Journal Subline
LNCS 7790

Alfredo Cuzzocrea · Umeshwar Dayal
Guest Editors

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

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

Springer

Lecture Notes in Computer Science 7790

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

Abdelkader Hameurlain Josef Küng
Roland Wagner Alfredo Cuzzocrea
Umeshwar Dayal (Eds.)

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

Special Issue on Advances in Data Warehousing and Knowledge Discovery

Springer

ISSN 0302-9743 (LNCS) e-ISSN 1611-3349 (LNCS)
ISSN 1869-1994 (TLDKS)
ISBN 978-3-642-37573-6 e-ISBN 978-3-642-37574-3
DOI 10.1007/978-3-642-37574-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013934729

CR Subject Classification (1998): H.2.4, H.2.7-8, I.2.6, H.3.4, J.1

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

Printed on acid-free paper

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

Preface

Data warehousing and *knowledge discovery* is an extremely active research area where a number of methodologies and paradigms converge, with coverage of both theoretical issues and practical solutions. The area of data warehousing and knowledge discovery has been widely accepted as a key technology for enterprises and organizations, as it allows them to improve their abilities in data analysis, decision support, and the automatic extraction of knowledge from data. With the exponentially growing amount of information to be included in the decision-making process, the data to be considered are becoming more and more complex in both structure and semantics. As a consequence, novel developments, both at the methodological level, e.g., complex analytics over data, and at the infrastructural level, e.g., cloud computing architectures, are necessary. Orthogonal to the latter aspects, the knowledge discovery and retrieval process from huge amounts of heterogeneous complex data represents a significant challenge for this research area.

In line with these traditional principles but with a novel emphasis, data warehousing and knowledge discovery is now moving its attention to exciting emerging contexts such as social network data, spatio-temporal data, streaming data, non-standard pattern types, complex analytical functionalities, multimedia data, as well as real-world applications (e.g., bio-medical and biological mining tools). The wide range of topics bears witness to the fact that the data warehousing and knowledge discovery field is dynamically responding to the new challenges posed by novel types of data and applications.

In order to fulfill the innovative requirements posed by the realization of data warehousing and knowledge discovery in emerging fields, this special issue on "*Advances in Data Warehousing and Knowledge Discovery*" of the *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems* presents a rigorous selection of the best papers from the 13^{th} *International Conference on Data Warehousing and Knowledge Discovery* (DaWaK 2011), held in Toulouse, France, during August 29 – September 2, 2011. Following its successful tradition, DaWaK 2011 attracted a large number of submissions, and, after a rigorous selection process only 12 papers were invited for submission to the *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems* special issue on "*Advances in Data Warehousing and Knowledge Discovery*". After two additional rigorous review rounds, only 8 papers were accepted for final publication in the special issue.

The aim of the special issue is to offer an innovative, modern research perspective on data warehousing and knowledge discovery models, methods and algorithms over large-scale data repositories, by highlighting recent top-quality contributions and results in this scientific context, and, at the same time, stimu-

lating further investigation in the field. In the following, we provide a summary of the papers contained in the special issue.

The first paper, titled "*ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce*", by Xiufeng Liu, Christian Thomsen, and Torben, Bach Pedersen, investigates the issue of improving the performance of Extract-Transform-Load (ETL) processes via emerging high-performance MapReduce paradigms, ETL being one of the most critical processes for data warehousing systems. In line with this main assumption, authors recognize that an increasing challenge for ETL flows is the need to process huge volumes of data quickly. MapReduce, on the other hand, is establishing itself as the de-facto standard for large-scale data-intensive processing. However, MapReduce lacks support for high-level ETL specific constructs, resulting in low ETL programmer productivity. This paper thus presents a scalable dimensional ETL framework, called ETLMR, based on MapReduce. ETLMR has built-in native support for operations on DW-specific constructs such as star schemas, snowflake schemas, and slowly changing dimensions. This enables ETL developers to construct scalable MapReduce-based ETL flows with very few code lines. To achieve good performance and load balancing, a number of dimension and fact-processing schemes are presented in the paper, including techniques for efficiently processing different types of dimensions. Also, the paper evaluates the ETLMR's performance on large realistic data sets. The experimental results show that ETLMR achieves very good scalability and compares favorably with other MapReduce data warehousing tools.

The second paper, titled "*The Planning OLAP Model - A Multidimensional Model with Planning Support*", by Bernhard Jaecksch and Wolfgang Lehner, explores an innovative capability of the OLAP multidimensional model, namely the planning functionality. Authors first recognize that a wealth of multidimensional OLAP models have been suggested in the past, tackling various problems of modeling multidimensional data. However, all of these models focus on navigational and query operators for grouping, selection, and aggregation. Hence, authors argue that planning functionality is, next to reporting and analysis, an important part of OLAP in many businesses and as such should be represented as part of a multidimensional model. Navigational operators are not enough for planning, instead new factual data is created or existing data is changed. Due to the fact that the main data entities of a typical multidimensional model are used both by planning and reporting, authors concentrate on the extension of an existing model by adding a set of novel operators that support an extensive set of typical planning functions. A comprehensive case study completes the analytical contributions of the paper.

The third paper, titled "*Query Optimization for the NOX OLAP Algebra*", by Ahmad Taleb, Todd Eavis, and Hiba Tabbara, discusses query optimization issues for a native OLAP query execution language, called NOX, and provides some effective solutions to them. As authors correctly recognize, current OLAP servers are typically implemented as either extensions to conventional relational databases or as non-relational array-based storage engines. In the for-

mer case, the unique modeling and processing requirements of OLAP systems often make for a relatively awkward fit with RDBMS systems. In the latter case, the proprietary nature of the MOLAP implementations has largely prevented the emergence of a standardized query model. Considering the two technologies (i.e., RDBMS and MOLAP), authors discuss an algebra for the specification, optimization, and execution of OLAP-specific queries, including its ability to support the native language query framework NOX. In addition to this main contribution, authors ground the conceptual work by incorporating the query optimization and execution facilities into a fully-functional OLAP-aware DBMS prototype. Experimental results clearly demonstrate the potential of the new algebra-driven system relative to both the un-optimized prototype and a pair of popular enterprise servers.

The fourth paper, titled "*Finding Critical Thresholds for Defining Bursts in Event Logs*", by Bibudh Lahiri, Ioannis Akrotirianakis, and Fabian Moerchen, takes into consideration bursts, i.e., unusually high frequencies of occurrences of events in a time-window, which are interesting in many monitoring applications that give rise to temporal data in terms of abnormal activities. While the problem of detecting bursts from time-series data has been well-addressed, the question of what choice of thresholds, on the number of events as well as on the window size, makes a window unusually bursty remains a relevant one. Starting from this main motivation, authors focus on the problem of finding critical values of both these thresholds. Indeed, since, for most applications, any apriori idea of what combination of thresholds is critical is hardly available, the range of possible values for either threshold can be very large. Hence, authors formulate finding the combination of critical thresholds as a two-dimensional search problem and design efficient deterministic and randomized divide-and-conquer heuristics to this end. For the deterministic heuristic, authors show that, under some weak assumptions, the computational overhead is logarithmic in the sizes of the ranges. Under identical assumptions, the expected computational overhead of the randomized heuristic in the worst case is also logarithmic. Finally, authors conduct extensive simulations on data obtained from logs of medical equipment, thus reinforcing the theoretical results obtained, and showing that, on average, the randomized heuristic beats its deterministic counterpart in practice.

The fifth paper, titled "*Concurrent Semi-supervised Learning with Active Learning of Data Streams*", by Hai-Long Nguyen, Wee-Keong Ng, and Yew-Kwong Woon, focuses the attention on semi-supervised learning algorithms over data streams. Authors first analyze conventional stream mining algorithms, which mainly implement stand-alone mining tasks, and then they argue that, given the single-pass nature of data streams, it makes sense to maximize throughput by performing multiple complementary mining tasks concurrently. On the basis of this main intuition, authors investigate the potential of concurrent semi-supervised learning on data streams and propose an incremental algorithm called CSL-Stream (Concurrent Semi-supervised Learning of Data Streams) that performs clustering and classification at the same time. Experiments using common synthetic and real data sets show that CSL-Stream outperforms prominent

clustering and classification algorithms, such as D-Stream and SmSCluster, in terms of accuracy, speed, and scalability. Moreover, enhanced with a novel active learning technique, CSL-Stream only requires a small number of queries to work well with very sparsely labeled data sets. The success of CSL-Stream paves the way for a new research direction in understanding latent commonalities among various data mining tasks in order to exploit the power of concurrent stream mining.

The sixth paper, titled "*Efficient Single Pass Ordered Incremental Pattern Mining*", by Yun Sing Koh and Gillian Dobbie, focuses the attention on the problem of supporting incremental pattern mining over data streams, which has been stimulated by the introduction of the FP-growth algorithm along with the FP-tree data structure. Most incremental mining adapts the Apriori algorithm. However, using a tree-based approach increases performance as compared with the candidate generation and testing mechanism used in Apriori. Despite this, FP-tree still requires two scans through a data set. To improve efficiency, authors present a novel tree structure, called Single Pass Ordered Tree (SPO-Tree), which captures information with a single scan for incremental mining. All items in a transaction are inserted/sorted based on their frequency. The tree is reorganized dynamically when necessary. Experimental results clearly show that SPO-Tree allows for easy maintenance in an incremental or data stream environment.

The seventh paper, titled "*Finding Interesting Rare Association Rules Using Rare Pattern Tree*", by Sidney Tsang, Yun Sing Koh, and Gillian Dobbie, addresses rare pattern mining problems over large databases, specially considering (rare) associations rules. Indeed, while most association rule mining techniques concentrate on finding frequent rules, rare association rules are in some cases more interesting than frequent association rules, since rare rules represent unexpected or unknown associations. All current algorithms for rare association rule mining use an Apriori level-wise approach which has computationally expensive candidate generation and pruning steps. Contrary to this main trend, authors propose RP-Tree, a method for mining a subset of rare association rules using a tree structure, and an information gain component that helps to identify the more interesting association rules. Empirical evaluation using a range of real-world data sets shows that RP-Tree itemset and rule generation is more time efficient than modified versions of FP-Growth and ARIMA, and discovers 92-100% of all the interesting rare association rules. Additional evaluation using synthetic data sets also shows that RP-Tree is more efficient, in addition to showing how the execution time of RP-Tree changes with transaction length and rare-item itemset size.

Finally, the eighth paper, titled "*Discovering Frequent Patterns from Uncertain Data Streams with Time-Fading and Landmark Models*", by Carson Kai-Sang Leung, Alfredo Cuzzocrea, and Fan Jiang, proposes novel techniques for discovering frequent patterns from uncertain data streams that exploit the time-fading model and the landmark model, respectively. Indeed, as authors recognize, streams of data can be continuously generated by sensors in various real-life applications such as environment surveillance. Partially due to the inherited lim-

itation of the sensors, data in these streams can be uncertain. To discover useful knowledge in the form of frequent patterns from streams of uncertain data, a few algorithms have been developed. They mostly use the sliding window model for processing and mining data streams. However, for some applications, other stream processing models such as the time-fading model and the landmark model are more appropriate. On the basis of this main assumption, authors provide mining algorithms that use these two latter models to discover frequent patterns from streams of uncertain data, by completing with an extensive experimental assessment and analysis on both synthetic and real-life data stream sets.

The editors would like to express their sincere gratitude to the editors-in-chief of the *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems*, Prof. Abdelkader Hameurlain, Prof. Josef Küng, and Prof. Roland Wagner, for accepting their proposal for a special issue focused on advances in data warehousing and knowledge discovery, and for assisting them whenever required. The editors would also like to thank all the reviewers who worked within a tight schedule and whose detailed and constructive feedback to authors contributed to a substantial improvement in the quality of the final papers.

December 2012

Alfredo Cuzzocrea
Umeshwar Dayal

Table of Contents

ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce

Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen

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

Abstract. Extract-Transform-Load (ETL) flows periodically populate data warehouses (DWs) with data from different source systems. An increasing challenge for ETL flows is to process huge volumes of data quickly. MapReduce is establishing itself as the de-facto standard for large-scale data-intensive processing. However, MapReduce lacks support for high-level ETL specific constructs, resulting in low ETL programmer productivity. This paper presents a scalable dimensional ETL framework, *ETLMR*, based on MapReduce. ETLMR has built-in native support for operations on DW-specific constructs such as star schemas, snowflake schemas and slowly changing dimensions (SCDs). This enables ETL developers to construct scalable MapReduce-based ETL flows with very few code lines. To achieve good performance and load balancing, a number of dimension and fact processing schemes are presented, including techniques for efficiently processing different types of dimensions. The paper describes the integration of ETLMR with a MapReduce framework and evaluates its performance on large realistic data sets. The experimental results show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

1 Introduction

In data warehousing, ETL flows are responsible for collecting data from different data sources, transformation, and cleansing to comply with user-defined business rules and requirements. Traditional ETL technologies face new challenges as the growth of information explodes nowadays, e.g., it becomes common for an enterprise to collect hundreds of gigabytes of data for processing and analysis each day. The vast amount of data makes ETL extremely time-consuming, but the time window assigned for processing data typically remains short. Moreover, to adapt to rapidly changing business environments, users have an increasing demand of getting data as soon as possible. The use of parallelization is the key to achieve better performance and scalability for those challenges.

In recent years, a novel "cloud computing" technology, *MapReduce* [12], has been widely used for parallel computing in data-intensive areas. A MapReduce program implements the *map* and *reduce* functions, which process key/value pairs and are executed in many parallel instances. MapReduce provides programming flexibility, cost-effective scalability and capacity on commodity machines. A MapReduce framework provides off-the-shelf functionality for inter-process communication, fault-tolerance,

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 1–31, 2013.

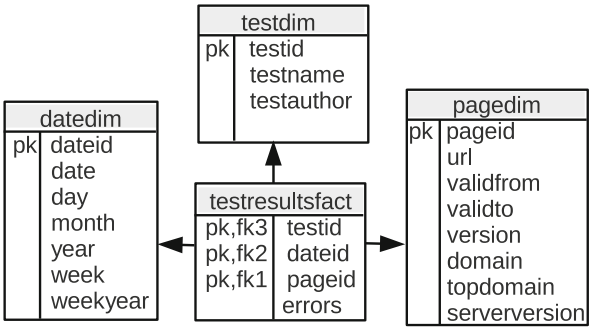


Fig. 1. Star schema of the running example

load balancing and task scheduling to a parallel program. Now MapReduce is becoming increasingly popular and is establishing itself as the de-facto standard for large-scale data-intensive processing. It is thus interesting to see how MapReduce can be applied to the field of ETL programming.

The data processing in ETL exhibits the *composable* property. For example, the processing of dimensions and facts can be split into smaller computation units and the partial results from these computation units can be merged to constitute the final results in a DW. This complies well with the MapReduce paradigm in term of *map* and *reduce*. Thus, MapReduce is a good foundation for the ETL parallelization. However, MapReduce is only a generic programming model. It lacks support for high-level DW/ETL specific constructs, such as the dimensional constructs of star schemas, snowflake schemas, and SCDs. An ETL program is inherently complex, which is due to the ETL-specific activities, such as transformation, cleansing, filtering, aggregating and loading. To implement a parallel ETL program is costly, error-prone, and this leads to low programmer productivity.

In this paper, we present a parallel dimensional ETL framework based on MapReduce, named *ETLMR*, which directly supports high-level ETL-specific dimensional constructs such as star schemas, snowflake schemas, and SCDs. We believe this to be the first paper to specifically address ETL for *dimensional* schemas on MapReduce. The paper makes several contributions: We leverage the functionality of MapReduce to the ETL parallelization and provide a scalable, fault-tolerable, and very lightweight ETL framework which hides the complexity of MapReduce. We present a number of novel methods which are used to process the dimensions of a star schema, snowflaked dimensions, SCDs and data-intensive dimensions. In addition, we introduce the offline dimension scheme which scales better than the online dimension scheme when handling massive workloads. The evaluations show that ETLMR achieves very good scalability and compares favourably with other MapReduce data warehousing tools.

The Running Example: To show the use of ETLMR, we use a running example throughout this paper. This example is inspired by a project which applies different tests to web pages. Each test is applied to each page and the test outputs the number of errors detected. The test results are written into a number of tab-separated files, which serve as the data sources. The data is processed to be stored in a DW with the star

schema shown in Fig. 1. We will consider a partly snowflaked (i.e., normalized) variant of the schema in Section 4.3. The star schema comprises a fact table and three dimension tables. Note that *pagedim* is a slowly changing dimension table which contains many more rows than the other two dimension tables. We use this example, instead of a more common example such as TPC-H [28], because it has an SCD, a data-intensive dimension, and can be represented by both a star schema and a snowflake schema. The chosen example thus allows us to illustrate and test our system more comprehensively.

The remainder of this paper is structured as follows: Section 2 gives a brief review of the MapReduce programming model. Section 3 gives an overview of ETLMR. Sections 4 and 5 present dimension processing and fact processing, respectively. Section 6 describes the implementation of ETLMR in the Disco MapReduce framework. Section 7 presents experimental evaluations. Section 8 reviews related work. Finally, Section 9 concludes the paper and provides ideas for future work. The appendix compares ETLMR to Hive and Pig by means of concrete ETL solutions.

2 MapReduce Programming Model

MapReduce is a parallel programming model derived from functional programming concepts, and is proposed by Google for large-scale data processing in a distributed environment [12]. The MapReduce programming model is depicted in Fig. 2, in which the computations are expressed by means of two functions called *map* and *reduce*.

```
map: (k1, v1) ->  list(k2, v2)
reduce: (k2, list(v2)) -> list(k3, v3)
```

The *map* function, defined by users, runs in parallel on many nodes. It takes as input a key/value pair $(k1, v1)$ read from a data split D_i in a distributed file system (DFS), and produces a list of intermediate key/value pairs $(k2, v2)$. On each node, the intermediate key/value pairs are partitioned on the key values, possibly combined into fewer pairs, and finally written to the local file system (combining is optional and is shown as a dashed line in Fig. 2). The MapReduce framework then sorts and merges the key value pairs before it shuffles them to the reducers over the network. The *reduce* function, also defined by users, takes as input a pair consisting of a key $k2$ and a list of all values for this key. It merges or aggregates the values to form a possibly smaller (or even empty) list of key/value pairs $(k3, v3)$, and then writes them to the DFS.

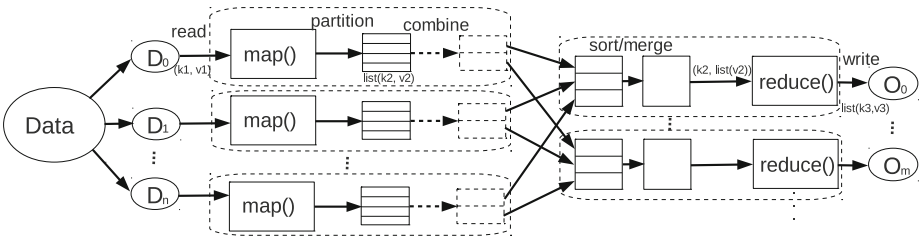


Fig. 2. MapReduce programming model

In addition to the map and reduce functions of the programming model, there are five other functions offered by most MapReduce frameworks, including functions for input reading, data partitioning, combining map output, sorting/merging, and output writing (see Fig. 2). Users can (but do not have to) specify these functions to fit to their specific requirements. A MapReduce framework achieves parallel computations by executing the implemented functions in parallel on clustered computers, each processing a chunk of the data sets.

3 Overview

In this section, we give an overview of ETLMR on a MapReduce framework and describe the data processing phases in ETLMR. Fig. 3 illustrates the data flow using ETLMR on MapReduce. In ETLMR, the dimension processing is done at first in a MapReduce job, then the fact processing is done in another MapReduce job. A MapReduce job spawns a number of parallel map/reduce task (map/reduce task denotes map tasks and reduce tasks running separately) for processing dimension or fact data. Each task consists of several steps, including reading data from a distributed file system (DFS), executing the map function, partitioning, combining the map output, executing the reduce function and writing results. In dimension processing, the input data for a dimension table can be processed by different processing methods, e.g., the data can be processed by a single task or by all tasks. In fact processing, the data for a fact table is partitioned into a number of equal-sized data files which then are processed by parallel tasks. This includes looking up dimension keys and bulk loading the processed fact data into the DW. The processing of fact data in the reducers can be omitted (shown by dotted ellipses in Fig. 3) if no aggregation of the fact data is done before it is loaded.

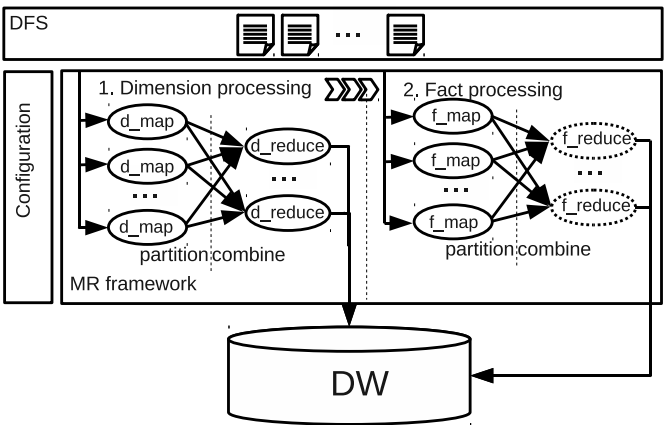


Fig. 3. ETLMR Data flow

Algorithm 1. ETL process on MapReduce framework

1: Partition the input data sets;
2: Read the configuration parameters and initialize;
3: Read the input data and relay the data to the map function in the map readers;
4: Process dimension data and load it into online/offline dimension stores;
5: Synchronize the dimensions across the clustered computers, if applicable;
6: Prepare fact processing (connect to and cache dimensions);
7: Read the input data for fact processing and perform transformations in mappers;
8: Bulk-load fact data into the DW.

Algorithm 1 shows the details of the whole process of using ETLMR. The operations in lines 2-4 and 6-7 are the MapReduce steps which are responsible for initialization, invoking jobs for processing dimensions and facts, and returning processing information. Line 1 and 5 are the non-MapReduce steps which are used for preparing input data sets and synchronizing dimensions among nodes (if no DFS is installed). The data reader in ETLMR partitions the data while it reads lines from the input files. Specifically, it supports the following two partitioning schemes.

- *Round-robin partitioning*: This method distributes row number n to task number $n \% nr\_map$ where nr_map is the total number of tasks and $\%$ is the modulo operator. By this method, the data sets are equally divided and processed by the parallel tasks. Thus, it ensures the load balance.
- *Hash partitioning*: This method partitions data sets based on the values of one or several attributes of a row. It computes the hash value h for the attribute values and assigns the row to task number $h \% nr\_map$.

All parameters are defined in a configuration file, including declarations of dimension and fact tables, dimension processing methods, number of mappers and reducers, and others. The parameteres are summarized in Table 1. The use of parameters gives the user flexibility to configure the tasks to be more efficient. For example, if a user knows that a dimension is a data-intensive dimension, she can add it to the list $bigdims$ so that an appropriate processing method can be chosen to achieve better performance and load balancing. The configuration file is also used for specifications of user-defined functions (UDFs) for data processing.

4 Dimension Processing

In ETLMR, each dimension table has a corresponding definition in the configuration file. For example, we define the object for the dimension table *testdim* of the running example by *testdim = CachedDimension(name='testdim', key='testid', defaultidvalue =-1, attributes=['testname', 'testauthor'], lookupatts=['testname',])*. It is declared as a cached dimension which means that (all or parts of) the dimension data will be kept in main memory during the processing. ETLMR also offers other dimension classes for declaring different dimension tables, including *SlowlyChangingDimension*

Table 1. The configuration parameters

Parameters	Description
Dim_i	Dimension table definition, $i = 1, \ldots, n$
$Fact_i$	Fact table definition, $i = 1, \ldots, m$
Set_{bigdim}	data-intensive dimensions whose business keys are used for partitioning the data sets if applicable
$Dim_i(a_0, a_1, ..., a_n)$	Define the relevant attributes $a_0, a_1, ..., a_n$ of Dim_i in data source
$DimScheme$	Dimension scheme, online/offline (online is the default)
nr_reduce	Number of reducers
nr_map	Number of mappers

and *SnowflakedDimension*, each of which is configured by means of a number of parameters for specifying the name of the dimension table, the dimension key, the attributes of dimension table, the lookup attributes (which identify a row uniquely), and others. Each class offers a number of functions for dimension operations such as *lookup*, *insert*, *ensure*, etc.

ETLMR employs MapReduce's *map*, *partition*, *combine*, and *reduce* to process data. This is, however, hidden from the user who only specifies transformations to apply to the data and declarations of dimension tables and fact tables. A map/reduce task reads data by iterating over lines from a partitioned data set. A line is first processed by *map*, then by *partition* which determines the target reducer, and then by *combine* which groups values having the same key. The data is then written to an intermediate file (there is one file for each reducer). In the reduce step, a reduce reader reads a list of key/values pairs from an intermediate file and invokes *reduce* to process the list. In the following, we present different approaches to process dimension data.

4.1 One Dimension One Task

In this approach, map tasks process data for all dimensions by applying user-defined transformations and by finding the relevant parts of the source data for each dimension. The data for a given dimension is then processed by a single reduce task. We name this method *one dimension one task* (*ODOT* for short).

The data unit moving around within ETLMR is a dictionary mapping attribute names to values. Here, we call it a *row*, e.g., *row*={*'url':'www.dom0.tl0/p0.htm','size': '12553', 'serverversion':'SomeServer/1.0', 'downloaddate':'2011-01-31','lastmoddate': '2011-01-01', 'test':'Test001', 'errors':'7'*}. ETLMR reads lines from the input files and passes them on as rows. A mapper does projection on rows to prune unnecessary data for each dimension and makes key/value pairs to be processed by reducers. If we define dim_i for a dimension table and its relevant attributes, $(a_0, a_1..., a_n)$, in the data source schema, the mapper will generate the map output, $(key, value) = (dim_i.name, \prod_{a_0,a_1,...,a_n}(row))$ where $name$ represents the name of dimension table. The MapReduce partitioner partitions map output based on the key, i.e., $dim_i.name$, such that the data of dim_i will go to a single reducer (see Fig. 4). To optimize, the values with identical keys (i.e., dimension table name) are combined in the combiner before they are sent

to the reducers such that the network communication cost can be reduced. In a reducer, a row is first processed by UDFs to do data transformations, then the processed row is inserted into the dimension store, i.e., the dimension table in the DW or in an offline dimension store (described later). When ETLMR does this data insertion, it has the following reduce functionality: If the row does not exist in the dimension table, the row is inserted. If the row exists and its values are unchanged, nothing is done. If there are changes, the row in the table is updated accordingly. The ETLMR dimension classes provide this functionality in a single function, $dim_i.ensure(row)$. For an SCD, this function adds a new version if needed, and updates the values of the SCD attributes, e.g., the attributes *validto* and *version*.

We have now introduced the most fundamental method for processing dimensions where only a limited number of reducers can be utilized. Therefore, its drawback is that it is not optimized for the case where some dimensions contain large amounts of data, namely data-intensive dimensions.

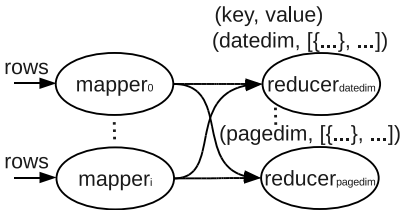


Fig. 4. ODOT

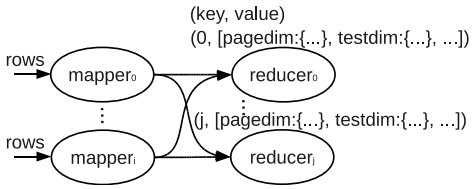


Fig. 5. ODAT

4.2 One Dimension All Tasks

We now describe another approach in which all reduce tasks process data for all dimensions. We name it *one dimension all tasks* (*ODAT* for short). In some cases, the data volume of a dimension is very large, e.g., the *pagedim* dimension in the running example. If we employ ODOT, the task of processing data for this dimension table will determine the overall performance (assume all tasks run on similar machines). We therefore refine the ODOT in two places, the map output partition and the reduce functions. With ODAT, ETLMR partitions the map output by round-robin fashion such that the reducers receive equally many rows (see Fig. 5). In the reduce function, two issues are considered in order to process the dimension data properly by the parallel tasks:

The first issue is how to keep the uniqueness of dimension key values as the data for a dimension table is processed by all tasks. We propose two approaches. The first one is to use a global ID generator and use *post-fixing* (detailed in Section 4.4) to merge rows having the same values in the dimension *lookup* attributes (but different key values) into one row. The other approach is to use private ID generators and post-fixing. Each task has its own ID generator, and after the data is loaded into the dimension table, post-fixing is employed to fix the resulting duplicated key values. This requires the uniqueness constraint on the dimension key to be disabled before the data processing.

The second issue is how to handle concurrency problem when data manipulation language (DML) SQL, such as UPDATE, DELETE, etc., is issued by several tasks. Consider, for example, the type-2 SCD table *pagedim* for which INSERTs and UPDATEs are frequent (the SCD attributes *validfrom* and *validto* are updated). There are at least two ways to tackle this problem. The first one is row-based commit in which a COMMIT is issued after every row has been inserted so that the inserted row will not be locked. However, row-based commit is more expensive than transaction commit, thus, it is not very useful for a data-intensive dimension table. Another and better solution is to delay the UPDATE to the post-fixing which fixes all the problematic data when all the tasks have finished.

In the following section, we propose an alternative approach for processing snowflaked dimensions without requiring the post-fixing.

4.3 Snowflaked Dimension Processing

In a snowflake schema, dimensions are normalized meaning that there are foreign key references and hierarchies between dimension tables. If we consider the dependencies when processing dimensions, the post-fixing step can be avoided. We therefore propose two methods particularly for snowflaked dimensions: *level-wise processing* and *hierarchy-wise processing*.

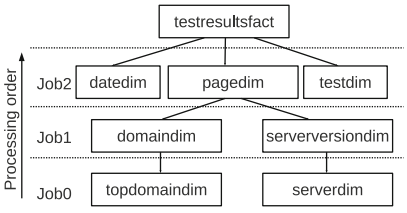


Fig. 6. Level-wise processing

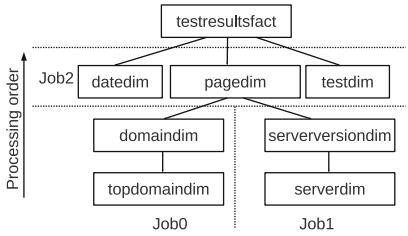


Fig. 7. Hierarchy-wise processing

Level-Wise Processing. This refers to processing snowflaked dimensions in an order from the leaves towards the root (the dimension table referred by the fact table is the root and a dimension table without a foreign key referencing other dimension tables is a leaf). The dimension tables with dependencies (i.e., with foreign key references) are processed in sequential jobs, e.g., *Job1* depends on *Job0*, and *Job2* depends on *Job1* in Fig. 6. Each job processes independent dimension tables (without direct and indirect foreign key references) by parallel tasks, i.e., one dimension table is processed by one task. Therefore, in the level-wise processing of the running example, *Job0* first processes *topdomaindim* and *serverdim* in parallel, then *Job1* processes *domaindim* and *serverversiondim*, and finally *Job2* processes *pagedim*, *datedim* and *testdim*. It corresponds to the configuration *loadorder = [('topdomaindim', 'serverdim'), ('domaindim', 'serverversiondim'), ('pagedim', 'datedim', 'testdim')]*. With this order, a higher level dimension table (the referencing dimension table) is not processed until the lower level

ones (the referenced dimension tables) have been processed and thus, the referential integrity can be ensured.

Hierarchy-Wise Processing. This refers to processing a snowflaked dimension in a branch-wise fashion (see Fig. 7). The root dimension, *pagedim*, derives two branches, each of which is defined as a separate snowflaked dimension, i.e., *domainsf = SnowflakedDimension([(domaindim, topdomaindim)])*, and *serverversionsf = SnowflakedDimension([(serverversiondim, serverdim)])*. They are processed by two parallel jobs, *Job0* and *Job1*, each of which processes in a sequential manner, i.e., *topdomaindim* followed by *domaindim* in *Job0* and *serverdim* followed by *serverversiondim* in *Job1*. The root dimension, *pagedim*, is not processed until the dimensions on its connected branches have been processed. It, together with *datedim* and *testdim*, is processed by the *Job2*.

4.4 Post-fixing

As discussed in Section 4.2, post-fixing is a remedy to fix problematic data in ODAT when all the tasks of the dimension processing have finished. Four situations require data post-fixing: 1) using a global ID generator which gives rise to duplicated values in the lookup attributes; 2) using private ID generators which produce duplicated key values; 3) processing snowflaked dimensions (and *not* using level-wise or hierarchy-wise processing) which leads to duplicated values in lookup and key attributes; and 4) processing slowly changing dimensions which results in SCD attributes taking improper values.

Example 1 (Post-fixing). *Consider two map/reduce tasks, task 1 and task 2, that process the* `page` *dimension which we here assume to be snowflaked. Each task uses a private ID generator. The root dimension,* `pagedim`*, is a type-2 SCD. Rows with the lookup attribute value url='www.domain2.tl2/p0.htm' are processed by both the tasks.*

domaindim

taskid	domainid	domain	topdomainid
1	1	www.domain1.tl1	1
1	2	www.domain2.tl2	2
2	1	www.domain2.tl2	1

topdomaindim

taskid	topdomainid	topdomain
1	1	tl1
1	2	tl2
2	1	tl2

pagedim

taskid	pageid	url	validfrom	validto	version	domainid
1	1	www.domain1.tl1/page0.htm	2010-01-01	NULL	1	1
1	2	www.domain2.tl2/page0.htm	2010-01-01	NULL	1	2
2	1	www.domain2.tl2/page0.htm	2010-12-31	NULL	1	1

Fig. 8. Before post-fixing

Fig. 8 depicts the resulting data in the dimension tables where the white rows were processed by task 1 and the grey rows were processed by task 2. Each row is labelled with the `taskid` *of the task that processed it. The problems include duplicated IDs in each dimension table and improper values in the SCD attributes,* `validfrom`*,* `validto`*, and* `version`*. The post-fixing program first fixes the* `topdomaindim` *such that rows with the same value for the lookup attribute (i.e.,* `url`*) are merged into*

one row with a single ID. Thus, the two rows with `topdomain` *= tl2 are merged into one row. The references to* `topdomaindim` *from* `domaindim` *are also updated to reference the correct (fixed) rows. In the same way,* `pagedim` *is updated to merge the two rows representing www.domain2.tl2. Finally,* `pagedim` *is updated. Here, the post-fixing also has to fix the values for the SCD attributes. The result is shown in Fig. 9.*

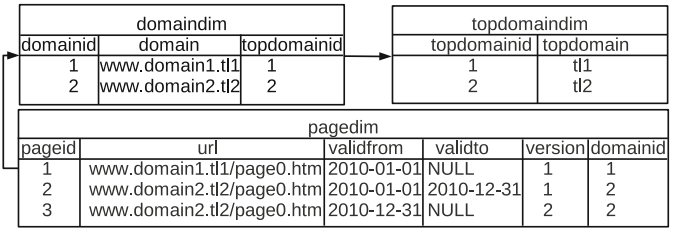


Fig. 9. After post-fixing

Algorithm 2. post_fix(*dim*)

refdims ← The referenced dimensions of *dim*
for *ref* in *refdims* **do**
  itr ← post_fix(*ref*)
  for ((*taskid*, *keyvalue*), *newkeyvalue*) in *itr* **do**
    Update *dim* set *dim.key* = *newkeyvalue* where *dim.taskid*=*taskid* and *dim.key*=*keyvalue*
ret ← An empty list
Assign *newkeyvalue*s to *dim*'s keys and add ((*taskid*, *keyvalue*), *newkeyvalue*) to *ret*
if *dim* is not the root **then**
  Delete the duplicate rows, which have identical values in *dim*'s lookup attributes
if *dim* is a type-2 SCD **then**
  Fix the values of SCD attributes, e.g., dates and version
return *ret*

The post-fixing invokes a recursive function (see Algorithm 2) to fix the problematic data in the order from the leaf dimension tables to the root dimension table. It comprises four steps: 1) assign new IDs to the rows with duplicate IDs; 2) update the foreign keys on the referencing dimension tables; 3) delete duplicated rows which have identical values in the business key attributes and foreign key attributes; and 4) fix the values in the SCD attributes if applicable. In most cases, it is not needed to fix something in each of the steps for a dimension with problematic data. For example, if a global ID generator is employed, all rows will have different IDs (such that step 1 is not needed) but they may have duplicate values in the lookup attributes (such that step 3 is needed). ETLMR's implementation uses an embedded SQLite database for data management during the post-fixing. Thus, the task IDs are not stored in the target DW, but only internally in ETLMR.

4.5 Offline Dimension

In ODOT and ODAT, the map/reduce tasks interact with the DW's ("*online*") dimensions directly through database connections at run-time and the performance is affected by the outside DW DBMS and the database communication cost. To optimize, the *offline dimension* scheme is proposed. In this scheme, the tasks do not interact with the DW directly, but with distributed offline dimensions residing physically in all nodes. It has several characteristics and advantages. First, a dimension is partitioned into multiple smaller-sized sub-dimensions, and small-sized dimensions can benefit dimension *lookup*s, especially for a data-intensive dimension such as *pagedim*. Second, high-performance storage systems can be employed to persist dimension data. Dimensions are configured to be fully or partially cached in main memory to speed up the *lookup*s when processing facts. In addition, offline dimensions do not require direct communication with the DW and the overhead (from the network and the DBMS) is greatly reduced. ETLMR has offline dimension implementations for one dimension one task (*ODOT (offline)* for short) and *hybrid*, which are described in the following.

ODOT (offline). Fig. 10 depicts the run-time architecture when we use two map/reduce tasks to process data. The data for each dimension table is saved locally in its own store in the node that processes it or in the DFS (shown in the center of the Fig. 10). The data for a dimension table is processed by one and only one reduce task (as in the online ODOT) which does the following: 1) Select the values of the fields that are relevant to the dimension table in mappers; 2) Partition the map output based on the names of dimension tables; 3) Process the data for dimension tables by using user-defined transformation functions in the reducers (a reducer only processes the data for one dimension table); 4) When all map/reduce tasks have finished, the data for all dimension tables are synchronized across the nodes if no DFS is installed (here, only the data files of the offline dimension stores are synchronized).

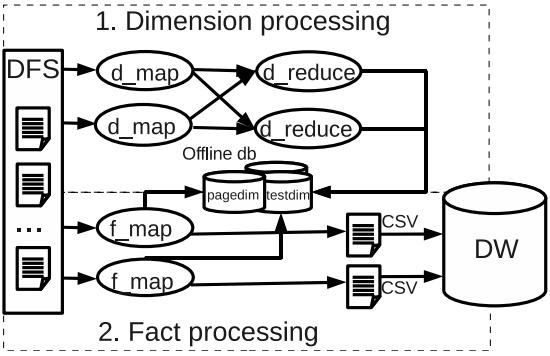


Fig. 10. ODOT (offline)

Hybrid. Hybrid combines the characteristics of ODOT and ODAT. In this approach, the dimensions are divided into two groups, the most data-intensive dimension and the other dimensions. The input data for the most data-intensive dimension table is partitioned based on the business keys, e.g., on the `url` of `pagedim`, and processed by all the map tasks (this is similar to ODAT), while for the other dimension tables, their data is processed in reducers, a reducer exclusively processing the data for one dimension table (this is similar to ODOT).

This corresponds to the following steps: 1) Choose the most data-intensive dimension and partition the input data sets, for example, on the business key values; 2) Process the chosen data-intensive dimension and select the required data for each of the other dimensions in the mappers; 3) Round-robin partition the map output; 3) Process the dimensions in the reducers (each is processed by one reducer); 4) When all the tasks have finished, synchronize all the processed dimensions across the nodes if no DFS is installed, but keep the partitioned data-intensive dimension in all nodes.

Example 2 (Hybrid). *Consider using two parallel tasks to process the dimension tables of the running example (see the upper part in Fig. 11). The data for the most data-intensive dimension table,* `pagedim`*, is partitioned into a number of chunks based on the business key* `url`*. The chunks are processed by two mappers, each processing a chunk. This results in two offline stores for* `pagedim`*,* `pagedim-0` *and* `pagedim-1`*. However, the data for the other dimension tables is processed similarly to the offline ODOT, which results in the other two offline dimension stores,* `datedim` *and* `testdim`*, respectively.*

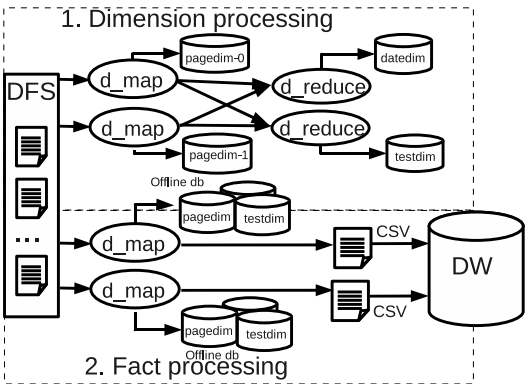


Fig. 11. Hybrid

In the offline dimension scheme, the dimension data in the offline stores is expected to reside in the nodes permanently and will not be loaded into the DW until this is explicitly requested.

5 Fact Processing

Fact processing is the second phase in ETLMR. In this phase, ETLMR looks up dimension keys for the facts, does aggregation of measures (if applicable), and loads the processed facts into the DW. Similarly to the dimension processing, the definitions and settings of fact tables are also declared in the configuration file. ETLMR provides the *BulkFactTable* class which supports bulk loading of facts to the DW. For example, the fact table of the running example is defined as *testresultsfact=BulkFactTable(name='testresultsfact', keyrefs=['pageid', 'testid', 'dateid'], measures=['errors'], bulkloader=UDF_pgcopy, bulksize=5000000)*. The parameters are the fact table name, a list of the keys referencing dimension tables, a list of measures, the bulk loader function, and the size of the bulks to load. The bulk loader is a UDF such that ETLMR can be used with different types of DBMSs.

Algorithm 3 shows the pseudo code for processing facts.

Algorithm 3. $process\_fact(row)$

Require: A *row* from the input data and the *config*
1: *facttbls* ← the fact tables defined in *config*
2: **for** *facttbl* in *facttbls* **do**
3: *dims* ← the dimensions referenced by *facttbl*
4: **for** *dim* in *dims* **do**
5: *row[dim.key]* ← *dim*.lookup(*row*)
6: *rowhandlers* ← *facttbl.rowhandlers*
7: **for** *handler* in *rowhandlers* **do**
8: *handler*(*row*)
9: *facttbl*.insert(*row*)

The function can be used as the map function or as the reduce function. If no aggregations (such as *sum*, *average*, or *count*) are required, the function is configured to be the map function and the reduce step is omitted for better performance. If aggregations are required, the function is configured to be the reduce function since the aggregations must be computed from all the data. This approach is flexible and good for performance. Line 1 retrieves the fact table definitions in the configuration file and they are then processed sequentially in line 2–8. The processing consists of two major operations: 1) look up the keys from the referenced dimension tables (line 3–5), and 2) process the fact data by the *rowhandlers*, which are user-defined transformation functions used for data type conversions, calculating measures, etc. (line 6–8). Line 9 invokes the insert function to insert the fact data into the DW. The processed fact data is not inserted into the fact table directly, but instead added into a configurably-sized buffer where it is kept temporarily. When a buffer becomes full, its data is bulk loaded into the DW. Each map/reduce task has a separate buffer and bulk loader such that tasks can do bulk loading in parallel.

6 Implementation and Optimization

ETLMR is designed to achieve plug-in like functionality to facilitate the integration with Python-supporting MapReduce frameworks. In this section, we introduce the ETL programming framework *pygrametl* which is used to implement ETLMR. Further, we give an overview of MapReduce frameworks with a special focus on Disco [4] which is our chosen MapReduce platform. The integration and optimization techniques employed are also described.

6.1 pygrametl

`pygrametl` was implemented in previous work [25] and has a number of characteristics. It is a code-based ETL framework which enables efficient development due to its simplicity and use of Python. `pygrametl` supports processing of dimensions in both star schemas and snowflake schemas and it provides direct support for slowly changing dimensions. Regardless of the dimension types, the ETL implementation is very concise and convenient. It uses an object to represent a dimension. Only a single method call such as `dimobj.insert(row)` is needed to insert new data. In this method call, all details, such as key assignment and SQL generation, are transparent to users. `pygrametl` supports the most commonly used operations on a dimension, such as `lookup`, `insert`, `ensure`. In the implementation of ETLMR, most functionality offered by `pygrametl` can be re-used, but some parts have been extended or modified to support the MapReduce operations.

6.2 MapReduce Frameworks

There are many open source and commercial MapReduce frameworks available. The most popular one is Apache Hadoop [6], which is implemented in Java and includes a distributed file system (HDFS). Hadoop is embraced by a variety of academic and industrial users, including Amazon, Yahoo!, Facebook, and many others [3]. Google's MapReduce implementation is extensively used inside the company, but is not publicly available. Apart from them, many companies and research units develop their own MapReduce frameworks for their particular needs, such as [22,30,17].

For ETLMR, we choose the open source framework Disco [4] as the MapReduce platform. Disco is developed by Nokia using the Erlang and Python programming languages. Disco is chosen for the following reasons. First, Disco's use of Python facilitates rapid scripting for distributed data processing programs, e.g., a complicated program or algorithm can be expressed in tens of lines of code. This feature is consistent with our aim of providing users a simple and easy means of implementing a parallel ETL. Second, Disco achieves a high degree of flexibility by providing many customizable MapReduce programming interfaces. These make the integration of ETLMR very convenient by having a plug-in-like effect. Third, unlike the alternatives, it provides direct support for the distributed programs written in Python. Some MapReduce frameworks implemented in other programming languages also claim to support Python programs, e.g., Hadoop, but they require bridging middleware and are, thus, not implementation-friendly.

6.3 Integration with Disco

Disco's architecture is similar to the Google and Hadoop MapReduce architectures in which intermediate results are stored as local files and accessed by appropriate reduce tasks. However, the used version (0.2.4) does not include a built-in distributed file system (DFS), but supports any POSIX-compatible DFS such as GlusterFS. If no DFS is installed, the input files are required to be distributed initially to each node so that they can be read locally [4]. To use Disco, the functions to use as map and reduce are passed on as arguments to Disco.

We present the runtime architecture of ETLMR on Disco in Fig. 12, where all ETLMR instances are running in parallel on many processors of clustered computers (or nodes). This architecture is capable of shortening the time of an ETL job by scaling up to the number of nodes in the cluster. It is an one-master-many-workers architecture. The master is responsible for scheduling the components (tasks) of the jobs to run on the workers, assigning partitioned data sets to the workers, tracking the status, and monitoring the status of the workers. When the master receives ETL jobs, it puts them into a queue and distributes them to the available workers. In each node, there is a worker supervisor started by the master which is responsible for spawning and monitoring all tasks on that particular node. When a worker receives a task, it runs this task exclusively in a processor of this node, processes the input data, and saves the processed data to the DW.

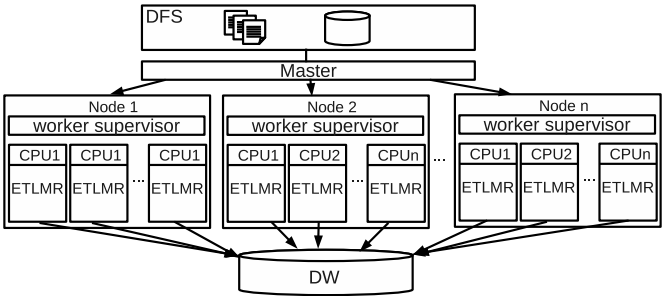


Fig. 12. Parallel ETL overview

The master-workers architecture has a highly fault-tolerant mechanism. It achieves reliability by distributing the input files to the nodes in the cluster. Each worker reports to the master periodically with the completed tasks and their status. If a worker falls silent for longer than a certain interval, the master will record this worker as dead and send the node's assigned tasks to other workers.

6.4 Optimizations

In ETLMR, the data sets from different storage systems are prepared into data files. Each file gets a unique address for map readers, such as a URL, file path, or distributed

file path. In many cases, however, a data source may be a database or an indexed file structure such that we can make use of its indices for efficient filtering, i.e., instead of returning all the data, only the needed columns or subsets of the data are selected. If the data sets are pre-split, such as several data files from heterogeneous systems, the split parts are directly processed and combined in MapReduce. Different map readers are implemented in ETLMR for reading data from different storage systems, such as the DBMS reader supporting user-defined SQL statements and text file reader. If the data sets are not split, such as a big data file, Dean and Ghemwat [11] suggest utilizing many MapReduce processes to run complete passes over the data sets, and process the subsets of the data. Accordingly, we implement such a map reader (see Program 1). It supports reading data from a single data source, but does not require the data sets to be split before being processed. In addition, we implement the offline dimension scheme by using the Python `shelve` package [5] in which main-memory database systems, such as `bsdbdb`, can be configured to persist dimension data. At run-time, the dimension data is fully or partially kept in main memory such that the *lookup* operation can be done efficiently.

Program 1. Map reader function

```
def map_reader(content, bkey, thispartition=this_partition()):
  while True:
    line = content.next()
    if not line:
        break
    if (hash(line[bkey])%Task.num_partitions)==thispartition:
        yield line
```

7 Experimental Evaluation

In this section, we present the performance improvements achieved by the proposed methods. Further, we evaluate the system scalability on various sizes of tasks and data sets and compare with other business intelligence tools using MapReduce.

