It is necessary to understand that this program is implementing an optimization strategy for fitting a neural network model, composed of 2 convolutional layers with 32 3×3 kernel filters and 2 dense layers with 128 and 10 neurons respectively. It is important to be aware that fitting this model requires a relatively large data set and that the only way of minimizing the loss function, cross-entropy in this case, is by using minibatch stochastic gradient descend. We need to know how to find the optimal minibatch size to speed up the optimization process in a specific machine and also to select the optimal non linearity function. Automatic differentiation is hidden in the fitting function, but it is absolutely necessary to deal with the optimization of the complex mathematical expression that results from this model specification.

This paper is only a basic introduction to some of the background knowledge is hidden behind this model specification, but to fully appreciate the power of deep learning the reader is advised to deepen in these areas ([18,19]): she will not be disappointed!

Acknowledgement. This work was partially supported by TIN2015-66951-C2 and SGR 1219 grants. I thank the anonymous reviewers for their careful reading of the manuscript and their many insightful comments and suggestions. I also want to acknowledge the support of NVIDIA Corporation with the donation of a Titan X Pascal GPU. Finally, I would like to express my sincere appreciation to the organizers of the Seventh European Business Intelligence & Big Data Summer School.

References

1. Hebb, D.O.: The Organization of Behavior. Wiley & Sons, New York (1949)
2. Rosenblatt, F.: the perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev. **65**(6), 386–408 (1958). Cornell Aeronautical Laboratory
3. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS 2012), pp. 1097–1105. Curran Associates Inc., USA (2012)
5. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
6. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-2010), pp. 807–814 (2010)
7. Csáji, B.C.: Approximation with Artificial Neural Networks, vol. 24, p. 48. Faculty of Sciences, Etvs Lornd University, Hungary (2001)
8. Sutton, R.S.: Two problems with backpropagation and other steepest-descent learning procedures for networks. In: Proceedings of 8th Annual Conference Cognitive Science Society (1986)
9. Duchi, J., Hazan, E., Singer, Y.: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)
10. Kingma, D., Jimmy B.A.: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Bottou, L.: Online Algorithms and Stochastic Approximations. Online Learning and Neural Networks. Cambridge University Press, Cambridge (1998)
12. Mozer, M.C.: A focused backpropagation algorithm for temporal pattern recognition. In: Chauvin, Y., Rumelhart, D. Backpropagation: Theory, Architectures, and Applications, pp. 137–169. ResearchGate, Lawrence Erlbaum Associates, Hillsdale (1995). Accessed 21 Aug 2017
13. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
14. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling arXiv:1412.3555 (2014)
15. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: a system for large-scale machine learning. In: OSDI, vol. 16, pp. 265–283 (2016)

16. Paszke, A., Gross, S., Chintala, S.: PyTorch. GitHub repository (2017). https://github.com/orgs/pytorch/people
17. Chollet, F.: (2017). Keras (2015). http://keras.io
18. Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep learning, vol. 1. MIT press, Cambridge (2016)
19. Nielsen, M.A.: Neural Networks and Deep Learning. Determination Press (2015)

Author Index