7.1 Experimental Setup

All experiments are conducted on a cluster of 6 nodes connected through a gigabit switch and each having an Intel(R) Xeon(R) CPU X3220 2.4GHz with 4 cores, 4 GB RAM, and a SATA hard disk (350 GB, 3 GB/s, 16 MB Cache and 7200 RPM). All nodes are running the Linux 2.6.32 kernel with Disco 0.2.4, Python 2.6, and ETLMR installed. The GlusterFS DFS is set up for the cluster. PostgreSQL 8.3 is used for the DW DBMS and is installed on one of the nodes. One node serves as the master and the others as the workers. Each worker runs 4 parallel map/reduce tasks, i.e., in total

20 parallel tasks run. The time for bulk loading is not measured as the way data is bulk loaded into a database is an implementation choice which is independent of and outside the control of the ETL framework. To include the time for bulk loading would thus clutter the results. We note that bulk loading can be parallelized using off-the-shelf functionality.

7.2 Test Data

We continue to use the running example. We use a data generator to generate the test data for each experiment. In line with Jean and Ghemawat's assumption that MapReduce usually operates on numerous small files rather than a single, large, merged file [11], the test data sets are partitioned and saved into a set of files. These files provide the input for the dimension and fact processing phases. We generate two data sets, *bigdim* and *smalldim* which differ in the size of the *page* dimension. In particular, 80 GB *bigdim* data results in 10.6 GB fact data (193,961,068 rows) and 6.2 GB *page* dimension data (13,918,502 rows) in the DW while 80 GB *smalldim* data results in 12.2 GB (222,253,124 rows) fact data and 54 MB *page* dimension data (193,460 rows) in the DW. Both data sets produce 32 KB *test* (1,000 rows) and 16 KB *date* dimension data (1,254 rows).

7.3 Scalability of the Proposed Processing Methods

In this experiment, we compare the scalability and performance of the different ETLMR processing methods. We use a fixed-size *bigdim* data set (20 GB), scale the number of parallel tasks from 4 to 20, and measure the total elapsed time from start to finish. The results for a snowflake schema and a star schema are shown in Fig. 13 and Fig. 14, respectively. The graphs show the *speedup*, computed by $T_{4,odot,snowflake}/T_n$ where $T_{4,odot,snowflake}$ is the processing time for *ODOT* using 4 tasks in a snowflake schema and T_n is the processing time when using n tasks for the given processing method.

We see that the overall time used for the star schema is less than for the snowflake schema. This is because the snowflake schema has dimension dependencies and hierarchies which require more (level-wise) processing. We also see that the offline hybrid scales the best and achieves almost linear speedup. The ODAT in Fig. 14 behaves similarly. This is because the dimensions and facts in offline hybrid and ODAT are processed by all tasks which results in good balancing and scalability. In comparison, ODOT, offline ODOT, level-wise, and hierarchy-wise do not scale as well as ODAT and hybrid since only a limited number of tasks are utilized to process dimensions (a dimension is only processed in a single task). The offline dimension scheme variants outperform the corresponding online ones, e.g., offline ODOT vs. ODOT. This is caused by 1) using a high performance storage system to save dimensions on all nodes and provide in-memory lookup; 2) The data-intensive dimension, *pagedim*, is partitioned into smaller chunks which also benefits the lookups; 3) Unlike the online dimension scheme, the offline dimension scheme does not communicate directly with the DW and this reduces the communication cost considerably. Finally, the results show the relative efficiency for the optimized methods which are much faster than the baseline ODOT.

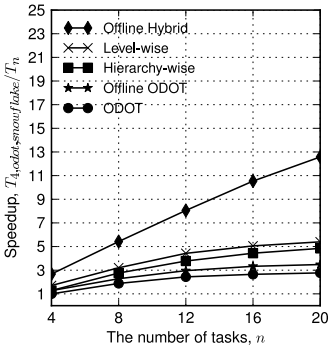


Fig. 13. Load for snowflake schema, 20 GB

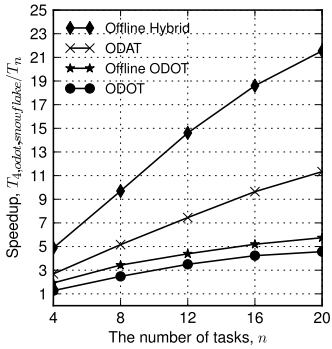


Fig. 14. Load for star schema, 20 GB

7.4 System Scalability

In this experiment, we evaluate the scalability of ETLMR by varying the number of tasks and the size of the data sets. We select the hybrid processing method, use the offline dimension scheme, and conduct the testing on a star schema, as this method not only can process data among all the tasks (unlike ODOT in which only a limited number of tasks are used), but also showed the best scalability in the previous experiment. In the dimension processing phase, the mappers are responsible for processing the data-intensive dimension *pagedim* while the reducers are responsible for the other two dimensions, *datedim* and *testdim*, each using only a single reducer. In the fact processing phase, no reducer is used as no aggregation operations are required.

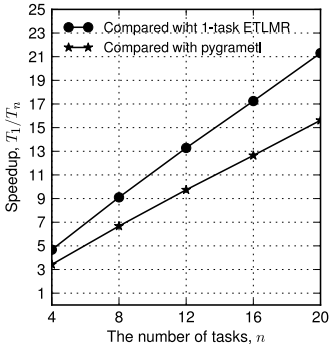


Fig. 15. Speedup with increasing tasks, 80 GB

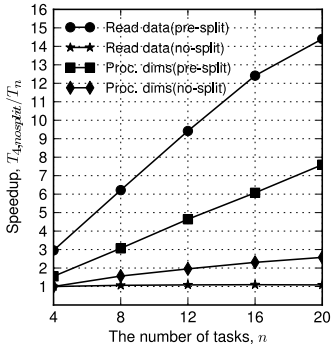


Fig. 16. Speedup of pre-split and no split, 20 GB

We first do two tests to get comparison baselines by using one task (named *1-task ETLMR*) and (plain, non-MapReduce) *pygrametl*, respectively. Here, pygrametl also employs 2-phase processing, i.e., the dimension processing is done before the fact processing. The tests are done on the same machine with a single CPU (all cores but one

are disabled). The tests process 80 GB *bigdim* data. We compute the speedups by using T_1/T_n where T_1 represents the elapsed time for 1-task ETLMR or for pygrametl, and T_n the time for ETLMR using n tasks. Fig. 15 shows that ETLMR achieves a nearly linear speedup in the number of tasks when compared to 1-task ETLMR (the line on the top). When compared to pygrametl, ETLMR has a nearly linear speedup (the lower line) as well, but the speedup is a little lower. This is because the baseline, 1-task ETLMR, has a greater value due to the overhead from the MapReduce framework.

Table 2. Execution time distribution, 80 GB (min.)

Testing data	Phase	Task Num	Part. Input	Map func.	Part.	Comb.	Red. func.	Others	Total
bigdim data (results in 10.6GB facts)	dims	4	47.43	178.97	8.56	24.57	1.32	0.1	260.95
		8	25.58	90.98	4.84	12.97	1.18	0.1	135.65
		12	17.21	60.86	3.24	8.57	1.41	0.1	91.39
		16	12.65	47.38	2.50	6.54	1.56	0.1	70.73
		20	10.19	36.41	1.99	5.21	1.32	0.1	55.22
	facts	4	47.20	183.24	0.0	0.0	0.0	0.1	230.44
		8	24.32	92.48	0.0	0.0	0.0	0.1	116.80
		12	16.13	65.50	0.0	0.0	0.0	0.1	81.63
		16	12.12	51.40	0.0	0.0	0.0	0.1	63.52
		20	9.74	40.92	0.0	0.0	0.0	0.1	50.66
smalldim data (results in 12.2GB facts)	facts	4	49.85	211.20	0.0	0.0	0.0	0.1	261.15
		8	25.23	106.20	0.0	0.0	0.0	0.1	131.53
		12	17.05	71.21	0.0	0.0	0.0	0.1	88.36
		16	12.70	53.23	0.0	0.0	0.0	0.1	66.03
		20	10.04	42.44	0.0	0.0	0.0	0.1	52.58

To learn more about the details of the speedup, we break down the execution time of the slowest task by reference to the MapReduce steps when using the two data sets (see Table 2). As the time for dimension processing is very small for *smalldim* data, e.g., 1.5 min for 4 tasks and less than 1 min for the others, only its fact processing time is shown. When the *bigdim* data is used, we can see that partitioning input data, map, partitioning map output (dims), and combination (dims) dominate the execution. More specifically, partitioning input data and map (see the *Part.Input* and *Map func.* columns) achieve a nearly linear speedup in the two phases. In the dimension processing, the map output is partitioned and combined for the two dimensions, *datedim* and *testdim*. Also here, we see a nearly linear speedup (see the *Part.* and *Comb.* columns). As the combined data of each is only processed by a single reducer, the time spent on reducing is proportional to the size of data. However, the time becomes very small since the data has been merged in combiners (see *Red. func.* column). The cost of post-fixing after dimension processing is not listed in the table since it is not required in this case where a global key generator is employed to create dimension IDs and the input data is partitioned by the business key of the SCD *pagedim* (see section 4.4).

In the fact processing, the reduce function needs no execution time as there is no reducer. The time for all the other parts, including map and reduce initialization, map output partitioning, writing and reading intermediate files, and network traffic, is relatively small, but it does not necessarily decrease linearly when more tasks are added (*Others* column). To summarize (see *Total* column), ETLMR achieves a nearly linear speedup when the parallelism is scaled up, i.e., the execution time of 8 tasks is nearly half that of 4 tasks, and the execution time of 16 tasks is nearly half that of 8 tasks.

Table 2 shows some overhead in the data reading (see *Part. input* column) which uses nearly 20% of the total time. We now compare the two approaches for reading data, namely when it is split into several smaller files ("pre-split") and when it is available from a single big file ("no-split"). We use *bigdim* data set and only process dimensions. We express the performance improvement by the speedup, $T_{4,no\text{-}split}/T_n$, where $T_{4,no\text{-}split}$ is the time taken to read data and process dimensions on the no-split data using 4 tasks (on 1 node), and T_n is the time when using n tasks. As illustrated in Fig. 16, for no-split, the time taken to read data remains constant even though there are more tasks as all read the same data sets. When the data is pre-split, the time taken to read data and process dimensions scales much better than for no-split since a smaller sized data set is processed by each task. In addition, pre-split is inherently faster – by a factor of 3 – than no-split. The slight sub-linear scaling is seen because the system management overhead (e.g., the time spent on communication, adjusting, and maintaining the overall system) increases with the growing number of tasks. However, we can conclude that pre-split is by far the best option.

We now proceed to another experiment where we for a given number of tasks size up the data sets from 20 to 80 GB and measure the elapsed processing time. Fig. 17 and Fig. 18 show the results for the *bigdim* and *smalldim* data sets, respectively. It can be seen that ETLMR scales linearly in the size of the data sets.

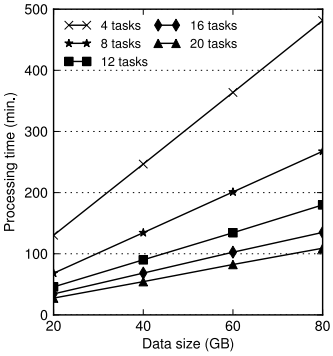


Fig. 17. Scale up the size of bigdim data

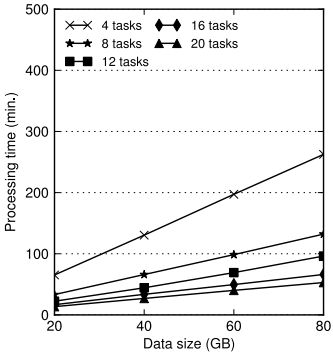


Fig. 18. Scale up the size of smalldim data

7.5 Comparison with Other Data Warehousing Tools

There are some MapReduce data warehousing tools available, including Hive [26,27], Pig [19] and Pentaho Data Integration (PDI) [7]. Hive and Pig both offer data storage on the Hadoop distributed file system (HDFS) and the scripting languages which have some limited ETL abilities. They are both more like a DBMS instead of a full-blown ETL tool. Due to the limited ETL features, they are not suitable for processing an SCD which requires UPDATEs. It is possible to process star and snowflake schemas, but it is complex and verbose. To load data into a *simplified* version of our running example (with *no* SCDs) requires 23 statements in Pig and 40 statements in Hive (we do not count the statements for user-defined transformations). In ETLMR – which in contrast to Pig and Hive is *dimensional* – only 14 statements are required. ETLMR can also support SCDs with the *same* number of statements, while this would be extremely cumbersome to do in Pig and Hive where every single update would have to emulated by overwriting the entire table. The details of the comparison are seen in Appendix A.

PDI is an ETL tool and provides Hadoop support in its 4.1 GA version. However, there are still many limitations with this version. For example, it only allows to set a limited number of parameters in the job executor, customized combiner and mapper-only jobs are not supported, and the transformation components are not fully supported in Hadoop. We only succeeded in making an ETL flow for the simplest star schema, but still with some compromises. For example, a workaround is employed to load the processed dimension data into the DW as PDI's *table output* component repeatedly opens and closes database connections in Hadoop such that the performance suffers.

In the following, we compare how PDI and ETLMR perform when they process the star schema (with *page* as a normal dimension, not an SCD) of the running example. To make the comparison neutral, the time for loading the data into the DW or the HDFS is not measured, and the dimension lookup cache is enabled in PDI to achieve a similar effect of ETLMR using offline dimensions. Hadoop is configured to run 4 parallel task trackers in maximum on each node, and scaled by adding nodes horizontally. The task tracker JVM option is set to be -Xmx256M while the other settings are left to the default.

Table 3 shows the time spent on processing 80 GB *smalldim* data when scaling up the number of tasks. As shown, ETLMR is significantly faster than PDI for Hadoop in processing the data. Several reasons are found for the differences. First, compared with ETLMR, the PDI job has one more step (the reducer) in the fact processing as its job executor does not support a mapper-only job. Second, by default the data in Hadoop is split which results in many tasks, i.e., 1192 tasks for the fact data. Thus, longer initialization time is observed. Further, some transformation components are observed to run with low efficiency in Hadoop, e.g., the components to remove duplicate rows and to apply JavaScript.

Table 3. Time for processing star schema (no SCD), 80 GB *smalldim* data set, (min.)

Tasks	4	8	12	16	20
ETLMR	246.7	124.4	83.1	63.8	46.6
PDI	975.2	469.7	317.8	232.5	199.7

8 Related Work

Besides MapReduce, other parallel programming models also exist. Multi-threading is an alternative and it is possible for an ETL framework to provide abstractions that make it relatively easy for the ETL developer to use several threads. Multi-threading is, however, only useful when a solution is scaled up on a SMP machine and not when scaled out to several machines in a cluster. For very large data sets it is necessary to scale out and the MapReduce framework does this elegantly and provides support for the "tedious" details. Another, alternative is Message Passing Interface (MPI), but again the MapReduce framework makes the developer's work easier by its abstractions and automatic handling of crashes, stragglers working unusually slowly, etc. In recent years, other massively parallel data processing systems have also been proposed. These include Clustera [14], Dryad [18], and Percolator [21]. Clustera and Dryad are systems that support many kinds of tasks ranging from general cluster computations to data management with parallel SQL queries. Their different approaches to this are interesting and give promising results, but the tools are still only available as academic prototypes and are not available to the general public for commercial use. In comparison, MapReduce so far has a far more wide-spread use. It is supported by many different frameworks and is also available as a service, e.g., from Amazon. Percolator was developed by Google motivated by the fact that web indexing with MapReduce requires a re-run on the entire data set to handle updates. Unlike this situation, Percolator supports incremental updates. It does so by letting the user define "observers" (code) that execute when certain data in Google's BigTable is updated. Further, Percolator adds transactional support. It is, however, closely related to Google's BigTable which makes it less general for ETL purposes. Further, it is not publicly available. MapReduce, on the other hand, can be used with several types of input data and many free implementations exist.

Though MapReduce is a framework well suited for large-scale data processing on clustered computers, it has been criticized for being too low-level, rigid, hard to maintain and reuse [19,26]. In recent years, an increasing number of parallel data processing systems and languages built on the top of MapReduce have appeared. For example, besides Hive and Pig (discussed in Section 7.5), Chaiken et al. present the SQL-like language SCOPE [8] on top of Microsoft's Cosmos MapReduce and distributed file system. Friedman et al. introduce SQL/MapReduce [15], a user-defined function (UDF) framework for parallel computation of procedural functions on massively-parallel RDBMSs. These systems or languages vary with respect to how they are implemented and what functionality they provide, but overall they give good improvements to MapReduce such as high-level languages, user interfaces, schemas, and catalogs. They process data by using query languages, or UDFs embedded in the query languages, and execute them on MapReduce. However, they do not offer direct constructs for processing star schemas, snowflaked dimensions, and slowly changing dimensions. In contrast, ETLMR runs separate ETL processes on a MapReduce framework to achieve parallelization and ETLMR directly supports common ETL operations for these *dimensional* schemas.

Another well-known distributed computing system is the parallel DBMS which first appeared two decades ago. Today, there are many parallel DBMSs, e.g., Teradata, DB2, Objectivity/DB, Vertica, etc. The principal difference between parallel DBMSs and MapReduce is that parallel DBMSs run long pipe-lined queries instead of small independent tasks as in MapReduce. The database research community has recently compared the two classes of systems. Pavlo et al. [20], and Stonebraker et al. [23] conduct benchmarks and compare the open source MapReduce implementation Hadoop with two parallel DBMSs (a row-based and a column-based) in large-scale data analysis. The results demonstrate that parallel DBMSs are significantly faster than Hadoop, but they diverge in the effort needed to tune the two classes of systems. Dean et al. [11] argue that there are mistaken assumptions about MapReduce in the comparison papers and claim that MapReduce is highly effective and efficient for large-scale fault-tolerance data analysis. They agree that MapReduce excels at complex data analysis, while parallel DBMSs excel at efficient queries on large data sets [23].

In recent years, ETL technologies have started to support parallel processing. Informatica PowerCenter provides a thread-based architecture to execute parallel ETL sessions. Informatica has also released PowerCenter Cloud Edition (PCE) in 2009 which, however, only runs on a specific platform and DBMS. Oracle Warehouse Builder (OWB) supports pipeline processing and multiple processes running in parallel. Microsoft SQL Server Integration Services (SSIS) achieves parallelization by running multiple threads, multiple tasks, or multiple instances of a SSIS package. IBM InfoSphere DataStage offers a process-based parallel architecture. In the thread-based approach, the threads are derived from a single program, and run on a single (expensive) SMP server, while in the process-based approach, ETL processes are replicated to run on clustered MPP or NUMA servers. ETLMR differs from the above by being based on MapReduce with the inherent advantages of multi-platform support, scalability on commodity clustered computers, light-weight operation, fault tolerance, etc. ETLMR is also unique in being able to scale automatically to more nodes (with no changes to the ETL flow itself, only to a configuration parameter) while at the same time providing automatic data synchronization across nodes even for complex structures like snowflaked dimensions and SCDs. We note that the licenses of the commercial ETL packages prevent us from presenting comparative experimental results.

Big data represents both a challenge and an opportunity that is receiving much attention. Cuzzocrea et al. [10] point out that MapReduce can be considered as the evolution of next-generation DW systems and in particular with regards to the ETL processing. Further, Cuzzocrea et al. discuss further research issues in the field of analytics over big data. There are a variety of MapReduce systems proposed for managing big data in the past few years. Chen [9] proposes the MapReduce-based DW system Cheetah which is designed for an online advertising application. HadoopDB [1] uses a hybrid of DBMS and MapReduce technologies for analytical workloads. Relational systems such as AsterData [2] and GreenPlum [16] have also been extended to support MapReduce jobs. While there exist different analytical systems built on top of Hadoop MapReduce, including Pig and Hive, they are designed for generic big data analytics and processing and not specifically for *dimensional* ETL processing. ETLMR is targeted specifically at making dimensional ETL processing easy and efficient in a MapReduce setting.

9 Conclusion and Future Work

As business intelligence deals with continuously increasing amounts of data, there is an increasing need for ever-faster ETL processing. In this paper, we have presented ETLMR which builds on MapReduce to parallelize ETL processes on commodity computers. ETLMR contains a number of novel contributions. It supports high-level ETL-specific dimensional constructs for processing both star schemas and snowflake schemas, SCDs, and data-intensive dimensions. Due to its use of MapReduce, it can automatically scale to more nodes (without modifications to the ETL flow) while it at the same time provides automatic data synchronization across nodes (even for complex dimension structures like snowflakes and SCDs). Apart from scalability, MapReduce also gives ETLMR a high fault-tolerance. Further, ETLMR is open source, light-weight, and easy to use with a single configuration file setting all run-time parameters. The results of extensive experiments show that ETLMR has good scalability and compares favourably with other MapReduce data warehousing tools.

ETLMR comprises two data processing phases, dimension and fact processing. For dimension processing, this paper proposed a number of dimension management schemes and processing methods in order to achieve good and load balancing. The online dimension scheme directly interacts with the target DW and employs several dimension specific methods to process data, including *ODOT*, *ODAT*, and *level-wise* and *hierarchy-wise* processing for snowflaked dimensions. The offline dimension scheme employs high-performance storage systems to store dimensions distributedly on each node. The methods, *ODOT* and *hybrid* allow better scalability and performance. In the fact processing phase, bulk-load is used to improve the loading performance.

Currently, we have integrated ETLMR with the MapReduce framework Disco. In the future, we intend to port ETLMR to Hadoop and explore a wider variety of data storage options. In addition, we intend to implement dynamic partitioning which automatically adjusts the parallel execution in response to additions/removals of nodes from the cluster, and automatic load balancing which dynamically distributes jobs across available nodes based on CPU usage, memory, capacity and job size through automatic node detection and algorithm resource allocation.

Acknowledgments. This work was in part supported by Daisy Innovation and the European Regional Development Fund and the eGovMon project co-funded by the Research Council of Norway under the VERDIKT program (project no. Verdikt 183392/S10).

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. PVLDB 2(1), 22–933 (2009)
2. AsterData (September 10, 2012), `www.asterdata.com`
3. Applications and organizations using Hadoop (September 10, 2012), `wiki.apache.org/hadoop/PoweredBy`
4. Disco project (September 10, 2012), `discoproject.org`

5. Shelve - Python object persistence (September 10, 2012), docs.python.org/library/shelve.html
6. The Apache Hadoop Project (October 6, 2011), hadoop.apache.org
7. (September 10, 2012), www.pentaho.com
8. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. PVLDB 1(2), 1265–1276 (2008)
9. Chen, S.: Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. PVLDB 3(1), 1459–1468 (2010)
10. Cuzzocrea, A., Song, I.Y., Davis, K.C.: Analytics Over Large-scale Multidimensional Data: the Big Data Revolution! In: Proc. of the ACM 14th International Workshop on Data Warehousing and OLAP, pp. 101–104 (2011)
11. Dean, J., Ghemawat, S.: MapReduce: A Flexible Data Processing Tool. CACM 53(1), 72–77 (2010)
12. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of OSDI, pp. 137–150 (2004)
13. Dittrich, J., Quiane-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB 3(1) (2010)
14. DeWitt, D., Robinson, E., Shankar, S., Paulson, E., Naughton, J., Krioukov, A., Royalty, J.: Clustera: An Integrated Computation and Data Management System. PVLDB 1(1), 28–41 (2008)
15. Friedman, E., Pawlowski, P., Cieslewicz, J.: SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. PVLDB 2(2), 1402–1413 (2009)
16. GreenPlum (September 10, 2012), www.greenplum.com
17. Kovoor, G., Singer, J., Lujan, M.: Building a Java MapReduce Framework for Multi-core Architectures. In: Proc. of MULTIPROG (2010)
18. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: Proc. of EuroSys, pp. 59–72 (2007)
19. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-so-foreign Language for Data Processing. In: Proc. of SIGMOD, pp. 1099–1110 (2008)
20. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-scale Data Analysis. In: Proc. of SIGMOD, pp. 165–178 (2009)
21. Peng, D., Dabek, F.: Large-scale Incremental Processing Using Distributed Transactions and Notifications. In: Proc. of OSDI, pp. 251–264 (2010)
22. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proc. of HPCA, pp. 13–24 (2007)
23. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and Parallel DBMSs: friends or foes? CACM 53(1), 64–71 (2010)
24. Thomsen, C., Pedersen, T.B.: Building a Web Warehouse for Accessibility Data. In: Proc. of DOLAP, pp. 43–50 (2009)
25. Thomsen, C., Pedersen, T.B.: pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers. In: Proc. of DOLAP, pp. 49–56 (2009)
26. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A Warehousing Solution Over a Map-reduce Framework. PVLDB 2(2), 1626–1629 (2009)
27. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive-A Petabyte Scale Data Warehouse Using Hadoop. In: Proc. of ICDE, pp. 996–1005 (2010)
28. TPC-H (September 10, 2012), http://tpc.org/tpch/

29. Vassiliadis, P., Simitsis, A.: Near Real Time ETL. In: Kozielski, S., Wrembel, R. (eds.) New Trends in Data Warehousing and Data Analysis, pp. 1–31. Springer (2008)
30. Yoo, R., Romano, A., Kozyrakis, C.: Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System. In: Proc. of IISWC, pp. 198–207 (2009)

Appendix

A Comparison with Other MapReduce Tools for BI

ETL tools are used for extracting, transforming and loading data into a DW. While this in some cases can be done by only using DBMS software, the typical scenario is that a specialized, stand-alone ETL tool is used. The reason is that ETL tools allow users to do things that are difficult to do with DBMS software such as reading different file formats, copying and writing files in different formats, sending email notifications, connecting to web services, etc. The large amount of different ETL tools available on the market in itself proves the industry-need for stand-alone ETL tools. In the following we compare ETLMR to Hive [26,27] and Pig [19] which are generic MapReduce-based data warehouse systems for storing data and analysis. This is thus somewhat similar to comparing ETL tools and DBMSs, but for completeness we include the comparisons here. Unlike Hive and Pig, ETLMR does not have its own data storage (note that the offline dimension store is only for speedup purpose), but is an ETL tool suitable for processing large scale data in parallel. Hive and Pig share large similarity, such as using Hadoop MapReduce, using Hadoop distributed file system (HDFS) as their data storage, integrating a command line user interface, implementing a query language, being able to do some ETL data analysis, and others. In the following, we compare their ETL features with ETLMR.

Table 4. The comparison of ETL features

Feature	ETLMR	HIVE	PIG
User Interface	Configuration file	Shell/HiveQL/Web JDBC/ODBC	Shell/Pig Latin
ETL knowledge required	Low	High	High
User Defined Functions (UDF)	Yes	Yes	Yes
Filter/Aggregation/Join	Yes	Yes	Yes
Star Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Snowflake Schema	Yes (explicit)	By handcode (implicit)	By handcode (implicit)
Slowly Changing Dimension (SCD)	Yes (explicit)	No	No
ETL details to users	Transparent	Fine-level	Fine-level

Table 4 summarizes the comparison. First, each system has a user interface. Hive provides an SQL-like language HiveQL and a shell, Pig provides a scripting language Pig Latin and a shell, and ETLMR provides a configuration file to declare dimensions,

facts, UDFs, and other run-time parameters. Unlike Hive and Pig which require users to write data processing scripts explicitly, ETLMR is intrinsically an ETL tool which implements ETL process within the framework. The advantage is that users do not have to learn the details of each ETL step, and are able to craft a parallel ETL program even without much ETL knowledge. Second, each system supports UDFs. In Hive and Pig, an external function or user customized code for a specific task can be implemented as a UDF, and integrated into their own language, e.g., functions for serialization/deserialization data. In ETLMR, UDFs are a number of *rowhandler*s (see Section 4 and 5) integrated into *map* and *reduce* functions. These UDFs are defined for data filtering, transformation, extraction, name mapping. ETLMR also provides other ETL primitive constructs, such as hash join or merge join between data sources, and the aggregation of facts by plugable function, i.e., used as the reduce function (see Section 5). In contrast, Hive and Pig achieve the functionlity of ETL constructs through a sequence of user-written statements, which are later translated into execution plans, and executed on Hadoop. Third, as ETLMR is a specialized tool developed for fast implementation of parallel ETLs, it explicitly supports the ETLs for processing different schemas, including star schemas, snowflake schemas, SCDs, and very large dimensions. Therefore, the implementation of a parallel ETL program is very concise and intuitive for these schemas, i.e., only fact tables, their referenced dimensions and *rowhandlers* if necessary must be declared in the configuration file. Although Hive and Pig both are able to process star and snowflake schemas technically, implementing an ETL, even the most simple star schema, is not a trivial task as users have to dissect the ETL, write the processing statements for each ETL step, implement UDFs, and do numerous testing to make them correct. Moreover, as the HiveQL and Pig Latin lack UPDATE and DELETE operations, they are not able to process SCDs, which require UPDATE operation on a dimension's valid date or/and version. Fourth, ETLMR is an alternative to traditional ETL tools but offer better scalability. In contrast, Hive and Pig are obviously not optimal for the situation, where an external DW is used.

In order to make the comparison more intuitive, we create ETL programs for processing the snowflaked schema for the running example (see Fig. 6) in each of the tools. The scripts are shown in Appendix A.1, A.2 and A.3, respectively. In each script, the UDFs are not shown, but indicated by self-explaining names (starting with *UDF_*). As described, the implementation of ETLMR only includes a number of declarations in the configuration file, such as dimensions, fact tables, data sources and other parameters, and a single line to start the program. All the ETL details are transparent to users. In contrast, the scripts of Hive and Pig include the finest-level details of the ETL. In ETLMR, as only declarations are required, its script is more concise and readable, e.g., only containing 14 statements for processing the snowflaked schema (a statement may span several lines by use of "\" in Python). In contrast, the implementations consist of 23 and 40 statements by using HiveQL and Pig Latin, respectively (each statement ends with ";"). In addition, although we have a clear picture of the ETL processing of this schema, we still spent several hours to write scripts for Hive and Pig (the time of implementing UDFs is not included), and test each step. In contrast, it is of high efficiency to script in ETLMR.

A.1 ETLMR

```
# The configuration file, config.py
# Declare all the dimensions:
datedim = Dimension(name='date',key='dateid',attributes=['date','day','month',\
          'year','week','weekyear'],lookupatts=['date'])
testdim = Dimension(name='test',key='testid',defaultidvalue=-1,\
          attributes=['testname'],lookupatts=['testname'])
pagedim = SlowlyChangingDimension(name='page',key='pageid',lookupatts=['url'],
          attributes=['url', 'size', 'validfrom','validto','version','domain',\
          'serverversion'],versionatt='version',srcdateatt='lastmoddate',\
          fromatt='validfrom',toatt='validto',srcdateatt='lastmoddate')
topdomaindim = Dimension(name='topdomain',key='topdomainid',\
          attributes=['topdomain'],lookupatts=['topdomain'])
domaindim = Dimension( name='domain',key='domainid', attributes=['domain', \
          'topdomainid'], lookupatts=['domain'])
serverdim = Dimension(name='server',key='serverid',attributes=['server'],\
          lookupatts=['server'])
serverversiondim = Dimension(name='serverversion',key='serverversionid',\
          attributes = ['serverversion','serverid'], lookupatts = \
          ['serverversion'])

# Define the references in the snowflake:
pagesf = [(pagedim, [serverversiondim, domaindim]),(serverversiondim, serverdim),\
         (domaindim, topdomaindim)]

# Declare the fact table:
testresultsfact = BulkFactTable(name='testresults',keyrefs=['pageid','testid',\
          'dateid'], measures=['errors'], bulkloader=UDF_pgcopy,bulksize=5000000)

# Define the settings of dimensions, including  data source schema, UDFs,
# dimension load order:
dims ={pagedim: {'srcfields':('url','serverversion','domain','size',\
      'lastmoddate'),'rowhandlers':(UDF_extractdomain, UDF_extractserver)},\
      datedim: {'srcfields':'downloaddate',),'rowhandlers':(UDF_explodedate,) },\
      testdim:{'srcfields':('test',), 'rowhandlers':(, )},}

# Define the processing order of snowflaked dimesions:
loadorder = [('topdomaindim', 'serverdim'),('domaindim', 'serverversiondim'),\
      ('pagedim', 'datedim', 'testdim')]

# Define the settings for the fact processing:
facts = {testresultsfact:{'refdims':(pagedim, datedim, testdim),'rowhandlers':(, )},}

# Define the input data:
    inputdata = ['dfs://localhost/TestResults0.csv', 'dfs://localhost/TestResults1.csv']

# The main ETLMR program: paralleletl.py
# Start the ETLMR program:
ETLMR.load('localhost',inputdata,required_modules=[('config','config.py'),],\
          nr_maps=4,nr_reduces=4)
```

A.2 HIVE

```
-- Copy the data sources from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Create staging tables for the data sources:
CREATE EXTERNAL TABLE downloadlog(localfile STRING, url STRING, serverversion
STRING, size INT, downloaddate STRING,lastmoddate STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/downloadlog';

CREATE EXTERNAL TABLE testresults(localfile STRING, test STRING, errors INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/testresults';
```

```
-- Load the data into the staging tables:
LOAD DATA INPATH "/user/test/input/DownloadLog.csv" INTO TABLE downloadlog;
LOAD DATA INPATH "/user/test/input/TestResults.csv" INTO TABLE testresults;

-- Create all the dimension tables and fact tables:
CREATE EXTERNAL TABLE datedim(dateid INT, downloaddate STRING, day STRING,
month STRING, year STRING, week STRING, weekyear STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/datedim';

CREATE EXTERNAL TABLE testdim(testid INT, testname STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/testdim';

CREATE EXTERNAL TABLE topdomaindim(topdomainid INT, topdomain STRING) ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/topdomaindim';

CREATE EXTERNAL TABLE domaindim(domainid INT, domain STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/domaindim';

CREATE EXTERNAL TABLE serverdim(serverid INT, server STRING) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/serverdim';

CREATE EXTERNAL TABLE serverversiondim(serverversionid INT, serverversion STRING,
serverid INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/serverversiondim';

CREATE EXTERNAL TABLE pagedim(pageid INT, url STRING, size INT, validfrom STRING,
validto STRING, version INT, domainid INT,serverversionid INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE LOCATION
'/user/test/pagedim';

CREATE EXTERNAL TABLE testresultsfact(pageid INT, testid INT, dateid INT,
error INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE
LOCATION '/user/test/testresultsfact';

-- Load data into the non-snowflaked dimension tables testdim and datedim:
INSERT OVERWRITE TABLE  datedim SELECT UDF_getglobalid() AS dateid, downloaddate,
UDF_extractday(downloaddate), UDF_extractmonth(downloaddate),
UDF_extractyear(downloaddate), UDF_extractweek(downloaddate),
UDF_extractweekyear(downloaddate) from downloadlog;

INSERT OVERWRITE TABLE testdim SELECT UDF_getglobalid() AS testid, A.testname FROM
(SELECT DISTINCT test AS testname FROM testresults) A;

-- Load data into the snowflaked dimension tables from leaves to the root:
INSERT OVERWRITE TABLE topdomaindim SELECT UDF_getglobalid() AS topdomainid,
A.topdomain FROM (SELECT DISTINCT UDF_extracttopdomain(url) FROM downloadlog) A;

INSERT OVERWRITE TABLE domaindim SELECT UDF_getglobalid() AS domainid, A.domain,
B.topdomainid FROM (SELECT DISTINCT UDF_extractdomain(url) AS domain,
UDF_extracttopdomain(url) AS topdomain FROM downloadlog) A JOIN topdomaindim B
ON (A.topdomain=B.topdomain);

INSERT OVERWRITE TABLE serverdim SELECT UDF_getglobalid() AS serverid, A.server
FROM (SELECT DISTINCT UDF_extractserver(serverversion) AS server FROM downloadlog) A;

INSERT OVERWRITE TABLE serverversiondim SELECT UDF_getglobalid() AS serverversionid,
A. serverversion, B.serverid FROM (SELECT DISTINCT  serverversion,
UDF_extractserver(serverversion) as server FROM downloadlog) A JOIN serverdim B
ON (A.server=B.server);

INSERT OVERWRITE TABLE pagedim SELECT UDF_getglobalid() AS pageid, A.url, A.size,
A.lastmoddate, B.domainid, C.serverversionid FROM (SELECT url, size, lastmoddate,
UDF_extractdomain(uri) AS domain, serverversion FROM downloadlog) A JOIN domaindim B
ON (A.domain=B.domain) JOIN serverversiondim C JOIN (A.serverversion=C.serverversion);

CREATE EXTERNAL TABLE pagedim_tmp(pageid INT, url STRING, size INT, lastmoddate
STRING, domainid INT, serverversionid INT) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE LOCATION '/user/test/pagedim_tmp';
```

```
-- Load data into the fact table testresultstact:
INSERT OVERWRITE TABLE testresultsfact SELECT C.pageid, E.testid, D.dateid,
B.errors FROM downloadlog A JOIN testresults B ON (A.localfile=B.localfile)
JOIN pagedim C ON (A.url=C.url) JOIN datedim D ON
(A.downloaddate=D.downloaddate) JOIN testdim E ON (B.test=E.testname);
```

A.3 PIG

```
-- Copy the data from local file system to HDFS:
hadoop fs -copyFromLocal /tmp/DownloadLog.csv /user/test;
hadoop fs -copyFromLocal /tmp/TestResults.csv /user/test;

-- Load the data into PIG:
downloadlog = LOAD 'DownloadLog.csv' USING PigStorage('\t')
              AS (localfile, url, serverversion,size,downloaddate,lastmoddate);
testresults = LOAD 'TestResults.csv' USING PigStorage('\t')
              AS (localfile, test, errors);

-- Load the dimension table testdim:
testers = FOREACH testresults GENERATE test AS testname;
distincttestname = DISTINCT testers;
testdim = FOREACH distincttestname GENERATE UDF_getglobalid()
          AS testid, testname;
STORE testdim INTO '/tmp/testdim' USING PigStorage();

-- Load the dimension table datedim:
downloadates = FOREACH downloadlog GENERATE downloaddate;
distinctdownloadates = DISTINCT downloadates;

datedim = FOREACH distinctdownloadates GENERATE UDF_getglobalid()
AS dateid, downloaddate, UDF_extractday(downloaddate) AS day,
UDF_extractmonth(downloaddate) AS month, UDF_extractyear(downloaddate) AS year,
UDF_extractweek(downloaddate) AS week, UDF_extractweekyear(downloaddate)
AS weekyear;

STORE datedim INTO '/tmp/datedim' USING PigStorage();

-- Load the snowflaked dimension tables:
urls = FOREACH downloadlog GENERATE url;

serverversions = FOREACH downloadlog GENERATE serverversion;

domains =  FOREACH downloadlog GENERATE UDF_extractdomain(url) AS domain;

distinctdomains = DISTINCT domains;

topdomains = FOREACH distinctdomains GENERATE UDF_extracttopdomain(domain)
             AS topdomain;

distincttopdomains = DISTINCT topdomains;

topdomaindim = FOREACH distincttopdomains GENERATE UDF_getglobalid()
               AS topdomainid, topdomain;
STORE topdomaindim INTO '/tmp/topdomaindim' USING PigStorage();

ndomains = FOREACH distinctdomains GENERATE domain AS domain,
           UDF_extracttopdomain(domain) AS topdomain;

ndomainjoin = JOIN ndomains BY topdomain, topdomaindim BY topdomain;

domaindim = FOREACH ndomainjoin GENERATE UDF_getglobalid()
            AS domainid, domain, topdomainid;
STORE domaindim INTO '/tmp/domaindim' USING PigStorage();

distinctserverversions = DISTINCT serverversions;
```

```
nserverversions = FOREACH distinctserverversions GENERATE serverversion
                AS serverversion, UDF_extractserver(serverversion) AS server;

servers = FOREACH nserverversions GENERATE server AS server;
distinctservers = DISTINCT servers;

serverdim = FOREACH distinctservers GENERATE UDF_getglobalid() AS serverid, server;
STORE serverdim INTO '/tmp/serverdim' USING PigStorage();

nserverversionjoin = JOIN nserverversions BY server, serverdim BY server;

serverversiondim = FOREACH nserverversionjoin GENERATE UDF_getglobalid()
                  AS serverversionid, serverversion, serverid;

STORE serverversiondim INTO '/tmp/serverversiondim' USING PigStorage();

joindomservversion = JOIN (JOIN downloadlog BY UDF_extractdomain(url),
                     domaindim by domain) BY serverversion, serverversiondim
                     BY serverversion;

pagedim = FOREACH joindomservversion GENERATE UDF_getglobalid() AS pageid,
          url, size, lastmoddate, serverversionid, domainid;

STORE pagedim INTO '/tmp/pagedim' USING PigStorage();

-- Load the fact tables:
testresults = JOIN downloadlog BY localfile, testresults BY localfile;

joinpagedim = JOIN testresults BY url, pagedim BY url;

joindatedim = JOIN joinpagedim BY downloaddate, datedim BY downloaddate;

jointestdim = JOIN joindatedim BY test, testdim BY testname;

testresultsfact = FOREACH jointestdim GENERATE dateid, pageid, testid, errors;

STORE testresultsfact INTO '/tmp/testresultsfact' USING PigStorage();
```

The Planning OLAP Model – A Multidimensional Model with Planning Support

Bernhard Jaecksch and Wolfgang Lehner

TU Dresden, Institute for System Architecture,
Database Technology Group, 01062 Dresden, Germany
bernhard.jaecksch@mailbox.tu-dresden.de, wolfgang.lehner@tu-dresden.de

Abstract. A wealth of multidimensional OLAP models has been suggested in the past, tackling various problems of modeling multidimensional data. However, all of these models focus on navigational and query operators for grouping, selection and aggregation. We argue that planning functionality is, next to reporting and analysis, an important part of OLAP in many businesses and as such should be represented as part of a multidimensional model. Navigational operators are not enough for planning, instead new factual data is created or existing data is changed. To our knowledge we are the first to suggest a multidimensional model with support for planning. Because the main data entities of a typical multidimensional model are used both by planning and reporting, we concentrate on the extension of an existing model, where we add a set of novel operators that support an extensive set of typical planning functions.

1 Introduction

With the rise of decision-support systems and the use of data warehouses in many modern companies in order to tame and harness the massive amounts of data that are collected during business execution, the research community devised various models to support the multidimensional analysis in the process of On-Line Analytical Processing (OLAP) [1]. The common data entities to model such multidimensional data are so-called cubes consisting of a set of orthogonal dimensions and mostly numerical fact-data that is characterized by the values of the different dimensions. The main aspect of OLAP is the navigation through and aggregation of the multidimensional data. Therefore, the models provide an algebra of operators that often contain typical operators of relational algebra transferred to the multidimensional scenario and extended by navigational operators to group, select and aggregate data, also termed as slice/dice and roll-up/drill-down operations.

Business planning is an important task in many companies where business targets are defined for future periods in order to provide specific guidelines for current operations and a means of comparison whether goals have been reached or not. As such, planning is an important part of many practically used decision-support-systems. However, to our knowledge, none of the existing multidimensional models supports planning functionality and as a result specific planning

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 32–52, 2013.

applications are necessary to provide planning functionality. We strive to overcome the limitation of existing models to support planning functionality as part of OLAP. As the basic data entities are the same for planning and reporting, we build on an existing OLAP model an extend its set of operations with novel operators to support a list of typical planning functions. In addition to [2] we discuss computational complexity for our proposed operators and introduce a new operator, which is essential to planning to create new facts, since all previous operators we have introduced are based on existing fact data and only modify or transform it.

The paper is structured as follows: in the next section we briefly describe related work in the field of OLAP models as well as the multidimensional model that serves as foundation for our OLAP model with planning support and why we have chosen it. Section 3 introduces a list of typical planning functions by example. Our novel operators to support planning are introduced in Section 4 where we show how to express the planning functions with the set of extended operators. We finish with a conclusion in Section 5 providing an outlook for an implementation of our model.

2 Foundation and Related Work

Starting with the extension of SQL with the Data Cube operation by Gray et al. [3], a wealth of multidimensional models have been proposed. Similar to the Data Cube, the first models by Li et al. [4] and Gysses et al. [5] were extensions to the relational model. Their main focus are operators to group and aggregate relational data and provide a sound model for these operations. Yet, they are all based on relational algebra and do not introduce a true concept of multidimensionality. The field of statistical databases also dealt with the quantitative analysis of large amounts of scientific data and, faced with similar problems of grouping and aggregating, suggested a range of models with a strong focus on statistical analysis. Prominent candidates are the Summary Tables model by Ozsoyoglu et al. [6] and the graphical model for Statistical Object Representation (STORM) by Rafanelli et al. [7]. Summary tables focus on a tabular representation of grouped data by introducing tables that have set-valued attributes and provide operators to manipulate and navigate these set-valued relations and summary tables. Again, the focus is on relational data and the concept of multi-dimensionality is only implicitly addressed. The STORM model directly addresses the concept of multidimensionality and the issues and limitations that arise from strictly staying to a 2-dimensional tabular representation. It also introduces a meta-data level that can convey more information about the relation between different model entities and categories as can be in a pure data representation in tabular form. While all these models in principal divide the data into *qualifying* and *quantifying* information, most modern models are based on the concept that the qualifying information defines a multidimensional space represented by a cube where each axis is called a dimension and the quantifying information at the intersection points, called measures or facts, is characterized

by the dimensions. Typical and often cited representatives are the Multidimensional Database Model by Agrawal et al. [8], the F-Table Calculus by Cabibbo et al. [9], the Cube Operations model by Vassiliadis et al. [10], the Multidimensional Object model by Lehner [11] and the Cube Data model by Datta et al. [12]. Vassiliadis provided a good classification and survey of these models in [13]. They compare them whether they are cube oriented or not, whether they provide procedural and or declarative query languages, whether they support hierarchies and provide a relational mapping or not. While these general properties are important, Pedersen et al. [14] evaluated the suitability of the models to implement a practical and complex data warehouse scenario. They did so according to an extensive set of typical requirements of a data warehouse implementation such as explicit hierarchies, multiple and flexible hierarchies per dimension, symmetric treatment of dimensions and measures and explicit aggregation semantics. Since none of the previous models fulfilled all requirements they suggested their own model, the Extended Multidimensional Data model (EMDM). However, all models that we listed here, do not cover the aspect of planning in a data warehouse. Instead of only drilling through data and slicing and dicing it, during planning new data is created and existing data is transformed and modified. All while staying consistent with the defined data model. Therefore, all these models lack a set of operators to address the characteristics of planning and therefore, we propose an extension for one of the models to explicitly support planning functionality. As EMDM satisfies all the requirements defined by Pedersen et al. [14] we considered it a suitable foundation for our planning extensions.

The basic EMDM model entity is the multidimensional object $MO = (S, F, Dim, R)$, which is a four-tuple consisting of an *n-dimensional fact schema* S, a set of facts F, a set of *dimensions* Dim and a set of corresponding *fact-dimension relations* R, that map the facts to elements of the dimensions. A key aspect of the model is that everything that characterizes a fact is regarded dimensional. That includes measures and as such dimensions and measures are treated symmetrically. An n-dimensional fact schema S is a two-tuple (FS, D) with FS describing a fact type and D being a set of dimension types $D = \{T_i, i = 1..n\}$. Each dimension type T itself is a four-tuple $(C, \prec_T, \top_T, \bot_T)$, where C is a set of category types $\{C_j, j = 1..k\}$ of T that form a partial ordering $\prec_T$ with $\top_T$ and $\bot_T$ as the top and bottom elements of the ordering. There is always a single top element that contains all other elements. For certain category types it often makes sense to aggregate them with an aggregation function. To support the different aggregation types in the model, three different classes of aggregation functions exist: constant c, average functions ϕ and sum functions Σ. For these classes an ordering exists such that $c \subset \phi \subset \Sigma$. For each dimension type T the model provides a function that determines the aggregation type for a category type. A dimension Dim_i has a dimension type T that is defined in the fact schema of an MO as explained in the previous section. $Dim_i = (Ca, \prec)$ is a two-tuple with Ca being a set of categories $\{Ca_j\}$ and $\prec$ a partial ordering on all dimension values e in each category Ca_j with $Type(e) = C_j$. Furthermore, all values e in the dimension Dim_i are smaller than value $\top$ and the most granular

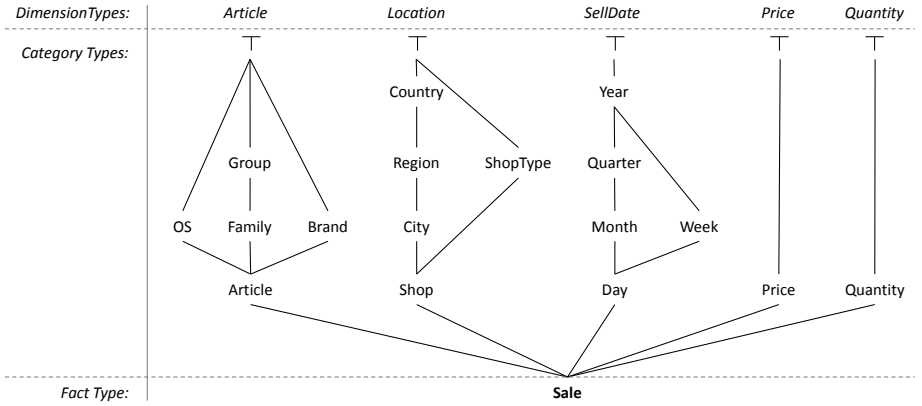


Fig. 1. An example schema

values are contained in category $\perp_T$. To establish a connection between facts and dimensions, fact-dimension relations are introduced. A fact-dimension relation R is a set of two-tuples $\{(f, e)\}$ where f is a fact and e is a dimension value. Therefore, the fact is characterized by the dimension value e. Values from different dimension categories can determine the same fact. Also it must be ensured in the model that each fact in R is characterized by at least one dimension value. Thus, if there is no suitable dimension value to characterize a fact, the value $\top$ is used. Based on these entities an algebra with a list of operators is part of the model. As basic operators, all operations from relational algebra like *selection*, *projection*, *rename*, *union*, *difference* and *join* are adopted to operate on MOs. In addition, the *aggregate formation* operator allows to build aggregates and group facts. Typical OLAP operators like roll-up, *drill-down*, *SQL-like aggregation* and *star-joins* are expressed in terms of the basic operators.

3 Common Planning Functions by Example

The example used throughout the paper has the schema shown in Figure 1, consisting of 5 dimensions with dimension types *Article, Location, SellDate, Price* and *Quantity*. They characterize the *Sale* of a product. Usually, *Price* and *Quantity* would be considered as measures, so we call them the measure dimensions. Each dimension consists of different categories that form one or more hierarchies per dimension. For the two dimensions *Article* and *Location* Figures 2 and 3 show dimension values and their partial ordering. The measure dimensions contain numerical values, where quantities are integral numbers and prices are values taken from real numbers. In Table 1 we list the facts for our example in column 1 and each row represents a fact. The other columns represent the dimensions that characterize the fact and the value(s) in each row represent the dimension value(s) forming, together with the fact, (an) element(s) of the fact-dimension relations.

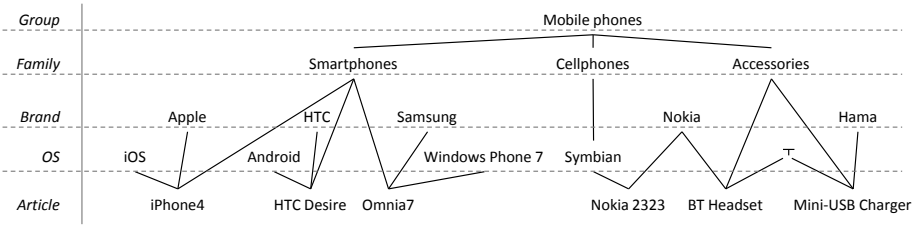


Fig. 2. Dimension values and partial ordering for dimension *Article*

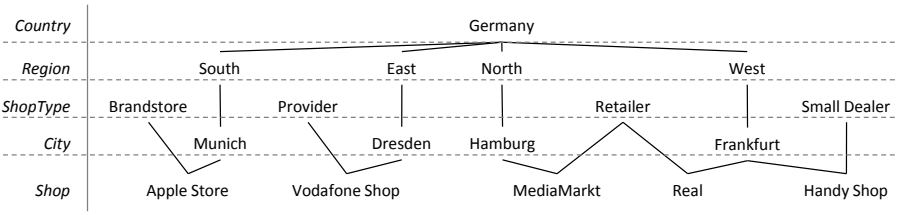


Fig. 3. Dimension values and partial ordering for dimension *Shop*

Table 1. Lists of facts in the example schema

Fact	$R_{Article}$	$R_{Location}$	$R_{SellDate}$	R_{Price}	$R_{Quantity}$
f_1	iPhone4,iOS, Apple,Smartphones	AppleStore,Munich, BrandStore	2011-05-06, 05, Q2, 2011	699.00	50
f_2	Desire,Android, HTC,Smartphones	Vodafone Shop, Dresden,Provider	2011-04-23, 04, Q2, 2011	479.00	35
f_3	Omnia 7,Windows Phone 7,Samsung, Smartphones	Media Markt, Hamburg,Retailer	2011-12-14, 12, Q4, 2011	389.00	10
f_4	2323,Symbian, Nokia, Cellphones	Real,Frankfurt, Retailer	2011-01-11, 01, Q1, 2011	79.95	110
f_5	BT Headset,Nokia, Accessories	HandyShop,Dresden, Smalldealer	2011-03-27, 03, Q1, 2011	24.55	70
f_6	USB Charger,Hama, Accessories	Real,Frankfurt, Retailer	2011-08-13, 08, Q3, 2011	12.99	45

The following example introduces a list of common planning functions in a typical business planning scenario. Our model should be able to express each of these functions. Assume a company that sells phones and accessories to shops and retailers. Our example schema shows a multidimensional object with a list of facts that capture sales for the year 2011. It is the task of a controller to plan sales quantities and prices for year 2012.

Step 1. As a first step he wants to base his plan on the values of the previous year and therefore he needs a planning function that **copies** data from 2011 into year 2012. The new MO would now contain twice as much facts as before.
Step 2. Because of recent market trends the company decides to sell only smartphones in 2012 and therefore a **delete** planning function deletes all standard cellphone sales from 2012.
Step 3. For the year 2012 the prices are lowered by 5% compared to the previous year. Therefore the planner calls a **revalue** planning function to apply these changes. Furthermore, he wants to know the planned revenue that is based on sales quantities and price.
Step 4. For a each retailer the planner requests the estimated quantity of sold items from the sales person that is responsible for this customer. These quantities are now entered to the plan at aggregated customer level and must be distributed to individual facts using a **disaggregation** planning function.
Step 5. Finally, the controller wants to use the current data of 2011 and the planned data of 2012 to predict a sales quantity trend for 2013. He requires a planning function that calculates a **forecast** and generates a set of forecasted facts.

After all these steps, the complete plan data is created and can be used for reports and comparisons with the actual data of 2012. The list of planning functions that was involved includes copying data, deletion of facts, calculating expressions to revalue quantitative values, distribute new values entered at an aggregated level to the most granular fact level and calculate new values using a forecasting function.

4 An OLAP Model for Planning

In the following section we extend the EMDM and develop our novel model algebra to support planning. One major aspect of planning is that it changes the fact data. All the models existing so far make the assumption that the fact data is a read only set of values and all operators have navigational, i.e., read only semantic. With planning new facts will be created or existing facts are manipulated. Therefore, the novel operators for planning must support this. From the list of basic planning functions shown in Section 3 not all require a separate operator. Similar to the original model, where typical OLAP operators like roll-up and drill down are expressed in terms of the basic aggregation formation operator, we only need a few basic operators to keep the extended algebra simple and minimal. In addition we give a short complexity estimation for each basic operator, in order to evaluate the feasibility of an actual implementation of each operator.

4.1 Basic Planning Operators

Create Fact Combinations. From a business point of view it is sometimes suggested to start planning with an empty sheet, i.e. not to immediately reuse the

values from previous plans or previous years, in order to avoid wrong prospective plan values by using historical values as a misleading orientation. To start with a clean slate, a multidimensional object might not contain any facts at all at the beginning. Also, a plan usually is done before execution and hence it is common in planning to start out with no factual data. Since all other operators of this model rely on existing factual data, this following operator is important to create new fact data that can then be modified and transformed by the other basic operators. To create a set of facts, for example, for all combinations of articles and locations the *create fact combinations* operator can be used. Starting with a possibly empty MO, a list of dimension categories $Ca_1, .., Ca_k$ and a list of dimension values $e_{k+1}, .., e_n$ the operator creates a new multidimensional object MO' that contains a set of newly created facts and respective fact-dimension relations contain new entries for each combination of dimension values $(e_1, .., e_n)$ with $e_1, .., e_n \in Ca_1 \times \cdots \times Ca_k \times e_{k+1} \times \cdots \times e_n$. For all dimensions where no $Ca_i, i = 1..k$ is given, but an explicit dimension value, the fact is characterized by given value e_i of this dimension. We formally define the create combinations operator as

Definition 1. $\otimes [Ca_1, .., Ca_k, e_{k+1}, .., e_n] (MO) = MO' = (S', F', Dim', R')$ *where*

1. $S' = S$
2. $F' = \{f' | f' \Rightarrow_1 e_1 \wedge \cdots \wedge f' \Rightarrow_n e_n \wedge (e_1, .., e_n) \in Ca_1 \times \cdots \times Ca_k \times e_{k+1} \times \cdots \times e_n \vee (f' \in F)\}$
3. $Dim' = Dim$
4. $R' = \{R'_i, i = 1..n\}$
5. $R'_i = \{(f', e'_i) | f' \in F' \wedge e_i = e'_i \wedge \forall (e_1, .., e_n) \in Ca_1 \times \cdots \times Ca_k \times e_{k+1} \times \cdots \times e_n\}$

As an example consider an $MO = (S, F, Dim, R)$ from our example schema, which has been filtered to contain only facts where the *SellDate* is 2012. Beginning with the sets of facts in Table 1 the MO would have an empty set of facts $F = \emptyset$, $R = \{R_i, i = 1..n\}$ and each fact-dimension relation $R_i = \emptyset$ would be empty. If we now apply the create fact combinations operator with $Ca_{Article}, Ca_{Shop}, e_{Year} = 2012$ and $e_{Price} = 0.00, e_{quantity} = 0$ the operator generates $6 \cdot 5 = 30$ new facts and corresponding fact-dimension entries for a combination of each of the six Articles and five Shops. It is not necessary that the input MO contains no facts. In that case, each existing fact is kept and only missing combinations are added. Furthermore, the operator creates new facts on every level of granularity. It is no requirement that the $Ca_i, i = 1..k$ parameters are all categories with the finest granularity level. If the example parameters are slightly modified Ca_{OS}, Ca_{City} then the resulting set of facts would represent all combinations of smartphone operating systems in all cities and the granularity for the Article and Location dimensions would be greater than $\perp_{Article}$ and $\perp_{Location}$ respectively. As this operation can produce possibly invalid facts

referring to invalid combinations of dimension values, a join can be used to restrict the result to a set of valid facts.

The complexity of this operator is defined by the number of non-constant categories that are involved in the operation and their cardinality. Let $|Ca_1| = N_1, .., |Ca_n| = N_n$ then $O(\otimes) = O(N_1 \times \cdots \times N_n) = O(N^n)$.

Disaggregation. After having created a set of new facts or reusing a copy of existing facts, a typical planning function is to assume a new aggregated value for a group of facts and then calculate the individual contribution to the target value for each fact according to some function. This is usually called disaggregation (or distribution) as it can be viewed as reverse operation to the aggregation. In contrast to the drill-down operation that only *displays* existing detailed values that make up a sum, the disaggregation operation defines a new sum value and *changes* all contributing facts and values of the measure dimensions accordingly. With disaggregation being an important function for planning, the Planning OLAP model defines a disaggregation operator similar to the aggregation formation operator for normal OLAP.

In our algebra disaggregation can be seen as inverse operator α^{-1} to the aggregate formation, although it is not strictly the inverse as it can not directly invert the result of an aggregate formation operator. As input it takes a set of dimension values $e_1, \ldots, e_n$ that define the level where the new sum value is entered. Thus it does not change the fact-schema S as is the case with the aggregation operator. Additional parameters are the distribution function $g^{-1}(e_{current}, e_{new}, e_{old})$ and an aggregate function g that determines how the values of the reference dimension are aggregated. The distribution function g^{-1} takes as input the current dimension value $e_{current}$, the new sum e_{new} to be distributed and the old sum e_{old} that is obtained as result of applying g to the reference dimension. Finally, the new dimension value e_{new} that should be disaggregated, is given with index t of the target dimension and index r of the reference dimension. It is allowed that $t = r$, in which case the new value is distributed according to the original fractions of that dimension. It follows the formal definition of the disaggregation operator:

Definition 2. $\alpha^{-1}\left[e_1, \ldots, e_n, g^{-1}, g, e_{new}, t, r\right](MO) = MO' = (S', F', Dim', R')$ *where*

1. $S' = S$
2. $F' = F$
3. $Dim' = \{Dim_i, i = 1..n \wedge i \neq t\} \cup \{Dim'_t\}$
4. $Dim'_t = (Ca'_t, \prec'_t)$
5. $\prec'_t = \prec_{t_{|Dim'}}$
6. $Ca'_t = \{Ca'_{tj} \in Dim_t | (Type(Ca'_{tj}) = \perp_{Dim_t} \wedge$
   $SUM(Group(e'_1, .., e'_n)) = e_{new} \wedge (e'_1, .., e'_n) \in Ca'_1 \times \cdots \times Ca'_n \wedge$
   $e_{old} = g(Group(e_1, .., e_n)) \wedge (e_1, .., e_n) \in Ca_1 \times \cdots \times Ca_n \wedge$
   $e'_{tj} = g^{-1}(e_{rj}, e_{new}, e_{old}) \wedge e_{rj} \in Ca_{rj} \wedge Type(Ca_{rj}) = \perp_{Dim_r}) \vee$
   $Type(Ca'_{tj}) = Ca_{tj}\}$

7. $R' = \{R'_i, i = 1..n \wedge i \neq t\} \cup \{R'_t\}$
8. $R'_i = \{(f', e') | f' \in F' \wedge e' \in Dim'_i\}$
9. $R'_t = \{(f', e'_i) | \exists (e_1, .., e_n) \in Ca_1 \times \cdots \times Ca_n \wedge f' \in F' \wedge e'_i \in Dim'_t \wedge e'_i = g^{-1}(e_{ri}, e_{new}, e_{old}) \wedge \forall e_i \in Dim_t \exists e_{ri} \in Dim_r\}$

We now begin to dissect the result of this operator and explain each expression in detail. The fact-schema S' of MO' is the same as that of the original multidimensional object, because only values are changed and no dimensions are added or restricted. The set of facts is the same as that of the input MO. The reason is, the disaggregation operator does not introduce new facts, but only maps facts for the target dimension to new values. The set of dimensions is again taken from the original MO, but the target dimension Dim'_t changes in the sense that intuitively the new dimension values are calculated based on the new sum and fractions of the given reference dimension Dim_r. Category attributes Ca'_{tj} of the target dimension are either the top attribute or they are in the class of the most granular attribute and their new dimension values are calculated using distribution function g^{-1}. The distribution function calculates new dimension values e'_{tj} using new sum e_{new} as input, the old reference aggregate value e_{old} and the respective dimension value e_{rj} of the reference dimension. The aggregate value e_{old} is obtained by applying the reference aggregate function g to the grouping of $Group(e_1, .., e_n)$ at the level of the given input dimension values. Finally, the fact-dimension mapping R'_t is adapted such that facts are now mapped to new dimension values calculated by the distribution function. This includes the requirement that for each fact-dimension mapping in the target dimension there exists a fact-dimension mapping in the reference dimension.

By allowing arbitrary functions for the distribution function g^{-1} and the reference aggregate function g, different types of distribution can be achieved. For example, a standard distribution function that calculates the new fraction based on the percentile of the reference value from the old sum value is $g^{-1}_{std}(e_r, e_{new}, e_{old}) = e_r \cdot e_{new}/e_{old}$ together with $g = SUM$. As already mentioned, the reference dimension can be the same as the target dimension. To arrive at a uniform distribution, the distribution function is $g^{-1} = e_{new}/e_{old}$ and the reference aggregate function to obtain e_{old} should be $g = COUNT$. For a constant distribution $g^{-1}(e_r, e_{new}, e_{old}) = e_{new}$ and the reference function can return an arbitrary value, since it is ignored in this case.

In theory the distribution function calculates an exact value and thus the complexity of the disaggregation is $O(|Ca_t|) = O(N)$ with N being the cardinality of the target dimension level. In a practical implementation one has to address rounding issues, since an exact fraction cannot always be stored. An implementation might need to distribute remainders in a second step. However, the worse case complexity is still linear bound by N. The following three examples illustrate how disaggregation is applied to our data set from Section 3.

Example 1. *In the first example we want to distribute a new article quantity of* 384 *to all sales facts in Germany for year 2011. The input MO contains all five dimensions Article, Location, SellDate, Price and Quantity and is not restricted. The parameters for the disaggregation are:*

$$\alpha^{-1}[\top_{Article}, Germany, 2011, \top_{Price}, \top_{Quantity}, g_{std}^{-1}, g = SUM, e_{new} = 384, t = 5, r = 5](MO)$$

The distribution function g_{std}^{-1} *is the standard function explained in the previous paragraph and the reference aggregate function* g *is* SUM*. The target dimension Quantity contains the new dimension values that would result in the increased sum* 384 *(old sum is* 320*) when the reverse operation, i.e. the aggregate formation, would be applied to the resulting* MO'*. The disaggregation affects all facts* $f_1, .., f_6$ *in the example and fact-dimension mapping* $R_{Quantity}$ *would change from*

$$\{(f_1, 50), (f_2, 35), (f_3, 10), (f_4, 110), (f_5, 70), (f_6, 45)\}$$

to

$$R'_{Quantity} = \{(f_1, 60), (f_2, 42), (f_3, 12), (f_4, 132), (f_5, 84), (f_6, 54)\}$$

Example 2. *The second example continues with* MO' *obtained by the previous example. We want to raise for all Retailers in the city of Frankfurt the quantities to be sold by another* 50% *and set it from its value of* 186 *in* MO' *to* 279*. We apply the disaggregation operator to* MO' *obtained from Example 1 with the following parameters:*

$$MO'' = \alpha^{-1}[\top_{Article}, Frankfurt, Retailer, 2011, g^{-1}, g, e_{new} = 279, t = 5, r = 5](MO')$$

This time the new sum does not affect all facts in the multidimensional object, but only the ones that match the given dimension values for category attributes Location and ShopType. Hence, the parameters of the disaggregation operator describe a selection. Again, only dimension values of the target Quantity dimension change and the corresponding dimension mapping: $R''_{Quantity} = \{(f_1, 60), (f_2, 42), (f_3, 12), (f_4, 198), (f_5, 84), (f_6, 81)\}$*. The distribution function and reference aggregate function are the same as in Example 1.*

Example 3. *The third example sets a new Price for all Accessories. This time we want a constant distribution and therefore* $g_{const}^{-1} = e_{new}$ *sets the new value of the Price dimension and adjusts the fact-dimension mapping for the selected facts such that* f_5 *and* f_6 *now map to the new price. The disaggregation operator is called as follows:*

$$MO''' = \alpha^{-1}\left[Accessories, \top_{Shop}, 2011, g_{const}^{-1}, g, e_{new} = 15.99, t = 6, r = 6\right](MO'')$$

The resulting fact dimension mapping R_{Price} is

$$R'''_{Price} = \{(f_1, 699.00), (f_2, 479.00), (f_3, 389.00), (f_4, 79.95), (f_5, 15.99), (f_6, 15.99)\}$$

Duplication. To introduce new facts, as it is necessary for a copy function, we add the duplication operator to the model, which duplicates a multidimensional object by duplicating the underlying facts of the MO. The resulting MO' has identical dimension structure and fact-dimension relations. With the exception that for each fact f in the original MO, there is a new fact f' in the result MO' that is characterized by the same dimension attributes and values, but has a different identity such that $f \neq f'$. This results in the following definition for the duplication operator:

Definition 3. *Duplication is defined as* $\tau(MO) = MO' = (S', F', Dim', R')$ *where*

1. $S' = S$
2. $F' = \{f' | \exists f \in F \wedge e_1 \rightarrow_1 f, .., e_n \rightarrow_n f \wedge f' \neq f \wedge e'_1 \rightarrow_1 f'_1, .., e'_n \rightarrow_n f'_n \wedge e_1 = e'_1 \wedge \ldots \wedge e_n = e'_n\}$
3. $Dim' = Dim$
4. $R' = \{R'_i, i = 1..n\}$
5. $R'_i = \{(f', e'_i) | \forall (f, e_i) \in R_i \wedge f' \in F' \wedge e'_i \in Dim'_i\}$

Complexity for the duplication operator is linear in the number N of facts that need to be duplicated $O(N)$.

Value Mapping. Another basic operation that is important, for example, for the copy function, is *value mapping*. When new planning data is generated one can either use the create combinations operator as explained earlier or one can copy facts, for example, from a previous year. That means, for a set of facts one or more dimension values change. Referring to the former example, the values for the time dimension must change from one year to another. To apply this change, the mapping operator takes as input a set of mapping functions, which map a combination of source dimension values to corresponding target values. As fact-dimension relations are the glue between facts and dimensions, the mapping operator has to modify these relations. We can now formally define the value mapping operator.

Definition 4. *We denote the mapping operator with* $v\left[\{m_r, r = 1..s\}\right](MO) = (S', F', Dim', R')$*. As parameters it takes a set of mapping functions* m_r *which have the form* $m_r(e_1, .., e_i, .., e_n) \mapsto e'_i$ *and*

1. $S' = S$
2. $F' = F$
3. $Dim' = Dim$
4. $R' = \{R'_i, i = 1..n\}$
5. $R'_i = \{(f', e'_i) | f' \in F' \wedge e'_i = m_r(e_1, .., e_i, .., e_n) \wedge e_1 \rightarrow_1 f, .., e_i \rightarrow_i f, .., e_n \rightarrow_n f \wedge e'_i \rightarrow_i f'\}$

Intuitively a mapping function m_r is applied to a fact-dimension relation R_i and, depending on the mapping function, it maps the input value to itself or, for matching values, maps them to a different dimension value. Furthermore, it

is important that the mapping function is aware of dimension hierarchies and provides a complete mapping from one part of the hierarchy lattice structure to another part. The runtime of an implementation of the mapping operator will depend on the structure of the dimension hierarchy that is mapped as well as on the complexity of the mapping function. In most cases, one could use a hash-map approach for the mapping function that for each value looks up the mapping result. Thus the complexity of the mapping operation is bound by the number of elements in the dimension hierarchy that is mapped and the cost for the mapping function. As a result the complexity of this operator is $O(N + h_c)$ where N is the number of elements in the dimension hierarchy, h_c the cost of the hash-map creation and a hash lookup is assumed to have a fixed cost $O(1)$.

If we consider, for example, the *SellDate* hierarchy of our example schema and want to map from year 2011 to year 2012, a mapping function $f_{2011 \mapsto 2012}$ must provide a mapping for all dimension values that are in the partial ordering below 2011 to the respective values in the hierarchy below 2012. Thus *Q1-2011* would be mapped to *Q1-2012*, *W32-2011* to *W32-2012* and so on.

Calculated Dimensions. Another cornerstone of planning functionality is to calculate various expressions, like arithmetic expressions or boolean expressions on multidimensional data. It is therefore necessary to allow multidimensional objects to participate in expressions. This is similar to the model of Lehner [11] where unary and binary operators perform cell-based operations on one or two multidimensional objects. A cell references a numeric fact value or in more general terms a measure. In the Extended Multidimensional Data Model that our Planning OLAP model is based on, we want to achieve something similar. Because, everything is a dimension in the model and facts are objects that are purely described by dimension values, an expression is not a cell-based operation. Instead it is an operation on dimension values. In order to realize expressions within our Planning OLAP model, we define a *calculated dimension* $\overline{Dim}$ of type $T = (C_j, \prec_T, \top_T, \bot_T)$ similar to basic dimensions as a two-tuple $\overline{Dim} = (Ca, \prec)$. The set Ca contains categories of the dimension and $\prec$ is the partial ordering on all dimension values of that dimension. The difference is that a category attribute $Ca_i \in \overline{Dim}_{n+1}$ is now defined in terms of an expression where the operands are category attributes of other dimensions $Ca_i = \otimes(Ca_j, Ca_k)$ with $Ca_j \in Dim_r$, $Ca_k \in Dim_s$ and $\otimes$ being an arbitrary binary operator from the following list $\{+, -, \cdot, /, \wedge, \vee\}$. In the same manner expressions that contain arbitrary scalar functions and operators are possible by extending the definition of a calculated category type to the general form $Ca_i = \otimes(\{Ca_j\})$ where $Ca_j \in Dim_r, i \in \{1 \dots l\}, j \in \{1 \dots m\}, r \in \{1 \dots n+1\}, i \neq j$ and $\otimes$ is an arbitrary n-nary operator or scalar function applied to a number of category attributes. It is possible that expressions reference other category attributes of the calculated dimension. This is useful for example to add a constant to the expression by defining a category attribute that only has one dimension value and reference it in other expressions. To calculate such an expression, it is necessary to add a calculated dimension to a multidimensional object. Introducing the *add dimension* operator to the model accomplishes just that. The operator takes as

input a multidimensional object MO and a calculated dimension $\overline{Dim}_{n+1}$ where the calculated category attributes can reference categories of the input MO.

Definition 5. *We define the operator as* $MO' = +\left[\overline{Dim}_{n+1}\right](MO) = (S', F', Dim', R')$ *with*

1. $S' = (FS', D')$
2. $D' = \{T'_i, i = 1..n\} \cup \{T_{n+1}\}$
3. $T'_i = T_i$
4. $F' = F$
5. $Dim' = \{Dim'_i, i = 1..n\} \cup \{Dim_{n+1}\}$
6. $\overline{Dim}'_i = Dim_i$
7. $\overline{Dim}_{n+1} = (Ca_{n+1}, \prec_{n+1})$
8. $Ca_{n+1} = \{Ca_{n+1,i} = \otimes(\{Ca_{rj} | Ca_{rj} \in Dim_r, i \in \{1 \dots l\}, j \in \{1 \dots m\}, r \in \{1 \dots n\}\})\}$
9. $R' = \{R'_i, i = 1..n\} \cup \left\{R'_{n+1}\right\}$
10. $R'_i = R_i$
11. $R'_{n+1} = \left\{(f', e'_{n+1,i}) | f' \in F' \wedge e'_r i \to f' \wedge e'_{n+1,i} = \otimes(e'_{ri}) \wedge e'_{ri} \in Ca_r\right\}$

To calculate an expression between values of two different MOs, first a join operator should be applied to create a combined MO and then a calculated dimension containing the expression is added. To give a complexity analysis for this operator is not directly feasible, since it is a means of the model to express dimensions that are calculated based on other dimensions. Although, it depends on the complexity of the calculation, in most cases it will be $O(1)$, since the calculated expression only depends on the number of values that it references in other dimensions.

Example 4. *As an example we will calculate the revenue for our mobile phone data. Therefore we add a dimension* $Dim_{revenue} = (Ca, \prec)$ *with* $Ca = (Revenue, \top)$, $Revenue = Qty \cdot Price$ *and add this dimension to our multidimensional object* $+[Dim_{Revenue}](MO)$. *The resulting* MO' *now has an additional Revenue dimension where dimension values of the Revenue category attribute are calculated for each fact as product of the Quantity and Price dimension values. If we take again facts from Table 1, the new fact-dimension relation* $R_{revenue}$ *contains the entries*

$$\{(f_1, 34.950.00), (f_2, 16.765.00), (f_3, 3.890.00), (f_4, 8.794.50), (f_5, 1.718.50), (f_6, 584.55)\}$$

Forecast. The need for a *forecast* operator is directly motivated by the respective forecast planning function. Besides copying new values and entering future plan values manually on higher levels in a hierarchy, it is often useful to use a certain forecasting function fc to project trends of historical data up to a certain point into the future. For the forecast operator this means creating a set of new fact values along a specified category attribute Ca_t of a dimension (most often a time dimension). The forecast is based on existing values and a given

ordering $O[Ca_t]$ for all dimension values $e_{ti}|Type(e_{ti}) = Ca_t$. Let $O[Ca_t] = 1..m$ with $O[Ca_t](e_{ti}) < O[Ca_t](e_{tj}), i \neq j, i = 1..m, j = 1..m$ if the value of e_{ti} is smaller than the value of e_{tj} in terms of the ordering O. The forecast function $f(F, O[Ca_t], tgt, k)$ can be an arbitrary forecasting algorithm and forecasting model like exponential smoothing or ARMA models [15]. To abstract away from these details, we assume that the forecast function fc encapsulates a model, a set of parameters of the model and implements the forecasting algorithm. The parameter values might be obtained by training the model. This can either be done within the operator or externally. The trained parameters are then supplied to the operator. The forecast function takes as input a set of facts F, an ordering $O[Ca_t]$ for these facts, based on a given dimension Dim_t with $Ca_t \in Dim_t$. Furthermore, it takes a category attribute Ca_v and an integral number tgt, which specifies the forecast horizon. This is the number of new facts it should produce together with their forecast value mapping to the category attribute containing the values Ca_v. It also takes an index $k = 1..tgt$ that specifies the position of the value from the number of forecast values, which it should return. We can now write the following definition for the forecast operator, where $\Lambda_{i=1}^{n}(\ldots)$ is the conjunction of the expression in brackets for all $i = 1 \ldots n$:

Definition 6. *$\psi[f, O, Ca_t, Ca_v, tgt](MO) = MO' = (S', F', Dim', R')$ with the fact schema staying the same $S' = S$, the set of facts is extended by tgt new facts and*

1. $F' = F \cup \{f''_j | j = 1..tgt \wedge \Lambda_{i=1}^{n}(\top_i \rightarrow_i f''_j) \wedge i \neq t \wedge i \neq v \wedge e'_t \rightarrow_t f''_j \wedge e'_v \rightarrow_v f''_j\}$
2. $Dim' = Dim$
3. $R' = \{R'_i | i = 1..n \wedge i \neq t \wedge i \neq v\} \cup \{R'_t, R'_v\}$
4. $R'_i = R_i \cup \left\{(f''_j, \top_i) | j = 1..tgt \wedge f''_j \in F'\right\}$
5. $R'_t = R_t \cup \left\{(f''_j, e'_t) | j = 1..tgt \wedge f''_j \in F' \wedge O[Ca'_t](e'_t) = max(O[Ca'_t](e_k)) + j \wedge k = 1..n\right\}$
6. $R'_v = R_v \cup \left\{(f''_j, e'_v) | j = 1..tgt \wedge f''_j \in F' \wedge e'_v = f(F, O[Ca_t], tgt, j)\right\}$

In essence, the forecast operator produces tgt new facts, which are mapped to an ordered set of dimension values such that the new facts are mapped to the tgt successors of the last dimension value from the existing facts. Furthermore, for a given (measure) dimension, each new fact is mapped to its projected new value according to the prediction of the forecasting function. The complexity of the operator depends on the complexity of the forecasting function f. An example is shown in the next section, when the forecasting planning function is discussed.

4.2 Expressing Typical Planning Functions

The following section lists typical planning functions that are necessary to support planning applications working on multidimensional data as motivated in the introduction. For each of these functions we describe it in terms of operators of our novel Planning OLAP model. With our extended model, all planning functions could be expressed in terms of existing multidimensional operators and novel planning operators. This is similar to the roll-up and drill-down functions

for the existing Extended Multidimensional Data Model, which are both expressed in terms of the aggregate formation operator. We describe each function and explain how a combination of basic operators yields the expected behavior.

Delete. The delete operator deletes fact values from a multidimensional object. The planning operator delete can be expressed similar to the selection operator from the EMDM. However, there is the distinction between an MO', where some facts have only been filtered out and thus still exist in the base MO that was input to the filter operator, and actual deletion with the delete operator. In accordance with the selection operator σ, we have to supply a predicate p as parameter and a multidimensional object MO as input. The predicate p must select all facts that should be deleted from the MO. As a result, all facts matching the predicate are removed from the MO. Let $MO = (S, F, Dim, R)$ and p an arbitrary predicate, then $\eta\,[p]\,(MO) = MO' = (S', F', Dim', R')$ where $S' = S$, $F' = \{f \in F | \exists e_1 \in Dim_1, \ldots, e_n \in Dim_n (\neq p(e_1, \ldots, e_n) \wedge f \rightarrow_1 e_1 \wedge \cdots \wedge f \rightarrow_n e_n)\}$, $Dim' = Dim$, $R' = \{R'_i\}$ and $R'_i = \{(f', e) \in R_i | f' \in F'\}$.

Example 5. *To wrap this up with an example, we might decide to sell no cellphones anymore only smartphones. Consequently, we take a predicate* $p_{Cellphones} : Ca_{Family} = Cellphone$ *and apply the delete operator to our initial* MO*:* $\eta\,[p_{Cellphones}]\,(MO) = MO' = (S', F', Dim', R')$ *such that the facts of the resulting* MO' *are reduced to the set* $\{f_1, f_2, f_3, f_5, f_6\}$.

Copy. When we introduced the *value mapping* operator, we already emphasized that copying is an important part of planning to set a starting point for subsequent planning operations with data based on historic values. The copy operator can be expressed in terms of the value mapping operation combined with the duplication operator. The duplication operator is necessary, since copy has to create new facts and the only two operators that create new facts are the create combination operator or the duplicate operator. The new multidimensional object MO' is created by applying a mapping to an MO^{Copy} that contains a set of duplicate facts of the original MO and a union of the result with the original MO. Let $M = \{m_r, r = 1..s\}$ be a set of mapping functions, then copying is defined as:

$$MO' = \cup(\gamma\,[M]\,(\tau(MO)), MO)$$

The resulting multidimensional object now contains all facts and their respective mappings to the dimension values from the original MO as well as a new set of facts, where the fact-dimension mappings have been adjusted according to the mapping functions.

Example 6. *If we pick up the value mapping example, where we want to copy our sales facts from year 2011 to year 2012, then we can use the copy operator to achieve that. Let* m_1 *be a mapping function that maps for all SellDate dimension values* $e \prec 2011$ *below 2011 to* $e' \prec 2012$*, then the copy function would be:*

$$\cup(\gamma\,[M = \{m_1\}]\,(\tau(MO)), MO) = MO'$$

The new MO' now contains twice as much facts and the contents of the fact-dimension relations are shown in Table 2.

Table 2. Lists of facts in the example schema after copying from 2011 to 2012

Fact	$R_{Article}$	$R_{Location}$	$R_{SellDate}$	R_{Price}	$R_{Quantity}$
f_1	iPhone4, iOS, Apple, Smartphones	AppleStore, Munich, BrandStore	2011-05-06, 05, Q2, 2011, W18	699.00	50
f_2	Desire, Android, HTC, Smartphones	Vodafone Shop, Dresden, Provider	2011-04-23, 04, Q2, 2011, W16	479.00	35
f_3	Omnia 7, Windows Phone 7, Samsung, Smartphones	Media Markt, Hamburg, Retailer	2011-12-14, 12, Q4, 2011, W50	389.00	10
f_4	2323, Symbian, Nokia, Cellphones	Real, Frankfurt, Retailer	2011-01-11, 01, Q1, 2011, W2	79.95	110
f_5	BT Headset, Nokia, Accessories	HandyShop, Dresden, Smalldealer	2011-03-27, 03, Q1, 2011, W12	24.55	70
f_6	USB Charger, Hama, Accessories	Real, Frankfurt, Retailer	2011-08-13, 08, Q3, 2011, W32	12.99	45
f_7	iPhone4, iOS, Apple, Smartphones	AppleStore, Munich, BrandStore	2012-05-06, 05, Q2, 2012, W18	699.00	50
f_8	Desire, Android, HTC, Smartphones	Vodafone Shop, Dresden, Provider	2012-04-23, 04, Q2, 2012, W16	479.00	35
f_9	Omnia 7, Windows Phone 7, Samsung, Smartphones	Media Markt, Hamburg, Retailer	2012-12-14, 12, Q4, 2012, W50	389.00	10
f_{10}	2323, Symbian, Nokia, Cellphones	Real, Frankfurt, Retailer	2012-01-11, 01, Q1, 2012, W2	79.95	110
f_{11}	BT Headset, Nokia, Accessories	HandyShop, Dresden, Smalldealer	2012-03-27, 03, Q1, 2012, W12	24.55	70
f_{12}	USB Charger, Hama, Accessories	Real, Frankfurt, Retailer	2012-08-13, 08, Q3, 2012, W32	12.99	45

Repost. Repost is very similar to copying, but instead of keeping the source, only the targets, i.e. the newly created facts are kept. As a result it can either be described informally as a copy, with a delete of the source afterwards, or in terms of basic planning operators it is essentially the application of the value mapping operator.

Example 7. *For example, if instead of copying as in the previous paragraph, we want to repost values from* 2011 *to* 2012*, we apply the set of mapping functions M using only the $\gamma\,[M]\,(MO) = MO'$ operator. The fact-dimension relations of the resulting multidimensional object will then be represented by the last 6 rows of Table 2.*

Disaggregation and Assign. The disaggregation planning function can be directly mapped to the disaggregation operator of the model. The assign function is very similar to the disaggregation function. The function is used to assign values to certain dimension values. However, while the disaggregation changes an aggregated value and, therefore, the underlying values that contribute to the aggregate must be adapted to correctly reflect the change within an MO, the assign planning function directly changes a non summarizable dimension value without the need for a distribution. Nevertheless, for both purposes the disaggregation operator can be used. Examples for the disaggregation have already been shown in Section 4.1, thus we give two examples for the assign operator:

Example 8. *Suppose we want to assign a new name to the Omnia 7 smartphone in* 2012*. We call the disaggregation operator*

$$\alpha^{-1}\left[Omnia7, \perp_{Shop}, 2012, g^{-1}, g, e_{new} = Satia, t = 1, r = 1\right](MO) = MO'$$

The distribution function g^{-1} just outputs the new value e_{new} similar as for the constant distribution and for all other dimensions we select the most granular value as dimension value. Therefore, the list of dimension values serves as a predicate that selects the values that should get the new value assigned. Furthermore, we use the disaggregation operator here to work on dimensions that are not measure dimensions. This is possible, since the model makes no actual distinction here and because the disaggregation operator uses a generic distribution function, which must not necessarily return numeric values.

Example 9. *As another example for the assign operator, assume we want to enter a new sales quantity for the iPhone for year 2012 based on the example data that was obtained by copying from 2011 to 2012 in previous Section 4.2. Calling the disaggregation operator as follows:*
$\alpha^{-1}\left[iPhone4, \perp_{Shop}, 2012, g^{-1}, g, e_{new} = 80, t = 5, r = 5\right](MO') = MO''$ *will change line 7 in Table 2. An excerpt of the result of both assign operations is shown in Table 3.*

Table 3. Excerpt of the changed fact-dimension relations after assigning new values

Fact	$R_{Article}$	$R_{Location}$	$R_{SellDate}$	R_{Price}	$R_{Quantity}$
...	...	...	...	...	...
f_7	iPhone4,iOS, Apple,Smartphones	AppleStore,Munich, BrandStore	2012-05-06, 05, Q2, 2012, W18	699.00	80
f_9	Satia,Windows Phone 7,Samsung, Smartphones	Media Markt, Hamburg,Retailer	2012-12-14, 12, Q4, 2012, W50	389.00	10
...	...	...	...	...	...

Revalue. The revalue planning function is used to change a set of values according to a formula. In its simplest form, re-evaluating can be used to increase a revenue value by a certain percentage. To execute such calculations within our Planning OLAP model, we use the feature of calculated dimensions, and if necessary rename and projection operators to revalue existing dimension values.

Example 10. *Continuing with our example data, we want to increase prices by* 10%. *Therefore, we use the add dimension operator to add a new calculated dimension* $Dim_{NewPrice}$ *that consists of a category attribute Percentage with one dimension value equal to* $e_{Percentage} = 0.1$ *and a calculated category attribute* $NewPrice = Price \cdot Percentage$. *From the resulting* MO' *we project out the original price dimension* Dim_{Price} *and rename the calculated dimension* $Dim_{NewPrice}$ *to* Dim_{Price}. *The complete revalue expression would be:*

$$\rho[S'](\pi[Dim_{Article}, Dim_{Location}, Dim_{SellDate}, Dim_{Quantity}, Dim_{NewPrice}] \\ (+[Dim_{NewPrice}](MO)))$$

With the expressive power of calculated dimensions, the model allows arbitrary formulas for the revaluation function.

Forecasting. Similar to the disaggregation planning function, the *forecasting* planning functions has a direct representation as an operator in our Planning OLAP model and can be expressed with this operator. The forecasting technique or model that should be used can be a parameter of the planning function and is determined by the forecast function fc that calculates new forecast values.

Example 11. *Using our running example scheme again, we present the application of a simple forecasting method that takes the average of previous values as new value to illustrate the usage of the forecast operator. To apply the operator, we first need a small time series that serves as historic data. From our examples, we take the resulting MO that we obtained by copying the data from 2011 to 2012 and assigned new values. The sales quantity of iPhone4 will serve as a mini time-series and we predict the next sales quantity for 2012. Therefore, we apply a selection operator selecting only facts that are characterized by Article iPhone4. For example purposes, we use the most simple forecasting function* fc_{MA}, *simply carrying forward a moving average, which calculates the* $i + 1th$ *element as* $e_{i+1} = \frac{e_i + e_{i-1} + \ldots + e_{i-k+1}}{k}$ *with* k *being* 2, *because our time-series only has two historic data points. The ordering* O_{year} *provides an ordering on the Year category attribute of the SellDate dimension and the quantity value should be predicted by the forecast for the next period (*$tgt = 1$*):*

$$\psi[fc_{MA}, O_{Year}, Ca_{Year}, Ca_{Quantity}, tgt = 1](MO) = MO'$$

As a result a new fact is created with the fact-dimension relations shown in Table 4:

Table 4. A new fact and its fact-dimension relations after forecasting

Fact	$R_{Article}$	$R_{Location}$	$R_{SellDate}$	R_{Price}	$R_{Quantity}$
f_1	iPhone4,iOS, Apple,Smartphones	AppleStore,Munich, BrandStore	2011-05-06, 05, Q2, 2011, W18	699.00	50
f_7	iPhone4,iOS, Apple,Smartphones	AppleStore,Munich, BrandStore	2012-05-06, 05, Q2, 2012, W18	699.00	80
f_{13}	iPhone4,iOS, Apple,Smartphones	AppleStore,Munich, BrandStore	2012		65

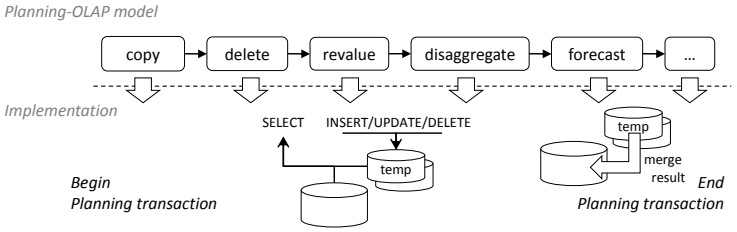


Fig. 4. Implementation scheme for the Planning-OLAP model

5 Impact and Conclusion

When comparing the Planning-OLAP model with traditional OLAP models then the distinction is, that the latter is based purely on read operations whereas the planning operators write or generate data. As planning has a simulation character, it is often the case that generated data is continuously adjusted until a final result is obtained. As such it can be viewed as a long running transaction containing a mixture of read (*roll-up, drill-down, slice and dice*) and write (*disaggregate, copy, revalue, forecast and delete*) operations. While the transaction bracket is not necessary for the read-only traditional OLAP, it makes sense for the Planning-OLAP model where only the final result of a planning transaction should become visible and persistent. The scheme in Figure 4 outlines how this usage paradigm of the Planning-OLAP model can be mapped to a relational system using SQL. At the beginning of a planning transaction one or more temporary tables are created that will contain the modifications of the current transaction and final results are merged into the original tables at the end of the transaction. Each operator is mapped to a combination of SELECT and INSERT/UPDATE/DELETE statements that modify the temporary tables. For example the disaggregation operator has to update every measure value that contributes to the overall sum. However, as the disaggregation operator contains possibly complex distribution logic, the orchestration of these statements to yield the correct result must either be done by the application or may be encapsulated in a stored procedure. Clearly, a direct integration of this functionality into the database system as a native operator would facilitate standardized and

application independent behaviour and allow for optimizations. This is similar for other operations such as value-mapping and forecasting. Therefore, we argue that in the future, these operations should be first-class citizens in a database system, equal to many navigational OLAP operators that are already supported natively by major database systems today.

Although there exists a wealth of OLAP models in the literature, none of the existing models incorporated planning functionality. Many requirements have been formulated for OLAP modeling to support real-world scenarios. While many requirements are already met by some of the models, none of it explicitly supported planning, which is a vital part of OLAP today. We proposed a novel Planning-OLAP model that uses the Extended Multidimensional Data Model (EMDM) as a foundation. By introducing a set of novel planning operators our model is capable of supporting an extensive list of standard planning functions, which we illustrated by examples. Since the Planning-OLAP model contains operators that change data according to complex semantics, the challenge on the implementation level is to have planning operators as first-class citizens within a database system.

References

1. Codd, E., Codd, S., Salley, C.: Providing OLAP (On-Line Analytical Processing) to User-Analysis: An IT Mandate (1993)
2. Jaecksch, B., Lehner, W.: The Planning OLAP Model - A Multidimensional Model with Planning Support. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 14–25. Springer, Heidelberg (2011)
3. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In: ICDE, pp. 152–159 (1996)
4. Li, C., Wang, X.S.: A Data Model for Supporting On-Line Analytical Processing. In: Proceedings of the Fifth International Conference on Information and Knowledge Management, CIKM 1996, vol. 199, pp. 81–88 (1996)
5. Gyssens, M., Lakshmanan, L.V.S.: A Foundation for Multi-Dimensional Databases. In: Proceedings of the International Conference on Very Large Data Bases, pp. 106–115. Citeseer (1997)
6. Ozsoyoglu, G., Ozsoyoglu, Z., Mata, F.: A Language and a Physical Organization Technique for Summary Tables. In: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, pp. 3–16. ACM (1985)
7. Rafanelli, M., Shoshani, A.: STORM: A Statistical Object Representation Model. Statistical and Scientific Database Management, 14–29 (1990)
8. Agrawal, R., Gupta, A., Sarawagi, S.: Modeling Multidimensional Databases. In: Proc. of 13th. Int. Conf. on Data Engineering (ICDE), pp. 232–243 (1997)
9. Cabibbo, L., Torlone, R.: Querying Multidimensional Databases. In: Cluet, S., Hull, R. (eds.) DBPL 1997. LNCS, vol. 1369, pp. 319–335. Springer, Heidelberg (1998)
10. Vassiliadis, P.: Modeling Multidimensional Databases, Cubes and Cube Operations. In: Proceedings of the Tenth International Conference on Scientific and Statistical Database Management (Cat. No.98TB100243), pp. 53–62 (1998)
11. Lehner, W.: Modeling Large Scale OLAP Scenarios. In: Advances in Database Technology EDBT 1998, p. 153 (1998)

12. Datta, A.: The Cube Data Model: a Conceptual Model and Algebra for On-line Analytical Processing in Data Warehouses. Decision Support Systems 27(3), 289–301 (1999)
13. Vassiliadis, P., Sellis, T.: A Survey on Logical Models for OLAP Databases. SIGMOD Record 28, 64–69 (1999)
14. Pedersen, T., Jensen, C., Dyreson, C.: A Foundation for Capturing and Querying Complex Multidimensional Data. Information Systems 26(5), 383–423 (2001)
15. Box, G., Jenkins, G.: Time Series Analysis: Forecasting and Control (1970)

Query Optimization for the NOX OLAP Algebra

Ahmad Taleb[1], Todd Eavis[2], and Hiba Tabbara[3]

[1] Najran University, Najran, Saudia Arabia
ahmadtaleb@hotmail.com
[2] Concordia University, Montreal, Canada
eavis@cs.concordia.ca
[3] Concordia University, Montreal, Canada
h_tabarra@encs.concordia.ca

Abstract. Current OLAP servers are typically implemented as either extensions to conventional relational databases or as non-relational array-based storage engines. In the former case, the unique modeling and processing requirements of OLAP systems often make for a relatively awkward fit with RDBM systems. In the latter case, the proprietary nature of the MOLAP implementations has largely prevented the emergence of a standardized query model. In this paper, we discuss an algebra for the specification, optimization, and execution of OLAP-specific queries, including its ability to support a native language query framework. In addition, we ground the conceptual work by incorporating the query optimization and execution facilities into a fully functional OLAP-aware DBMS prototype. Experimental results clearly demonstrate the potential of the new algebra-driven system relative to both the un-optimized prototype and a pair of popular enterprise servers.

1 Introduction

Data warehousing and Online Analytical Processing (OLAP) are two of the most important components of contemporary Decision Support Systems (DSS). Collectively, they allow organizations to make effective decisions regarding both their current and future state. In practice, warehouse databases are implemented via array-based multi-dimensional storage engines (MOLAP) or as extensions to the more familiar relational database management systems (ROLAP). While the MOLAP tools offer impressive performance, their limited scalability often restricts their use to environments with more modest resource requirements (e.g., departmental data marts). Conversely, enterprise ROLAP systems tend to scale quite well, but offer design and implementation models that are constrained by conceptual and architectural elements intended primarily for transaction processing systems.

Moreover, current warehouse/OLAP systems utilize query mechanisms that were designed decades ago. Specifically, they rely upon a combination of string based query languages such as SQL and MDX, along with various proprietary extensions. These languages (and their APIs) have little in common with the safe, flexible Object Oriented languages commonly used in today's development

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 53–88, 2013.

environments. Ultimately, they make client side programming less effective due to the following fundamental constraints:

- The absence of compile-time checking (both data types and query logic) not only complicates code development but increases the likelihood of undiscovered run-time errors and exceptions.
- Code re-factoring — in response to schema changes, for example — is extremely limited and typically consists of error prone "search and replace" operations.
- Application development requires that programmers work concurrently in both imperative (e.g., Java) and declarative (e.g., SQL) programming styles, a task that is problematic for many less experienced developers.
- There are few, if any, possibilities for the code re-use afforded by Objected Oriented Processing (OOP) concepts such as query inheritance and polymorphism. (Note that while Object Relational Mapping, frameworks such as Hibernate do provide some additional functionality in this regard, it tends to be quite limited for the complex, multi-table aggregation queries seen in OLAP domains).

Finally, we note that the use of traditional query languages makes it very difficult for the DBMS server to effectively exploit OLAP-specific constructs at query resolution time. In other words, the requirement to work with existing query languages and APIs largely prevents the backend server from effectively optimizing user queries to take full advantage of either OLAP conceptual structures (e.g., concept hierarchies and aggregation paths) or physical layer extensions (e.g., enhanced indexing or sorting opportunities).

For this reason, we believe that new OLAP query interfaces are required. In an earlier work [19], we introduced and provided the motivation for a framework called NOX — Native language OLAP query eXecution — that would allow data cube queries to be written in native OOP languages such as Java. Specifically, the paper presented the core components of a software model for the transparent translation of DBMS queries. Primary elements included source code pre-parsing tools, client-side libraries that formed the programmatic API, and the multi-dimensional ResultSet Object. While the use of a domain specific grammar was discussed, the paper provided no formal representation of the algebra, nor did it discuss algebraic optimization strategies. In addition, the paper did not address the integration of the NOX model with the underlying DBMS platform and provided no performance results for such a platform. A follow up paper, however, did address these issues [29]. There, we extended the initial research with a more thorough discussion of an algebra used to support the language libraries visible to the client side programmer. Moreover, we discussed the integration of the algebra with a robust DBMS backend known as Sidera that not only natively supports the algebraic operators but is able to optimize query plans by applying a series of transformations to the initial parse trees. The fully optimized plan can then be passed to an execution engine that, in turn, exploits indexes and algorithms designed expressly for this purpose.

Given the complexity and breadth of the integrated NOX model, however, a number of the associated components could only be treated in a cursory manner. In the current paper, we extend the previous work with a far more extensive discussion of the algebra and its optimization strategies. To this end, we present a series of ten Transformation Rules that are used by the query engine to manipulate query expression trees so that execution cost is likely to be significantly reduced. Extensive examples, illustrations, and notational formalisms are provided to support the various optimization strategies. We note that while a number of the transformations are intended to be utilized by the highly tuned Sidera server (e.g., by explicitly exploiting its indexes), other optimization rules can potentially be utilized by servers exposing an OLAP or cube-aware access language, even if the other NOX advantages described above are not available. MDX-based servers would be candidates in this regard.

The paper is organized as follows. Section 2 briefly reviews related work. An overview of the Sidera data model and architecture is provided in Section 3, including its application to native language querying. In Section 4, we discuss the formal properties of the current algebra, including a series of illustrative examples. Section 5 is the core contribution of the paper and presents an extensive overview of optimization strategies for the OLAP algebra. Key experimental results are then presented in Section 6. Section 7 concludes the paper with a few final observations.

2 Related Work

Over the past decade or so, numerous attempts have been made to simplify, extend, or otherwise improve DBMS query interfaces, languages and data models. One common theme has been the adaptation of APIs to include Object Oriented semantics and syntax. Object Relational Mapping (ORM) frameworks — including Java Data Objects (JDO) [1] and Hibernate [7] — have been used to define *transparent object persistence* for DBMS-backed OOP applications. Still, the query language extensions — including JDOQL (JDO) and HQL (Hibernate) — required to execute joins, complex selections, and sub-queries, produce a development environment that often seems as complex as the model it was meant to replace. More recently, Safe Query Objects (SQO) [13] have been introduced. Rather than explicit mappings, safe queries are defined by a class containing, in its simplest form, a *filter* and *execute* method. The compiler checks the validity of query types, relative to the objects defined in the filter. The *execute* method is then rewritten as a JDO call to the remote database.

Other approaches target the language itself. For example, one can point to language extensions such as those found in Ruby's Active Records [5], HaskellDB [2], and Microsoft's Language Integrated Query (LINQ) extensions for its C# and VisualBasic environments [9]. Here, however, one must note that none of these languages are in any way OLAP-aware and, thus, have no native support for concepts such as cubes, dimensions, aggregation hierarchies, granularity levels, and drill down relationships. By contrast, Microsoft's popular Multidimensional

eXpressions (MDX) query language [30] — while syntactically reminiscent of SQL — provides direct support for both multi-level dimension hierarchies and a crosstab data model. Still, MDX remains an embedded string based language and, as such, cannot provide comprehensive compile-time type checking, a single unified application/DBMS development language, OOP functionality (e.g., inheritance and polymorphism), or efficient source code re-factoring.

In terms of OLAP and Business Intelligence (BI) specific design themes, most contemporary research builds in some way upon the OLAP *data cube* operator [21]. In addition to various algorithms for cube construction, including those with direct support for dimension hierarchies [26], researchers have identified a number of new OLAP *operators* [14], each designed to minimize in some way the relative difficulty of implementing core operations in "raw SQL". In some cases, direct extensions to existing DBMS servers have been proposed, including techniques such as UDF functions for in-memory cube lattices [11] and in-memory matrices for analytics computation [31]. Few, if any, of these proposals, however, were concerned with the underlying algebra or its optimization.

Performance optimization has been another fairly popular target. At various times, researchers have focused on view materialization [23,24], improved indexing [12,18], and parallelization and partitioning [25,22]. In general, all such approaches build on techniques that were developed for OLTP databases. For environments in which exact answers are not necessarily required, approximate query answering has also been utilized as a means to dramatically improve query response time by significantly compressing the data set. Approaches that build upon multi-dimensional histograms have been a common theme [10], with more recent work attempting to minimize the effect of data skew during the partitioning process [16]. Sampling has also been employed when quick approximate results are required [6], including OLAP-specific methods that pre-process input data to minimize the impact of outliers on the sample data [17]. An interesting approach to approximate query answering has also been proposed in the mobile setting [15]. Here, flattened subsets of cube data are compressed using a quad tree technique and passed to mobile devices for further offline query processing. While the general goal of enhanced query performance is similar to that described in the current paper, we note however that the target is quite different in that our proposed methods seek primarily to reduce server workload rather than to minimize the effect of limited bandwidth and connectivity instability. In any case, it is important to point out that such techniques can be seen as complimentary, rather than competitive, as approximate query processing/compression can be used in conjunction with the methods discussed in this work.

Finally, there has also been some interest in the design and exploitation of supporting algebras [27]. The primary focus of this work has been to define an API that would ultimately lead to transparent, intuitive support for the underlying data cube, and in a more general sense, to the identification of the core elements of the OLAP conceptual data model. OLAP-specific optimization based upon query re-writing has also been proposed. For example, using an OLAP algebra that highlights the visual representation of the data cube, Bellatreche et al. propose a set

of rules to re-structure OLAP queries executed against fact and dimension tables (i.e., Star Schema) stored in a standard relational DBMS [8]. Though improved performance is suggested, there is no concrete DBMS implementation (or physical operators) by which to fully quantify or evaluate the proposal.

3 Preliminary Material

We emphasize at the outset that the Sidera/NOX focus is clearly on the OLAP/BI domain. In fact, the DBMS is intended to specifically support analytics services and is not expected to be utilized as a general purpose database management system that might, for example, be employed for transaction processing. The primary motivation for this approach is the rejection of the "be all things to all people" mantra that tends to plague systems that must maintain a fully generic, lowest common denominator profile [28]. In the current context, the targeting of a specific application domain ultimately relieves the designer from having to manually construct a comprehensive data model, along with its constituent processing constructs.

We begin with a review of the conceptual and physical model upon which the Sidera DBMS is constructed. We remind the reader that the methods discussed in this paper are part of the larger NOX framework [19]. To summarize, NOX provides the following seven components:

- **OLAP conceptual model**. NOX allows developers to write code directly at the conceptual level. No knowledge of the physical or even logical level is required.
- **OLAP algebra**. Given the complexity of directly utilizing the relational algebra in the OLAP context, we define fundamental query operations against a cube-specific OLAP algebra.
- **OLAP grammar**. Closely associated with the algebra is a DTD-encoded OLAP grammar that provides a concrete foundation for client language queries.
- **Client side libraries**. NOX provides a small suite of OOP classes corresponding to the objects of the conceptual model.
- **Programming API**. Collectively, the exposed methods of the libraries form a clean programming API that can be used to instantiate OLAP queries.
- **Augmented compiler**. At its heart, NOX is a query re-writer. During a pre-processing phase, the framework's compilation tools (JavaCC/JJTree) effectively re-write source code to provide transparent model-to-DBMS query translation.
- **Cube result set**. OLAP queries essentially extract a subcube from the original space. The NOX framework exposes the result in a logical read-only multi-dimensional array.

Figure 1 provides a concise illustration of the NOX processing stack. In short, the developer's view of the OLAP environment consists of the elements of the top three levels. More to the point, from the developer's perspective, all OLAP

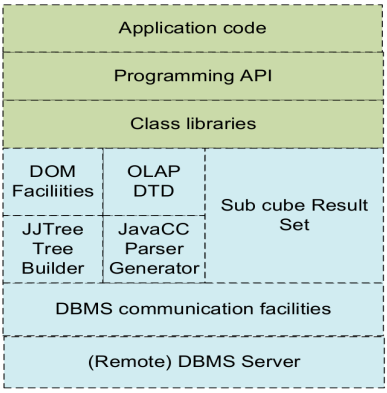


Fig. 1. NOX processing stack

data is housed in a series of cube objects housed in local memory. The fact that these repositories are not only remote, but likely much larger than local memory, is largely irrelevant.

3.1 Conceptual Model

As noted in the previous section, NOX allows one to program directly against a conceptual data model. Briefly, we consider analytical environments to consist of one or more *data cubes*. Each cube is composed of a series of d dimensions (sometimes called *feature* attributes) and one or more *measures*. The dimensions can be visualized as delimiting a d-dimensional hyper-cube, with each axis identifying the *members* of the parent dimension (e.g., the days of the year). Cell values, in turn, represent the aggregated measure (e.g., sum) of the associated members. Figure 2(a) provides an illustration of a very simple three dimensional cube. We can see, for example, that 12 units of Product AM54 were sold in the Berkeley location during the month of January (assuming a Count measure).

Beyond the basic cube, however, the conceptual OLAP model relies extensively on aggregation hierarchies provided by the dimensions themselves. In fact, hierarchy traversal is one of the more common and important elements of analytical queries. In practice, there are many variations on the form of OLAP hierarchies (e.g., symmetric, ragged, non-strict). NOX supports virtually all of these, and does so by augmenting the conceptual model with the notion of an arbitrary graph-based hierarchy that may be used to *decorate* one or more cube dimensions. Figure 2(b) illustrates a simple geographic hierarchy that an organization might use to identify intuitive customer groupings.

3.2 Native Language Queries

NOX provides a set of client libraries that map directly to the conceptual model described above. In addition to base classes representing OLAP objects such

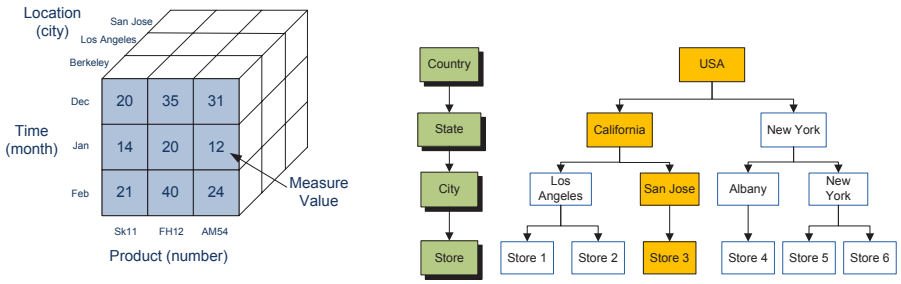


Fig. 2. (a) NOX conceptual query model (b) A simple symmetric hierarchy

as dimensions, hierarchies, cells, and aggregation paths, the framework include a core OLAPQuery class that exposes methods corresponding to the algebra described in Section 4. The programmer therefore defines queries not by embedding a non-OOP text string, but by over-riding and extending the OLAPQuery base class and adding just those constraints relevant to the current query. In so doing, the NOX environment is able to provide compile time type checking, semantic verification (as per the client libraries), refactoring facilities, and OOP functionality (e.g., query inheritance). Figure 3 illustrates an MDX query and the corresponding NOX query (written in Java). Note that as queries become larger and more complex, NOX queries tend to maintain their readability much better than the corresponding MDX queries.

Though the translation and submission of NOX queries is a somewhat complex process [19], the reader should note the following. The client side query depicted in Figure 6 is not executed directly. Instead, the NOX processor parses the source code, identifies the NOX class constructs, and transparently re-writes

```
SELECT
 { [Product].[Type].ALLMEMBERS } ON COLUMNS,
 { [Customer].[Province].ALLMEMBERS  }  ON ROWS

FROM  [Order]

WHERE (
  [Measures].[Quantity_Ordered],
  [Time].[Year].[2007],
  [Time].[Month].[May],
  [Time].[Month].[June],
  [Customer].[Age].[45],[Customer].[Age].[55]
)
```

(a)

```
class SimpleQuery extends OlapQuery {
  public boolean select() {
     DateDimension date = new DateDimension();
     Customer customer = new Customer() ;
     OlapProperty dateMonth = new OlapProperty(date.getMonth());
     return (customer.getAge() > 40 && date.getYear() == 2007 &&
          dateMonth.inRange(5, 10));
  }
  public Object[] project() {
     Customer customer = new Customer() ;
     Product product = new Product() ;
     Measure measure = new Measure() ;
     Object[] projections = {product.getType(),
          customer.getProvince(),
          measure.getQuantity_Ordered()};
     return projections;
  }
}
```

(b)

Fig. 3. (a) A simple MDX query (b) The NOX equivalent

the programmer's source code. In place of the original OLAPQuery definition, the processor inserts a network call to the Sidera DBMS. Within the network packet is a query representation that has already been reduced to its algebraic components. It is this form of the query that is actually optimized and executed by the server at runtime.

3.3 The Sidera Architecture

Sidera is a research DBMS that targets analytics environments. To this end, it provides native, OLAP-specific support for indexing (bitmaps and R-trees), fault tolerance (network heartbeat), caching (spatial query representation), lightweight graphical interfaces (via the Google Web toolkit) and, of course, query languages (NOX). It is also designed from the ground up as a parallel DBMS that is intended to scale to ROLAP sizes, while giving something close to MOLAP performance. Essentially, Sidera is constructed as a federation of sibling servers that function more or less independently, each accessing and processing a slice of the current query. A Parallel Service Interface (PSI) offers global coordination and merging services as required. Figure 4 illustrates the the structure of the parallel system, including the cooperative siblings.

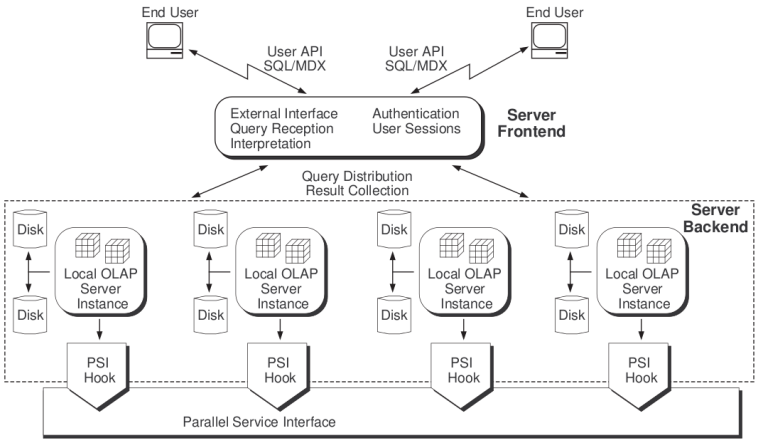


Fig. 4. The core architecture of the parallel OLAP server

In this section, we discuss those elements of the architecture that support the execution of translated NOX queries. Specifically, we will look at the storage and indexing model with which the query costing and optimization is associated. We begin with the physical representation of the NOX conceptual model described above. Traditionally, relational warehouses use a Star Schema, consisting of a *Fact* table and one or more *Dimension* tables. Process metrics are housed in the Fact table, with dimension tables containing feature information typically used

to constrain user queries. A Sidera database is roughly analogous to this design. However, rather than a Fact table, Sidera employs a materialized cube (fully or partially, as space permits) that is constructed as a set of Hilbert packed R-trees, then minimized using a form of tuple differential compression [18]. In short, data points are not stored in standard row format. Rather, multi-dimensional points are ordered as per their occurrence along the length of a Hilbert space filling curve and subsequently represented as the distance between consecutive positions along the curve (i.e., a single compressed integer). We then incorporate the (open source) Berkeley DB embedded libraries [3] into the Sidera code base so as to efficiently encode the Fact Structure. Specifically, we have extended the default Berkeley API to include R-tree generation and storage functionality. Note that we refer to measure data as a Fact Structure, rather than a Fact table, as the storage format (i.e, integer differentials) bears little resemblance to the traditional table that one would expect to find in a row-based relational database.

Figure 5 illustrates the internal structure of the Sidera/Berkeley cube. Note that the letters A-B-C are simply used as a shorthand for Dimension names such as Product, Date, etc. We see that the cube consists of a packed sequence of meta data, measure data, and index blocks for each aggregated view or cuboid, as well as a master B-tree that locates the relevant view data, as per the current query specification. We note as well that Sidera is specifically designed for the workloads and query patterns one would expect to see in typical OLAP environments; in other words, common OLAP domains consist of one of more cubes (Star Schemas), each made up of perhaps 15 dimensions or less. For environments in which very high dimension counts are actually required for individual cubes (as opposed to being the result of poor schema design), column oriented storage is worthy of investigation.

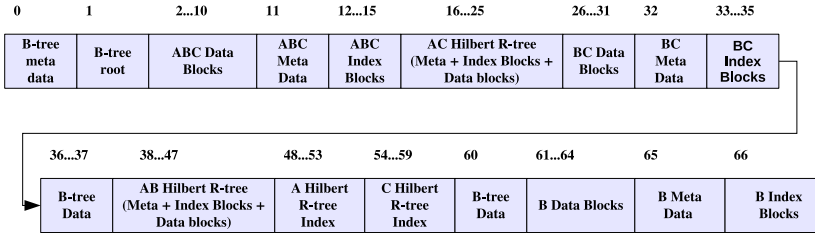


Fig. 5. The physical structure of the indexed cube

Dimension data is stored independently of the central cube structure, as it requires distinct forms of indexing and representation. Specifically, Sidera is aware of both *hierarchical* and *non-hierarchical* elements. By hierarchical, we mean those values associated with user-defined aggregation pathways (e.g., the common Day-Month-Year Time hierarchy). Sidera uses a structure known as

mapGraph to efficiently translate cell values between arbitrary hierarchy levels at run-time [20]. Figure 6(a) shows mapGraph's representation of the meta data associated with a simple symmetric Product-Type-Category hierarchy (Sidera can also support support more complex hierarchies). Note that the integer values in the figure corresponds to ranges of Product ID values (i.e., Product keys) that are encapsulated within the compressed tuple differential values encoded in the Fact Structure. Sidera always stores cell values at the lowest level of granularity so as to permit arbitrary bi-directional translation between hierarchy levels. In effect, the DBMS uses the in-memory mapGraph structure as a join index between the Fact Structure and the hierarchy values.

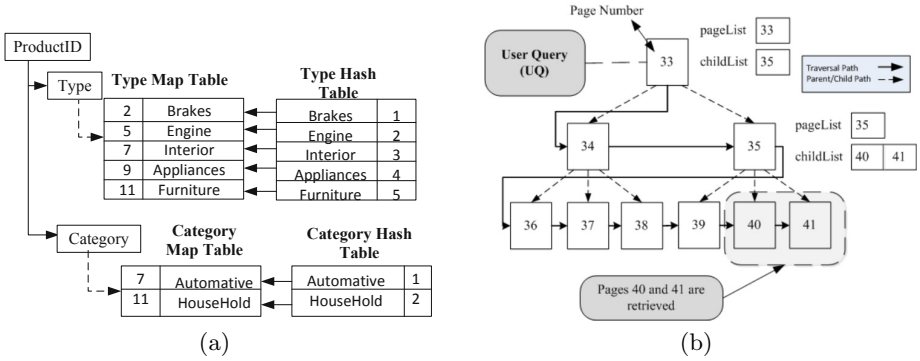


Fig. 6. (a) A simple mapGraph translation map (b) Linear Breadth First R-tree search

Non hierarchical attributes such as age, on the other hand, may be used to constrain user queries but are not associated with identifiable aggregation paths. In this case, dimension attributes are encoded with FastBit [4], an efficient compressed bitmap indexing mechanism. Sidera's Fastbit attribute processing essentially produces contiguous sequences of key values that can be mapped against the Fact Structure. Because the encoded Berkeley R-trees are internally packed level-by-level and processed with a *breadth first* search strategy (rather than the conventional depth first approach), Fastbit key sequence matching can be accomplished with a single pass through the cube index. Figure 6(b) illustrates how the R-tree search algorithm sweeps across levels of the index identifying sequences of pages that correspond to the consecutive key values produced by the bitmap indexes.

Figure 7 provides an intuitive representation of the core components of the storage engine, albeit for a very simple cube example with just two dimensions. One can see how the dimension data is accompanied by a (memory-resident) hMap structure to support hierarchical attributes, multiple bitmaps for non-hierarchical attributes, and system generated surrogate keys (i.e., simple integers) shared with the Fact Structure. During query resolution, these keys pass

in/through the query engine to the storage backend. At that point, records can be matched against Hilbert compressed data, using the Berkeley Master B-tree to locate the required view/block combinations.

Finally, we re-iterate that Sidera is constructed as a parallel DBMS and runs on commodity Linux clusters (multi-core and GPU extensions are currently being investigated). Figure 8 illustrates the processing model of the individual sibling servers, showing the relationship between the the native components as well as the Berkeley extensions. Note that the View Manager is responsible for the identification of the most cost effective group-by inside the Berkeley Fact Structure — and is initialized by scanning the primary Master B-tree database that contains references to all indexed group-bys — while the Hierarchy Manager builds and maintains the in-memory mapGraph structures. Note as well that OLAP Caching has nothing to do with the Berkeley caching component that stores recently accessed disk blocks, but refers instead to a native, multi-dimensional OLAP query cache.

4 The Sidera Algebra

While the language of OLAP algebras has yet to be standardized, it is nevertheless the case that a core set of operations has been consistently identified in the literature [27]. However, before defining the operators of the Sidera algebra, we will first provide a more formal representation of the conceptual model presented in Section 3.1.

An N-dimensional **cube** C is constructed as $<D, F, M, BasicCube>$ where:

- D is a set of dimension D_i of C, where $D = \{D_1, D_2, ..., D_N\}$, and$1 \leq i \leq N$.
- F is a set of feature attributes F_i of C, $F = \{F_1, F_2, \ldots, F_N\}$, where $1 \leq i \leq N$.

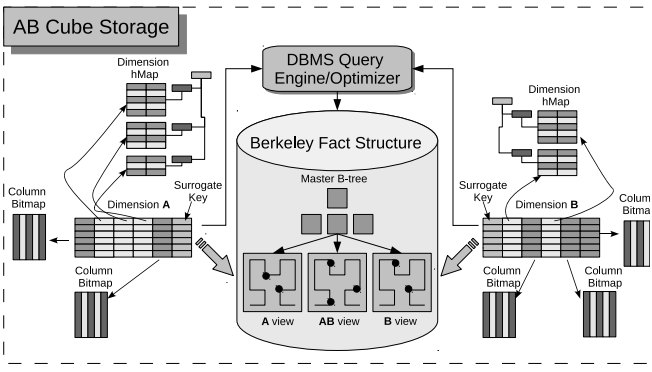


Fig. 7. The architecture of the individual nodes of the cluster DBMS

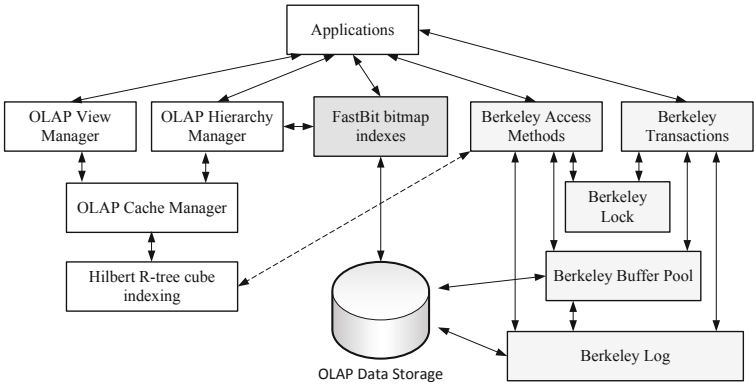


Fig. 8. The architecture of the individual nodes of the cluster DBMS

- M is a list of measure attributes M_j of C, $M = \{M_1, M_2, \ldots, M_k\}$, where $j \leq k$.
- *BasicCube* is a set of cells that describes the *facts* (measure attributes) at the particular level of detail specified by F.

A **dimension** (D_i) is defined by a schema written as *schema*(D_i) =<*ColumnList, Key, Hierarchy*> where:

- *ColumnList* is a set of dimension attributes $D_i.A_j$ of D_i,*ColumnList* = $\{D_1.A_1, \ldots, D_1.A_n\}$, where n is the number of attributes in dimension D_i.
- *Key* is an attribute $D_i.A_k$ of *ColumnList*, where $D_i.A_k$ is the deepest level of detail for dimension D_i, where $1 \leq k \leq n$.
- *Hierarchy* is a set of hierarchies $D_i.H_j$ of D_i, with *Hierarchy* =$\{D_i.H_1, D_i.H_2, \ldots, D_i.H_z\}$, where $j \leq z$ and z is the number of hierarchies associated with dimension D_i. Each hierarchy $D_i.H_j$ is of the form $D_i.H_j = \{H_j, D_i.A_r \rightarrow \ldots \rightarrow D_i.A_l\}$, where $D_i.A_r$ is the root hierarchal attribute level, while $D_i.A_l$ is the leaf level in hierarchy H_j of dimension D_i.

A **Feature Attributes** F_i refers to a specific attribute A_j in dimension D_k, where $i,k \subseteq [1,N]$. It is of the form $F_i = \{D_k.A_j\}$, where F_i is an attribute in the *ColumnList* of dimension D_k.

A **BasicCube** is a multidimensional end user representation with a schema of the form schema*(BasicCube)* $= \{F, M\}$. An instance of a *BasicCube* is the set of *cells/facts/records/tuples* that are described by the values of measure attributes M at the level defined by F. Through the remainder of this paper, we will use the terms $cells, facts, records$, and $tuples$ interchangeably.

SELECTION. The SELECTION operator identifies one or more cells from within the full d-dimensional search space and its application produces what

is commonly referred to as "slicing and dicing". When applied to a data cube it produces a subset of the same cube. More formally, we can define the SELECTION operator on cube **C** as

$$\sigma_{(D_i.A_j\ OP\ \phi)}C$$

where $D_i.A_j$ is an attribute in dimension D_i, **OP** is a conditional operator such as $\{<, >, =, \ldots,$ etc.$\}$, and ϕ is one or more values in domain($D_i.A_j$).

The result of $\sigma_{(D_i.A_j\ OP\ \phi)}C$ is a cube *C1<D, F, M, BasicCube1>*, where sets *D, F,* and *M* are equivalent to those in the input cube *C* and schema(*BasicCube1*) = schema(*BasicCube*). From the user's perspective, the query is executed against the physical data cube such that the SELECTION criteria will be iteratively evaluated against each and every cell. If the SELECTION test evaluates to true, the cell is included in the result; if not, then it is ignored.

Let us assume that the three dimensional cube SOLD is defined as per Figure 9(a). Here, SOLD is composed of three dimensions (Product, Time and Location) and one measure attribute (UnitsSold)). We can see, for example, that 77 units of Product FH12 were sold in the Ottawa location during the month of January (assuming a Count measure). The full result of the OLAP expression $\sigma_{(Time.Month=Jan)}SOLD$ is depicted in Figure 9(b).

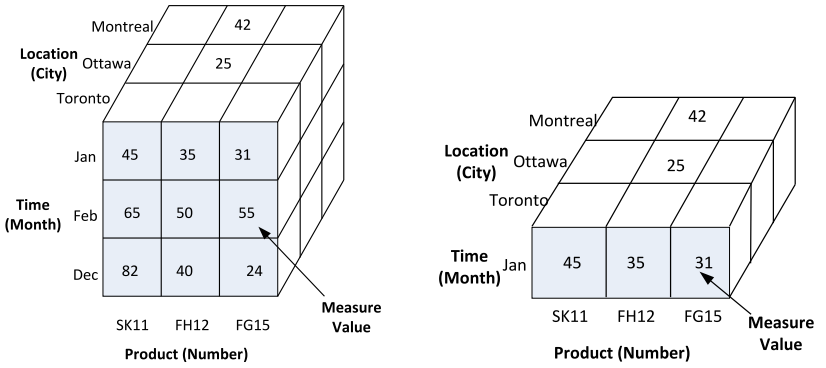


Fig. 9. (a) The three dimensional cube *SOLD*. (b) The result of $\sigma_{(Time.Month=Jan)}SOLD$.

PROJECTION. Used for the identification of presentation attributes — including both measure attribute(s) and dimension members — a projection extracts from the original cube a new cube composed of only those elements specified with the PROJECTION operator. Formally, PROJECTION can be written as:

$$\pi_{(D_i.A_j,\ y)}C,$$

where $D_i.A_j$ is a list of dimension attributes, and y $\subset M$. The resulting cube is *C1<D1, F1, M1, BasicCube1>*, where $D1$ is a set of dimensions, $F1$ = list of dimension attributes $D_i.A_j$, $M1 = y$, and Schema(BasicCube1) = F1, M1. Note that the measure value(s) $M1$ of BasicCube1 are aggregated at the level of the attribute(s) in $F1$.

Again consider the three dimensional cube *SOLD* of Figure 9(a). We can project this cube onto a new cube with two dimension attributes (Product.Number and Time.Month) and one measure UnitsSold with the following OLAP algebra expression:

$$\pi_{(Time.Month,\ Product.Number,\ UnitsSold)}\ SOLD.$$

The resulting cube is depicted in Figure 10(a). Note how the measure values *(UnitsSold)* of product FH12 are re-calculated accordingly. For example, the total sales for Jan/FH12 = 35 + 25 + 42 = 102.

CHANGE LEVEL. This operator allows the user to navigate amongst levels of a concept hierarchy, each with a distinct aggregation granularity. We typically refer to these processes as "roll-up" and "drill down." Formally, we denote the Change Level operator as:

$$\Updownarrow_{(D_i.A_j \rightarrow D_i.A_k)}\ C,$$

such that $D_i \in D$, $D_i.A_j$ is a feature attribute of cube C and $D_i.A_k$ is a hierarchical attribute level in dimension D_i. The resulting cube is of the form *C1 = <D, F1, M, BasicCube1>*. Note that while the result cube C1 maintains the same dimensions and measure attribute(s), it will have a new feature set($F1 = F - D_i.A_j + D_i.A_k$). The Change Level operator may also

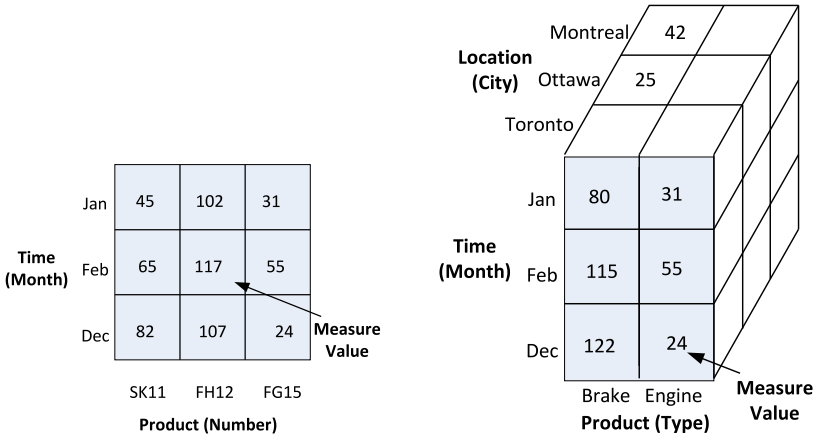


Fig. 10. (a) The result of Projection operator (b) Result of a Change Level operation

produce multiple level changes as follows: $\Updownarrow_{(D_i.A_j \rightarrow D_i.A_k, D_r.A_s \rightarrow D_r.A_t, \ldots)} C$, where $i, r = [1 \ldots N]$.

Consider the three dimensional cube *SOLD* of Figure 9(a). Figure 10(b) illustrates how the "Product" dimension, originally listed at a more detailed level number, is aggregated in order to provide a break down by Brake and Engine product types. Figure 10(b) is the result of the following OLAP expression: $\Updownarrow_{(Product.Number \rightarrow Product.Type)} SOLD$. In this expression, Product.Number is a feature attribute in the source cube SOLD and Product.Type is a hierarchical attribute level in dimension Product. Note how attribute Product.type in Figure 10(b) becomes a feature attribute instead of the attribute Product.Number.

CHANGE BASE. This operator represents the addition or deletion of one or more dimensions from the current result cube **C**. Aggregated cell values must be re-calculated accordingly. Change Base may be represented as:

$$\pm_{(D_i.A_j \rightarrow Action)C},$$

where *Action* $\in$ (*Remove*, *Add*). The resulting cube C1 = ¡$D1, F1, M, BasicCube1$¿ has different dimensions, feature attributes and BasicCube relative to that of the source cube **C**.

Again, we consider the *SOLD* cube of Figure 9(a). The result of the following expression $\pm_{(Time.Month \rightarrow Remove, Location.City \rightarrow Remove)} SOLD$ is depicted in Figure 11.

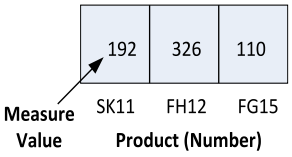


Fig. 11. Result of the Change Base operation

PIVOT. This is a presentation-specific operation that allows users to re-organize the axes of the cube. No recalculation of cell values is required. Formally, we have:

$$\circlearrowleft_{(D_i.A_j \rightarrow D_k.A_l)} C$$

where $D_i.A_j$ and $D_k.A_l$ are feature attributes in cube C. This operator reorganizes the axes of cube C so that $D_k.A_l$ is viewed instead of $D_i.A_j$, and vice versa. The result cube is equivalent in construction to the source cube.

Figure 12(b) provides a simple example of how the pivot operation works on the original two-dimensional view in Figure 12(a). The expression is written as $\circlearrowleft_{(Product.Number \rightarrow Time.Month)} Purchase$.

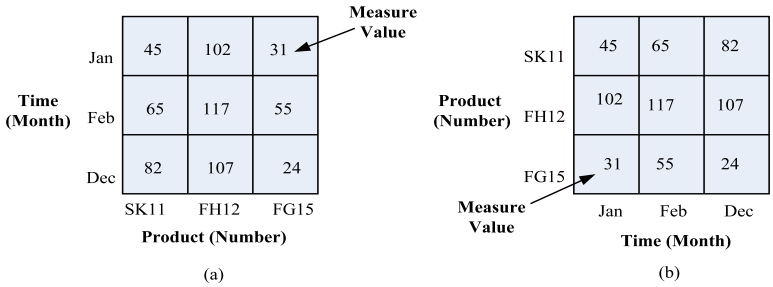


Fig. 12. PIVOT Operation: (a) the original view. (b) the result of the PIVOT operation.

DRILL ACROSS. Here, we denote the integration of two independent cubes, with each possessing common dimensional axes, so as to compare their measure attributes. In effect, this is a cube "join" (possibly a self join) that changes or extends the subject of analysis. Consider two cubes C1 = <D1, F1, M1, BasicCube1> and C2 = <D1, F1, M2, BasicCube2> having the same set of dimensions and feature attributes but with different sets of measure attributes (M1 and M2). We therefore have:

$$C1(M1) \leftrightarrows C2(M2)$$

The result of this operation is another cube C = <D1, F1, M, BasicCube>, where M is the union of sets M1 and M2 and BasicCube contains the union of BasicCube1 and BasicCube2, with the new measure attributes M.

Consider the two cubes (*C1* and *C2*) of Figure 13(a), with both cubes having the same feature attributes (Product Number and Time Month). The measure attribute in *C1* is the total number of units sold during each month for each product number. However, the measure attribute in *C2* is the ordered quantity during each month for each product. The DRILL ACROSS result is found with the following OLAP operation: $C1(UnitsSold) \leftrightarrows C2(QuantityOrdered)$.

SET Operations. The basic Set operations — UNION, INTERSECTION, and DIFFERENCE — may also be applied to data cubes **C1** and **C2**. In all cases, **C1** and **C2** must be composed of the same feature attribute set (i.e., they must possess the same dimensional axes).

Let C1<D, F, M, BasicCube1> and C2<D, F, M, BasicCube2> be two cubes sharing common dimensions, feature attributes and measure attributes. In other words, they have common schemas but may possibly have different cell values. Formally, we can define set operators as:

– UNION: $C1 \cup C2$ is the union of two cubes sharing common dimensional axes. If two cells from C1 and C2 have the same feature attribute values, then we can add their measure attribute values (measure attributes are assumed to be numeric).

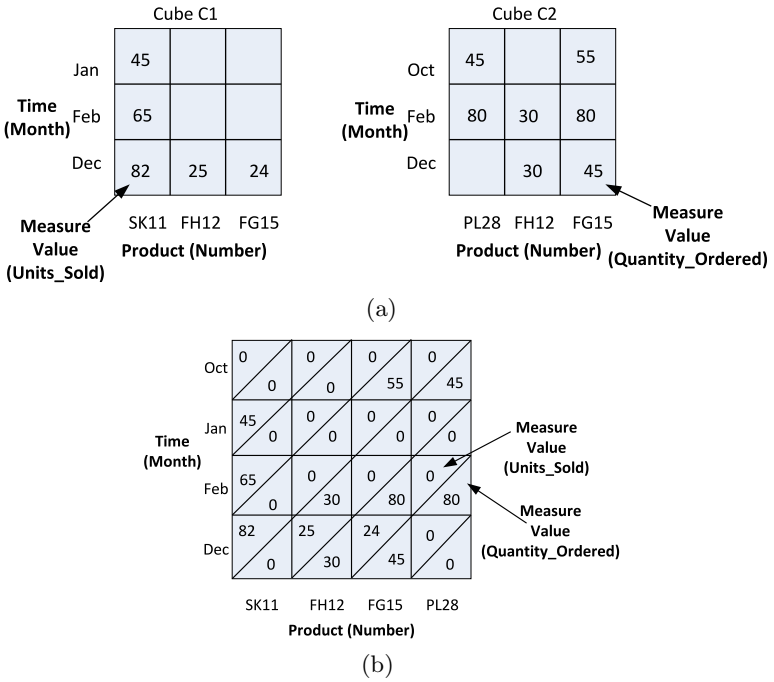


Fig. 13. (a) Two cubes (C1 and C2) with different measure attributes. (b) Two resulting Drill Across on C1 and C2.

- Intersection: $C1 \cap C2$ is the intersection of two cubes sharing common dimensional axes. If two cells are intersected, then we subtract the larger measure attribute values from the smaller.
- Difference: $C1 - C2$ is the difference of two cubes sharing common dimensional axes. When two cells have the same feature attribute values, then the cell value of C1 will be included in the output if its measure value is greater than that of C2.

In the above definitions, the resulting cube has the same schema (<D, F, M>) as any of the source cubes but cell values are calculated according to the operation being performed. Note that C1 and C2 can sometimes be the results of other, intermediate, algebraic expressions.

To further illustrate, suppose we have two cubes *C1* and *C2*, as in Figure 14. Note that both have the same schema (i.e., dimensions, feature attributes, and measure attribute). The result cubes are shown in Figures 15(a), 15(b), 15(c) respectively.

5 Query Optimization

As noted in Section 3.2, native language client-side queries are decomposed into the associated algebraic operators and passed to the Sidera DBMS at runtime.

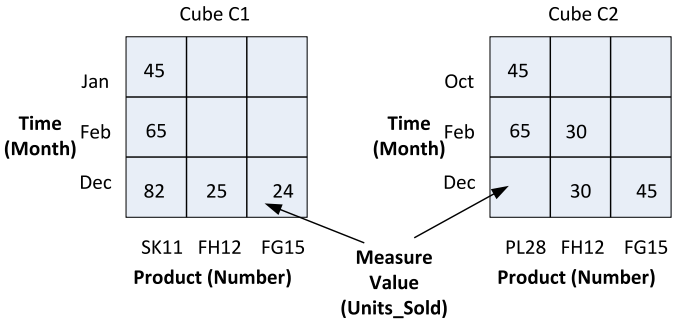


Fig. 14. Two Cubes(*C1* and *C2*) share common feature and measure attributes

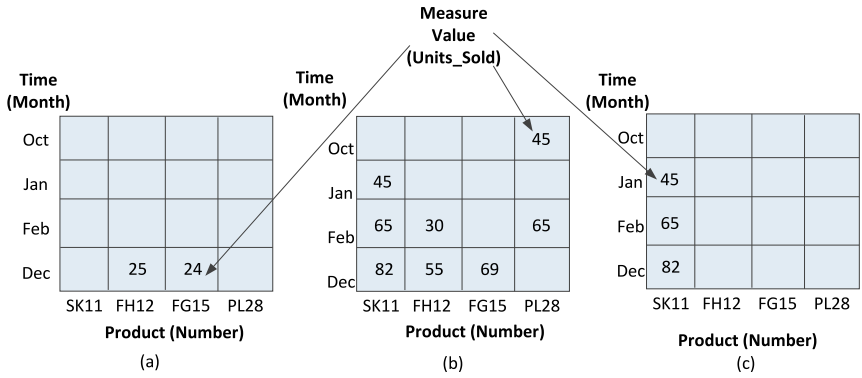


Fig. 15. (a) $C1 \cap C2$ (b) $C1 \cup C2$ (c) $C1 - C2$

That being said, this initial query form likely does not represent the most efficient execution plan for an OLAP DBMS, as no attempt has been made to either exploit the physical representation of the cube (e.g., indexes, materialized views) or the properties of the algebra itself (e.g., re-ordering logical operations to reduce intermediate cube sizes). In fact, similar optimization strategies are employed by conventional DBMS platforms that manipulate the more familiar relational algebra and its associated indexes (e.g., B-trees).

In this section, we will discuss optimization principles relevant to OLAP aware servers in general, and to Sidera in particular. The strategies are organized as a set of Transformation Rules that are used by the DBMS backend to reduce the computational cost of operator execution. While proofs of the transformations are typically straightforward, we will demonstrate the correctness of several rules for the sake of completeness. Finally, we note that the current Sidera prototype

transparently applies these transformation rules at runtime. In Section 6, we will provide experimental results that demonstrate the impact of the optimization process.

5.1 SELECTION

Sidera does not perform traditional relational sort or hash-based joins. Instead, it uses the FastBit indexes to retrieve the relevant dimension keys values, then uses these to directly perform a SELECTION on the Fact Structure. Because the Fact Structure is encoded as a packed R-tree, and is accessed by a linear breadth first search, the "standard" Star Schema query effectively becomes a single pass SELECTION. Rule 1, below, formalizes this notion:

RULE 1: $\sigma_{Dim_1(C_1),Dim_2(C_2),...Dim_n(C_n)}C = \sigma_{Dim_1ID=L_1,Dim_2ID=L_2...Dim_nID=L_n}C$

Proof. Suppose a cell c is in the result of the left expression. Then there must be a record r that satisfies the restriction on dimension Dim_1 and a record s that satisfies the restriction on dimension Dim_2. Moreover, r and s must agree with c on every attribute that each record shares with cell c (Dim_1ID and Dim_2ID). When we evaluate the expression on the right, x is a set of Dim_1 IDs satisfying the restriction associated with Dim_1, while y is a set of Dim_2 IDs satisfying the restriction on dimension Dim_2. Dim_1ID of record r must be in set x and Dim_2ID of record s must be in set y. Thus a cell $c1$ is in the result of the right expression. Consequently, Dim_1ID and Dim_2ID of cell $c1$ must agree with one value from set x (Dim_1ID of record r) and one value from set y (Dim_2ID of record s). Therefore, we can say that $c1$ and $c2$ are the same cell.

Figure 16 provides an illustration of a very simple two dimensional view (called Sales) with the most detailed values stored for each dimension in the cube. Suppose that the schema of this view is (ProductID, EmployeeID, Measure(s)). Feature attributes (EmployeeID and ProductID) allow connections to dimension tables Employee and Product respectively. Now consider the SELECTION expression depicted in Figure 17 and specified as $\sigma_{(Product.Type=Brakes\ AND\ Employee.Age>30)}Sales$.

Using RULE 1,the SELECTION can be written as $\sigma_{(ProductID=x\ AND\ EmployeeID=y)}Sales$ such that $x = \{1, 2\}$ (ProductID 1 and 2 have attribute Type equals Brakes) and $y = \{2, 3\}$ (EmployeeID 2 and 3 have age greater than 30). Figure 18 shows the resulting expression tree after applying RULE 1 to the initial expression tree of Figure 17. In Figure 17, the view (Sales) must be joined with the Product and Employee dimension tables. However, in Figure 18, we access each dimension (Product and Employee) to obtain the most detailed level values that satisfy the condition associated with it; then the view Sales can be accessed alone to answer the query. In other words, the relevant view is accessed in order to return those rows that intersect with the sets of the dimension's detailed values satisfying the query restriction, thereby ensuring efficient use of the supporting R-tree.

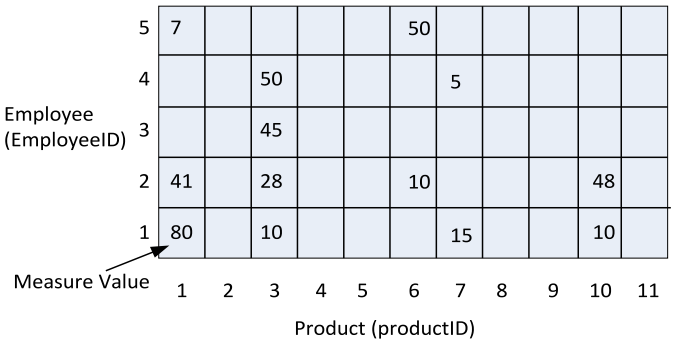


Fig. 16. Two-dimensional cube with the most detailed level values

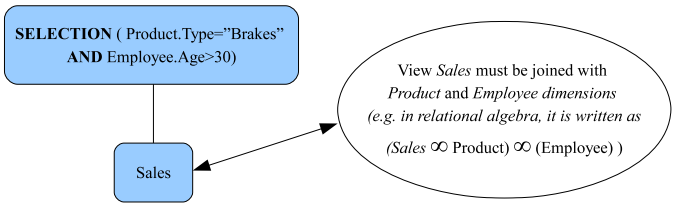


Fig. 17. Initial expression tree

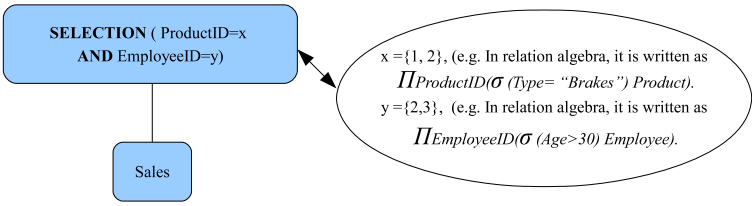


Fig. 18. After applying Rule 1 to the initial tree of Figure 17

Combining Conditions. Multiple SELECTION operators can be integrated into a single SELECTION operator by combining their conditions with the AND operator(s). Thus, our second rule for SELECTION is the *combining* rule:

Rule 2: $\sigma_{(Cond1)}(\sigma_{(Cond2)}C) = \sigma_{(Cond1\ AND\ Cond2)}C$

Proof. Suppose that a cell c is in the result of the left expression. Then the result of $\sigma_{(Cond2)}C$ is a sub-cube $C1$ that contains cell c that satisfies *cond*2. We apply $\sigma_{(Cond1)}$ to $C1$. The result is a sub-cube of the left expression that contains c that also satisfies *cond*2. When we evaluate the right condition, cell c will again be in the result since c satisfies *cond*1 and *cond*2.

Since Sidera provides a very efficient multi-dimensional indexing scheme (i.e., the Hilbert R-tree index), application of this rule allows the SELECTION operation to benefit from this multi-dimensional indexing. Specifically, instead of accessing the appropriate R-tree index view to answer the first condition and then using the result cube to answer the second condition, the multi-dimensional index view can be efficiently used to answer both conditions simultaneously.

In addition, we note that two SELECTION operators can be combined into a single SELECTION if there is a UNION operator between them. Here, the conditions of both SELECTIONs are connected with the OR operator.

Rule 3: $\sigma_{(Cond1)}C\ UNION\ \sigma_{(Cond2)}C = \sigma_{(Cond1\ OR\ Cond2)}C$

Pushing Transformations. SELECTION is arguably the most important operation in terms of OLAP query optimization. In particular, SELECTION tends to reduce the size of intermediate cubes. A primary objective in this regard is to move the SELECTION down the tree as far as it will go without changing what the expression tree actually does. Moreover, pushing SELECTION down the tree makes it possible to efficiently utilize the most appropriate (multi-dimensional) indexed view. Below, we present a *family* of rules (a *RuleSet*) allowing SELECTION to be pushed through other OLAP operators. We therefore refer to this group of transformations as the *pushing laws*.

Rule(set) 4:

1. For a UNION, SELECTION must be pushed to both arguments of the UNION. For example, $\sigma_{(cond)}C1 \cup C2 = \sigma_{(cond)}C1 \cup \sigma_{(cond)}C2$.
2. For a DIFFERENCE, SELECTION must be pushed to the first argument of the operator or to both arguments. For example, $\sigma_{(cond)}C1 - C2 = \sigma_{(cond)}C1 - \sigma_{(cond)}C2$.
3. For an INTERSECTION, SELECTION can be pushed to one of the arguments or both. For example, $\sigma_{(cond)}C1 \cap C2 = \sigma_{(cond)}C1 \cap \sigma_{(cond)}C2$.
4. For CHANGE LEVEL and CHANGE BASE, SELECTION is pushed down to the argument. For example, $\sigma_{(cond)} \pm_{A_i \to action} C1 = \pm_{A_i \to action}\sigma_{(cond)}C1$.
5. For a DRILL ACROSS, SELECTION must be pushed to both arguments. For example, $\sigma_{(cond)}C1(M1) \leftrightarrows C2(M2) = \sigma_{(cond)}C1(M1) \leftrightarrows \sigma_{(cond)}C2(M2)$.

We shall provide a proof of one of the above variations — pushing the SELECTION under UNION — as an example. Proofs for the remaining cases are straightforward.

Proof. Suppose that a cell c is in the result of $\sigma_{(Cond)}C1 \cup C2$. Then the result of $C1 \cup C2$ has a cell c that satisfies the condition parameter of the SELECTION operator. In addition, c can be a cell found only in $C1$, $C2$, or the result of two cells from both cubes $C1$ and $C2$. When we evaluate the right expression, $\sigma_{(Cond)}C1 \cup \sigma_{(Cond)}C2$, c will again be found in the result because c matches the condition and can be found in ($C1$, $C2$), or the result of the UNION.

For example, consider the two cuboids/views C1(Product.Number, Time.Month, UnitsSold) and C2(Product.Number, Time.Month, UnitsSold) of Figure 14. Figure 19(a) illustrates an initial OLAP expression tree. Using Rule(set) 4, we can push the SELECTION operator to both arguments as depicted in Figure 19(b). It is an improvement to do so since we reduce the size of both $C1$ and $C2$ before the INTERSECTION. Moreover, if $C1$ and $C2$ are stored on disk by the server, then we can efficiently retrieve those cells from $C1$ and $C2$ satisfying the condition (e.g., Time.Month = Jan) by utilizing the R-tree index of $C1$ and $C2$.

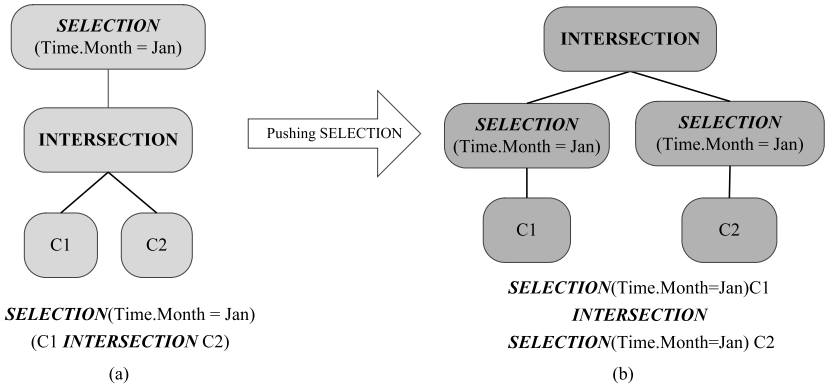


Fig. 19. (a) Initial OLAP expression tree and (b) its equivalent after applying the SELECTION pushing laws

Pulling Transformations. Pushing a SELECTION down an OLAP expression tree is one of the most important steps performed by the query optimizer. However, we have found that in some situations it is essential to pull the SELECTION up the expression tree as far as it will go, and then push it down all possible branches. Consider the two views/cuboids (C1 and C2) of Figure 14. Assume that we have the following OLAP algebra expression:

$$\pi_{(Product.Number,Time.Month,UnitsSold)}(\sigma_{(Time.Month=Dec)}C2 \cap \pi_{(Product.Number,Time.Month,UnitsSold)}C1)$$

The OLAP expression tree of the above OLAP algebra expression is shown in Figure 20(a). Here, there is no way to push the SELECTION down the tree because it is already as far as it would go. However Rule(set) 4.1 can be applied

from right to left, to bring SELECTION (Time.Month = Dec) up above the INTERSECTION. Since $C1$ and $C2$ have the same schemas, we may then push the SELECTION to both arguments (C1 and C2). We can pull the SELECTION above the INTERSECTION and then push down *if and only if* the output of the INTERSECTION contains all attributes mentioned within the SELECTION. For example, because Time.Month is in the output schema of Figure 20(a), we can pull the SELECTION operator up and then down. Figure 20(b) illustrates the resulting expression tree. This mechanism of pulling up and then pushing down the SELECTION operator is advantageous because the size of the view $C1$ is reduced in the INTERSECTION. Moreover, if $C1$ is stored by the Sidera server, then its R-tree index can be efficiently used to find those rows satisfying the condition (Time.Month=Dec). However, without this condition, all cells in $C1$ must be accessed and read into main memory.

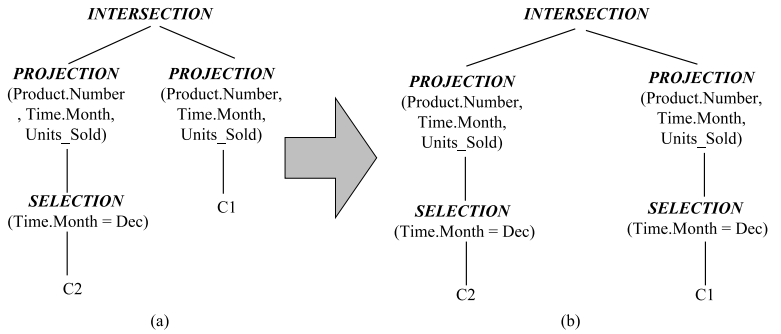


Fig. 20. (a) Initial OLAP expression tree. (b) Improving the initial expression by pulling SELECTION up and then pushing it down the tree.

In addition to the INTERSECTION operator, we can use Rule(set) 4 to pull up the SELECTION(s) that can be combined with the OR operator(s) (Rule 3) and then push them down to both arguments. Again, it is important to mention here that this technique — pull up and then pushing down — can be used only if all attributes mentioned within the SELECTION(s) are in the output schema of the binary operation.

5.2 PROJECTION

Recall that an OLAP query almost always contains a PROJECTION since one or more display attributes are generally required. In other words, the PROJECTION operation determines the schema of the resulting cube query. Therefore, the guiding principle for the PROJECTION rules is that a new PROJECTION may be introduced in the expression tree somewhere *below* an existing PROJECTION. If we do so, the new PROJECTION is guaranteed to only eliminate attributes from the cube that are never used by any of the OLAP operators above.

Rule 5: $\pi_{(L,M)}\sigma_{(Cond)}C = \pi_{(L,M)}\sigma_{(Cond)}\pi_{(L1,M)}C$

where $L1$ is the list of dimension attributes of cube C that are either used within the condition (Cond) of the SELECTION operator or are input attributes of L, and L is the list of output dimension attributes and M is the list of output measure attributes.

To illustrate the importance of this rule, consider the three-dimensional view of Figure 9(a). Figure 21 illustrates the application of the rule. Assume that the user wants to see the total UnitsSold in all cities during the month of January (Jan). The initial expression tree is depicted in Figure 21(a). By applying Rule 5, a new PROJECTION is introduced below the SELECTION to eliminate attribute Product.Number, since the only required attributes/axes are Time.Month and Location.City. Figure 21(b) illustrates the resulting expression tree.

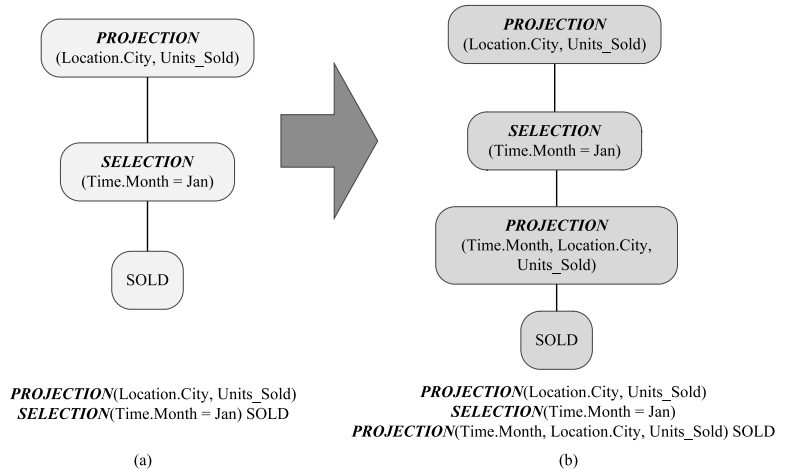


Fig. 21. (a) Initial OLAP expression tree.(b) Improving the initial expression by introducing new PROJECTION.

Because CHANGE LEVEL and CHANGE BASE are relevant to an existing result set, PROJECTION must be pushed below these operators. Below are the pushing rules for the PROJECTION operator.

Rule(set) 6:

1. $\pi_{(L)}\ \Updownarrow_{(M)} C = \Updownarrow_{(M)}\ \pi_{(L)}C$
2. $\pi_{(L)} \pm (M)C = \pm_{(M)}\pi_{(L)}C$

We note that a new PROJECTION cannot be introduced below the binary operations (UNION, INTERSECTION, DIFFERENCE, and DRILL ACROSS).

Decomposition. Recall that Sidera stores the cube only for the most detailed dimension values (e.g., ProductID, EmployeeID, etc.). As was noted above, PROJECTION results depend on inner/natural joins between the cube and dimension tables to produce descriptive OLAP reports, since it is the dimension tables that store descriptive attributes. Below is the decomposition rule for PROJECTION:

$$\textbf{Rule 7: } \pi(L, M)C = \Updownarrow_{(L1 \to L)} \Updownarrow_{(L1, M)} C$$

where L is a list of hierarchical attributes, $L1$ is the list of feature attributes in C that link the cube C with its corresponding dimensions mentioned in L, and M is one or more measure attributes in C. This transformation is able to improve query plans since it defers the joins between the cube and the dimension tables to a later step of the query execution. As a result, the PROJECTION can be answered from the appropriate view (e.g., C) without dimension table joins, with the joins required for the CHANGE LEVEL operation being deferred to a later stage of the query.

Consider the two dimensional cuboids in cube Sales of Figure 16 and its surrounding dimension tables (Employee and Product). Figure 22(a) illustrates the initial OLAP expression tree in which only one PROJECTION operation needs to be joined with dimension tables. Using Rule 7, the optimizer produces the expression tree depicted in Figure 22(b).

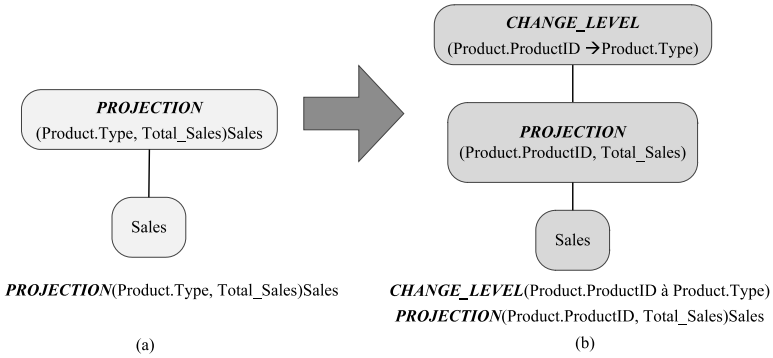


Fig. 22. (a) Initial OLAP expression tree. (b) Improving the initial expression by decomposing the PROJECTION.

5.3 CHANGE LEVEL and CHANGE BASE

As noted, the CHANGE LEVEL and CHANGE BASE operators are relevant only to an existing result set. Therefore, they must be located above the PROJECTION operator that defines the schema of the output cube result (Rule 6).

Removal Transformation. The CHANGE BASE operator can be removed from an OLAP expression tree as follows:

Rule 8: $\pm_{(L1 \rightarrow Add, L2 \rightarrow Remove)} \pi_{(L,M)} C = \pi_{(K,M)} C.$

where L, $L1$ and $L2$ are lists of attributes such that all attributes that are removed by the CHANGE BASE operator (L2) must be in the input cube that is itself the result of the PROJECTION operator ($L2 \subset L$). K is a list of attributes that are in L, $L1$ and not in $L2$ ($K = L \cup L1 \notin L2$). This law provides potential benefit in that only one PROJECTION will be executed instead of a PROJECTION *and* a CHANGE BASE.

Pushing and Pulling Rules. We define a pair of push/pull rules for the CHANGE LEVEL and CHANGE BASE operators:

Rule 9: If the CHANGE LEVEL operator changes the result data from the most summarized (up) to the most detailed (down) along a concept hierarchy (Drill Down), then we pull the operator up the tree until it reaches another CHANGE LEVEL.

In general, this transformation reduces the size of intermediate results due to the fact that it pulls up the Drill Down operation that would otherwise increase the size of the result cube. In addition, the root of the resulting expression tree becomes a CHANGE LEVEL operator. As a concrete example, consider the OLAP expression tree of Figure 23(a) (e.g., CHANGE LEVEL(K) corresponds to a drill down operation). The resulting expression tree, after applying Rule 9, is depicted in Figure 23(b). The second tree is an improvement over the first one because after pulling the CHANGE LEVEL operation up, we reduce the size of the intermediate result. For example, in Figure 23(a) the size of the intermediate results (e.g., R) before the UNION is larger than that of Figure 23(b) (e.g., R1) because CHANGE LEVEL in the former transforms the result data from the most summarized level to the most detailed level.

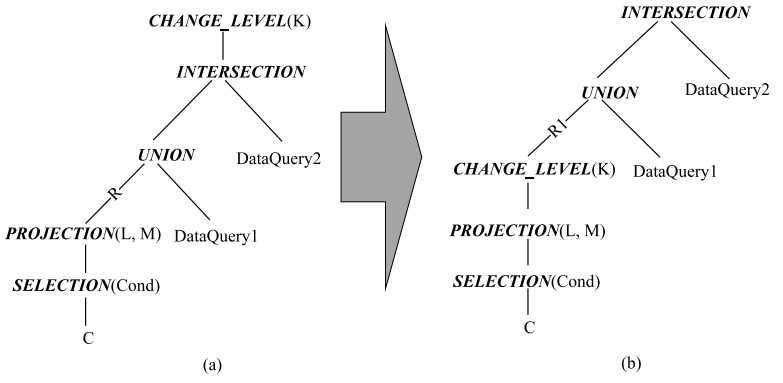


Fig. 23. (a) Initial OLAP expression tree. (b) Result of pulling up CHANGE LEVEL(K).

Rule 10: If the CHANGE LEVEL operator navigates among levels of data ranging from the most detailed (down) to the most summarized (up) (Roll Up), then we push it down the tree until it reaches a PROJECTION operator. In general, pushing this type of CHANGE LEVEL (Roll-up) operator reduces the size of intermediate results because the result of the roll up operation changes from the most detailed level to the most summarized (e.g., 12 month values is one year value). Note that the CHANGE LEVEL can be pushed below the UNION, INTERSECTION and DRILL ACROSS but *not* below the DIFFERENCE operator.

Finally, we note that if a CHANGE LEVEL involves both operations (a Drill Down and Roll Up) of an existing result set, then it is important to estimate the size of intermediate results before deciding whether to pull it up or push it down.

Merging. We refer to the following transformation concerning the CHANGE LEVEL operator as the *merging rule*, as it merges two or more consecutive CHANGE LEVELS into one.

$$\textbf{Rule 11: } \Updownarrow_{(LI \to LO)} \Updownarrow_{(MI \to MO)} C = \Updownarrow_{(KI \to KO)} C$$

where LI, LO, MI, MO, KI and KO are lists of hierarchical attributes. Recall that CHANGE LEVEL is applied to an existing result cube to change one or more attribute level values. Therefore, all attributes mentioned in MI must be in the result view C. The parameters of the CHANGE LEVELS are merged as follows:

- $KI = LI \cup MI$, for all attributes in MI and LI. However, if one or more attributes in MI and LI belong to the same hierarchy, then we select only attributes from LI to be in KI.
- $KO = LO \cup MO$, for all attributes in LO and MO. However, if they have attributes belonging to the same hierarchy, then we select those attributes from LO.

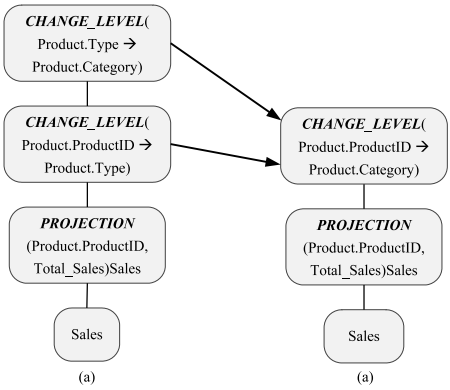


Fig. 24. (a) Initial OLAP expression tree. (b) Result of merging two CHANGE LEVELS.

For example, consider the two-dimensional view Sales of Figure 16 and its surrounding dimensions (Employee and Product). Figure 24(a) illustrates the initial expression tree with two consecutive CHANGE LEVELS. Here, LI = Product.Type, LO = Product.Category, MI = Product.ProductID and MO = Product.Type. Figure 24 illustrates the result expression tree produced by using Rule 11. In this example, $KI = MI$ and $KO = LO$ because (LI and MI) and (LO and MO) have attributes that belong to the same hierarchy (Product). The transformation rule is likely to improve the performance if the CHANGE LEVELS involve attributes from the same hierarchy due to the fact that it reduces the number of translations between hierarchical levels.

5.4 Commutative and Associative Transformations

Several of our OLAP algebraic operators are both associative and commutative. Figure 25 provides a listing of the Associative and Commutative rules relevant to the algebra. Note that, strictly speaking, these are not a new rules; they are merely a property of existing formalisms.

6 Experimental Results

Sidera is a relatively sophisticated prototype and, as such, lends itself to meaningful experimental evaluation. We stress that Sidera is a DBMS in the true sense of the word. In other words, it is not simply an interface to a relational or even multidimensional server. Rather, it provides data storage, indexing, query parsing, optimization, and caching services. As such, the experimental results listed below provide a reasonable representation of the potential for this type of OLAP model (i.e., one that uses OLAP-specific indexes, storage and algebraic operations to provide scalable OLAP functionality).

In terms of the environment, tests were conducted on a dual-boot workstation running Windows Vista and a Fedora Linux distribution (2.6.x kernel). (Note that we perform single-node evaluation in this paper rather than utilizing the full cluster architecture). The machine uses 1GB of main memory and houses

r1	C1 ***UNION*** C2 = C2 ***UNION*** C1
r2	(C1 ***UNION*** C2) ***UNION*** C3 = C1 ***UNION*** (C2 ***UNION*** C3)
r3	C1 ***INTERSECTION*** C2 = C2 ***INTERSECTION*** C1
r4	(C1 ***INTERSECTION*** C2) ***INTERSECTION*** C3 = C1 ***INTERSECTION*** (C2 ***INTERSECTION*** C3)
r5	C1(M1) ***DRILL_ACROSS*** C2(M2) = C2(M2) ***DRILL_ACROSS*** C1(M1)
r6	(C1(M1) ***DRILL_ACROSS*** C2(M2)) ***DRILL_ACROSS*** C3(M3) = C1(M1) ***DRILL_ACROSS*** (C2(M2) ***DRILL_ACROSS*** C3(M3))

Fig. 25. Commutative and Associative rules

a standard 160 GB SATA hard drive. The analytics database consists of six dimensions with cardinalities ranging between 300 and one million. Each dimension also has a three or four level hierarchy. Dimension data was generated with an open source data generation tool so as to more accurately represent real (i.e., text) values. In terms of the the Fact Structure, relevant feature attributes (i.e., with matching keys) and measure attributes were produced by a generator designed specifically for Sidera. While the generator has the ability to produce skewed data, the distribution in the current case is essentially uniform as skew is largely irrelevant for the current round of testing. Moreover, we note that while skewed data tends to more challenging/expensive for indexing and processing algorithms in general, that is not the case for the packed, tuple compressed Hilbert indexes used by Sidera. In fact, uniform distributions, which maximize the geometric distance between points/rows in the multi-dimensional space, are actually the most expensive in terms of storage and IO. With respect to data set size, row counts typically vary from 100,000 records to 10,000,000 records. Once generated, Rtrees and Bitmaps are constructed as required.

Because no true OLAP query benchmark currently exists, we developed a set of "Star Schema queries" representing common OLAP operations (slice and dice, drill down, roll up, etc). We note that while TPC-H is considered to be a Decision Support standard, it does not actually utilize a proper Star Schema and its focus is upon somewhat lower level ad hoc queries. Still, the general style of the queries developed for the current testing is inspired by the TPC-H approach. In the Appendix, we have included a list of 12 queries found in one of our test sets. Other sets are similar, but utilize different dimensions, hierarchies, and query conditions. Each individual input query was encoded in three formats, as required by the target platform: SQL (relational OLAP), MDX (multi-dimensional OLAP,) and XML (Sidera). Unless otherwise indicated, batches of 10-20 such queries are used in a given test, with the average of five runs recorded. Finally, query and OS caches are cleared between runs.

We begin by looking at the performance of the Fact Structure described in Section 3.3. In most environments, indexing demonstrates increasingly poor performance once query selectivity reaches a certain point, typically about 5% of the records in the data set. However, Sidera's Berkeley R-tree storage — with its breadth first traversal pattern that limits access to a single sequential pass — does not degrade in this manner. Figure 26(a) illustrates that for a 12-query batch, with selectivity ranging between 1% and 25%, Sidera's query performance remains 3-4 times faster than that of a sequential scan of the data set. Figure 26(b), on the other hand, demonstrates Sidera's ability to exploit R-tree Fact Structures containing a fully materialized cube (i.e., all aggregation levels included) generated by a Sidera ETL module. Specifically, for data sets of 10 million records, the same query batch completes in one tenth of the time (black bars) if aggregates are available (Note that the query optimizer transparently determines the optimal summary view).

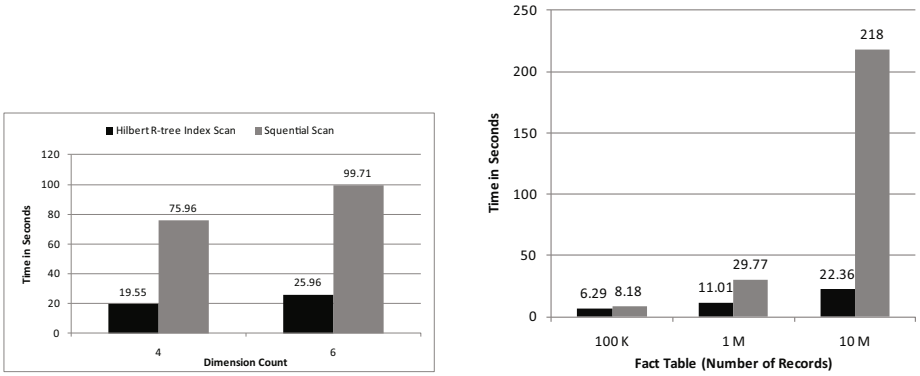


Fig. 26. (a) Rtree performance versus sequential scan (10M records) (b) Fact structure performance by record count

Figure 27 shows query performance — relative to record and dimension counts — on a batch of 16 OLAP queries that have been parsed into the algebraic operations described in the paper and re-written by Sidera using the join and pushing optimizations described in Section 5. We remind the reader that such optimization is analogous to that employed by conventional relational DBMSs when optimizing incoming queries. Here, we see performance improve by a factor of 5-15 when optimization steps are undertaken (black bars).

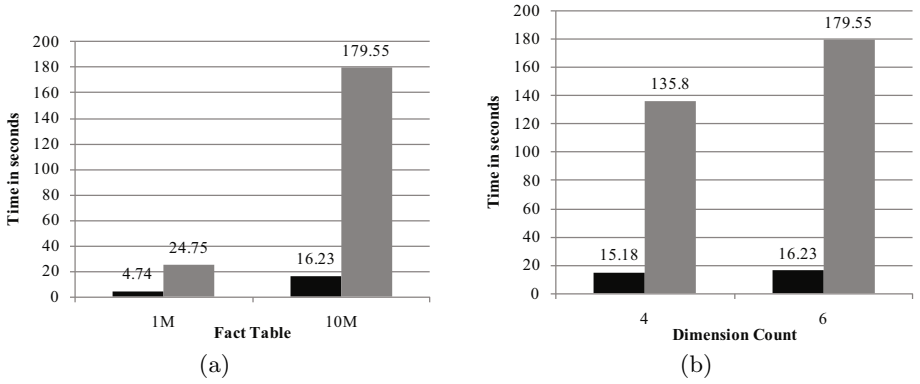


Fig. 27. SELECTION optimization by (a) Record count (b) Dimension count

We have also compared Sidera to DBMS systems often used in industrial database environments, namely the open source MySQL server and Microsoft's Analysis Services. In this case, we reproduce the database stored by Sidera and load it into both DBMS platforms in the standard Star Schema format. Queries

are re-written in SQL form to match. Figure 28 shows comparative results for both platforms and demonstrates that the MySQL server takes approximately 10-15 times as long to resolve the same queries, while Microsoft's Analysis Service — running in ROLAP mode — is three to six times slower. In all three cases, the DBMS query optimizers are free to transparently re-write the algebraic representation of the input query as they see fit. We note, of course, that additional manual tuning could improve the execution time of each of the three DBMS systems. The relational systems, for example, could be improved with additional indexes while the Sidera server is designed to exploit additional partial aggregates if permitted. However, the objective in this test was to provide a baseline evaluation in which both systems were run in an "out-of-the-box" fashion, without benefit of supplemental indexes, materialization, or data structures. In practice, for the relational servers this implies standard B-tree indexes on the Dimension tables and a composite, multi-column B-tree index on the Fact table.

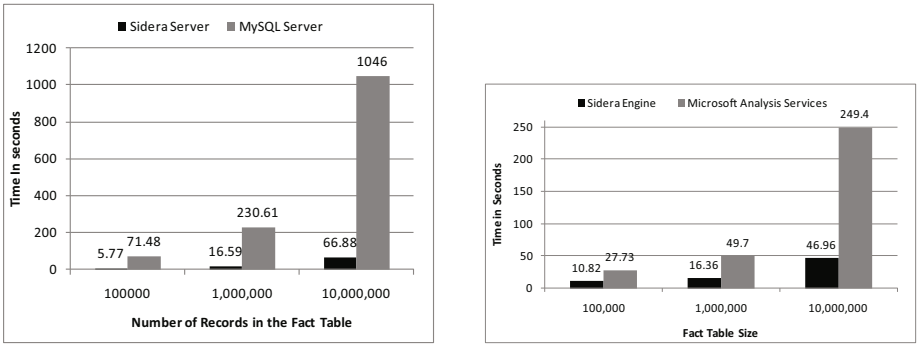


Fig. 28. Sidera versus (a) MySQL (b) MS Analysis Services (ROLAP)

Of course, one can argue that MOLAP (i.e., native cube) offers superior performance to ROLAP (i.e. tabular) configurations, even when relational tables are augmented with extensive indexing. In fact, for cubes fitting entirely in main memory, little or no indexing is required at all on the main cube structure. Therefore, to test a system closer to the "state of the art" in OLAP processing (i.e., the DBMS is augmented in some way for OLAP query loads), we loaded the same Star Schema data using the MOLAP mode of Microsoft's Analysis Services. Figure 29(a) shows that MOLAP does indeed outperform the Sidera DBMS by a factor of about 5 to 1. However, we note that in this test, Sidera was not permitted to materialize any additional data; it was essentially just an efficient Star Schema, as configured for the ROLAP comparison above. In Figure 29(b), we see the result once the server is allowed to augment its basic structures with additional aggregate data. We stress that the server performs this materialization transparently in its standard execution mode — no view

identification or manual tuning is required by the end user. While Microsoft's MOLAP server still has a slight advantage in this follow up test, we note that (i) the Microsoft DBMS benefits from years of optimization, and (ii) MOLAP is ideally suited to the scale of the current test (i.e., 1-10 million records). Given that the Sidera DBMS framework is not constrained by the limits of array-based storage, these preliminary results suggest that the Sidera DBMS (and its optimization engine) has the potential to provide MOLAP-style performance with ROLAP-style scalability.

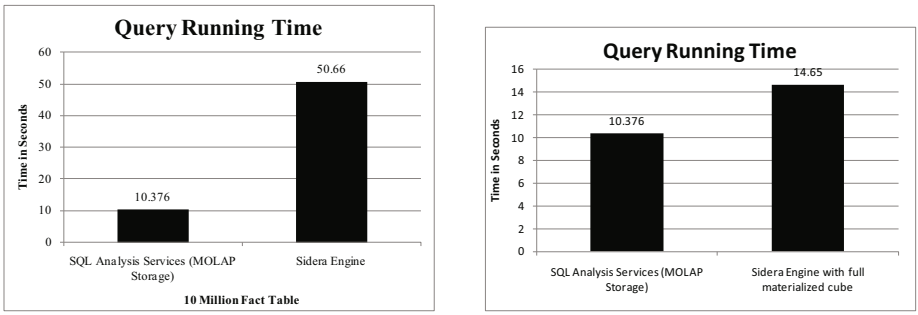


Fig. 29. (a) MOLAP versus non-materialized Sidera (b) MOLAP versus materialized Sidera

7 Conclusions

OLAP servers have traditionally relied either on extensions to DBM systems designed primarily for OLTP environments or on array-based servers that lack a formal query model and tend to provide limited scalability. In this paper, we have discussed the integration of an OLAP-oriented algebra with a DBMS prototype designed specifically for analytical processing. The use of the algebraic operators lends itself to both a clean, native language query interface for end users and a query execution engine that is able to optimize performance by manipulating initial parse trees to more efficiently exploit the available index and storage structures. We have provided an extensive analysis of the optimization of the algebra, describing general strategies in terms of a series of parse tree transformation rules. Initial testing demonstrates that not only does the DBMS provide a contemporary OOP interface for end users, but that it is already competitive in performance to commercial systems optimized for in-memory OLAP. Given that Sidera is ultimately designed as a scalable parallel system, we believe the current work suggests that MOLAP-level performance — at commodity prices — is indeed possible for practical environments in which one cannot expect data sets to reside entirely in main memory.

References

1. JSR 243: Java Data Objects 2.0 - An Extension to the JDO specification (2008), http://java.sun.com/products/jdo/
2. HaskellDB (2010), http://www.haskell.org/haskellDB/
3. Berkeleydb (2011), http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html
4. Fastbit indexing (2011), http://crd.lbl.gov/~kewu/fastbit/index.html
5. Ruby programming language (2011), http://www.ruby-lang.org/en/
6. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, pp. 539–550. ACM, New York (2003)
7. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co., Greenwich (2006)
8. Bellatreche, L., Giacometti, A., Laurent, D., Marcel, P., Mouloudi, H.: Olap query optimization: A framework forcombining rule-based and cost-based approaches. In: EDA (2005)
9. Blakeley, J.A., Rao, V., Kunen, I., Prout, A., Henaire, M., Kleinerman, C.: NET database programmability and extensibility in Microsoft SQL Server. In: ACM SIGMOD International conference on Management of Data, pp. 1087–1098. ACM, New York (2008)
10. Bruno, N., Chaudhuri, S., Gravano, L.: Stholes: a multidimensional workload-aware histogram. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD 2001, pp. 211–222. ACM, New York (2001)
11. Chen, Z., Ordonez, C.: Efficient olap with udfs. In: Proceedings of the ACM 11th International Workshop on Data Warehousing and OLAP, DOLAP 2008, pp. 41–48 (2008)
12. Chmiel, J., Morzy, T., Wrembel, R.: Time-hobi: indexing dimension hierarchies by means of hierarchically organized bitmaps. In: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP, DOLAP 2010, pp. 69–76. ACM, New York (2010)
13. Cook, W.R., Rai, S.: Safe query objects: statically typed objects as remotely executable queries. In: International Conference on Software Engineering (ICSE), pp. 97–106 (2005)
14. Cunningham, C., Graefe, G., Galindo-Legaria, C.A.: PIVOT and UNPIVOT: Optimization and execution strategies in an RDBMS. In: International Conference on Very Large Data Bases (VLDB), pp. 998–1009 (2004)
15. Cuzzocrea, A., Furfaro, F., Saccà, D.: Enabling olap in mobile environments via intelligent data cube compression techniques. J. Intell. Inf. Syst. 33(2), 95–143 (2009)
16. Cuzzocrea, A., Serafino, P.: Lcs-hist: taming massive high-dimensional data cube compression. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT 2009, pp. 768–779. ACM, New York (2009)
17. Cuzzocrea, A., Wang, W.: Approximate range—sum query answering on data cubes with probabilistic guarantees. J. Intell. Inf. Syst. 28(2), 161–197 (2007)
18. Eavis, T., Cueva, D.: The lbf r-tree: Efficient multidimensional indexing with graceful degradation. In: Proc. 11th International Database Engineering and Applications Symposium, IDEAS 2007, September 6-8, pp. 241–250 (2007)

19. Eavis, T., Tabbara, H., Taleb, A.: The NOX Framework: Native Language Queries for Business Intelligence Applications. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DAWAK 2010. LNCS, vol. 6263, pp. 172–189. Springer, Heidelberg (2010)
20. Eavis, T., Taleb, A.: Mapgraph: efficient methods for complex olap hierarchies. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, pp. 465–474. ACM, New York (2007)
21. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In: International Conference on Data Engineering (ICDE), pp. 152–159. IEEE Computer Society, Washington, DC (1996)
22. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: Hyrise: a main memory hybrid storage engine. Proc. VLDB Endow 4, 105–116 (2010)
23. Hanusse, N., Maabout, S., Tofan, R.: A view selection algorithm with performance guarantee. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT 2009, pp. 946–957. ACM, New York (2009)
24. Hose, K., Klan, D., Marx, M., Sattler, K.-U.: When is it time to rethink the aggregate configuration of your olap server? Proc. VLDB Endow 1, 1492–1495 (2008)
25. Lauer, T., Datta, A., Khadikov, Z., Anselm, C.: Exploring graphics processing units as parallel coprocessors for online aggregation. In: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP, DOLAP 2010, pp. 77–84. ACM, New York (2010)
26. Morfonios, K., Ioannidis, Y.: CURE for cubes: cubing using a ROLAP engine. In: International Conference on Very Large Data Bases (VLDB), pp. 379–390. VLDB Endowment (2006)
27. Romero, O., Abelló, A.: On the Need of a Reference Algebra for OLAP. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 99–110. Springer, Heidelberg (2007)
28. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: International conference on Very Large Data Bases (VLDB), pp. 1150–1160 (2007)
29. Taleb, A., Eavis, T., Tabbara, H.: The NOX OLAP Query Model: From Algebra to Execution. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 167–183. Springer, Heidelberg (2011)
30. Whitehorn, M., Zare, R., Pasumansky, M.: Fast Track to MDX. Springer-Verlag New York, Inc., Secaucus (2005)
31. Witkowski, A., Bellamkonda, S., Bozkaya, T., Dorman, G., Folkert, N., Gupta, A., Shen, L., Subramanian, S.: Spreadsheets in rdbms for olap. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD 2003, pp. 52–63 (2003)

A Star Schema Queries

Below, we provide a sample of 12 "TPC-H" inspired SQL queries used in our experimental testing (additional query batches were also used). They are based upon a typical six dimensional cube (Customer, Vendor, Employee, Product, Store, Time), represented as a Star Schema called Sales. (Note that just four dimensions were used in this particular batch). The Sidera schema, while not relational, was configured to mimic this same design.

1. **SELECT** c.Region, SUM(s.Tolal_Sales)
   FROM customer as c , sales as s
   WHERE s.C_ID = c.C_ID and c.Age = 50 and c.Region = 'Quebec'
   GROUP BY c.Region
2. **SELECT** t.Month, SUM(s.Tolal_Sales)
   FROM time as t, sales as s
   WHERE s.T_ID = t.t_ID and t.Year = 2005 and DayName ='Monday' and t.Quarter ='Q1'
   GROUP BY t.Month
3. **SELECT** p.Type, SUM(s.Tolal_Sales)
   FROM product as p, sales as s
   WHERE s.P_ID = p.P_ID and p.ProdDesc = 'Urna' and p.Category ='Automotive' and p.Quantity=200
   GROUP BY p.Type
4. **SELECT** s.StoreCity, t.Month, SUM(ss.Tolal_Sales)
   FROM time as t , store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and ((t.year=2005 and t.DayName = 'Monday') and (t.Quarter='Q1' or t.Quarter='Q2')) and s.StoreState='Ontario'
   GROUP BY s.StoreCity, t.Month
5. **SELECT** c.Region, p.Category, SUM(ss.Tolal_Sales)
   FROM customer as c, product as p, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.P_ID = p.P_ID and c.Age = 40 and c.Country = 'Canada' and p.Quantity=200 and p.Category='Automotive'
   GROUP BY c.Region, p.Category
6. **SELECT** c.country, t.Month, SUm(ss.Tolal_Sales)
   FROM customer as c , time as t, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and (((t.year=2005 and t.DayName='Monday') and (t.Month='May' or t.Month='June')) and (c.Age = 40 and c.Region = 'Quebec'))
   GROUP BY t.Month,c.country
7. **SELECT** c.Region, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
   FROM customer as c, product as p , store as s, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and s.StoreState= 'Ontario' and p.Category = 'Household'
   GROUP BY c.Region, p.Type, s.StoreCity

8. **SELECT** s.StoreCity, t.Month, SUM(ss.Tolal_Sales)
   FROM time as t,customer as c , store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and (t.year=2005 and t.DayName='Monday') and (c.Age = 40 and c.Region = 'Quebec') and s.StoreState = 'Ontario'
   GROUP BY s.StoreCity, t.Month
9. **SELECT** t.Quarter, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
   FROM time as t, product as p , store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and t.Year = 2005 and t.Quarter = 'Q1' and s.StoreState= 'Ontario' and p.Category ='Household'
   GROUP BY t.Quarter,p.Type, s.StoreCity
10. **SELECT** t.Quarter,c.Region, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
    FROM time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and s.StoreState= 'Ontario' and p.Category = 'Household'
    GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter
11. **SELECT** p.Type, s.StoreState, SUM(ss.Tolal_Sales) **FROM** time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and t.Quarter= 'Q1' and p.Category = 'Household'
    GROUP BY p.Type, s.StoreState
12. **SELECT** t.Quarter, SUM(ss.Tolal_Sales)
    FROM time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Age = 40 and t.year =2005 and s.StoreState= 'Ontario' and p.Type = 'Engine'
    GROUP BY t.Quarter

Finding Critical Thresholds for Defining Bursts in Event Logs

Bibudh Lahiri[1,⋆], Ioannis Akrotirianakis[2], and Fabian Moerchen[2]

[1] Case Commons, New York, NY, USA 10003
bibudh@casecommons.org
[2] Siemens Corporate Research, Princeton, NJ, USA 08540
{ioannis.akrotirianakis,fabian.moerchen}@siemens.com

Abstract. A burst, i.e., an unusally high frequency of occurrence of an event in a time-window, is interesting in many monitoring applications that give rise to temporal data as it often indicates an abnormal activity. While the problem of detecting bursts from time-series data has been well addressed, the question of what choice of thresholds, on the number of events as well as on the window size, makes a window "unusally bursty" remains a relevant one. We consider the problem of finding *critical* values of both these thresholds. Since for most applications, we hardly have any apriori idea of what combination of thresholds is critical, the range of possible values for either threshold can be very large. We formulate finding the combination of critical thresholds as a two-dimensional search problem and design efficient deteministic and randomized divide-and-conquer heuristics. For the deterministic heuristic, we show that under some weak assumptions, the computational overhead is logarithmic in the sizes of the ranges. Under identical assumptions, the expected computational overhead of the randomized heuristic in the worst case is also logarithmic. Using data obtained from logs of medical equipment, we conduct extensive simulations that reinforce our theoretical results, and show that on average, the randomized heuristic beats its deteministic counterpart in practice.

Keywords: Analytics for temporal data, Massive data analytics: algorithms.

1 Introduction

A burst is a window in time when an event shows an unusally high frequency of occurrence. For applications where the data arrives as a time series or a stream, a burst indicates a period in time when some interesting event took place. For example, following the death of Michael Jackson on June 25, 2009, there was a flood of related tweets on Twitter, and analytics tools like TwitterMonitor

⋆ This work was done when this author was an intern at Siemens Corporate Research, Princeton. A preliminary version of this paper appeared in the Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery, 2011.

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 89–112, 2013.

[13] have been developed to detect such trends through burst of keywords from Twitter streams. If we consider text streams derived from news articles or blogs, a historically important event like the 9/11 attacks did give rise to a burst of the keywords like "twin towers" or "World Trade Center". In a pay-per-click revenue model [15], a burst in clicks to an online advertisement might indicate a click fraud [27]. Over the Internet, network instrusions often exhibit a bursty traffic pattern [24] - the traffic from the attacker to the victim shows a sudden deluge, does the damage, and then fades away. In astrophysics, a Gamma ray burst might indicate an interesting phenomenon [29,28].

Labelling a window in time as "bursty" calls for at least two thresholds - one on the number of events (k) and the other on the length of the window (t). We call a window (k,t)-bursty if at least k events occur in a time interval of length at most t. While the problem of identifying (k,t)-bursty windows, given k and t, is interesting in itself, knowing the right thresholds is part of the problem, as even Zhu and Shasha [29] acknowledged. For a given t, to know what value of k should be termed "unusually high", we first need to know typically how many events to expect in a window of length t. For example, the keyword 'Obama' might appear in 50 tweets per minute under normal circumstances, so a frequency of 4000 tweets per minute should be counted as a burst. Similarly, for a given k, to know what value of t is "unusually low", we first need to know typically how long it takes to generate k events.

Before we formally defined the problem of finding the critical thresholds, we had to quantify the notion of "usual" and "unusual". Thus, we defined the following metric which we call "coverage":

Definition 1. *Given a threshold pair (k,t), and a sequence of timestamps of n events, the coverage $C_{k,t}$ is the fraction of the n events that were included in* some (k,t)*-bursty window.*

Note that, a single event can be part of more than one (k,t)-bursty windows. For a given pair (k,t), if we find that $C_{k,t}$ is quite close to 1, then actually we are not interested in such a pair (k,t), because that implies having at least k events in a window of length at most t is not unusual, and hence can hardly be labelled as a burst. On the other hand, if $C_{k,t}$ is quite close to 0 but still positive, then it implies having k events in a window of length at most t is not usual, and hence demands attention. For example, for a particular event code, we found that $C_{2,21} = 0.29$, but $C_{3,21} = 0.0125$, which implies (loosely speaking), in a window of at most 21 minutes, an event has about 30% chance of being part of a "pair", but less than 2% chance of being part of a triplet, so a triplet is (23 times) more likely to qualify as a "burst".

Note that, the way we define the coverage ensures that $C_{k,t}$ will always be within the real interval $[0,1]$, and gives us a quick, high-level idea of whether a (k,t)-bursty window should actually be called "bursty" for an application, even if we do not have other information like the number of (k,t)-bursty windows or the length of the sequence n. This also makes the "coverage" monotonically

non-increasing in k and non-decreasing in t, properties that we prove in Section 4 and take advantage of in Section 5 while designing our algorithms.

We focus on identifying *critical* pairs (k^*, t^*) such that C_{k^*,t^*} is abruptly different from values of $C_{k,t}$ for pairs (k, t) which are in the neighborhood of (k^*, t^*) - this implies having k^* events in a window of length at most t^* is not a regular phenomenon, yet there are some rare situations when this has happened. Note that, for a given pair (k, t), $C_{k,t}$ can be computed by making a single pass over the data, but if the range of possible values for k and t have sizes K and T respectively, then evaluating $C_{k,t}$ at every point in the two-dimensional space would have a computational overhead of $O(KT)$. Since for most applications, we hardly have any apriori idea of what combination of thresholds is critical, each of K and T can be rather large, e.g., t might range from a few minutes to a few hours, depending on the nature of the application, and k might take any value from 2 to a few thousand.

Our contributions can be summarized as follows:

- We formally define the problem of finding critical threshold pairs that should label a subsequence of a time series data as *unusually bursty*. We formulate it as a two-dimesional search problem.
- We prove monotonicity properties of the coverage function rigorously, and exploit them to design deteministic and randomized divide-and-conquer algorithms that explore the search space efficiently. We analytically show that, under some weak assumptions, the deterministic algorithm computes $C_{k,t}$ at $O(\log K \log T)$ different points, and, under identical assumptions, the randomized algorithm also computes $C_{k,t}$ at $O(\log K \log T)$ different points (on expectation) in the worst case.
- Using logs generated during the operation of medical equipment at Siemens Healthcare, we compared the performance of our deterministic and randomized algorithms with that of a naive algorithm that evaluates $C_{k,t}$ at at most, but typically much less than, KT points. Even though we made some optimizations over the naive algorithm, the savings made by our algorithms are in the range of 41% to 97%.

2 Related Work

Detecting bursts or self-similarity in time series or streaming data, e.g. ethernet and Internet traffic, multimedia, disk traffic, has drawn significant attention in the research community. While some of the earlier effort [8,3,2,22] were devoted to finding the appropriate parameters for modelling bursts in real-life data, more recently, there has been work on finding efficient algorithms for burst detection [29,28]. Zhu and Shasha [29] considered the problem of "elastic" burst detection, where they kept a different threshold for each unique window size and identified windows over the time-series when an aggregate function (sum) computed over the window exceeded the corresponding threshold. The concept of "elastic" windows, i.e., windows of multiple sizes, was introduced since for most applications, we hardly have any apriori idea of what choice of window size might reveal

a burst. Their algorithm builds a Shifted (binary) Wavelet Tree (SWT) data structure on the time series of length n, and the complexity of their algorithm is $O(n + k)$, where k is the output size - the total number of windows where (true and false) bursts were reported. In [28], the same authors generalized the Shifted Wavelet Tree to create a faster data structure called Aggregation Pyramid. The experiments in [28] also revealed *correlated bursts* in stock exchange data streams, among stocks within the same business sector as well as among stocks from different sectors.

The correlated nature of bursts in multiple streams are observed in text streams or stock exchange data because in those applications, bursts are typically caused by high-impact real-life events, and when such an event occurs, it effects all sources that generate these streams. Correlated bursts, in the context of coordinated text streams, were also addressed by Wang *et al* [23], where it was assumed that a (coordinated) set of text streams is generated by a mixture of distributions over a vocabulary of words, and the authors proposed the use of expectation maximization to estimate the parameters of the mixture model. Correlated bursts can be interesting in the context of online analytical (OLAP) queries over multidimensional streams too, a topic which has been addressed in [5], [7] and [6], as in a multidimensional data stream, a burst along one dimension can cause a burst along another.

Kleinberg [9] investigated how frequencies of keywords show a bursty pattern in document streams like emails and news articles, and developed a framework in the form of an infinite-state automata to model a stream. He used the exponential distribution to model the "rate" at which messages arrive in a stream, where burts corresponded to higher rates and periods with regular intensity corresponded to lower rates. The model in [9] was generalized enough to reveal the "nested" structure of bursts, which arises because multiple short bursts with high intensity can all be parts of a longer burst with relatively shorter intensity. Kumar *et al* [10] extended the ideas of [9] to discover bursts in the hyperlinking among blogs in Blogspace, which occurs when one author publishes an entry on her blog that increases the traffic to her blog and stimulates activities in other blogs that focus on the same topic at the same time.

Vlachos *et al* [21] addressed the problem of burst detection in the context of search engine queries to detect periodic (e.g., weekly or monthly) trends. Their techniques are based on finding the best Fourier coefficients from a signal. Yuan *et al* [25] worked with high-speed short text streams, which may be relevant for trend detection from tweets or short messages sent from mobile phones. Yuan *et al* [26] proposed an improved counting bloom filter to maintain the frequencies of the items appearing in a data stream that can be helpful for burst detection when multiple item types (e.g., IP addresses) appear in the same stream.

Among other work, Mathioudakis and Koudas [13] developed TwitterMonitor to detect trends from bursty keywords in Twitter streams. Angel *et al* [1] developed Grapevine to detect the trending topics is given time windows and geographic regions and demographic groups. Leskovec *et al* [12] studied the

temporal pattern of generation and spreading of "meme"s on the web. Commercial applications like [19], [16], [14], [17], [18], [20] are being developed to monitor conversations over social media, detect trends from them and use that for promoting brands.

While all these earlier literature have focused on the *detection* of burts, we focus on finding the thresholds that *define* a burst. An algorithm like ours can be used to learn from historical data what choice of thresholds separates a burst from a non-burst, and can be used later in a real monitoring system for the same application to detect bursts, when some other burst-detection algorithm can also be used.

3 Problem Statement

We have a sequence of timestamps of events $S = (e_1, e_2, ...e_n)$ that were generated in some process that we monitored continuously for a length of time. Let t_e be the timepoint at which event e occurs, so the correspoding sequence of timestamps is $S = (t_{e_1}, t_{e_2}, ...t_{e_n})$. There are windows of time where these events show a "bursty" pattern - i.e., a lot of events occur within a short interval of time. Formally, we call a window over S as (k, t)-bursty if at least k events occur in a time interval of at most t. We define the *coverage* for the pair (k, t), denoted as $C_{k,t}$ as follows

$$N_{k,t} = \#\text{Events that are in some } (k,t)\text{-bursty window} \tag{1}$$

$$C_{k,t} = \frac{N_{k,t}}{n} \tag{2}$$

The reason for defining this quantity is that bursty windows can overlap with each other, and this definition of coverage makes dealing with overlapping bursty windows easier. An event which is included in multiple bursty windows contributes only 1 to the numerator $N_{k,t}$ of the definition in Equation 2. The size n of the sequence S in the denomenator helps to understand how significant are the bursts for a given sequence, and also becomes useful when comparing one sequence with another.

Let K_{min} and K_{max} be the minimum and maximum possible values of k, known apriori, and $K = K_{max} - K_{min}$. Similarly, let T_{min} and T_{max} be the minimum and maximum possible values of t, also known apriori, and $T = T_{max} - T_{min}$.

We want to identify a pair of critical thresholds: k on the number of events, and t on the window-length such that many events are part of $(k-1, t)$-bursty windows, but not many events are part of (k, t)-bursty windows. This should make the ratio $\frac{C_{k-1,t}}{C_{k,t}}$ high; hence, to accomplish this, we focus on the following problem:

Problem 1. Given the sequence S, and a user-given parameter $\theta > 1$, find a set $\alpha = \{(k^*, t^*)\}$ such that

$$\alpha \subset [K_{min} + 1, K_{max}] \times [T_{min}, T_{max}]$$

and for any pair $(k^*, t^*) \in \alpha$,

$$\frac{C_{k^*-1,t^*}}{C_{k^*,t^*}} \geq \theta$$

We first focus on simpler, one-dimensional versions of the problem. Assuming we are dealing with a fixed value of the maximum window length t, this becomes

Problem 2. For a fixed t, and a user-given parameter $\theta > 1$, find a subset $K^* \subset [K_{min} + 1, K_{max}]$ such that for any $k^* \in K^*$,

$$\frac{C_{k^*-1,t}}{C_{k^*,t}} \geq \theta$$

Alternatively, if we deal with a fixed value of the threshold k on the number of events, this becomes

Problem 3. For a fixed k, and a user-given parameter $\theta > 1$, find a subset $T^* \in [T_{min}, T_{max}]$ such that for any $t^* \in T^*$,

$$\frac{C_{k-1,t^*}}{C_{k,t^*}} \geq \theta$$

We observed from our experiments that $\frac{C_{k-1,t}}{C_{k,t}}$ remains close to 1 most of the time; however, for very few combinations of k and t, it attains values like 2 or 3 or higher - and these are the combinations we are interested in. Since K and T can be pretty large, searching for the few critical combinations calls for efficient search heuristics.

4 Monotonicity of the Coverage Function

Note that, $C_{k,t}$ is a monotonically non-increasing function of k, and a monotonically non-decreasing function of t. We prove the monotonicity through the following series of observations and lemmas:

Observation 1. *For $k > 1$, an event e with timestamp t_e is not covered by* any *(k, t)-bursty window iff $t_{e'} < t_e - t$ and $t_{e''} > t_e + t$, where e' and e'' are the immediately preceding and succeeding events of e in S.*

Note that, the above statement assumes e has both a preceding and a succeeding event; if it is the first event in the sequence, then only the second condition $(t_{e''} > t_e + t)$ is relevant. Similarly, if it is the last event in the sequence, then only the first condition $(t_{e'} < t_e - t)$ is relevant.

Lemma 1. *For $k > 1$, if an event e is covered by some (k, t_1)-bursty window, then for any $t_2 \geq t_1$, it is covered by some (k, t_2)-bursty window.*

Proof. We prove this by contradiction. Suppose an event e is covered by some (k, t_1)-bursty window, but for some $t_2 \geq t_1$, it is not covered by any (k, t_2)-bursty window. Then, according to Observation 1, $t_{e'} < t_e - t_2$ and $t_{e''} > t_e + t_2$. But $t_2 \geq t_1$, so together, these imply $t_{e'} < t_e - t_1$ and $t_{e''} > t_e + t_1$. As per Observation 1, this implies e is not covered by any (k, t_1)-bursty window, which is a contradiction.

Theorem 1. *$C_{k,t}$ is a monotonically non-decreasing function of t, i.e.,*

$$t_1 \leq t_2 \Rightarrow C_{k,t_1} \leq C_{k,t_2} \tag{3}$$

Proof. Once again, we prove by contradiction. Suppose $t_1 \leq t_2$ but $C_{k,t_1} > C_{k,t_2}$. Let $S_{k,t}$ be the set of events that get reported in some (k, t)-bursty window, i.e., $C_{k,t} = \frac{|S_{k,t}|}{n}$. Consider some event $e \in S_{k,t_1} - S_{k,t_2}$. e got covered by some (k, t_1)-bursty window but not by any (k, t_2)-bursty window, whereas $t_2 \geq t_1$. According to Lemma 1, this is a contradiction.

Next, we prove the monotonicity of $C_{k,t}$ in k.

Lemma 2. *For any k_1, k_2 with $k_2 \geq k_1$, any (k_2, t)-bursty window also gets reported as a (k_1, t)-bursty window by our algorithm.*

Proof. While looking for a (k, t)-bursty window, our algorithm includes as many events within the window as possible (irrespective of k), as long as the difference of timestamps between the first and the last event within the window does not exceed t. This implies if a window is reported as bursty for $k = k_2$, it will be reported as bursty for any $k_1 \leq k_2$.

Theorem 2. *$C_{k,t}$ is monotonically non-increasing in k, i.e.,*

$$k_1 \leq k_2 \Rightarrow C_{k_1,t} \geq C_{k_2,t} \tag{4}$$

Proof. We prove by contradiction. Suppose $k_1 \leq k_2$ but $C_{k_1,t} < C_{k_2,t}$. As in Theorem 1, let $S_{k,t}$ be the set of events that get reported in some (k, t)-bursty window, so that $C_{k,t} = \frac{|S_{k,t}|}{n}$. Now consider some event $e \in S_{k_2,t} - S_{k_1,t}$. e got covered by some (k_2, t)-bursty window but not by any (k_1, t)-bursty window, whereas $k_1 \leq k_2$. But by Lemma 2, this is not possible since any (k_2, t)-bursty window is always reported as a (k_1, t)-bursty window too.

5 A Divide-and-Conquer Algorithm for Finding Critical Thresholds

5.1 The One-Dimensional Problem

We first discuss the solution for Problem 2 - the solution to Problem 3 is somewhat similar. Given the sequence $S = (t_1, t_2, ...t_n)$, and a given pair (k, t), $C_{k,t}$

can be computed on S in a single pass by algorithm 1. A naive approach would be to invoke algorithm 1 with the pairs (k, t) $\forall k \in [K_{min} + 1, K_{max}]$, and check when $C_{k-1,t}/C_{k,t}$ exceeds θ. This would take $O(K)$ calls to algorithm 1. To cut down the number of invocations to algorithm 1, we take a simple divide-and-conquer approach, and exploit the monotonicity of the function $C_{k,t}$ proved in Section 4. We present two variations of the approach - one deterministic and the other randomized. The intuition is as follows:

Intuition: We split the range K of all possible inputs into two sub-intervals. We devise a simple test to decide which of the two sub-intervals this value k^* *may* lie within. The test is based on the observation that if a sub-interval $X = [k_s, k_e]$ contains a k^* such that $C_{k^*-1,t}/C_{k^*,t} \geq \theta$, then $C_{k_s-1,t}/C_{k_e,t} \geq \theta$. The correctness of the observation is proven in Lemma 3. Note that, *the reverse is not necessarily true*, as $C_{k_s-1,t}/C_{k_e,t}$ might exceed θ because there was a gradual change (of factor θ or more) from $k_s - 1$ to k_e. Thus, the test may return a positive result on a sub-interval *even if* there is no such value k^* within that sub-interval where there was an abrupt change. However, we repeat this process iteratively, cutting down the length of the interval in each iteration - the factor by which it is cut down depends on whether the algorithm is deterministic or randomized. Since the length of the interval is significantly reduced in each iteration, the number of iterations taken to reduce the original interval of width K to a point is logarithmic in K. Note that, in the case when there is no such point k^*, the intervals might pass the test for first few iterations (because of the gradual change from k_s to k_e), but then, eventually it will be reduced to some interval for which $C_{k_s-1,t}/C_{k_e,t}$ will fall below θ, and hence it will no longer pass the test.

If we consider the state-space graph/tree for this problem, it is a binary tree with the root node being the original interval $[K_{min} + 1, K_{max}]$ of width K, and for any node in the tree that represents an interval of width w, its two children are sub-intervals of width w' and w'' respectively, where $w' + w'' = w$. The goal node(s) is/are the interval(s) (of length 1) that contain(s) the point k^*. We perform a test, given by Lemma 3, at each node to see if any of its descendants can be a candidate for the goal node. Note that, because of the fact that any parent of the goal node will pass the test, but a node passing the test does not necessarily imply that one of its descendants will be the goal node, we may have to backtrack several times in the search process.

Deterministic vs. Randomized Divide-and-Conquer: In the deterministic divide-and-conquer, we always split an interval of width w into two equal intervals of length $w/2$ each; whereas in the randomized divide-and-conquer, we pick a point $p \in (0, 1)$ uniformly at random, and the sub-intervals created are of length $p \cdot w$ and $(1 - p) \cdot w$ respectively. If both the sub-intervals pass the test, then for the deterministic divide-and-conquer, we probe into the intervals serially; whereas for the randomized one, we first process that sub-interval which is *smaller* between the two. The reasons for probing the smaller sub-interval first are the following

1. If it contains a point k^*, then it can be found in fewer iterations.
2. If it does not contain any point k^*, and passed the test of Lemma 3 falsely, then, the algorithm would backtrack after few iterations.

Lemma 3. *For a fixed t, if a sub-interval $X = [k_s, k_e]$ contains a point k^* such that $C_{k^*-1,t}/C_{k^*,t} \geq \theta$, then $C_{k_s-1,t}/C_{k_e,t} \geq \theta$.*

Proof. Note that

$$k_s - 1 \leq k^* - 1 < k^* \leq k_e$$

By the monotonicity property of inequality 4, this implies

$$C_{k_s-1,t} \geq C_{k^*-1,t} \geq C_{k^*,t} \geq C_{k_e,t}$$

Combining $C_{k_s-1,t} \geq C_{k^*-1,t}$ and $C_{k_e,t} \leq C_{k^*,t}$ gives

$$\frac{C_{k_s-1,t}}{C_{k_e,t}} \geq \frac{C_{k^*-1,t}}{C_{k^*,t}} \geq \theta$$

The following lemma follows immediately from Lemma 3.

Lemma 4. *For a fixed t, if for a sub-interval $X = [k_s, k_e]$, $C_{k_s-1,t}/C_{k_e,t} < \theta$, then it does not contain any point k such that $C_{k-1,t}/C_{k,t} \geq \theta$.*

The search for t^* in Problem 3 proceeds similar to the search for k^* as we explained above, the difference being that the test on the sub-intervals is performed using the following lemma.

Lemma 5. *For a fixed k, if a sub-interval $X = [t_s, t_e]$ contains a point t^* such that $C_{k-1,t^*}/C_{k,t^*} \geq \theta$, then $C_{k-1,t_e}/C_{k,t_s} \geq \theta$.*

Proof. By the monotonicity property of Theorem 1, $t_e \geq t^* \Rightarrow C_{k-1,t_e} \geq C_{k-1,t^*}$ and $t_s \leq t^* \Rightarrow C_{k,t_s} \leq C_{k,t^*}$. Combining these two, we get,

$$\frac{C_{k-1,t_e}}{C_{k,t_s}} \geq \frac{C_{k-1,t^*}}{C_{k,t^*}} \geq \theta$$

5.2 The Two-Dimensional Problem

We now advance to the original and more general problem in two dimensions, i.e., Problem 1. Our algorithm for 2D is an extension of the algorithm for the 1D problem discussed in Section 5.1 in the sense that it progressively divides the 2D range of all possible values of k and t, i.e., $[K_{min}+1, K_{max}] \times [T_{min}, T_{max}]$, into four sub-ranges/rectangles. For the 2D problem, the pair(s) (k^*, t^*) for which $C_{k^*-1,t^*}/C_{k^*,t^*}$ exceeds θ will come from one or a few of these four sub-ranges. We devise a test similar to the one in Section 5.1 to identify which of the four sub-ranges *may* include the pair (k^*, t^*); and then probe into that sub-range in the next iteration, cutting down its size again, and so on. If the range of possible values for k and t are of unequal length, i.e., if $K \neq T$, then the length

of the range would reduce to unity for the smaller one, and the rest of the search becomes a 1D search on the other dimension, like the ones in Section 5.1.

The test for identifying the correct sub-range in our 2D algorithm is based on the observation in the following lemma.

Lemma 6. *If a sub-range* $X = [k_s, k_e] \times [t_s, t_e]$ *contains a point* (k^*, t^*) *such that* $C_{k^*-1,t}/C_{k^*,t} \geq \theta$*, then* $C_{k_s-1,t_e}/C_{k_e,t_s} \geq \theta$*.*

Proof. We know

$$k_s - 1 \leq k^* - 1 < k^* \leq k_e$$

By the monotonicity of inequality 4, this implies

$$C_{k_s-1,t^*} \geq C_{k^*-1,t^*} \geq C_{k^*,t^*} \geq C_{k_e,t^*}$$

Combining $C_{k_s-1,t^*} \geq C_{k^*-1,t^*}$ and $C_{k_e,t^*} \leq C_{k^*,t^*}$ gives

$$\frac{C_{k_s-1,t^*}}{C_{k_e,t^*}} \geq \frac{C_{k^*-1,t^*}}{C_{k^*,t^*}} \geq \theta \tag{5}$$

Also, since $t^* \in [t_s, t_e]$, by the monotonicity of inequality 3, these imply $C_{k_s-1,t_e} \geq C_{k_s-1,t^*}$ and $C_{k_e,t_s} \leq C_{k_e,t^*}$. Combining these two,

$$\frac{C_{k_s-1,t_e}}{C_{k_e,t_s}} \geq \frac{C_{k_s-1,t^*}}{C_{k_e,t^*}} \tag{6}$$

Combining inequality 5 with inequality 6, we get

$$\frac{C_{k_s-1,t_e}}{C_{k_e,t_s}} \geq \theta$$

The following lemma follows immediately from Lemma 6.

Lemma 7. *If for a 2D sub-range* $X = [k_s, k_e] \times [t_s, t_e]$*,* $C_{k_s-1,t_e}/C_{k_e,t_s} < \theta$*, then it does not contain any point* (k, t) *such that* $C_{k-1,t}/C_{k,t} \geq \theta$*.*

Like the algorithms in Section 5.1 for the 1D problems, here also we have two variants: one deterministic and the other randomized. If more than one of the four sub-intervals/rectangles pass the test in Lemma 6, then for the deterministic algorithm, we probe into them serially, and for the randomized one, we probe into the rectangles in increasing orders of their areas.

Algorithm 1 computes the coverage $C_{k,t}$, given a sequence of timestamps $(S = (t_{e_1}, t_{e_2}, \ldots, t_{e_n})$, a lower bound k on the number of events and an upper bound t on the window length. As we have alerady pointed out, even if a single timestamp t_{e_i} is included in multiple (k, t)-bursty windows, its contribution to the numerator of the definition of $C_{k,t}$, as in Equation 2, is only 1. Hence, we maintain a bitmap $(b_1, b_2, \ldots, b_n)$ of length n, one bit for each timestamp in S. We slide a window over S, marking the starting and ending points of the sliding window by s and f respectively all the time. Once s is fixed, f is incremented (lines 4-9 and 18-23) until one of the following two happens:

1. The timestamp of the event indexed by f, t_{e_f}, is no longer within the boundary of the current window ($t_{e_f} > t_{e_s} + t$)
2. All the timepoints are exhausated ($f > n$)

Once all the timepoints in a window are "picked up", we check (in lines 12 and 28) if the number of events in the current window $[s, f]$, i.e., $f - s + 1$, exceeds the threshold k. If it does, then all the bits in the sub-sequence $(b_s, \ldots, b_f)$ of the bitmap are set to 1 (lines 13 and 29) to indicate that the timepoints indexed by these bits are part of some bursty window.

Optimization: We designed Algorithm 1 in such a way that it does not consider windows where one window is completely "engulfed" within another, although it allows partial overlap between windows. For example, suppose we have the following sequence of timestamps: $(1, 3, 7, 14)$, and $k = 2$ and $t = 10$. As per the definition of (k, t)-bursty window, the subsequence $(1, 3, 7)$ ($[s = 1, f = 3]$) qualifies as a $(2, 10)$-bursty window, and so does the subsequence $(3, 7)$ ($[s = 2, f = 3]$), but we do not consider a subsequence like $(3, 7)$ since it is completely covered by the subsequence $(1, 3, 7)$. So far as our definition of $C_{k,t}$ in Equation 2 is concerned, considering $[s = 2, f = 3]$ as a separate window would not have made a difference, because in the bitmap (b_1, b_2, b_3, b_4), $[s = 2, f = 3]$ would have set the bits b_2 and b_3 which were already set by the window $[s = 1, f = 3]$, but it is an unnecessary operation, and moreover, if someone asks for the number of (k, t)-bursty windows , with the constraint that partial overlaps are allowed but complete overlaps are not, then this optimization can return the correct answer. Such a constraint might be motivated by the fact that we are actually interested in the "temporal clustering" pattern in an event sequence, and clusters that are engulfed within larger clusters do not interest us.

Algorithm 2 perfoms a one-dimensional search over the interval $[k_s, k_e]$, for a fixed value of the maximum windowlength t. This becomes useful for solving Problem 2, and is called as a subroutine during the two-dimensional search once the range of t-values reduces to a single point. Similarly, algorithm 3 perfoms a one-dimensional search over the interval $[t_s, t_e]$, for a fixed value of the minimum number of events k. This becomes useful for solving Problem 3, and is called as a subroutine during the two-dimensional search once the range of k-values reduces to a single point. In both algorithms 2 and 3, we reduce the number of calls to Algorithm 1 by *memoization*: whenever we need to compute $C_{k,t}$, we first check if it has already been computed, in which case, it should exist in a hashtable with $(k|t)$ being the key; otherwise, we compute it by invoking `ComputeCoverage`(S, t, k).

Note that, in lines 24 and 28 of Algorithm 2 (3), we are checking if the coverage evaluated at the start (end) point of the respective interval is greater than zero. Not only it helps to avoid division-by-zero error, but also works as a heuristic for pruning intervals in which all coverage values are zero and hence are not worth exploring further. Because of the monotonicity property in Theorem 2 (1), if the coverage at the start (end) point is 0, the coverage at the end (start) point must also be 0.

Algorithm 4 performs the search over the 2D interval $[k_s, k_e] \times [t_s, t_e]$ to solve Problem 1. Since the number of rectangles is only four, we used insertion sort while sorting them by their areas, which takes $O(1)$ time because the input size is constant.

The deterministic divide-and-conquer algorithms are very similar to their randomized counterparts (algorithms 2, 3 and 4), with the following differences:

- In line 21 of algorithms 2 and 3 and lines 24 and 25 of algorithm 4, k_q and t_q are midpoints ($\lfloor \frac{k_s+k_e}{2} \rfloor$ and $\lfloor \frac{t_s+t_e}{2} \rfloor$) of the intervals $[k_s, k_e]$ and $[t_s, t_e]$ respectively. So, in algorithm 2, if the interval $[k_s, k_e]$ is of even length, then the intervals $[k_s^{big}, k_e^{big}]$ and $[k_s^{small}, k_e^{small}]$ would be of equal length, and if the length of $[k_s, k_e]$ is odd, then the length of the bigger sub-interval would be one more than that of the length of the smaller, but that difference is trivial (same applies for $[t_s, t_e]$ in algorithm 3).
- In algorithm 4, we can do away with the sorting step of line 27, since the four rectangles would be of equal length when both $[k_s, k_e]$ and $[t_s, t_e]$ are of even length, and the lengths would differ only slightly when at least one of these intervals has an odd length.

6 Complexity Analysis

Let $C(K)$ be the number of calls made to Algorithm 1 for solving Problem 2. We compute $C(K)$ assuming that in Algorithm 2 and its deterministic equivalent, only one of the two sub-intervals passes the test of Lemma 3 between lines 24-30, so we never probe into the other interval.

Theorem 3. *For the deterministic counterpart of Algorithm 2,* $C(K) = O(\log K)$.

Proof. The input interval $[k_s, k_e]$ is split into two *equal* halves whenever the interval has 3 or more points. There is an $O(1)$ number of calls to Algorithm 1 at each step, and we only probe into one of the two intervals. So, the following recurrence relation holds true for $C(K)$:

$$C(K) = C(K/2) + O(1)$$

We solve this by the substitution method [4], guessing the solution is $C(K) = O(\log_2 K)$. We need to prove $C(K) \leq c \cdot \log_2 K$ for some constant $c > 0$. We start by assuming that this holds for all $l < K$, and $O(1) = a$ for some $a > 0$, so that substituting in the recurrence yields

$$\begin{aligned} C(K) &\leq c \cdot \log_2 (K/2) + a \\ &= c \cdot \log_2 K - c + a \\ &= c \cdot \log_2 K - (a - c) \\ &\leq c \cdot \log_2 K \text{ if } a \geq c \end{aligned}$$

Algorithm 1. ComputeCoverage $(S = (t_{e_1}, t_{e_2}, \ldots, t_{e_n}), t, k)$

output: $C_{k,t}$: the fraction of events that are in some (k,t)-bursty window
$n \leftarrow |S|$;
initialize a bitmap $(b_1, b_2, \ldots, b_n)$ to all zeros;
/* sliding window is $[s, \ldots, f], s \in \{1, \ldots, n\}$, $f \in \{1, \ldots, n\}$ */
$s \leftarrow 0, f \leftarrow 0$;
// Note: t_{e_i} is the i^{th} timepoint in S.
while $t_{e_f} < (t_{e_s} + t) \wedge f < n$ **do**
  $f \leftarrow f + 1$;
end
if $t_{e_f} > t_{e_s} + t$ **then**
  $f \leftarrow f - 1$;
end
while $f < n$ **do**
  $n_w = f - s + 1$;
  if $n_w \geq k$ **then**
    set the bits $(b_s, \ldots, b_f)$ to 1;
  end
  /* Move the window, storing pointers to the previous window */
  $s_p = s, f_p = f$;
  while $s \geq s_p \wedge f \leq f_p$ **do**
    $s \leftarrow s + 1$;
    while $t_{e_f} < (t_{e_s} + t) \wedge f < n$ **do**
      $f \leftarrow f + 1$;
    end
    if $t_{e_f} > t_{e_s} + t$ **then**
      $f \leftarrow f - 1$;
    end
  end
end
/* If the last point is within the last window, it will be counted.
   Otherwise, it is an isolated point and hence not interesting. */
if $f = n$ **then**
  $n_w \leftarrow f - s + 1$;
  if $n_w \geq k$ **then**
    set the bits $(b_s, \ldots, b_f)$ to 1;
  end
end
$C_{k,t} \leftarrow \sum_{j=1}^{n} b_j / n$;
return $C_{k,t}$

Algorithm 2. RandomSearchk* $(S = (t_{e_1}, t_{e_2}, \ldots, t_{e_n}), t, k_s, k_e, \theta)$

```
   /* C_{k,t} is always computed by invoking ComputeCoverage(S,t,k)        */
 1 if k_s = k_e then
     /* Interval has reduced to a single point                            */
 2   if C_{k_s,t} > 0 then
 3     r ← C_{k_s-1,t} / C_{k_s,t};
 4     if r ≥ θ then
 5       output (k_s, t) as a critical threshold pair;
 6     end
 7   end
 8   return;
 9
10 else if k_e − k_s = 1 then
     /* Interval of length 2. Simply check at two points individually
        */
11   foreach i ∈ [k_s, k_e] do
12     if C_{i,t} > 0 then
13       r ← C_{i-1,t} / C_{i,t};
14       if r ≥ θ then
15         output (i, t) as a critical threshold pair;
16       end
17     end
18   end
19   return;
20 else
     /* Interval of width 3 or more. The function U([a,b]) returns a
        number uniformly at random in [a,b]                              */
21   k_q ← U([k_s, k_e − 1]);
22   between [k_s, k_q] and [k_q + 1, k_e], let [k_s^big, k_e^big] be the bigger window and
     [k_s^small, k_e^small] be the smaller;
23   r_small ← C_{k_s^small − 1,t} / C_{k_e^small,t};
     /* Note: r might exceed θ because of a divison-by-zero error. In
        case that happens, we will explore the interval only if
        C_{k_s^small − 1,t} ≥ 0, because C_{k_s^small − 1,t} = 0 implies C_{k_e^small,t} = 0 by
        the monotonicity, and it is not worth exploring [k_s^small, k_e^small].
        */
24   if (r_small ≥ θ) ∧ (C_{k_s^small − 1,t} ≥ 0) then
25     RandomSearchk*(S, t, k_s^small, k_e^small, θ);
26   end
27   r_big ← C_{k_s^big − 1,t} / C_{k_e^big,t};
28   if (r_big ≥ θ) ∧ (C_{k_s^big − 1,t} ≥ 0) then
29     RandomSearchk*(S, t, k_s^big, k_e^big, θ);
30   end
31 end
```

Algorithm 3. RandomSearcht* $(S = (t_{e_1}, t_{e_2}, \ldots, t_{e_n}), k, t_s, t_e, \theta)$

```
   /* C_{k,t} is always computed by invoking ComputeCoverage(S, t, k)        */
 1 if t_s = t_e then
       /* Interval has reduced to a single point                            */
 2     if C_{k,t_s} > 0 then
 3         r ← C_{k-1,t_s} / C_{k,t_s};
 4         if r ≥ θ then
 5             output (k, t_s) as a critical threshold pair;
 6         end
 7     end
 8     return;
 9
10 else if t_e − t_s = 1 then
       /* Interval of length 2. Simply check at two points individually
          */
11     foreach i ∈ [t_s, t_e] do
12         if C_{k,i} > 0 then
13             r ← C_{k-1,i} / C_{k,i};
14             if r ≥ θ then
15                 output (k, i) as a critical threshold pair;
16             end
17         end
18     end
19     return;
20 else
       /* Interval of width 3 or more.                                      */
21     t_q ← U([t_s, t_e − 1]);
22     between [t_s, t_q] and [t_q + 1, t_e], let [t_s^big, t_e^big] be the bigger window and
       [t_s^small, t_e^small] be the smaller;
23     r_small ← C_{k-1,t_e^small} / C_{k,t_s^small};
       /* Note: r might exceed θ because of a divison-by-zero error         */
24     if (r_small ≥ θ) ∧ (C_{k-1,t_e^small} ≥ 0) then
25         RandomSearcht*(S, k, t_s^small, t_e^small, θ);
26     end
27     r_big ← C_{k-1,t_e^big} / C_{k,t_s^big};
28     if (r_big ≥ θ) ∧ (C_{k-1,t_e^big} ≥ 0) then
29         RandomSearcht*(S, k, t_s^big, t_e^big, θ);
30     end
31 end
```

Algorithm 4. RandomSearch2D $(S = (t_{e_1}, t_{e_2}, \ldots, t_{e_n}), k_s, k_e, t_s, t_e, \theta)$

```
   /* A rectangle is defined as a four-tuple (t_l, t_h, k_l, k_h), i.e., the set
      of all points in the 2D range [t_l, t_h] × [k_l, k_h]. Its area is
      (t_h − t_l + 1)·(k_h − k_l + 1).                                    */
 1 if (t_s = t_e) ∧ (k_s = k_e) then
       /* Rectangle has reduced to a single point                        */
 2     if C_{k_e,t_s} > 0 then
 3         r ← C_{k_s−1,t_e} / C_{k_e,t_s};
 4         if r ≥ θ then
 5             output (k_s, t_s) as a critical threshold pair;
 6         end
 7     end
 8     return;
 9 else if t_s = t_e then
       /* t fixed, rectangle reduced to a single line [k_s, k_e]         */
10     RandomSearchk*(S, t_s, k_s, k_e, θ)
11 else if k_s = k_e then
       /* k fixed, rectangle reduced to a single line [t_s, t_e]         */
12     RandomSearcht*(S, k, t_s, t_e, θ)
13 else if (t_e − t_s = 1) ∧ (k_e − k_s = 1) then
       /* 2 × 2 rectangle. Check four points individually                */
14     foreach (i, j) ∈ [k_s, k_e] × [t_s, t_e] do
15         if C_{i,j} > 0 then
16             r ← C_{i−1,j} / C_{i,j};
17             if r ≥ θ then
18                 output (i, j) as a critical threshold pair;
19             end
20         end
21     end
22     return;
23 else
       /* Split the rectangle randomly into four quadrants               */
24     k_q ← U([k_s, k_e − 1]);
25     t_q ← U([t_s, t_e − 1]);
26     let R be an array of rectangles with R[1] = (t_s, t_q, k_s, k_q),
       R[2] = (t_q + 1, t_e, k_s, k_q), R[3] = (t_s, t_q, k_q + 1, k_e) and
       R[4] = (t_q + 1, t_e, k_q + 1, k_e);
27     sort R in increasing order of areas of the rectangles;
28     for p = 1 to 4 do
29         let (t_l, t_h, k_l, k_h) be the 4-tuple for rectangle R[p];
30         r ← C_{k_l−1,t_h} / C_{k_h,t_l};
31         if (r ≥ θ) ∧ (C_{k_l−1,t_h} ≥ 0) then
32             RandomSearch2D(S, k_l, k_h, t_l, t_h, θ)
33         end
34     end
35 end
```

Let $C(T)$ be the number of calls made to Algorithm 1 for solving Problem 3. Analogous to Theorem 3, we can claim the following:

Theorem 4. *For the deterministic counterpart of Algorithm 3,* $C(T) = O(\log T)$.

For the two-dimensional version, let $C(K, T)$ be the number of calls made to Algorithm 1 for solving Problem 1.

Theorem 5. *For the deterministic counterpart of Algorithm 4,* $C(K, T) = O(\log K \log T)$.

Proof. The input interval $[k_s, k_e] \times [t_s, t_e]$ is split into four *equal* halves whenever the interval has 3 or more points. There is an $O(1)$ number of calls to Algorithm 1 at each step, and we only probe into one of the four intervals. So, the following recurrence relation holds true for $C(K, T)$:

$$C(K, T) = C\left(\frac{K}{2}, \frac{T}{2}\right) + O(1)$$

We again solve this by the substitution method, like Theorem 3. For brevity, we omit the full proof here, and it can be found in [11].

Theorem 6. *For Algorithm 2, the expected complexity in the worst case is* $E[C(K)] = O(\ln K)$.

Proof. Our analysis is similar to the analysis of the RANDOMIZED-SELECT algorithm in [4], except some differences that are specific to our problem. We do a worst-case analysis in the sense that we assume once an interval is split into two (potentially unequal) sub-intervals, always the *bigger* of the two passes the test of Lemma 3. For the sake of brevity, we do not present the proof here, and it can be found in [11].

Analogous to theorem 6, we can claim the following for the complexity $C(T)$ of algorithm 3 for problem 3.

Theorem 7. *For Algorithm 3, the expected complexity in the worst case is* $E[C(T)] = O(\ln T)$.

Once again, for the two-dimensional version, let $C(K, T)$ be the number of calls made to Algorithm 1 from algorithm 4 for solving Problem 1. Before analyzing the expected complexity of Algorithm 4, we prove the following lemma which will be useful in Theorem 8.

Lemma 8. *In Algorithm 4, for* $i \in \{1, 2, 3, 4\}$*, rectangle* $R[i]$ *is the largest of the four if it is larger than the two which share the same endpoints with* $R[i]$ *on the* t*-axis and the* k*-axis.*

Proof. If we consider $R[1] = (t_s, t_q, k_s, k_q)$, then it has the same endpoints on the k-axis as $R[2]$ since $R[2] = (t_q + 1, t_e, k_s, k_q)$, and $R[1]$ has the same endpoints

on the t-axis as $R[3]$ since $R[3] = (t_s, t_q, k_q + 1, k_e)$. What the statement claims is as follows: if $R[1]$ is larger than both $R[2]$ and $R[3]$, then it is the largest of all four, which implies it is larger than $R[4]$. Similarly, if $R[2]$ is larger than both $R[1]$ and $R[4]$, then it is the largest of all four, which implies it is larger than $R[3]$, and so on. For lack of space, we do not present the proof here, and it can be found in [11].

Theorem 8. *For Algorithm 4, the expected complexity in the worst case is* $E[C(K,T)] = O(\ln K \ln T)$.

Proof. For lack of space, we do not present the proof here, and it can be found in [11]. We do a worst-case analysis in the sense that we assume once a rectangle is split into four (potentially unequal) sub-intervals, always the *biggest* of the four passes the test of Lemma 6.

7 Evaluation

Dataset: We implemented both heuristics and compared them with a naive algorithm (which also gave us the ground truth to begin with), by running all 3 on a set of logs collected during the operation of large complex equipment sold by Siemens Healthcare. We chose 32 different types of events that occurred on these equipment, each event identified by a unique code. The event codes had upto 300,000 distinct time points. Each event code occurred on multiple (upto 291 different) machines, so we had to take care of some additional details while computing $C_{k,t}$ and finding the critical thresholds, which are as follows:

We defined a stream $S_e^{(m)} = (t_{e_1}^{(m)}, t_{e_2}^{(m)}, ...t_{e_n}^{(m)})$ as the sequence of (sorted) time points of occurrences of an event code e on a single machine m. While computing the coverage for event code e, which occurred on the set M of machines, we modify the definition of Section 3 by the following formulae:

$$N_{k,t}^{(e)} = \sum_{m \in M} \#\text{Events in } S_e^{(m)} \text{ covered in some } (k,t)\text{-bursty window} \quad (7)$$

$$n^{(e)} = \sum_{m \in M} \#\text{Occurrences of } e \text{ on } m \quad (8)$$

$$C_{k,t}^{(e)} = \frac{N_{k,t}^{(e)}}{n^{(e)}} \quad (9)$$

To compute $C_{k,t}^{(e)}$ as in equation 9, we invoked algorithm 1 over the stream $S_e^{(m)}$ for each machine $m \in M$ and found the number of occurrences of e in $S_e^{(m)}$ that are covered in some (k,t)-bursty window.

Experiments: We implemented our heuristics in Java on a Windows desktop machine with 2 GB RAM. We set $T_{min} = 1$ minute, $T_{max} = 100$ minutes, $K_{min} = 2$ and $K_{max} = 100$ for all the event codes. We made the following simple optimizations to the naive algorithm:

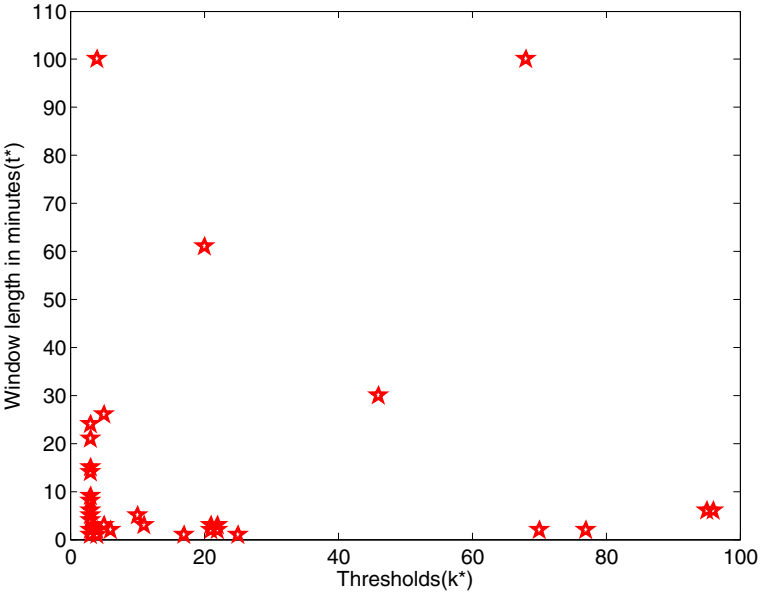


Fig. 1. The critical threshold pairs (k^*, t^*) for all the 32 event codes

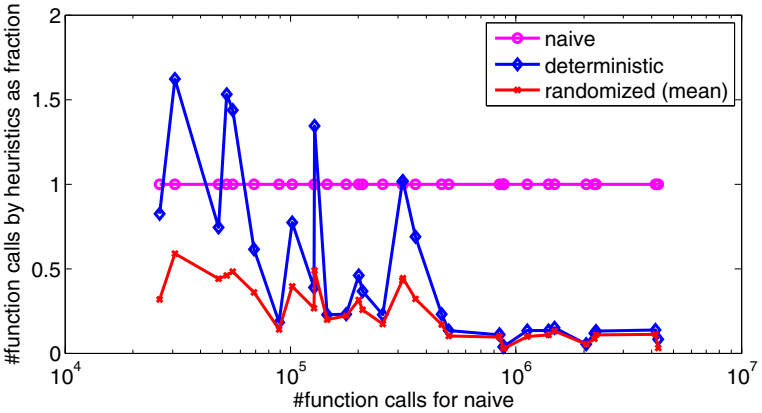


Fig. 2. The improvements by $\mathcal{N}_R$ and $\mathcal{N}_D$ over $\mathcal{N}_N$. On the X-axis we have $\mathcal{N}_N$ (the event codes are sorted by $\mathcal{N}_N$), on the Y-axis we have $\mathcal{I}_R$ and $\mathcal{I}_D$. Note that the savings are in general more for larger valus of $\mathcal{N}_N$, and the randomized heuristic consistently outperforms the deterministic one. The X-axis is logarithmic.

1. For each event code e, $C^{(e)}_{k,t}$ for each combination of k and t is computed at most once, stored in a hashtable with the key being $e|k|t$, a concatenation of e, k and t. The stored value of $C^{(e)}_{k,t}$ is used in evaluating both $C^{(e)}_{k-1,t}/C^{(e)}_{k,t}$ and $C^{(e)}_{k,t}/C^{(e)}_{k+1,t}$. We followed the same practice for our heuristics, too.

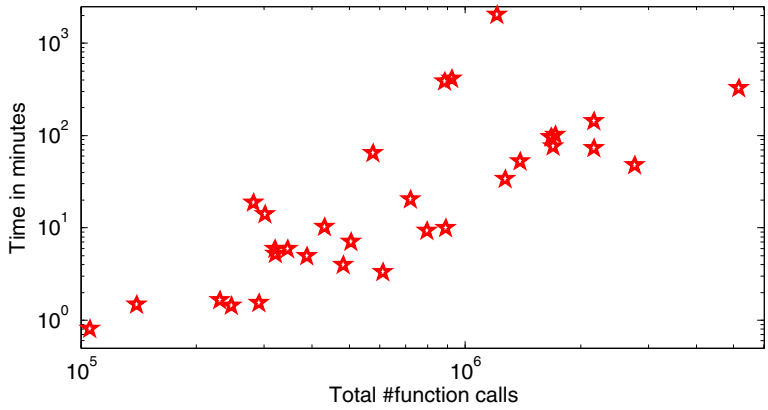


Fig. 3. X-axis shows total number of function calls for 1 run of deterministic and 10 runs of randomized heuristic for each eventcode, i.e., $10{\cdot}\mathcal{N}_R + \mathcal{N}_D$, and Y-axis shows total time (in minutes) for all these runs. Both axes are logarithmic.

2. Once $C^{(e)}_{k,t}$ reaches 0 for some k, $C^{(e)}_{k,t}$ is not computed for any larger value of k since it is known that they will be 0 because of the monotonicity property discussed in Section 4.

Ground Truth: The ratios $C^{(e)}_{k-1,t}/C^{(e)}_{k,t}$ for all possible combinations of k and t, obtained from the naive algorithm, formed our ground truth. While running our heuristics for each e, we picked the highest value of $C^{(e)}_{k-1,t}/C^{(e)}_{k,t}$, and set θ to that value. We had the following observations from running the naive algorithm:

- The maximum value of $C^{(e)}_{k-1,t}/C^{(e)}_{k,t}$ (and hence θ) for the different event codes varied from 1.55 to 55, with 23 out of the 32 values being within 10.
- For the remaining 9 event codes with θ above 10, k^* was mostly 3, which means that "pairs" of occurrences of these event codes within their respective streams were common, but clusters of any larger size were not, and hence qualified as bursts. The (k^*, t^*) pairs for the 32 event codes are plotted in Figure 1.
- While most event codes reported a single pair (k^*, t^*), we found two event codes, each of which reported a single value of k^* with t^* assuming values from a contiguous range. For example, for one of them, we found $\theta = 9.25$, and (k^*, t^*) taking all values from the set $\{(4, 88), (4, 89), (4, 90), \ldots, (4, 100)\}$. This is reasonable since by definition, for $t_1 \leq t_2$, a (k, t_1)-bursty window also qualifies as a (k, t_2)-bursty window, so in this case, the (4,88)-bursty windows, the (4,89)-bursty windows,...,the (4,99)-bursty windows are all actually (4,100)-bursty windows. Moreover, clusters of 3 events are almost 10 times as likely as clusters of 4 or more events, hence clusters of 4 (or more) events in about 100 minutes should qualify as burst.

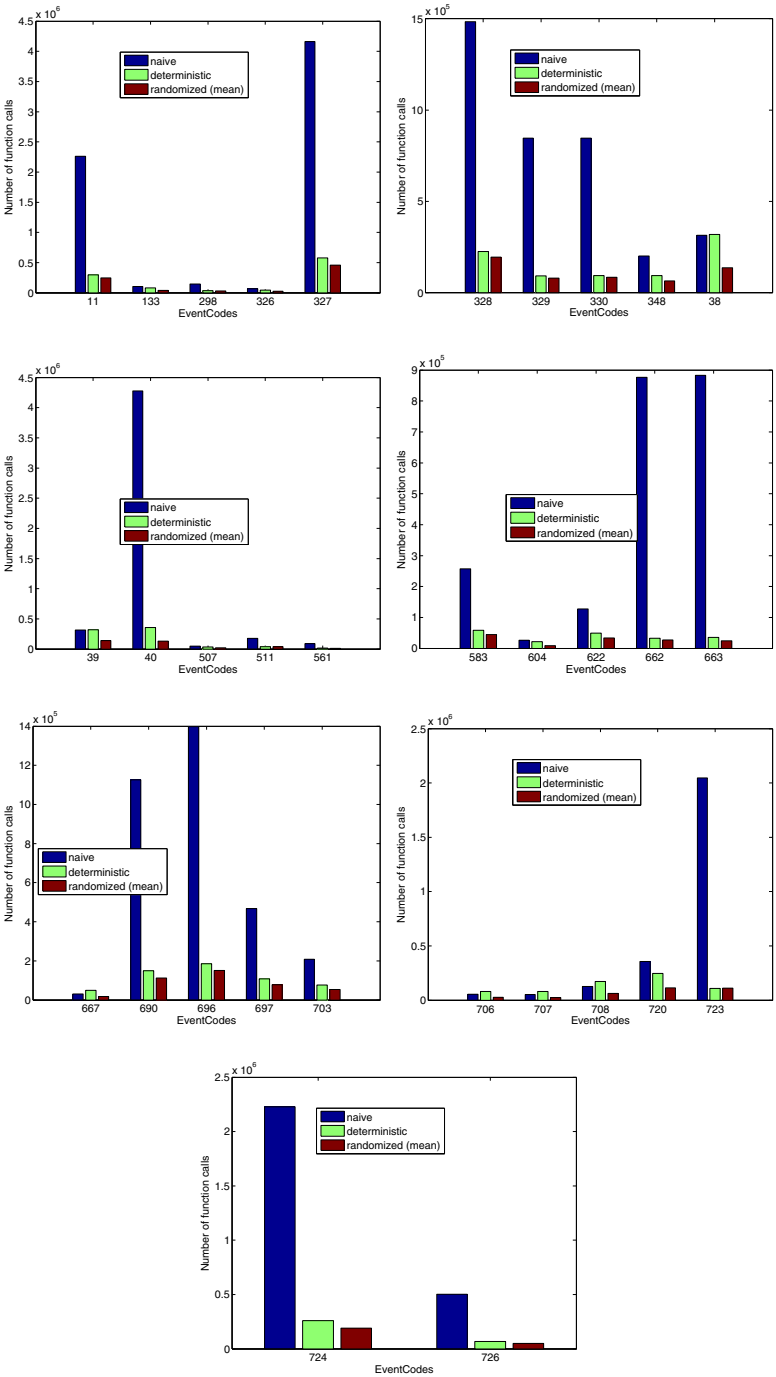


Fig. 4. $\mathcal{N}_N$, $\mathcal{N}_D$ and $\mathcal{N}_R$ for all eventcodes

We ran the heuristics for an event code only if θ set in this way was at least 1.5. For each event code, we ran the randomized heuristic 10 times, each time with a different seed for the pseudo-random number generator, noted the number of calls to Algorithm 1 for each, and calculated the mean ($\mathcal{N}_R$), the standard deviation (σ) and the coefficient of variation (CV = $\frac{\sigma}{\mathcal{N}_R}$).

Our observations about the number of calls to Algorithm 1 by the naive algorithm ($\mathcal{N}_N$), the deterministic heuristic ($\mathcal{N}_D$) and the mean for the randomized one ($\mathcal{N}_R$) are as follows:

1. For all but one event code, we found $\mathcal{N}_R < \mathcal{N}_D$. The probable reason is, after partitioning the original interval, when the four intervals are unequal, if the smaller interval does not contain (k^*, t^*), then it has less chance of falsely passing the test of Lemma 6 in line 31 of Algorithm 4. Also, as we discussed in Subsection 5.1, even if the smaller interval passes the test falsely, we are more likely to backtrack from it earlier because its sub-intervals have even less chance of falsely passing the test, and so on. Even for the single event code where $\mathcal{N}_D$ beats $\mathcal{N}_R$, the latter makes only 0.4% more function calls than the former. Depending on the event code, $\mathcal{N}_R$ is 4% to 70% less than $\mathcal{N}_D$.
2. We define the "improvement" by the randomized ($\mathcal{I}_R$) and the deterministic ($\mathcal{I}_D$) heuristics as $\mathcal{I}_R = \frac{\mathcal{N}_R}{\mathcal{N}_N}$ and $\mathcal{I}_D = \frac{\mathcal{N}_D}{\mathcal{N}_N}$, which are both plotted in Figure 2. The improvements are more when $\mathcal{N}_N$ is close to or more than a million - the improvement $\mathcal{I}$ in those cases is then 3-11%. In other cases, it is mostly in the range of 40-50%. Hence, the curves for both $\mathcal{I}_R$ and $\mathcal{I}_D$ in Figure 2 show a roughly decreasing pattern as we go from left to right.
3. For 28 out of 32 event codes, the CV ($\frac{\sigma}{\mathcal{N}_R}$) for the randomized heuristic is less than 0.1, which implies a quite stable performance across runs, and hence we would not need multiple runs (and obtain an average) in a real setting, and hence would not ruin the savings obtained by exploiting the monotonicity. In fact, for 22 out of these 28, the CV is less than 0.05. The maximum CV for any event code is 0.18 only.
4. We show the time taken (in minutes) for $10{\cdot}\mathcal{N}_R + \mathcal{N}_D$ function calls for each eventcode in Figure 3. The time taken increased as the number of function calls increased, which is quite expected. For 16 out of 32 eventcodes, the time taken for $10{\cdot}\mathcal{N}_R + \mathcal{N}_D$ function calls was less than 15 minutes, and for 27 out of 32 eventcodes, this time was less than 2 hours. As an example, an eventcode which took about 19 minutes for $10{\cdot}\mathcal{N}_R + \mathcal{N}_D$ function calls had $\mathcal{N}_N = 883,080$, $\mathcal{N}_D = 35,100$ and $\mathcal{N}_R = 24,655$, so the 19 minutes time plotted in Figure 3 is for $10{\cdot}24655 + 35100 = 281650$ function calls, which is less than 32% of $\mathcal{N}_N$.
5. We show the actual values of $\mathcal{N}_N$, $\mathcal{N}_R$ and $\mathcal{N}_D$ for all the 32 eventcodes in Figure 4. The lengths of the bars in these plots also corroborate the fact that the "improvement" is more for the eventcodes for which $\mathcal{N}_N$ is large.

8 Conclusion

We studied the problem of defining critical thresholds on the number of events and the window-lengths such that combinations of these thresholds can separate a burst from a non-burst. Although there is a significant body of literature that addresses the burst detection problem in different application domains, and some previous authors acknowledged that not having apriori knowledge of a good threshold makes the detection problem more difficult, to our knowledge, this is the first work that addresses the threshold definition problem. For a threshold pair, we conceptualize its "coverage" of the event sequence, and exploit its monotonicity to design efficient deterministic and randomized heuristics that find the critical thresholds. We plan to investigate if the number of passes made over the event sequence can be reduced to $o(\log K \log T)$, perhaps by using the advanced data structures designed for the burst detection algorithms [29,28]. The sooner we find the critical thresholds, the sooner we can deploy them for burst detection in a real application.

References

1. Angel, A., Koudas, N., Sarkas, N., Srivastava, D.: What's on the grapevine? In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1047–1050 (2009)
2. Barford, P., Crovella, M.: Generating representative web workloads for network and server performance evaluation. In: SIGMETRICS, pp. 151–160 (1998)
3. Beran, J.: Statistics for Long-Memory Processes. Chapman & Hall, New York (1994)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, McGraw-Hill Book Company (2009)
5. Cuzzocrea, A.: CAMS: OLAPing Multidimensional Data Streams Efficiently. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 48–62. Springer, Heidelberg (2009)
6. Cuzzocrea, A.: Retrieving Accurate Estimates to OLAP Queries over Uncertain and Imprecise Multidimensional Data Streams. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 575–576. Springer, Heidelberg (2011)
7. Cuzzocrea, A., Chakravarthy, S.: Event-based lossy compression for effective and efficient OLAP over data streams. Data and Knowledge Engineering 69(7), 678–708 (2010)
8. Garrett, M.W., Willinger, W.: Analysis, modeling and generation of self-similar vbr video traffic. In: SIGCOMM, pp. 269–280 (1994)
9. Kleinberg, J.M.: Bursty and hierarchical structure in streams. Data Mining and Knowledge Discovery 7(4), 373–397 (2003)
10. Kumar, R., Novak, J., Raghavan, P., Tomkins, A.: On the bursty evolution of blogspace. In: WWW, pp. 568–576 (2003)
11. Lahiri, B., Akrotirianakis, I., Moerchen, F.: Finding critical thresholds for defining bursts in event logs, `http://home.eng.iastate.edu/~bibudh/techreport/burst_detection.pdf`

12. Leskovec, J., Backstrom, L., Kleinberg, J.M.: Meme-tracking and the dynamics of the news cycle. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 497–506 (2009)
13. Mathioudakis, M., Koudas, N.: Twittermonitor: trend detection over the twitter stream. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1155–1158 (2010)
14. Lithium, http://www.lithium.com/
15. Google AdWords, http://www.google.com/ads/adwords2/
16. Radian, http://www.radian6.com/
17. Sysomos, http://www.sysomos.com/
18. Thoora, http://thoora.com/
19. Trendrr, http://trendrr.com/
20. Twitscoop, http://www.twitscoop.com/
21. Vlachos, M., Meek, C., Vagena, Z., Gunopulos, D.: Identifying similarities, periodicities and bursts for online search queries. In: SIGMOD Conference, pp. 131–142 (2004)
22. Wang, M., Chan, N.H., Papadimitriou, S., Faloutsos, C., Madhyastha, T.M.: Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In: ICDE, pp. 507–516 (2002)
23. Wang, X., Zhai, C., Hu, X., Sproat, R.: Mining correlated bursty topic patterns from coordinated text streams. In: KDD, pp. 784–793 (2007)
24. Xu, K., Zhang, Z.L., Bhattacharyya, S.: Reducing unwanted traffic in a backbone network. Appeared in the Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop, SRUTI (2005)
25. Yuan, Z., Jia, Y., Yang, S.: Online burst detection over high speed short text streams. In: International Conference on Computational Science (ICCS), pp. 717–725 (2007)
26. Yuan, Z., Miao, J., Jia, Y., Wang, L.: Counting data stream based on improved counting bloom filter. In: Proceedings of the Ninth International Conference on Web-Age Information Management (WAIM), pp. 512–519 (2008)
27. Zhang, L., Guan, Y.: Detecting click fraud in pay-per-click streams of online advertising networks. In: ICDCS (2008)
28. Zhang, X., Shasha, D.: Better burst detection. In: Proceedings of the 22nd International Conference on Data Engineering (ICDE), p. 146 (2006)
29. Zhu, Y., Shasha, D.: Efficient elastic burst detection in data streams. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 336–345 (2003)

Concurrent Semi-supervised Learning with Active Learning of Data Streams

Hai-Long Nguyen[1], Wee-Keong Ng[1], and Yew-Kwong Woon[2]

[1] Nanyang Technological University, Singapore
nguy0105@ntu.edu.sg, wkn@acm.org
[2] EADS Innovation Works South Asia
david.woon@eads.net

Abstract. Conventional stream mining algorithms focus on stand-alone mining tasks. Given the single-pass nature of data streams, it makes sense to maximize throughput by performing multiple complementary mining tasks concurrently. We investigate the potential of concurrent semi-supervised learning on data streams and propose an incremental algorithm called CSL-Stream (Concurrent Semi–supervised Learning of Data Streams) that performs clustering and classification at the same time. Experiments using common synthetic and real datasets show that CSL-Stream outperforms prominent clustering and classification algorithms (D-Stream and SmSCluster) in terms of accuracy, speed and scalability. Moreover, enhanced with a novel active learning technique, CSL-Stream only requires a small number of queries to work well with very sparsely labeled datasets. The success of CSL-Stream paves the way for a new research direction in understanding latent commonalities among various data mining tasks in order to exploit the power of concurrent stream mining.

1 Introduction

Nowadays, large volumes of data streams are generated from various advanced applications such as information/communication networks, real-time surveillance, and online transactions. These data streams are usually characterized as temporally ordered, read-once-only, fast-changing, and possibly infinite, compared to static datasets studied in conventional data mining problems. Although many studies have been conducted to deal with data streams [2,4,11,13,27], most existing data stream algorithms focus on stand-alone mining. In many practical applications, it is desirable to concurrently perform multiple types of mining in order to better exploit data streams. For example, website administrators will be interested to use clickstream data to classify users into specific types and cluster webpages of similar topics at the same time in order to enhance the user's experience. In addition, concurrent mining offers a potential synergy: The knowledge gained by one mining task may also be useful to other mining tasks. Unfortunately, there is currently little research on concurrent stream mining.

In order to demonstrate the potential of concurrent stream mining, we propose a stream mining algorithm called CSL-Stream (Concurrent Semi-supervised

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 113–136, 2013.

Learning of Stream Data) that concurrently performs clustering and classification. We choose to focus on classification and clustering as they share common assumptions that data objects within a cluster (class) are similar and data objects belonging to different clusters (classes) are dissimilar. In addition, it is observed that most data objects in a cluster belong to a dominant class, and a class can be represented by many clusters. Therefore, classifiers can leverage on clustering to improve its accuracy [31]. For example, webpages within a topic are most visited by a specific type of users, and a user can be interested in many topics.

Moreover, CSL-Stream is a semi-supervised algorithm that is applicable to many real applications where only a small portion of data is labeled due to expensive labeling costs. In order to exploit a valuable limited amount of labeled data, CSL-Stream maintains a class profile vector for each node in the synopsis tree of the data stream. The profile vectors play an important role as pivots for the clustering process. The two mining tasks not only run at the same time, but also mutually improve each other. The clustering considers class profile vectors to attain high-purity clustering results. Conversely, the classification benefits clustering models in achieving high accuracy, and even works well with unlabeled data or outliers. Finally, CSL-Stream uses an *incremental learning* approach where the clustering and classification models are continuously improved to handle concept drifts and where mining results can be delivered in a timely manner. By re-using information from historical models, CSL-Stream requires only constant time to update its learning models with acceptable memory bounds. In addition, when the portion of labeled instances is very small ($\simeq 1\%$), we propose a novel active learning method to select the most "informative" queries, thus improving CSL-Stream's performance.

To develop such a concurrent stream mining framework, there are three main technical challenges. First, as multiple mining tasks are involved, CSL-Stream needs to maintain a larger amount of information extracted from the data stream. Second, effective and efficient operations on the data must be developed to avoid losing information. Third, a semi-learning approach is needed to cope with unlabeled data; in real-time applications, 100% pre-labeling of data is not generally be available due to expensive labeling costs. Overcoming these challenges and achieving impressive experimental results, we hope that CSL-Stream will inspire a new research direction in understanding latent commonalities among various data mining tasks in order to determine the degree of concurrency possible among them; this requires further research to derive data mining primitives which represent the entire space of data mining tasks.

In summary, our contributions are as follows:

- We propose a new algorithm to perform concurrent semi-supervised learning on data streams. We have proven that concurrent mining achieves better results by exploiting the synergies between related mining tasks.
- We integrate the dynamic synopsis tree with class profile vectors so that CSL-Stream can work well with unlabeled data.

- We deploy an incremental approach that can provide the clustering and classification results instantly. We also introduce advanced techniques to keep the space and time complexity low with theoretical bounds and empirically evaluations.
- We further enhance CSL-Stream with an active learning method so that it can work well with very sparsely labeled datasets.

The rest of this paper is organized as follows: In the next section, we discuss related work in stream clustering, classification and semi-supervised learning. We propose the CSL-Stream algorithm and provide its formal proofs in Section 3. We perform rigorous experiments to evaluate our algorithm with both real and synthetic datasets in Section 4. Finally, we conclude the paper in the last section.

2 Related Work

Data streams are generated in huge volumes and they change rapidly. Some data stream management system (DSMS) have been designed to facilitate powerful and fast online analytical processing (OLAP) over data streams [14,15,16,18]. These systems are useful for the many tasks of data stream mining, since they support not only basic measures such as *sum, average, min, max* but also advance measures such as regression and data distribution monitoring. In the following sections, we go beyond OLAP and review the evolution of classical clustering, classification and semi-supervised learning techniques to cope with the demanding challenges posed by data streams.

2.1 Clustering

Various algorithms have been recently proposed for stream data clustering, such as CluStream [2], DenStream [11], and D-Stream [13]. These algorithms adopt an online-offline scheme where raw stream data is processed to produce summary statistics in the online component and clustering is performed in the offline component using summary statistics. A micro-cluster is an extension of clustering features in BIRCH with two more dimensions, time and square of time [34]. Both CluStream and DenStream store summary statistics into a collection of micro-clusters. D-Stream views the entire data space as a finite set of equal-size grids, each of which maintains summary information. The different algorithms vary in their online data processing schemes and offline clustering methods. CluStream applies k-means to perform offline clustering while DenStream and D-Stream perform DBSCAN-like clustering [24]. DenStream and D-Stream share some common features:(i) They both apply a fading model to discard outdated data in the stream, which successfully addressed the evolving characteristics of stream data. (ii) They are both able to discover clusters with arbitrary shapes and sizes. (iii) Both of them have a synopsis removal scheme to limit the memory usage of synopsis storage. (iv) Clustering performance is highly sensitive to the values of the parameters, such as the number of stored micro-clusters and grid size. Recently, some algorithms extend subspace clustering methods to deal with high dimensional data streams [3,22,27].

2.2 Classification

Research work on data stream classification mainly falls into three categories. The first category extends the decision tree from traditional classification algorithms to the Hoeffding tree, such as CVFDT [19]. The second category uses ensemble classifiers and modifies them whenever concept drifts appear [28,33]. The last category is k-NN classification on stream data. An example of k-NN stream classification is the On-Demand Classifier [4] that maintains sets of micro-clusters with single-class labels and performs classification on demand. In 2007, Aggrawal *et al.* proposed a summarization paradigm for the clustering and classification of data streams [1]. This paradigm can be considered as a generalization of CluStream [2] and On-Demand Classification [4]. Unfortunately, the classification and clustering algorithms run separately and there is no mutual relationship between them. The algorithm needs time to process the offline operations and cannot provide timely results. Moreover, the use of a pyramidal time window makes it unstable; the micro-structures become larger and larger over time and this degrades the model's performance [35].

2.3 Semi-supervised Learning

Semi-supervised learning has been proposed to cope with partially labeled data. Although many semi-supervised learning algorithms have been developed for traditional data [36], there is still insufficient work for data streams. Semi-supervised learning can be categorized into two approaches [36]. The first approach is graph-based semi-supervised learning in which labeled and unlabeled samples are connected by a graph and edges represent similarity between samples [8,10]. The goal is to estimate a smooth label function f on the graph so that it is close to the given labels of vertices. However, it is difficult to apply the graph-based approach for data stream as the cost for managing and retrieving a large graph is high. The second approach, semi-supervised Support Vector Machines (S3VMs)[20,12,29], incorporates unlabeled instances into learning by defining a hinge loss function that favors unlabeled instances outside of the margin and assigns them weights to alleviate the imbalance problem. However, this is a highly non-convex optimization problem which is difficult to solve. Recently, Masud *et al.* proposed a semi-supervised algorithm for data streams, called SmSCluster [25]. The algorithm utilizes a cluster-impurity measurement to construct an ensemble of k-NN classifiers. When a classifier has no knowledge of a certain class, the algorithm copies the knowledge from another classifier with an injection procedure in order to reduce the mis-classification rate. However, SmSCluster suffers from high computational complexity as it performs clustering based on the k-means algorithm [32,6]. The difficulty in choosing the optimal k value is an inherent problem of k-means and this is aggravated by the evolving nature of data streams which inevitably results in more dynamic clusters being formed.

In many applications, totally labeled training instances are rare because it is time-consuming and difficult to get experts to label instances. Moreover, instances may be incorrectly labeled which may significantly degrade the performance of the model. Active learning or "query learning" attempt to address

this problem by querying domain experts to selectively label a small number of unlabeled instances. It aims to achieve an accurate predictive model using as few labeled instances as possible, therefore, minimizing the labeling cost. Some heuristics have been proposed to determine the most useful instances, including uncertainty sampling [30] and query by a committee (QBC) [26]. In uncertainty sampling, instances with the most uncertain labels under the current trained model are selected for querying. For example, Tong and Koller [30] proposed a sampling strategy for support vector machines to select those instances that are closest to the linear decision boundary. The QBC approach is related to maintaining a committee of models each of which is trained on the current labeled set and can label query candidates. The most important instance is the most disagreed one among the committee members. In [26], Melville *et al.* proposed an ensemble-based method that generates queries to increase diversity of the ensemble. There are three main settings where queries can be generated: membership query synthesis, stream-based sampling, and pool-based sampling. In the query synthesis setting, the learner may ask for labels of any unlabeled instances in the input space. A limitation of this setting is such arbitrary queries can be awkward to human experts, for example, active learning works poorly when a human is used to classify handwritten characters [7]. The stream-based setting is sometimes called sequential active learning, as when each unlabeled instance comes, the learner must decide whether to query it or not. A typical method is to define a region of uncertainty and only queries instances that fall within this region. This approach is computationally expensive, as the region must be updated after each query. The pool-based sampling approach is motivated by a scenario where there is a large pool of unlabeled data and only a small set of labeled data [23]. It usually deploys a greedy method together with an *informativeness* measure to evaluate all instances in the pool.

To overcome weaknesses in the existing approaches reviewed here, we propose a concurrent active learning mining algorithm in the next section to seize the unexplored benefits of performing multiple mining tasks on sparsely labeled data streams.

3 Proposed Method

Figure 1 is an overview of our novel concurrent mining approach. CSL-Stream stores a dynamic tree structure to capture the entire multi-dimensional space as well as a statistical synopsis of the data stream. Unlike static grids in D-Stream [13] that use much memory, our dynamic tree structure requires far less storage because unnecessary nodes will be pruned and sibling nodes merged. We apply a fading model to deal with concept drifts; the properties of a tree node will *decrease* according to how long the node has not been updated.

The two mining tasks of CSL-Stream, semi-supervised clustering and semi-supervised classification, are concurrently performed. They leverage on each other in terms of improving accuracy and speed; the clustering process takes into account class labels to produce high-quality clusters; the classification process

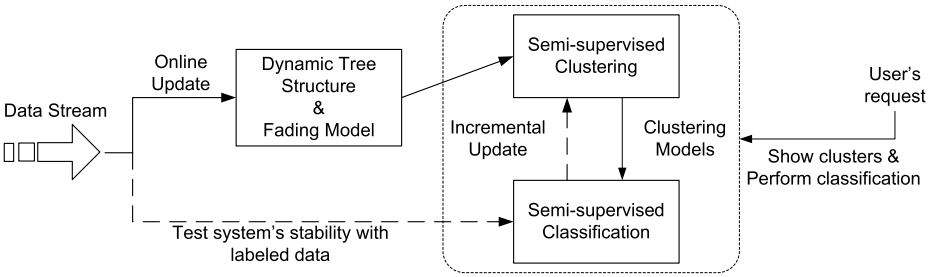


Fig. 1. Overview of CSL-Stream

uses clustering models together with a statistical test to achieve high accuracy and low running time.

CSL-Stream also employs an *incremental learning* approach. The system only needs to update whenever it becomes unstable. To check the system's stability, we select labeled data during each time interval t_p and test them with the classification model. If the accuracy is too low, we will incrementally update the clustering models.

3.1 Dynamic Tree Structure

We assume that the input data stream has d dimensions and forms a hyper-space S. We also assume a discrete time model where the time stamp is labeled by integers $t = 0, 1, 2, \ldots, n$. There is one record e of the data stream arriving at each time stamp, which consist of a coordinate vector $e_x = [x_1, x_2, \ldots, x_d]$, a class label e_l ($0 \leq e_l \leq L; e_l = 0$ if e is unlabeled), and a time stamp e_t.

We construct a dynamic tree structure to capture the entire space S with different levels of granularities as follows. Initially, a tree root at level 0 is created to hold the entire space S. Then for a tree node at level h, we partition each dimension into two equal-sized portions, and the tree node is partitioned into 2^d child nodes at level $h + 1$. We chose 2 as it provides sufficient granularity with reasonable storage requirements. This partitioning process terminates when it reaches a pre-defined maximum tree height H. The tree root contains the overall information of the entire space S. Each tree node at height h stores information of its subspace at granularity h. A tree node is only created if there are some instances belonging to it.

Figure 2 is an example of the tree structure with $d = 2$, $H = 2$. Figure 2(a) is a 2-dimensional data stream whose each dimension has a range [0-1]. Each data instance has a transaction identify, two coordinates $[x_1, x_2]$, a class label (1, 2, or 0 if unlabeled), and a time stamp. The tree root is created at level $h = 0$, then it is divided into four ($2^d = 2^2 = 4$) child nodes at level $h = 1$. Again, a tree node at level 1 is partitioned into four child nodes at level $h = 2$, and the partitioning process stops. Then, a tree node stores information of data instances belonging to its space. For example, in Figure 2(b), the data instance *tid*_02 with coordinates $[0.6, 0.2]$ is stored in the leaf node B at the bottom level $h = 2$.

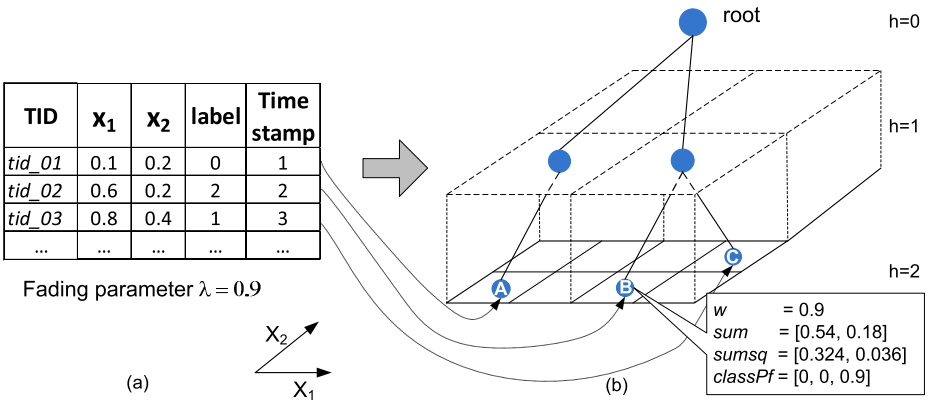


Fig. 2. Synopsis tree for the 2-dimensional data stream with $H = 2$

Node Storage. Suppose a tree node N receives m data instances $e^1, e^2, \ldots, e^m$. The following information will be stored in the tree node N:

1. Node weight w is the sum of weights of all data instances, $w = \sum_{i=1}^{m} e_w^i$. The weight of an instance is set to 1 at its arriving time, and then decreased over time to reflect its diminishing effect on the mining processes. Details of the weighting scheme are defined in the next section.
2. A d-dimensional vector sum is the weighted sum of the coordinates of all data instances, $sum = \sum_{i=1}^{m} e_w^i \times e_x^i$.
3. A d-dimensional vector $sumsq$ is the weighted sum of the squared coordinates of all data instances, $sumsq = \sum_{i=1}^{m} e_w^i \times {e_x^i}^2$
4. A $(L+1)$-dimensional vector $classPf$ stores the class profile of the node. The l^{th} element is the sum of weights of all data instances that have label l.

Storing the above information, we are able to derive the mean $\mu = sum/w$ and variance $\sigma = \sqrt{sumsq/w - (sum/w)^2}$ to describe the statistical class boundary of a node. A change of the class profile also reflects a change in class distribution.

Handling Concept Drift. Concept drift occurs in a data stream when there are changes in the underlying data distribution over time. There are two major approaches to handle concept drifts for data streams: (1) *instance selection*, and (2) *instance weighting*. The instance selection approach uses a sliding window to select recently arrived instances while ignoring older instances [21]. In instance weighting approach, data instances can be weighted according to their age, and their relevance with regard to the current concept, e.g. fading model [13].

In CSL-Stream, we apply a fading model with an exponential function to set the weight of a data instance e: $e_w = f(\Delta t) = \lambda^{\Delta t}$, where $0 < \lambda < 1$, and Δt is the time gap between the current time and the arriving time of the instance e. Figure 2(b) illustrates the fading model. Let us assume that the current time is 3, and the fading parameter λ is 0.9. The instance e with tid_02 that came at time stamp 2 has a weight of $e_w = \lambda^{(3-2)} = 0.9^1 = 0.9$. As the tree node B only

receives a data instance tid_02, the node B stores the following information: $w = 0.9$, $sum = 0.9 \times [0.6, 0.2] = [0.54, 0.18]$, $sumsq = 0.9 \times [0.6^2, 0.2^2] = 0.9 \times [0.36, 0.04] = [0.324, 0.036]$, and $classPf = [0, 0, 0.9]$ as the instance tid_02 has 2 as its class label .

3.2 Online Update

Given an instance e, this online-update process starts from the root and searches through the tree until the maximum height H is reached. At each height h, we update the node N that contains e. Then, we move down to search for the child node of N that contains e and update it. If no such child node exists, we create a new node. Using a hash technique with node pointers, the complexity of searching for the correct child node at any tree level is $O(1)$ so that the complexity of tree updating for each new data instance limits to $O(H)$, which is constant time in the stream environment.

As shown in Algorithm 1, the online-update process consists of two parts:

1. *Top-down updating* of the tree node's information whenever a new instance arrives (lines 6-16): Firstly, the time gap is calculated. All historical information is faded with the exponential function $f(\triangle t)$. The weight and element i-th of class profile are incremented by 1, while the *sum* vector and *sumsq* vector are added by the coordinates and the squares of coordinates respectively.
2. *Periodical pruning* of the tree to conserve memory and accelerate the mining process (line 19): For each time interval $t_p = \lfloor \log_\lambda(\alpha_L/\alpha_H) \rfloor$, which is the minimum time required for a node's density to be changed according to Lemma 2, CSL-Stream searches through the tree and deletes all sparse nodes (α_L, α_H to be defined in the next section). Next, it searches the tree to find a parent node whose all child nodes are dense. Then, all child nodes are merged and deleted, and the parent node contains the sum of their properties.

3.3 Concurrent Semi-supervised Learning

Semi-supervised Clustering. We use the node weight to define different types of nodes: dense, transitional and sparse nodes. The reachability property of dense nodes is also defined.

Node Density: The density of a node N is a ratio of its weight to its hypervolume (product of d-dimensional lengths). Suppose the volume of the entire data space S is $V(S)$ and the tree node N is at the height h, the volume of the node N is $V(N) = V(S)/2^{dh}$. According to Lemma 1, the upper bound of total weight of the synopsis tree is $lim_{t\to\infty} w_{total-tree} = \dfrac{1}{1-\lambda}$. Thus, the average density is defined as $lim_{t\to\infty} \dfrac{w_{total-tree}}{V(S)} = \dfrac{1}{(1-\lambda)V(S)}$.

Algorithm 1. Online Update

```
1: initialize tree T
2: while data stream is active do
3:    read next data point e
4:    N ← T.root
5:    while N.height ≤ H do
6:       Δt ← t − C.lastUpdateTime
7:       N.w ← N.w×f(Δt)+1
8:       N.sum ← N.sum×f(Δt)+e_x
9:       N.sumsq ← N.sumsq×f(Δt)+(e_x)^2
10:      i ← e_l;
11:      N.classPf[i] ← N.classPf[i]×f(Δt)
         + 1
12:      N ← N.getChild(e)
13:      if N is NULL then
14:         addSubNode(N)
15:      end if
16:   end while
17:   if t = t_p then
18:      Semi-Clustering()
19:   else if (t mod t_p = 0) then
20:      PruneTree()
21:      Incremental_Update()
22:   else if (user_request = true) then
23:      Show clustering results
24:      Semi-Classification()
25:   end if
26: end while
```

We define that a node is *dense* if its density is α_H times greater than the average density. Therefore, the dense condition can be written as:

$$\frac{w}{V(N)} = \frac{w}{V(S)/2^{dh}} \geq \alpha_H * \frac{1}{(1-\lambda)V(S)}$$

$$w \geq \alpha_H * \frac{V(S)}{2^{dh}} * \frac{1}{(1-\lambda)V(S)}$$

$$w \geq \alpha_H * \frac{1}{2^{dh}(1-\lambda)} = D_H. \quad (1)$$

Similarly, a node N is a sparse node if its density is α_L times less than the average density:

$$w \leq \alpha_L * \frac{1}{2^{dh}(1-\lambda)} = D_L. \quad (2)$$

And, a node is a transitional node if the density of the node is between D_H and D_L, i.e., $D_L < w < D_H$.

Node Neighborhood: Two nodes are *neighbors* if their minimum distance or single-link is less than $\frac{\delta}{2^H}$, where δ is a neighbor range parameter. A node N_p is *reachable* from N_q if there exists a chain of nodes $N_1, \ldots, N_i, N_{i+1}, \ldots, N_m$, $N_q = N_1$ and $N_p = N_m$; such that each pair (N_i, N_{i+1}) are neighbors. Moreover, a *cluster* is defined as a set of density nodes where any pair of two nodes is reachable.

The semi-supervised clustering of CSL-Stream is performed according to Algorithm 2. Initially, the algorithm traverses the tree to get a list of all dense nodes. Then, this list is divided into two sets: a set of labeled dense nodes *LDSet* and a set of unlabeled dense nodes *UDSet*. Firstly, we create a cluster for each labeled node (line 1). Next, we extend these clusters step by step with their

Algorithm 2. Semi-Clustering

Input:
A list of labeled dense nodes: $LDSet$
A list of unlabeled dense nodes: $UDSet$

1: Create a cluster for each node in $LDSet$.
2: Extend these clusters sequentially with neighbor nodes in $UDSet$.
3: Merge any 2 clusters if they have same labels.
4: Perform DBSCAN clustering for the remaining nodes in $UDSet$
5: Remove clusters whose weights are less than ϵ

Algorithm 3. Semi-Classification

Input: A test set: $testSet$
A list of clusters: $listCluster$

1: **while** $testSet$ is not empty **do**
2: $e \leftarrow testSet$.removeFirst()
3: Select the closet cluster, $\hat{C}$.
4: $stDv \leftarrow$ $\hat{C}$.getDeviation()
5: $mean \leftarrow$ $\hat{C}$.getMean()
6: $dist \leftarrow$calculateDistance(e,$mean$)
7: **if** $dist < \theta * stDv$ **then**
8: e.clusterID $\leftarrow$ $\hat{C}$.getClusterID()
9: e.class $\leftarrow$ $\hat{C}$.getDominantClass()
10: **else**
11: Set e as noise.
12: **end if**
13: **end while**

neighbor nodes. If the neighbor node does not belong to any cluster, we put it into the current cluster (line 2). If the neighbor node belongs to a cluster, we will merge these two clusters if they have the same label (line 3). Then, we continue to build clusters for the remaining nodes in *UDSet*. We perform DBSCAN[24] clustering to find the maximum set of remaining reachable unlabeled dense nodes in the *UDSet* (lines 4). Finally, we remove clusters whose weights are less than a threshold value, ϵ. These clusters are considered as noise.

Semi-supervised Classification. Details of the semi-supervised classification of CSL-Stream are given in Algorithm 3. The input to Algorithm 3 is a list of clusters. Each cluster in the list has a class profile array computed by the sum of class profiles of all its member nodes. For each testing instance, we find the closest cluster and then do a statistical check to decide whether the distance between the instance and the cluster's center is acceptable. The adequate range is less than θ times of the cluster's standard deviation (line 7). The cluster's center and deviation are computed accordingly $\mu = \frac{sum}{w}, \sigma = \sqrt{\frac{sumsq}{w} - \left(\frac{sum}{w}\right)^2}$. If the distance is acceptable, the dominant class of the cluster will be assigned to the testing sample. Else, the instance will be considered as noise.

Incremental Update. CSL-Stream is an incremental algorithm. It can detect and update its learning models incrementally whenever a concept drift occurs.

Algorithm 4. Incremental Update

Input: A list of historical clusters: $listClusters$
Output: A list of clusters

1: **if** $accuracy \leq threshold$ **then**
2: **for all** cluster $\hat{C}$ in $listCluster$ **do**
3: $\hat{C}$.removeSparseNodes()
4: $\hat{C}$.checkConsistence()
5: **end for**
6: Traverse the tree to get a list β_{dense} of *isolated* dense nodes.
7: Separate β_{dense} into two sets of labeled and unlabeled nodes, $LDSet$ and $UDSet$.
8: Semi-Clustering($LDSet$,$UDSet$);
9: **end if**

For each time interval t_p (Algorithm 1, line 21), we select labeled instances and test them with the classification model to check the system's stability. If the accuracy is greater than a predefined threshold value (experiments reveal that 0.9 is an appropriate value); the model remains stable, and we can skip the remaining steps. Else, we need to refine the learning models. CSL-Stream begins to fine-tune the historical clustering model by checking the consistency of each cluster. The algorithm removes sparse nodes, that are dense at the previous learning phase, and splits the cluster if it becomes unconnected or has low purity.

Then, CSL-Stream traverses the tree to get a list of isolated dense nodes β_{dense} (line 6). After separating β_{dense} into two sets of labeled and unlabeled dense nodes, it performs the semi-supervised clustering method to derive new arriving clusters. As the procedure reuses the historical clustering results, the size of β_{dense} is relatively small. Thus, the running time of the clustering process is reduced significantly.

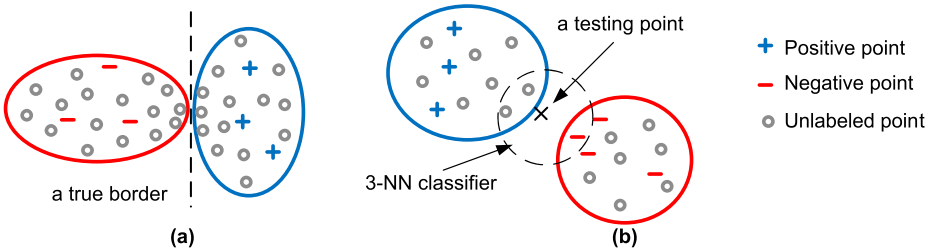


Fig. 3. Examples of the benefits of concurrent semi-learning

Example. Our concurrent approach maximizes computational resources and exploits advantages of the interplay between clustering and classification. CSL-Stream's semi-supervised clustering takes care of data labels by continuously updating the class profile array of each node. It ensures that two clusters are merged only if they have the same dominant class, finds the best borders among

clusters. For example, in Figure 3 (a), two clusters with different class labels share the same border. A stand-alone clustering algorithm will probably incorrectly consider them as a single cluster. For example, D-Stream will merge the two clusters as it views the data points at the border as connected due to their proximity to one another [13]. CSL-Stream does not merge them as they have different dominant classes, and thus CSL-Stream correctly considers them as two clusters.

By disregarding unlabeled data, a stand-alone classification method may not capture the classes' distribution. Using clustering results with unlabeled data, CSL-Stream can gain knowledge of the classes' distributions and improve its accuracy. For example, in Figure 3 (b), a 3-NN classifier wrongly classifies the testing point because most its neighbors are unlabeled. CSL-Stream selects the label of the nearest cluster, and correctly classifies the testing point as positive.

3.4 Enhancing Concurrent Learning with Active Learning

Here, we choose to apply the pool-based sampling setting as it can provide the best queries after collecting and evaluating the entire collection of unlabeled data. We further propose an effective heuristic to select the most important query. Particularly, we need to check and generate a query with two following cases:

- *Unlabeled clusters*: A cluster is considered as an unlabeled cluster if it contains no labeled data instance. In this case, the cluster's center representing the whole cluster is the most important data instance, so that we select the center to query for a label. Then, the label of the cluster is set to the suggested label.
- *Skewed labeled clusters*: A cluster's label is skewed if its labeled data instances only occupy one side of the cluster. As we are uncertain about the other side of the cluster, we choose the farthest instance from the other side to query for a label. If the cluster's label differs from the suggested label, we need to internally re-cluster this cluster.

Figure 4 shows an illustration of our active learning method. CSL-Stream processes the synopsis of data streams, which are represented by a dynamic tree structure. Hence, it may cause awkward problems to human experts if we simply select the center of a tree node to query. To overcome this problem, we attach each tree node to a representative data instance, and this instance is updated whenever a new sample arrives in the corresponding tree node. Details of our active learning method are given in Algorithm 5. As the number of queries is limited, we want to perform active learning with clusters having bigger weights first. Hence, we sort clusters according to their weights in descending order (line 1). Then, we check all the clusters and generate the most informative queries if the number is less than or equal to a maximum value (lines 2-18).

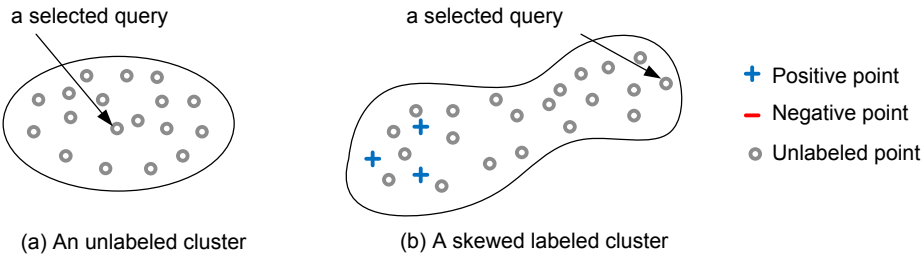


Fig. 4. Enhancing Concurrent Learning with Active Learning

Algorithm 5. Active Learning

Input: A list of clusters: *listCluster*

```
1: Descendingly sort the list listCluster according to clusters' weights.
2: for all cluster C in the list of clusters listCluster do
3:    if number of query > max − query then
4:       Exit.
5:    else
6:       if C is unlabeled then
7:          Select the tree node N that contains the cluster's center.
8:          Get the representative point p_rep of the tree node N.
9:          Query for the label of p_rep and set it as the dominant class label of C.
10:      else if C is skewed labeled then
11:         Get the "informative" node N that is farthest from labeled nodes of C.
12:         Query for the label of the representative point p_rep of N.
13:         if p_rep.class ≠ C.getDominantClass() then
14:            Perform Semi-Clustering() for cluster C.
15:         end if
16:      end if
17:   end if
18: end for
```

3.5 Theoretical Proofs

Lemma 1. *The total weight of the synopsis tree is less than or equal to* $\frac{1}{1-\lambda}$.

Proof. The total weight of the synopsis tree is the sum of the weights of all its tree nodes. Moreover, the weight of a tree node equals to the sum of data instances belonging to the node's subspace. Hence, given a current time t, the tree weight is the sum of weights of data instances that have arrived during the time gap $[0, 1, \ldots, t]$. As we apply the fading model, the weight of data instance arriving at time instance $t' \in [0, t]$ is: $f(\Delta t) = \lambda^{\Delta t} = \lambda^{t-t'}$. Thus, the total weight of the synopsis tree is:

$$W_{total-tree} = \sum_{t'=0}^{t} \lambda^{t-t'} = \frac{1-\lambda^{t+1}}{1-\lambda} \leq \frac{1}{1-\lambda}.$$

Alternatively, we can derive that $lim_{t\to\infty} W_{total-tree} = \frac{1}{1-\lambda}$.

Lemma 2. *The minimum time for a tree node to change from dense to sparse is* $t_p = \lfloor \log_\lambda(\alpha_L/\alpha_H) \rfloor$.

Proof. Let us assume that a tree node N is sparse at a time instance t_1. It then becomes sparse at a time instance t_2, $(t_1 < t_2)$. According to the definition of a dense node in Equation 1, the tree node's weight at time t_1 is:

$$w_1 \geq \alpha_H * \frac{1}{2^{dh}(1-\lambda)}.$$

Similarly, according to the definition of sparse node in Equation 2, the tree node's weight at time t_2 is:

$$w_2 \leq \alpha_L * \frac{1}{2^{dh}(1-\lambda)}.$$

We have:

$$\begin{aligned} w_2 &\geq w_1 * f(t_2 - t_1) = w_1 * \lambda^{(t_2-t_1)} \\ \alpha_L * \frac{1}{2^{dh}(1-\lambda)} &\geq \alpha_H * \frac{1}{2^{dh}(1-\lambda)} * \lambda^{(t_2-t_1)} \\ \alpha_L &\geq \alpha_H * \lambda^{(t_2-t_1)} \\ \alpha_L/\alpha_H &\geq \lambda^{(t_2-t_1)} \\ t_p = \lfloor \log_\lambda(\alpha_L/\alpha_H) \rfloor &\leq (t_2 - t_1). \end{aligned}$$

Space Complexity

Lemma 3. *If the density threshold to remove sparse node is* D_L*, then the maximum number of nodes in the synopsis tree is* $N_{max-node} \leq \lceil \frac{\log D_L}{log\lambda} \rceil = O(dH)$

Proof. In the worst case, we assume that every data point arrives at a different node and no merging is performed. When a node receives a data point, its weight is set to 1. We define t_{max} as the time needed for this node to become sparse with its weight $\leq D_L$, and is removed. After t_{max} time, whenever a data point arrives, a new tree node created, and another one is deleted. This means that t_{max} is the maximum number of tree nodes. We have:

$$\lambda^{t_{max}} = D_L \quad \Rightarrow \quad t_{max} \log \lambda = \log D_L \quad \Rightarrow \quad t_{max} \leq \lceil \frac{\log D_L}{\log \lambda} \rceil.$$

We also know that: $D_L = \alpha_L * \frac{1}{2^{dh}(1-\lambda)}$. Then,

$$\Rightarrow \quad t_{max} \leq \lceil \log_\lambda \alpha_L - \log_\lambda(1-\lambda) - dH \log_\lambda 2 \rceil \quad = O(dH).$$

Table 1. Characteristics of datasets used for evaluation

Name	#Instances	#Attributes	#Classes
RBF	100,000	10	5
HYP	100,000	10	2
LED	100,000	24	3
SHUTTLE	58,000	9	7
KDD'99	494,022	34	5
COVERTYPE	581,012	54	7

Time Complexity

Lemma 4. *The complexity of CSL-Stream is $O(n(dH + L))$, where n is the number of instances of the data stream and L is the number of classes.*

Proof. The proposed CSL-Stream has three main operations: online updating, clustering and classification. The online updating consists of top-down updating and periodical pruning. The top-down updating only requires a constant time to update a new data instance and its complexity is $O(nH)$. The running time of the pruning process depends on the number of tree nodes. In the previous section, we prove that the upper bound of the number of tree nodes is $O(dH)$, so the pruning process's complexity is $O(ndH)$.

The complexity of the clustering depends on its implementation. If we use R*-trees to implement the synopsis tree, the searching time for neighbor nodes is $O(log(dH))$. And the complexity of clustering operation is $O(dHlog(dH))$. The running time of the classification operation depends on the number of clusters, and this number is usually a multiple of the number of classes. Thus, the classification time can be expected to be $O(nL)$, where L is the number of classes.

In summary, the complexity of the algorithm is: $O(nH + ndH + dHlog(dH) + nL) = O(n(dH + L))$

4 Experiments and Analysis

4.1 Experimental Setup

We use both synthetic and real datasets in our experiments to demonstrate the practicality of our approach. Three synthetic datasets, Random RBF generator (RBF), Rotating Hyperplane (HYP), and LED dataset (LED), are generated from the MOA framework [9]. Concept drifts are generated by moving 10 centroids at a rate of 0.001 per instance in the RBF dataset, and changing 10 attributes at a rate of 0.001 per instance in the HYP dataset. We also use the three largest real datasets in the UCI machine learning repository: SHUTTLE, KDD'99, and Forest Covertype [17]. Table 1 shows the characteristics of the six datasets.

In order to illustrate the beneficial mutual relationship between clustering and classification, we created two variants of CSL-Stream that do not exploit this relationship as follows:

- *Alone-Clustering:* This variant of CSL-Stream does not consider class labels while clustering.
- *Alone-Classification:* This variant of CSL-Stream finds the nearest tree node to the testing instances and classifies them to the dominant class of the tree node.

We compare the performance of CSL-Stream to its stand-alone variants, D-Stream [13], and SmSCluster [25]. The experiments were conducted on a Windows PC with a Pentium D 3GHz Intel processor and 2GB memory.

Parameter Setting

Table 2. Parameter Table

Parameter	Description	Recommended Value
λ	fading parameter	[0.99,1]
α_H	dense ratio threshold	3
α_L	sparse ratio threshold	0.8
H	the maximum trees height	[5,10]
ϵ	noise threshold	[1,3]
δ	neighborhood range	[0,2]
θ	statistical range	[3,6]

Table 2 summarizes seven parameters that are used in CSL-Stream and their recommended values. The first three parameters, which do not greatly affect results, are fixed as follows: the dense ratio threshold α_H is set to 3.0, the sparse ratio threshold α_L is set to 0.8, and the fading factor λ is set to 0.998 as recommended in D-Stream [13]. For D-Stream and SmSCluster, we use the published default parameter configurations. The chunk size *chunk-size* is set to 1000 for all datasets. For CSL-Stream, the other parameters of CSL-Stream are set as follows: maximum tree's height $H = 5$, the noise threshold $\epsilon = 1$, the neighborhood range $\delta = 2$, and the statistical range factor $\theta = 3$. The scalability and sensitivity of these parameters are evaluated in the next section.

To simulate the data stream environment, we divided each dataset into many chunks of size *chunk-size*. These sequential chunks are numbered and their roles are set as follows: the odd chunks are used for training and the even chunks are used for testing. We run the experiments 10 times for each dataset and summarize their running time and performance with their means and standard deviations in Tables 3 and 5.

4.2 Clustering Evaluation

Here, we use the BCubed [5] measure to evaluate clustering results in our experiments. It computes the average correctness over the entire dataset and has been proven to be more useful than the purity, inverse purity, entropy, and F-measures.

We conducted experiments with the 100%-labeled datasets to assess the clustering results of CSL-Stream, Alone-Clustering and D-Stream. Table 3 shows the running time as well as B-Cubed comparisons among the three clustering methods.

$$Bcubed_P = Avg_e[Avg_{e'.C(e)=C(e')}[Correctness(e, e')]]$$
$$Bcubed_R = Avg_e[Avg_{e'.L(e)=L(e')}[Correctness(e, e')]],$$

where C denotes the set of clusters, L denotes the set of classes, and $Correctness(e, e') = 1$ if e and e' have the same class label and the same cluster, otherwise it is 0. Bcubed is the F-combination of its precision and its recall:

$$Bcubed = \frac{1}{\alpha(\frac{1}{Bcubed_P}) + (1-\alpha)(\frac{1}{Bcubed_R})}$$

Table 3. Comparison among CSL-Stream, Alone-Clustering and D-Stream with B-Cubed measure. Time is measured in seconds. For each dataset, the lowest running time is underlined, and the highest B-Cubed value is **boldfaced**.

	CSL-Stream		**Alone-Clustering**		**D-Stream**	
	Time	B-Cubed	Time	B-Cubed	Time	B-Cubed
RBF(10,0.001)	$10.64_{\pm 1.56}$	$\mathbf{37.67_{\pm 4.93}}$	$17.22_{\pm 1.72}$	$36.27_{\pm 2.32}$	$17.37_{\pm 1.36}$	$17.39_{\pm 2.41}$
HYP(10,0.001)	$13.37_{\pm 1.23}$	$\mathbf{65.24_{\pm 10.66}}$	$23.67_{\pm 1.66}$	$57.14_{\pm 4.01}$	$33.37_{\pm 1.46}$	$55.54_{\pm 4.66}$
LED	$53.9_{\pm 1.68}$	$\mathbf{68.38_{\pm 11.74}}$	$205.61_{\pm 2.34}$	$19.1_{\pm 0.58}$	$203.8_{\pm 2.94}$	$19.3_{\pm 0.63}$
SHUTTLE	$1.37_{\pm 0.16}$	$\mathbf{93.46_{\pm 1.04}}$	$1.37_{\pm 0.16}$	$89.07_{\pm 1.83}$	$1.45_{\pm 0.17}$	$88.1_{\pm 0.93}$
KDD'99	$39.78_{\pm 0.62}$	$\mathbf{76.89_{\pm 27.83}}$	$38.68_{\pm 0.84}$	$76.88_{\pm 27.83}$	$53.79_{\pm 1.06}$	$73.5_{\pm 31.24}$
COVERTYPE	$130.24_{\pm 1.87}$	$26.54_{\pm 9.68}$	$212.07_{\pm 2.45}$	$\mathbf{35.11_{\pm 10.79}}$	$152.46_{\pm 2.67}$	$12.55_{\pm 5.8}$

We observe that CSL-Stream achieves the lowest running time and the highest B-Cubed value for many datasets. Comparing with Alone-Clustering and D-Stream, CSL-Stream takes into account the class labels during clustering and guarantees that no two clusters with the same label are merged. Thus, it achieves better B-Cubed results than Alone-Clustering and D-Stream. Moreover, using the dynamic tree structure, CSL-Stream can adapt quickly to concept drifts. It is also the fastest method because the pruning process helps to remove unnecessary tree nodes. Although CSL-Stream has low B-Cubed values in some datasets, e.g., RBF, HYP, and COVERTYPE as each class has many clusters in these datasets, CSL-Stream still attains high purity in these cases ($> 90\%$) as shown in Table 4; purity is a measure of cluster homogeneity.

4.3 Classification Evaluation

We compare the running time and accuracy among CSL-Stream, Alone-Classification, and SmSCluster [25]. The upper part of Table 5 reports the

Table 4. Comparison among CSL-Stream, Alone-Clustering and D-Stream with purity measure. Time is measured in seconds. For each dataset, the lowest running time is underlined, and the highest purity value is boldfaced.

	CSL-Stream		**Alone-Clustering**		**D-Stream**	
	Time	Purity	Time	Purity	Time	Purity
RBF(10,0.001)	$10.64_{\pm 1.56}$	$\mathbf{100_{\pm 0}}$	$17.22_{\pm 1.72}$	$48.41_{\pm 10.66}$	$17.37_{\pm 1.36}$	$99_{\pm 2.05}$
HYP(10,0.001)	$13.37_{\pm 1.23}$	$\mathbf{99.99_{\pm 0.03}}$	$23.67_{\pm 1.66}$	$67.76_{\pm 8.09}$	$33.37_{\pm 1.46}$	$69.63_{\pm 8.54}$
LED	$53.9_{\pm 1.68}$	$\mathbf{100_{\pm 0.01}}$	$205.61_{\pm 2.34}$	$15.74_{\pm 2.41}$	$203.8_{\pm 2.94}$	$15.9_{\pm 2.45}$
SHUTTLE	$1.37_{\pm 0.16}$	$\mathbf{98.46_{\pm 0.44}}$	$1.37_{\pm 0.16}$	$97.78_{\pm 2.53}$	$1.45_{\pm 0.17}$	$98.22_{\pm 1.27}$
KDD'99	$39.78_{\pm 0.62}$	$\mathbf{99.99_{\pm 0.05}}$	$38.68_{\pm 0.84}$	$99.99_{\pm 0.07}$	$53.79_{\pm 1.06}$	$99.99_{\pm 0.03}$
COVERTYPE	$130.24_{\pm 1.87}$	$\mathbf{99.28_{\pm 0.53}}$	$212.07_{\pm 2.45}$	$79.32_{\pm 7.63}$	$152.46_{\pm 2.67}$	$91.98_{\pm 3.98}$

Table 5. Comparison with fully labeled datasets among CSL-Stream, Alone-Classification and SmSCluster. Time is measured in seconds. For each dataset, the lowest running time is underlined, and the highest accuracy is **boldfaced**.

	CSL-Stream		**Alone-Classification**		**SmSCluster**	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
RBF(10,0.001)	$49.29_{\pm 2.35}$	$\mathbf{71.57_{\pm 6.37}}$	$53.32_{\pm 3.24}$	$44.57_{\pm 7.18}$	$41.45_{\pm 3.35}$	$30.1_{\pm 12.38}$
HYP(10,0.001)	$15.78_{\pm 1.68}$	$\mathbf{87.88_{\pm 1.88}}$	$16.17_{\pm 2.03}$	$70.66_{\pm 2.09}$	$40.18_{\pm 2.65}$	$76.05_{\pm 2.61}$
LED	$34.17_{\pm 2.16}$	$\mathbf{72.73_{\pm 1.82}}$	$98.85_{\pm 3.12}$	$10.24_{\pm 1}$	$85.69_{\pm 3.87}$	$54.70_{\pm 3.45}$
SHUTTLE	$2.56_{\pm 0.35}$	$\mathbf{98.3_{\pm 0.3}}$	$2.64_{\pm 0.91}$	$98.28_{\pm 0.31}$	$20.35_{\pm 2.61}$	$97.50_{\pm 0.49}$
KDD'99	$83.06_{\pm 2.47}$	$\mathbf{98.06_{\pm 8.29}}$	$87.57_{\pm 2.13}$	$98.25_{\pm 8.24}$	$565.02_{\pm 3.87}$	$85.33_{\pm 33.39}$
COVERTYPE	$183.75_{\pm 3.05}$	$\mathbf{81.63_{\pm 10.43}}$	$194.41_{\pm 3.46}$	$78.96_{\pm 9.39}$	$320.65_{\pm 3.02}$	$49.23_{\pm 15.42}$

running time and accuracy comparisons among the above classification algorithms when the datasets are fully labeled. It is found that CSL-Stream is the fastest and the most accurate algorithm for many datasets. CSL-Stream is better than Alone-Classification in terms of speed and accuracy because it exploits clustering results for classification. CSL-Stream also takes less time as it only compares the testing instances to a small number of clusters and is resistant to noise. SmSCluster suffers from a high time complexity and low accuracy with non-spherical clusters since it is based on the k-Means algorithm.

Furthermore, we measure performances of the above algorithms when unlabeled data is present. We conduct experiments with the Hyperplane and KDD'99 datasets with different proportions of labeled data instances: *50%, 25%* and *10%*. Table 6 shows the running time and the accuracy of the above classification algorithms. We observe that CSL-Stream and SmSCluster can work well with partially labeled data. When the percentage of labeled data decreases, the accuracy of Alone-Classification drops quickly while the accuracy of CSL-Stream and SmSCluster remains high. It is obvious too that CSL-Stream outperforms SmSCluster in terms of speed and accuracy. These results also show that CSL-Stream has the best overall performance.

Table 6. Comparison with partially labeled datasets among CSL-Stream, Alone-Classification and SmSCluster. Time is measured in seconds. For each dataset, the lowest running time is underlined, and the highest accuracy is boldfaced.

	CSL-Stream		**Alone-Classification**		**SmSCluster**	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
RBF(10,0.001) 50%	$52.86_{\pm 2.13}$	$\mathbf{54_{\pm 10.13}}$	$40.36_{\pm 2.48}$	$26.12_{\pm 7.51}$	$48.33_{\pm 1.43}$	$30.47_{\pm 12.29}$
RBF(10,0.001) 25%	$49.53_{\pm 2.54}$	$\mathbf{51.15_{\pm 10.81}}$	$38.92_{\pm 2.67}$	$15.47_{\pm 5.86}$	$44.36_{\pm 1.86}$	$27.93_{\pm 12.3}$
RBF(10,0.001) 10%	$49.09_{\pm 2.01}$	$\mathbf{48.88_{\pm 10.06}}$	$38.56_{\pm 2.65}$	$8.16_{\pm 4.86}$	$30.73_{\pm 1.08}$	$28.43_{\pm 12.16}$
HYP(10,0.001) 50%	$20.08_{\pm 1.48}$	$\mathbf{81.6_{\pm 3.61}}$	$21.11_{\pm 2.16}$	$35.97_{\pm 3.02}$	$37.89_{\pm 2.14}$	$74.36_{\pm 2.68}$
HYP(10,0.001) 25%	$17.34_{\pm 1.32}$	$\mathbf{73.88_{\pm 4.25}}$	$18.65_{\pm 2.09}$	$18.29_{\pm 2.1}$	$46.26_{\pm 2.09}$	$64.36_{\pm 3.11}$
HYP(10,0.001) 10%	$15.97_{\pm 1.65}$	$61.18_{\pm 7.15}$	$17.47_{\pm 2.05}$	$7.57_{\pm 1.32}$	$31.56_{\pm 1.86}$	$\mathbf{61.41_{\pm 4.50}}$
LED 50%	$50.52_{\pm 2.41}$	$\mathbf{62.08_{\pm 9.33}}$	$108.78_{\pm 2.57}$	$6.53_{\pm 5.04}$	$133.3_{\pm 2.19}$	$52.14_{\pm 4.59}$
LED 25%	$54.3_{\pm 2.54}$	$\mathbf{51.85_{\pm 5.87}}$	$99.65_{\pm 2.88}$	$2.73_{\pm 4.53}$	$127.64_{\pm 2.76}$	$47.44_{\pm 4.11}$
LED 10%	$36.36_{\pm 2.87}$	$\mathbf{45.43_{\pm 4.07}}$	$97.77_{\pm 2.71}$	$0.88_{\pm 2.86}$	$74.75_{\pm 2.77}$	$44.93_{\pm 3.61}$
SHUTTLE 50%	$2.59_{\pm 0.34}$	$\mathbf{97.7_{\pm 2.3}}$	$2.47_{\pm 0.63}$	$97.54_{\pm 2.61}$	$18.88_{\pm 1.51}$	$97.69_{\pm 0.44}$
SHUTTLE 25%	$2.5_{\pm 0.42}$	$\mathbf{97.5_{\pm 1.09}}$	$2.5_{\pm 0.88}$	$97.31_{\pm 2.4}$	$18.34_{\pm 1.93}$	$97.41_{\pm 0.44}$
SHUTTLE 10%	$2.48_{\pm 0.26}$	$96.12_{\pm 2.15}$	$2.47_{\pm 0.47}$	$96.71_{\pm 2.36}$	$19_{\pm 1.91}$	$\mathbf{97.02_{\pm 1}}$
KDD'99 50%	$72.3_{\pm 2.65}$	$\mathbf{97.08_{\pm 8.71}}$	$77.26_{\pm 2.35}$	$95_{\pm 12.11}$	$691.25_{\pm 4.26}$	$75.33_{\pm 14.27}$
KDD'99 25%	$72.56_{\pm 2.69}$	$\mathbf{96.49_{\pm 9.25}}$	$78.12_{\pm 2.48}$	$92.82_{\pm 15.47}$	$703.56_{\pm 4.35}$	$75.03_{\pm 14.65}$
KDD'99 10%	$79.46_{\pm 2.13}$	$\mathbf{95.07_{\pm 11.7}}$	$85.44_{\pm 2.49}$	$90.63_{\pm 18.89}$	$821.56_{\pm 4.07}$	$75.25_{\pm 14.25}$
COVERTYPE 50%	$196.34_{\pm 3.4}$	$\mathbf{74.71_{\pm 11.66}}$	$178.95_{\pm 3.65}$	$53.56_{\pm 7.86}$	$400.62_{\pm 4.27}$	$35.95_{\pm 14.14}$
COVERTYPE 25%	$190.55_{\pm 3.57}$	$\mathbf{72.37_{\pm 11.33}}$	$168.12_{\pm 3.08}$	$38.27_{\pm 7.02}$	$394.8_{\pm 4.65}$	$33.08_{\pm 17.93}$
COVERTYPE 10%	$184.88_{\pm 3.79}$	$\mathbf{67.36_{\pm 11.99}}$	$165.11_{\pm 3.49}$	$25.09_{\pm 5.53}$	$350.9_{\pm 4.92}$	$28.05_{\pm 14.61}$

4.4 Enhancing Concurrent Mining with Active Learning

In order to evaluate the effectiveness of active learning, we set the proportion of labeled data to just 1%. Moreover, our active learning methods are configured with the following thresholds of the number of queries: 1%, 0.5%, 0.2%, and 0.1%. Table 7 shows the comparisons between CSL-Stream and its variants. We observe that CSL-Stream does not work well with datasets with very few labeled samples. Although it attains good results for SHUTTLE and KDD'99 datasets, its accuracy is low for remaining ones. With active learning, CSL-Stream increases its average accuracy by *5.81%, 4.11%, 3.42%, 2.91%* with query thresholds of 1%, 0.5%, 0.2%, and 0.1% respectively. We also find that our active learning method is more effective with real datasets than synthetic datasets, which are typically generated by random processes.

The impressive experimental results show that our active learning method is able to work well with very sparsely labeled datasets by exploiting the most informative queries.

4.5 Scalability Evaluation

Section 3.5 gives an upper bound for the maximum number of nodes and the running time of CSL-Stream. Here, we have examined the scalability of CSL-Stream in terms of the number of nodes and the running time w.r.t dimensionality and the tree's height.

Table 7. Comparison with sparsely labeled datasets *(1%)* among CSL-Stream and its variant enhanced with Active Learning with different thresholds of the number of queries

Dataset	**CSL-Stream**	**CSL-Stream + Active Learning**			
		#query≤1%	#query≤0.5%	#query≤0.2%	#query≤0.1%
RBF(10,0.001) 1%	$44.81_{\pm 12.99}$	$51.6_{\pm 11.55}$	$52.26_{\pm 11.21}$	$51.18_{\pm 11.37}$	$50.5_{\pm 11.6}$
HYP(10,0.001) 1%	$69.24_{\pm 5.94}$	$72.72_{\pm 4.41}$	$71.76_{\pm 4.7}$	$71.87_{\pm 4.51}$	$71.5_{\pm 4.25}$
LED 1%	$11.16_{\pm 2.6}$	$11.83_{\pm 2.7}$	$11.57_{\pm 3.12}$	$11.33_{\pm 3.1}$	$11.64_{\pm 2.69}$
SHUTTLE 1%	$74.38_{\pm 11.08}$	$87.47_{\pm 2.81}$	$79.53_{\pm 1.44}$	$79.41_{\pm 5.88}$	$79.33_{\pm 4.81}$
KDD'99 1%	$92.31_{\pm 14.81}$	$94.5_{\pm 12.56}$	$94.47_{\pm 12.38}$	$94.32_{\pm 12.22}$	$93.99_{\pm 13.12}$
COVERTYPE 1%	$59.26_{\pm 12.22}$	$67.92_{\pm 12.34}$	$66.24_{\pm 12.63}$	$63.6_{\pm 12.08}$	$61.67_{\pm 12.3}$
Average	$58.53_{\pm 9.94}$	$64.34_{\pm 7.73}$	$62.64_{\pm 7.58}$	$61.95_{\pm 8.19}$	$61.44_{\pm 8.13}$
Average Improvement		**5.81**	**4.11**	**3.42**	**2.91**

We conduct experiments with the KDD'99 dataset to assess the scalability w.r.t dimensionality d. We set the maximum tree height to 5, and increase d from 5 to 34 with unit steps. With Figures 5 and 6, we can clearly see that the maximum number of nodes and the running time of CSL-Stream increase linearly as dimensionality increases.

To evaluate the scalability w.r.t the maximum tree height H, we select the first 30K data instances of KDD'99 dataset and set the dimensionality to 34. We increase H from 3 to 10 with unit steps. Figures 7 and 8 show that the maximum number of nodes and the running time of CSL-Stream increase linearly as H increases.

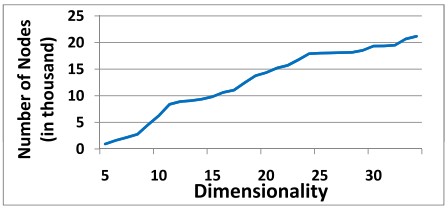


Fig. 5. Number of nodes vs. dimensionality

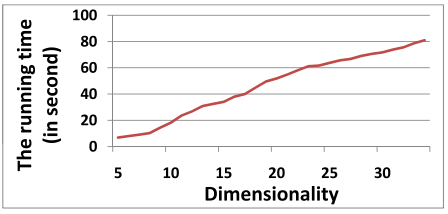


Fig. 6. Running time vs. dimensionality

Fig. 7. Number of nodes vs. tree height

Fig. 8. Running time vs. tree height

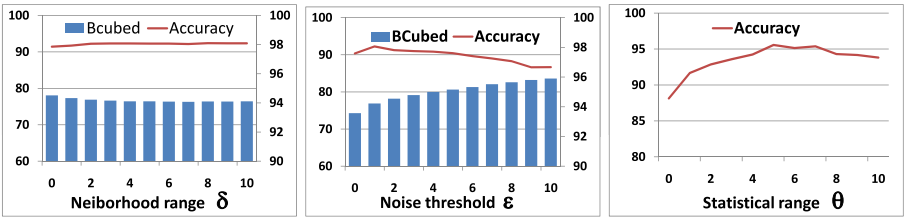


Fig. 9. Sensitivity analysis

4.6 Sensitivity Analysis

In the experiments, we set the neighborhood range $\delta = 2$, the noise threshold $\epsilon = 1$, and the statistical range factor $\theta = 3$. We test the sensitivity of δ, ϵ, and θ with the KDD'99 dataset. We add 10% of random noise to test the sensitivity of θ. Figure 9 shows the Bcubed values and accuracy of CSL-Stream with different parameters.

We observe that CSL-Stream is insensitive to the neighborhood range as the KDD'99 dataset is high-dimensional. When the noise threshold increases, CSL-Stream's accuracy decreases as some important clusters are ignored. Our algorithm is also sensitive to the statistical range factor, and it should be set between two to six to achieve good accuracy.

5 Conclusions

We have studied the new problem of concurrent mining for data streams and proposed CSL-Stream to integrate semi-supervised classification and clustering. We have conducted extensive experiments to show that CSL-Stream outperforms state-of-the-art data stream algorithms in terms of speed, accuracy and scalability. Experimental results also showed that it can produce mining results in a real-time manner, making it suitable for online applications.

With the success of CSL-Stream, we intend to perform an in-depth study on the relationships among basic mining tasks in an attempt to create a framework of mining primitives which will allow us to easily determine the degree of possible concurrency among various tasks. With such a framework, new synergies among different combinations of mining tasks may be discovered, which may lead to performance improvement or even spawn new research challenges and applications.

Furthermore, we have enhanced CSL-Stream with a novel active learning method so that it can achieve excellent results even with sparsely labeled datasets. However, in real applications, a domain expert is typically only able to handle a very small number of queries, for example, 100 queries for data streams with millions of instances. Hence, we plan to make our active learning method more

practical by significantly reducing the number of queries while maintaining reasonable accuracy. We intend to introduce more dynamism into our querying system too so that new queries can be proposed on-the-fly when the expert rejects queries that he is not confident of answering.

References

1. Aggarwal, C., Han, J., Wang, J., Yu, P.: On Clustering Massive Data Streams: A Summarization Paradigm. Advances in Database Systems, vol. 31, pp. 9–38. Springer US (2007)
2. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: A framework for clustering evolving data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 81–92. VLDB Endowment (2003)
3. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: A framework for projected clustering of high dimensional data streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, pp. 852–863. VLDB Endowment (2004)
4. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: A framework for on-demand classification of evolving data streams. IEEE Transactions on Knowledge and Data Engineering 18(5), 577–589 (2006)
5. Amigo, E., Gonzalo, J., Artiles, J.: A comparison of extrinsic clustering evaluation metrics based on formal constraints. Information Retrieval 12(4), 461–486 (2009)
6. Basu, S., Bilenko, M., Mooney, R.J.: A probabilistic framework for semi-supervised clustering. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 59–68. ACM (2004)
7. Baum, E., Lang, K.: Query learning can work poorly when a human oracle is used. In: Proceedings of the IEEE International Joint Conference on Neural Networks, pp. 335–340. IEEE Press (1992)
8. Belkin, M., Niyogi, P., Sindhwani, V.: Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. The Journal of Machine Learning Research 7, 2399–2434 (2006)
9. Bifet, A., Holmes, G., Kirkby, R.: Moa: Massive online analysis. The Journal of Machine Learning Research 11, 1601–1604 (2010)
10. Blum, A., Chawla, S.: Learning from labeled and unlabeled data using graph mincuts. In: Proceedings of the Eighteenth International Conference on Machine Learning, pp. 19–26. Morgan Kaufmann Publishers Inc. (2001)
11. Cao, F., Ester, M., Qian, W., Zhou, A.: Density-based clustering over an evolving data stream with noise. In: Proceedings of the 2006 SIAM International Conference on Data Mining, pp. 328–339 (2006)
12. Chapelle, O., Sindhwani, V., Keerthi, S.S.: Optimization techniques for semi-supervised support vector machines. The Journal of Machine Learning Research 9, 203–233 (2008)
13. Chen, Y., Tu, L.: Density-based clustering for real-time stream data. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 133–142. ACM (2007)
14. Cuzzocrea, A.: CAMS: OLAPing Multidimensional Data Streams Efficiently. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 48–62. Springer, Heidelberg (2009)

15. Cuzzocrea, A.: Retrieving Accurate Estimates to OLAP Queries over Uncertain and Imprecise Multidimensional Data Streams. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 575–576. Springer, Heidelberg (2011)
16. Cuzzocrea, A., Chakravarthy, S.: Event-based lossy compression for effective and efficient olap over data streams. Data & Knowledge Engineering 69(7), 678–708 (2010)
17. Frank, A., Asuncion, A.: UCI machine learning repository (2010), http://archive.ics.uci.edu/ml
18. Han, J., Chen, Y., Dong, G., Pei, J., Wah, B., Wang, J., Cai, Y.: Stream cube: An architecture for multi-dimensional analysis of data streams. Distributed and Parallel Databases 18(2), 173–197 (2005)
19. Hulten, G., Spencer, L., Domingos, P.: Mining time-changing data streams. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 97–106. ACM (2001)
20. Joachims, T.: Making large-scale support vector machine learning practical, pp. 169–184. MIT Press (1999)
21. Klinkenberg, R.: Learning drifting concepts: Example selection vs. example weighting. Intelligent Data Analysis 8(3), 281–300 (2004)
22. Kriegel, H.-P., Kröger, P., Ntoutsi, I., Zimek, A.: Density Based Subspace Clustering over Dynamic Data. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 387–404. Springer, Heidelberg (2011)
23. Lewis, D.D., Gale, W.A.: A sequential algorithm for training text classifiers. In: The ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 3–12. ACM/Springer (1994)
24. Martin, E., Hans-Peter, K., Jörg, S., Xiaowei, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, pp. 226–231. AAAI Press (1996)
25. Masud, M.M., Jing, G., Khan, L., Jiawei, H., Thuraisingham, B.: A practical approach to classify evolving data streams: Training with limited amount of labeled data. In: ICDM, pp. 929–934 (2008)
26. Melville, P., Mooney, R.J.: Diverse ensembles for active learning (2004)
27. Park, N.H., Lee, W.S.: Grid-based subspace clustering over data streams. In: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management, pp. 801–810. ACM (2007)
28. Sattar, H., Ying, Y., Zahra, M., Mohammadreza, K.: Adapted one-vs-all decision trees for data stream classification. IEEE Transactions on Knowledge and Data Engineering 21, 624–637 (2009)
29. Sindhwani, V., Keerthi, S.S.: Large scale semi-supervised linear svms. In: ACM SIGIR, pp. 477–484. ACM (2006)
30. Tong, S., Koller, D.: Support vector machine active learning with applications to text classification. The Journal of Machine Learning Research 2, 45–66 (2002)
31. Vilalta, R., Rish, I.: A Decomposition of Classes via Clustering to Explain and Improve Naive Bayes. In: Lavrač, N., Gamberger, D., Todorovski, L., Blockeel, H. (eds.) ECML 2003. LNCS (LNAI), vol. 2837, pp. 444–455. Springer, Heidelberg (2003)
32. Wagstaff, K., Cardie, C., Rogers, S.: Constrained k-means clustering with background knowledge. In: Proceedings of the Eighteenth International Conference on Machine Learning, pp. 577–584. Morgan Kaufmann Inc. (2001)

33. Wang, H., Fan, W., Yu, P.S., Han, J.: Mining concept-drifting data streams using ensemble classifiers. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 226–235. ACM (2003)
34. Zhang, T., Ramakrishnan, R., Livny, M.: Birch: An efficient data clustering method for very large databases. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp. 103–114. ACM (1996)
35. Zhou, A., Cao, F., Qian, W., Jin, C.: Tracking clusters in evolving data streams over sliding windows. Knowledge and Information Systems 15(2), 181–214 (2008)
36. Zhu, X., Goldberg, A.B., Brachman, R., Dietterich, T.: Introduction to Semi-Supervised Learning. Morgan and Claypool Publishers (2009)

Efficient Single Pass Ordered Incremental Pattern Mining

Yun Sing Koh and Gillian Dobbie

Department of Computer Science, University of Auckland, New Zealand
{ykoh,gill}@cs.auckland.ac.nz

Abstract. Since the introduction of FP-growth using FP-tree there has been a lot of research into extending its usage to data stream or incremental mining. Most incremental mining adapts the Apriori algorithm. However, we believe that using a tree based approach would increase performance as compared to the candidate generation and testing mechanism used in Apriori. Despite this, FP-tree still requires two scans through a dataset. In this paper we present a novel tree structure called Single Pass Ordered Tree, SPO-Tree, that captures information with a single scan for incremental mining. All items in a transaction are inserted/sorted based on their frequency. The tree is reorganized dynamically when necessary. SPO-Tree allows for easy maintenance in an incremental or data stream environment.

Keywords: Incremental Mining, Frequent Pattern Mining, FP-Tree.

1 Introduction

Frequent pattern mining is an important area in data mining and knowledge discovery. Frequent Pattern Tree (FP-Tree) based Frequent Pattern Growth (FP-Growth) mining proposed by Han et al. [12] was an efficient technique to mine frequent patterns based on using a single prefix-tree. Since the introduction of FP-Tree, a large amount of research has been carried out to solve the frequent pattern mining problem more efficiently. The main benefit of applying FP-Tree was the performance gain due to the compact nature of the data structure. As frequent patterns can be generated by traversing the prefix-tree, this avoids multiple scans of the dataset. Prefix-tree enables fast computation for the support of all the frequent patterns as well. However, most traditional frequent pattern mining algorithms only cope with static datasets, which provide a snapshot in time of the patterns found. With the growing importance of data streams, mining frequent patterns from data stream has become an important and challenging problem for a range of applications including real-time surveillance systems, communication networks, Internet traffic, online transactions in the financial market or retail industry, electric power grids, and remote sensors.

Incremental data mining algorithms perform knowledge updating incrementally to amend and strengthen what was previously discovered. Incremental data

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 137–156, 2013.

mining algorithms incorporate dataset updates without having to mine the entire dataset again. Examples of some incremental mining techniques include the FUP algorithm [4], the adaptive algorithm [19], and IncSpan [3]. The collective idea in these approaches is that previously mined information should be utilized to reduce maintenance costs. In these approaches, intermediate results, such as frequent patterns, are stored and checked against newly added transactions. This reduces the computation time for maintenance. The algorithms mentioned above are Apriori-based [2] techniques, that depend on a "generate and test" mechanism. Whereas, many of the conventional prefix-tree data mining algorithms cannot handle large and growing data sets, as they require two dataset scans.

The key contribution of this work is proposing and developing a novel tree structure for maintaining frequent patterns in an incremental dataset. We propose a novel tree structure called SPO-Tree (Single Pass Ordered Tree) for incremental mining. The tree captures the content of the transactions dataset in a single pass. The main benefit of this is when a transaction is inserted, deleted or modified, our approach would not require a rescan of the entire dataset.

The rest of the paper is organized as follows. In the next section, we look at previous work in the area. Section 3 introduces our work for SPO-Tree. Our experimental results are presented in Section 4. Finally we summarize our research contributions in Section 5 and outline directions for future work.

This paper is an extension of our previous paper submitted to the 13th International Conference on Data Warehousing and Knowledge Discovery.

2 Related Work

Traditional OLAP and data mining methods require multiple scans of the data and are therefore infeasible for stream data applications. Thus recent research has been in both the area of OLAP stream analysis [8,10,9] and stream data mining techniques including stream classification[1], stream clustering [18], and stream pattern mining. In this section we will concentrate on pattern mining on data streams. We will look closely at four existing FP-tree based algorithms that handle stream mining, namely (i) FELINE algorithms with the CATS tree, (ii) the AFPIM algorithm, (iii) CanTree algorithms, and (iv) CP-Tree.

Cheung and Zaiane [5] proposed the Compressed and Arranged Transaction Sequence tree (CATS) for interactive mining. The CATS tree stems from the idea of using FP-tree to improve storage compression. The aim of the work was to build a compact tree representation. This proposed technique requires one pass through the dataset to build the tree. New transactions are added at the root level. At each level, items of the new transaction are compared with children (or descendant) nodes, the transaction is then merged with the node with the highest frequency level. The remainder of the transaction is then added to the merged nodes. This process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch in the last merged node. If the frequency of a node becomes higher than its ancestors, then it has to swap with the ancestors to ensure that its frequency is lower or

equal to the frequencies of its ancestors. In CATS we are required to find the right path for each of the new transactions to merge in. It also requires swaps and merges of nodes during the updates, as the nodes in CATS trees are locally sorted.

Table 1. Example of Dataset

TID	Transactions
t_1	{a,b,c,d,e}
t_2	{a,f,b,e}
t_3	{b}
t_4	{d,a,b}
t_5	{a,c,b}
t_6	{c,b,a,e}
t_7	{a,b,d}
t_8	{a,b,d}

Example CATS Tree. Consider the dataset in Table 1. Figure 1 shows the resulting CATS tree after each transaction is added. Here we highlight some of the important steps. From the insertion of transactions t_1 to t_2, common items in both transactions $\{a, b, e\}$ are merged into the existing tree. In this step, item e is swapped with its ancestors c and d. Since there are no further common items, the remaining item in t_2, f is added as a new branch of e. When t_3 arrives, item b is swapped with item a and moved up. The rest of the transactions are inserted in the same manner.

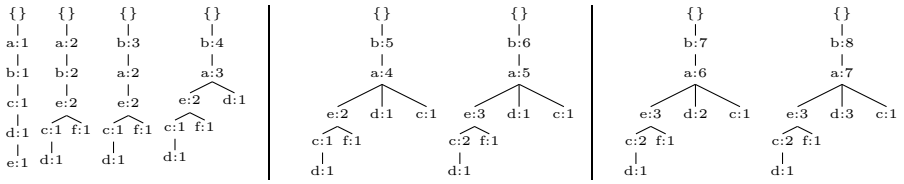


Fig. 1. CATS tree after each transaction is added

Leung et al. [14] proposed the Canonical-Ordered Tree for stream mining. This algorithm is designed so that it only requires one dataset scan. In CanTree, items are arranged in some canonical order, which can be determined by the user prior to the mining process or at runtime during the mining process. The items are arranged according to a prefix tree structure, thus are unaffected by the item frequency. CanTree generates compact trees if and only if the majority of the transactions contain a common pattern-base in canonical order. Otherwise, it may generate skewed trees with too many branches and hence with too many

nodes. Despite taking less time for tree construction, it requires more memory and more time for extracting frequent patterns from the generated tree.

Example CanTree Tree. Figure 2 shows the resulting CanTree tree after each transaction is added. Like CATS tree this technique keeps track of all items. In this tree, items are inserted in some form of canonical order (lexicographical or arrival). Items in t_1 are sorted in alphabetical order. The subsequent transactions are sorted in the same manner. In this step, item e is swapped with its ancestors c and d. Since there are no further common items the remaining item in t_2 which is item f, is added as a new branch to e. When t_3 arrives, item b is swapped with item a and moved up. The rest of the transactions are inserted in the same manner.

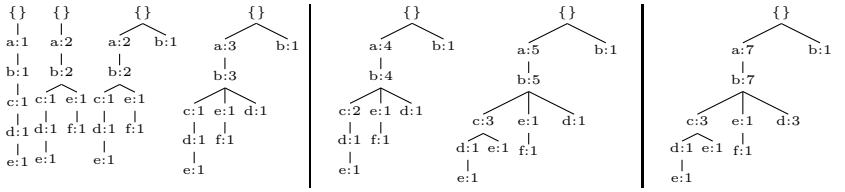


Fig. 2. CanTree in lexicographic order

Tanbeer et al. [20] proposed a tree structure, called CP-Tree that constructs a compact prefixed structure. CP-Tree has a frequency descending structure by capturing part by part data from the dataset and dynamically restructuring itself. The construction operation consists of two phases: insertion phase and restructuring phase. Insertion phase inserts transaction(s) into CP-Tree according to current sorted order of the item list and updates frequency based on the item list. Restructuring phase rearranges the list according to frequency-descending order of items and restructures the tree nodes according to the new ordered item list. The two phases are executed consecutively.

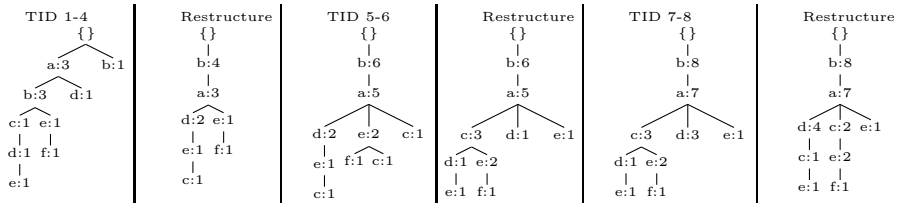


Fig. 3. CP Tree

Example CP Tree. Figure 3 shows the resulting CP-Tree after each transaction is added. In this example, we show that restructuring is carried out at the end of every block. In the insertion phase the transactions are inserted in the same

way as CanTree following the item-appearance order. In the restructuring phase, at first the items in the tree are rearranged in descending order, which is $I = \{b, a, d, e, c, f\}$ then the tree is restructured to that particular order. In the next insertion phase, items are inserted in the same order as in I. The restructuring phase is carried out as per the previous phases. All subsequent insertion and restructuring phases are carried out in a similar fashion. The authors showed that CP-Tree outperformed CanTree with a dense dataset. Despite CP-Tree taking a longer tree construction time, it outperformed the CanTree during the mining phase as it produced a more compact tree.

Koh and Shieh [13] developed the Adjusting FP-tree for Incremental Mining (AFPIM) algorithm. This algorithm uses the notion of FP-tree, whereby only the frequent items are kept in the tree. In this algorithm, an item is frequent if its support is no less than a threshold called preMinsup, which is lower than the usual minsup threshold. The frequent items are arranged in descending order of their frequency. Any insertion, deletion, or modification of transactions may affect the frequency of the items, and ordering of the items. As a correction step, the AFPIM algorithm reorders the tree using bubble sort. This may be computationally intensive when applied to all the branches affected by the change in item frequency. Incremental updating of items may also lead to the introduction of new items, which occurs when an infrequent item becomes frequent in the updated dataset. When faced with this scenario, the AFPIM algorithm has to rescan the entire dataset to build a new FP-tree.

The techniques mentioned above find the exact set of patterns. Another area of incremental mining in data stream mining [7,6], uses heuristics to approximate patterns found. This is to reduce complexity and number of patterns generated. The initial attempt to mine frequent patterns over the entire history of streaming data was proposed by Manku and Motwani [17]. They proposed two single-pass algorithms, Sticky-Sampling and Lossy Counting, both of which are based on the anti-monotone property; these algorithms provide approximate results with an error bound. Li et al. [15,16] proposed DSM-FI and DSM-MFI to mine frequent patterns using a landmark window. Each transaction is converted into k small transactions and inserted into an extended prefix-tree-based summary data structure called the item-suffix frequent itemset forest. Yu et al. [21] proposed an efficient algorithm to mine false negative or false positive frequent itemsets from high speed transactional data streams using the Chernoff bound. This approach uses a running error parameter to prune itemsets and a reliability parameter to control memory usage.

In the next section we discuss the details of our proposed approach, SPO-Tree, which mines exact sets of patterns.

3 Single Pass Ordered Tree (SPO-Tree)

The following is a formal definition of association rules. Let $I = \{i_1, i_2, \ldots, i_n\}$, be a set of items. A set $x = \{i_j, ...i_k\} \subseteq I$ where $j \leq k$ and $1 \leq j, k \leq n$ is called an itemset. A transaction is $T = (tid, Y)$ where tid is the transaction id and Y is

an itemset. If $X \subseteq Y$ is an itemset, then X occurs in T. A transactional dataset D over I is a set of transactions and $|D|$ is the number of transactions in the dataset. The support of an itemset X is the portion of transaction in the dataset that contains X, $supp(X) = \frac{count(X,D)}{|D|}$. An itemset is frequent if its support is no less than a user given support threshold called *minsup*. An association rule is an implication of the form $X \rightarrow Y$, where $X, Y \subset D$, and $X \cap Y = \emptyset$.

In this section we will discuss the preliminaries and the step-by-step construction of our SPO-Tree. The SPO-Tree has two phases:

Tree Construction Phase : This phase can be broken down into two additional phases, Insertion phase and Reorganization phase.

Insertion Phase : In the Insertion phase, items in a transaction are inserted into the tree based on a descending order of frequency.

Reorganization Phase : The tree is reorganized once the proportion of the edit distance of items in the sorted order changes above a certain threshold as shown in Equation 5.

Tree Mining Phase : The tree mining phase follows the FP-Growth mining technique. Once the SPO-Tree is constructed we use FP-growth to mine patterns with support above a user defined minsup. FP-Growth is used in the mining phase in both the CP-Tree and CanTree approaches.

Table 2. Example of Dataset

TID	Transactions
t_1	{a,b,c,d,e}
t_2	{a,b,e}
t_3	{b}
t_4	{d,a,b}
t_5	{a,c,b}
t_6	{c,b,a,e}
t_7	{a,b,d}
t_8	{a,b,d}

Example SPO Tree. Figure 4 to 6 show a step-by-step example of the SPO-Tree mechanism using the dataset in Table 2. Figure 4 shows the resulting tree after transactions t_1 to t_3 are added. The main difference between our technique and previous techniques, is that each transaction is sorted based on the order of frequency (and the order of appearance if the item has the same support) before insertion into the dataset.

In Figure 4, there are 3 tables. In the second and third tables, the first two columns so the items sorted according to frequency based on the current inserted transaction, and the last column shows the edit distance, d, between the sorted order based on the current transaction t_n and its previous transaction t_{n-1}. Here t_{n-1} represents the merged result of the previous $n-1$ transactions. In calculating the edit distance we consider the shift of the items upwards as positive edit distance and downwards as negative edit distance. In this example in transaction

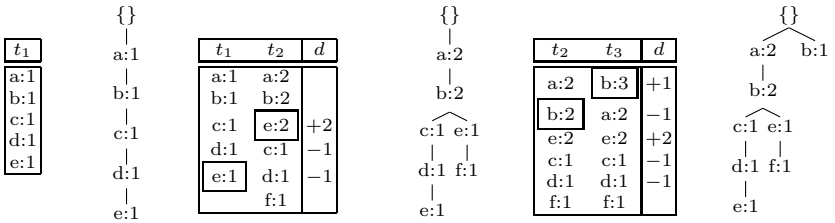


Fig. 4. Insertion of t_1 to t_3

t_1 item e was in the position 5 and in t_2 item e moved up to position 3. The edit distance of an item i can be defined as:

$$d_i = pos_{t_n}(i) - pos_{t_{n-1}}(i) \tag{1}$$

where pos_{t_n} represents the position of item i at transaction t_n and $pos_{t_{n-1}}$ represents the position of item i at transaction t_{n-1}.

The total edit distance of all items in the dataset is defined as:

$$\text{total edit distance} = \sum_{k=1}^{N} abs(d_{i_k}) \tag{2}$$

where N represents the unique number of items. For example, in the third table in Figure 4 the total absolute edit distance is 6 (1+1+2+1+1).

To find the maximum possible edit distance we defined the total maximum distance equation as:

$$\sum_{k=1}^{N} \max(k-1, N-k) \tag{3}$$

Using the same example the total maximum distance is 24 (5+4+3+3+4+5).

Given that in most cases, we will have a stable data stream the positions of items based on their frequencies should remain fairly stable. We believe that in some cases this would reduce the overall number of tree branches that need to be sorted as compared to CP-Tree.

Figure 5 shows the resorting of a tree after the insertion of t_3. We define a new measure called edit distance proportion:

$$\text{edit distance proportion} = \frac{\sum_{k=1}^{N} abs(d_{i_k})}{\sum_{k=1}^{N} \max(k-1, N-k)} \tag{4}$$

A tree is sorted once the fraction of the total absolute edit distance per total maximum edit distance is above a defined minimum edit distance. In this example this is set to 0.20. The resorting phase is prompted if:

$$\frac{\sum_{k=1}^{N} abs(d_{i_k})}{\sum_{k=1}^{N} \max(k-1, N-k)} \geq \text{minimum edit distance} \tag{5}$$

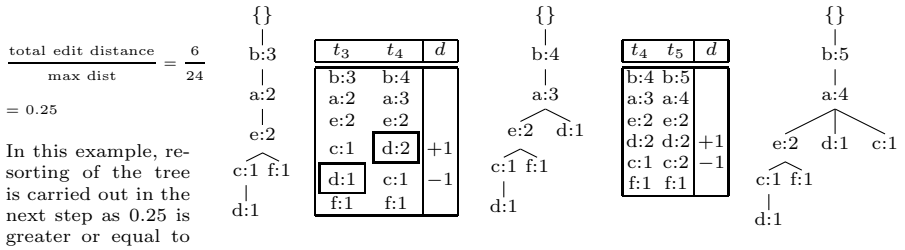


Fig. 5. Insertion of t_4 to t_5 with resorting

Table 3. Example of Calculating Total Maximum Edit Distance

Items	1	2	3	4	5	6	7	8	9	10
d_{10}										9
d_9									8	8
d_8								7	7	7
d_7							6	6	6	6
d_6						5	5	5	5	5
d_5					4	4	4	4	4	5
d_4				3	3	3	3	4	5	6
d_3			2	2	2	3	4	5	6	7
d_2		1	1	2	3	4	5	6	7	8
d_1	0	1	2	3	4	5	6	7	8	9
$\sum d$	0	2	5	10	16	24	33	44	56	70

Here N is the number of items, and d is the edit distance.

Table 3 shows how the maximum total edit distance is derived. From the table we notice that if we had 9 items the total maximum edit distance would be 56 and if we had 5 items the total edit distance would be 16. This is equivalent to the calculations of triangular numbers plus quarter squares. We can rewrite the total maximum edit distance as:

$$\sum_{k=1}^{N} \max(k-1, N-k) = \frac{k(k-1)}{2} + \text{floor}\left(\frac{k^2}{4}\right) \tag{6}$$

Following the sorting phase the edit distance, d, for the items are reset to 0. The subsequent transactions t_4, and t_5 are inserted as usual following the sorted frequency and appearance.

Figure 6 shows the insertion of transactions t_6 to t_8. The reorganizing phase is carried out when required. It does carry out an additional reorganizing phase at the end of each block of transactions.

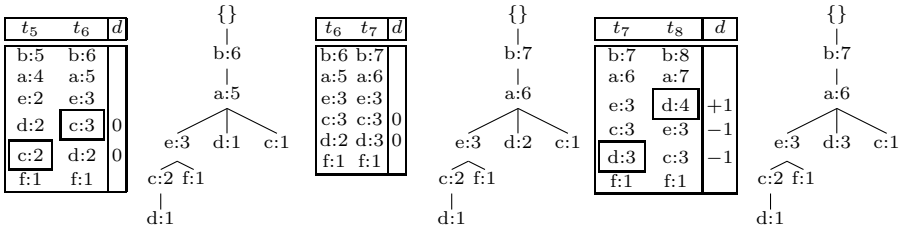


Fig. 6. Insertion of t_6 to t_8

SPO-Tree Algorithm. In this section, we look at the Tree Construction phase in the SPO-Tree algorithm. Figure 7 shows the pseudo code for our proposed Tree Construction phase. In the pseudo code, the *Frequency_Support_Order()* generates a comparator based on the descending frequency order. As a new transaction comes in we sort the transaction based on the comparator. The transaction will be sorted based on the current descending frequency order and then inserted into the tree.

Algorithm: Tree Construction for SPO-Tree

```
Input:  Transaction database DB
Output: SPO_Tree, SPO-Tree

SPO_Tree ← {}
cmp ← Frequency_Sorted_Order()
while (hasNextTransaction())
   Insertion Phase:
   trans ← getNextTransaction()
   SPO_Tree.Insert_Transaction(trans, cmp)
   Reorganization Phase:
   if (edit distance proportion≥ γ)
      // Generates a new comparator and reorganizes the tree,
      // SPO_Tree, using new comparator
      cmp ← Frequency_Support_Order()
      Sort(SPO_Tree, cmp)
   end if
end while
cmp ← Frequency_Sorted_Order()
Sort(SPO_Tree, cmp)
return SPO_Tree
```

Fig. 7. Pseudocode of SPO-Tree

In the algorithm, the minimum edit distance is calculated using Equation 5. If the *edit distance proportion* is greater or equal to a user defined minimum edit distance threshold, γ, the tree will be resorted based on the current descending frequency order, using the *Sort()* function. We carry out a final resorting before the SPO_{Tree} is passed to the tree mining phase. This is to ensure that all the items in the tree are sorted in a descending frequency order.

```
Algorithm: Sort()
Input:  SPO_Tree, cmp
Output: SPO_Tree
for each branch B in SPO_Tree
   for each path P in B
      If IsSorted(P)
         Sort_Branch(P, cmp)
      Else
         Sort_Path(P, cmp)
      End If
   End For
End For
return SPO_Tree
```

Fig. 8. Pseudocode of Sort()

```
Algorithm: Sort_Branch()
Input:  Branch P
Output: P
for each node n in P from leaf_p node
   for each subpath from n to leaf_k
      If the position of any item n and leaf_k is greater than n in cmp
         P = path (n,leaf_k)
         if IsSorted(P)
            Sort_Branch(P)
      Else
         P = path (root, leaf_k)
      Sort_Path(P, cmp)
return P
```

Fig. 9. Pseudocode of Sort_Branch()

We provide the *Sort*() algorithm in Figure 8. In general, our tree *Sort*() approach is similar to the branch sorting method used by CP-Tree. It first obtains the *cmp* by rearranging the items in a frequency-descending order and then performs the restructuring operation on SPO_{Tree}. This approach uses an temporary vector-based technique that performs the branch-by-branch restructuring process from the root of the tree SPO_{Tree}. Therefore, SPO_{Tree} may contain as

many branches as the number of children it has under the root. Each branch may consist of several paths. While sorting a branch, each path in the branch is sorted according to the new sort order. The path is removed from the tree into a temporary vector where it is sorted and then moved back into the tree. While processing a path, if it is found to already be in sorted order, the branch is skipped and no sorting operation is performed. Figure 9 shows the algorithm for *Sort_Branch*() algorithm. The *Sort_Path*() function uses merge sort to sort a path using a temporary vector according to the comparator *cmp*.

There are two advantages of our technique as compared to previous techniques. The first advantage is the tree is only reorganized when necessary. The second advantage of our technique is that we insert the incoming transactions based on a descending frequency order. Given the fact the incoming transactions from data streams will tend to be fairly stable, this means that the most transactions would have been inserted into our tree based on the correct final sorted order. This would reduce the amount of resorting necessary.

Complexity Analysis. The performance of the SPO-Tree relies on the degree of displacements of items between before and after the reorganization when the minimum edit threshold is reached, and the volatility of the stream. We use a merge sort approach to sort the nodes of any path. The degree of disorder does not have a large effect on the performance during sorting. The complexity of a merge sort is $O(n \log n)$, where n is the total number of items in the list. In a worst case scenario whereby a tree is constructed using transactions with the dataset that shares no prefix with each other, and the tree contains no sorted path. Thus the cost of sorting the tree is $O(mn \log n)$, where n is the average length of transactions and m is the number of paths in the tree. Given that the tree is required to be reorganized, k, times within the data stream, the complexity is $O(kmn \log n)$. Here k can be very high when dealing with a violate stream. In a worst case scenario a reorganization needs to be carried out at every $n.\gamma$ transactions. However, the worst case scenario is unlikely, as the items are inserted into the tree based on its current frequency descending order. In a stable data stream, the rate of items being introduced remains fairly constant. Thus there would be minimal changes to the order of the items based on frequency.

In the best case scenario, the cost of sorting the tree within the stream is $O(kn \log n)$. In the best case scenario the value for k is 1. In this scenario the tree only needs to be reorganized once and all the transactions are identical with length n.

Minimum Edit Threshold, γ. The minimum edit threshold allows the users to determines the amount of displacements of items based on a list of items in descending frequency order before resorting is carried out. It allows some logical control over the amount of re-sorting which is carried out. The SPO-Tree is resorted once the edit distance proportion is greater than a defined minimum edit threshold. In general the minimum edit threshold should be set to a low threshold *i.e.* 0.10. Setting a minimum edit distance to 0.10, allows 10% displacement before resorting is required. This means that if the number of unique items within

the dataset 1000, resorting would be carried out if 100 items are displaced from their correct descending frequency order.

Usually the rate of new items introduced is greater earlier in the stream. This means that the number of items added into the tree increases rapidly in the initial stages of reading a stream. This phenomenon is especially true if the incoming dataset is dense, and vice versa, this phenomenon may not occur in huge sparse datasets containing transactions with few co-occurrences among items. In these two cases, it can be suggested that a lower minimum edit threshold is used for a dense dataset, whereas a higher minimum edit threshold is used for a sparse dataset.

4 Experimental Results

In the experiments, we tested our program on real-world and synthetic datasets. The programs were written in Microsoft Visual C++, and run on Window 7 operating system on an Intel core 2 Duo machine in a time sharing environment with 4GB of main memory. In all experiments, runtime excludes I/O cost.

4.1 Real-World Datasets

We divided the testing into two sections which includes BMS-POS dataset [22] and several datasets from the UCI repository [11]. Using BMS-POS we carried out an in depth analysis of the performance of SPO-Tree versus CP-Tree. We later ran the SPO-Tree, CP-Tree, and CanTree on several different UCI datasets to examine the efficiency of the algorithms across a range of different datasets.

BMS-POS Dataset. Here we tested our SPO-Tree against CP-Tree using the BMS-POS dataset. In this experiment we divide the transactions in to the original datasets, and update portion of the dataset. Here we divided BMS-POS into 10 datasets with the initial block of 51,500 and an update of 50,000 subsequent blocks. In the first experiment we compare the tree construction time and the number of nodes produced by SPO-Tree as compared to CP-Tree.

Figure 10 shows the time for tree construction BMS-POS using CP-Tree and SPO-Tree. Results show that the overall restructuring efficiency notably increases as the dataset size increases. For CP-Tree we chose to use a range of the user given fixed slot for restructuring from 6,250 to 50,000. The largest fixed slot was fixed at 50,000 as the incremental blocks used were of size 50,000. We use a minimum edit distance of 0.10 for SPO-Tree.

Table 4 shows the number of nodes in the tree. As the number of nodes increases the time taken to mine will also inadvertently increase. From the table, both SPO-Tree and CP-Tree produce a similar number of nodes. On average SPO-Tree was faster than CP-Tree by 7.4% (minimum 1.1% to maximum 13.3%).

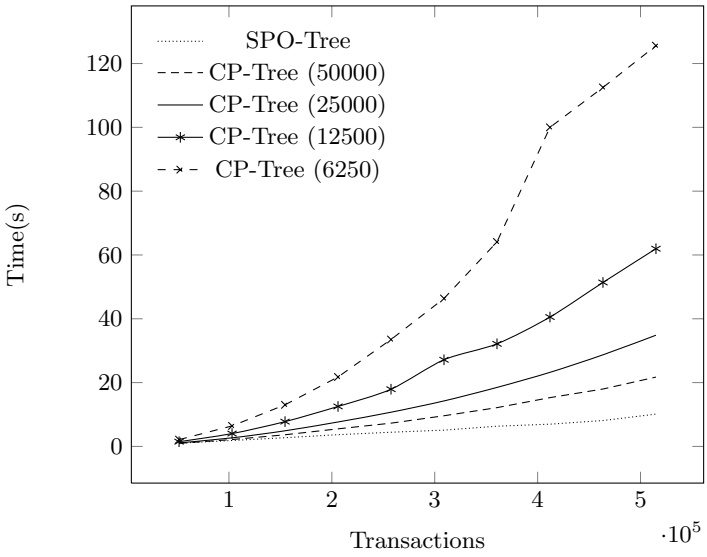


Fig. 10. Execution time for BMS-POS during Tree Construction Phase

Table 4. Comparison based on Number of Nodes in BMS-POS

Num Trans	SPO-Tree	CP-Tree			
		50000	25000	12500	6250
51500	208206	208196	208196	208196	208196
103000	381125	381152	381152	381152	381152
154500	545770	545794	545794	545794	545794
206000	698270	698292	698292	698292	698292
257500	811726	811726	811726	811726	811726
309000	924217	924175	924175	924175	924175
360500	1033709	1033639	1033639	1033639	1033694
412000	1204442	1204465	1204465	1204465	1204426
463500	1406072	1406124	1406124	1406124	1406029
515000	1593508	1593516	1593516	1593516	1593520

From Table 4 we note the number of nodes generated by the two techniques remains fairly close, but SPO-Tree is constructed more efficiently (as shown in Figure 10). In the last block where BMS-POS dataset has 515,000 transactions, the tree is reorganized seven times in SPO-Tree, whereas for CP-Tree it is reorganized 10 times for a user-given fixed slot of 50,000 and it is reorganized 62 times for a user given fixed slot of 6,250.

Next we tested how minsup value affects the runtime of the algorithms. Figure 12 shows the runtime for the SPO-Tree versus CP-Tree. We chose to

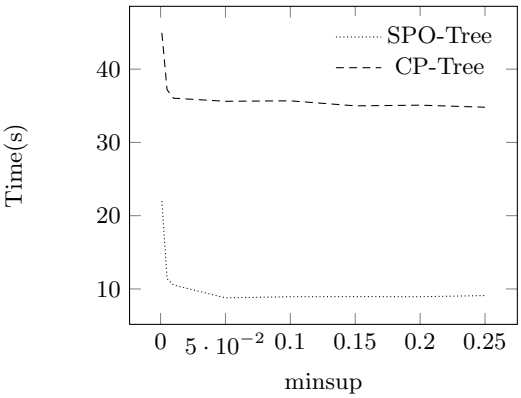


Fig. 11. Execution time by varying minsup

mine CP-Tree using a user-given fixed slot of 50,000. We noticed that when minsup decreases the runtime increased but overall the SPO-Tree was still more efficient than CP-Tree.

In the next experiment we test the effect choose to vary the minimum edit distance threshold of our algorithm. The edit distance threshold was set from 0.02 to 0.20. Notice that the time initially decreases when the minimum edit threshold increases however it increases slightly at the 0.14 mark. When a lower threshold was set the number of resorts carried out was higher. The number of resorts when the minimum edit distance is set to low is not optimal. When the minimum edit threshold increases the resorts are further delayed.

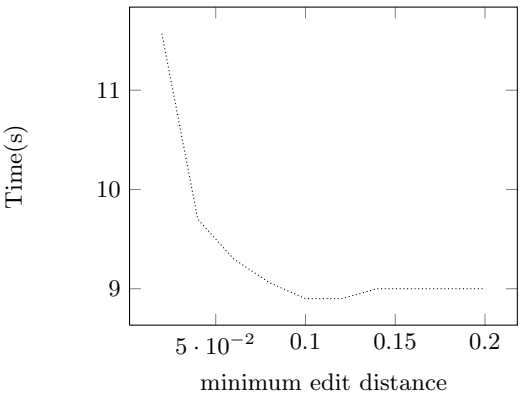


Fig. 12. Execution time by varying minimum edit distance

UCI Datasets. We also compared the execution of CP-Tree, CanTree, and SPO-Tree. Table 5 shows the results of execution time in seconds for these three algorithms. In these experiments there are on average nine restructuring phases for each of the datasets in CP-Tree. Overall the tree construction time for

Table 5. Comparison based on Execution Time

Dataset	CanTree			CP-Tree			SPO-Tree		
	Cons-truction	Mining	Total	Cons-truction	Mining	Total	Cons-truction	Mining	Total
Soybean	0.0	806.0	806.0	0.1	28.6	28.6	0.0	28.1	28.2
Mushroom	0.2	3.6	3.8	0.7	3.1	3.9	0.4	3.2	3.6
Adult	0.6	5.5	6.1	1.2	5.3	6.6	0.7	5.3	6.1
Accidents	3.3	70.4	73.8	16.4	62.5	79.0	5.8	62.9	68.8
Chess	0.2	1624.0	1624.3	0.9	543.0	543.9	0.6	534.6	535.2
Connect-4	2.1	10073.1	10075.3	11.5	669.8	681.3	4.5	657.8	662.4

CanTree is faster than CP-Tree or SPO-Tree. We use a minimum edit distance of 0.10 for SPO-Tree. However both CP-Tree and SPO-Tree produce a more compact representation of the tree, thus reducing the tree mining time. From the experiments we can glean that SPO-Tree is slightly faster than CP-Tree. The difference in time becomes more prominent as the density within a dataset increases. From the datasets below Soybean and Mushroom can be considered as the less dense datasets, and Connect-4 can be considered as a denser dataset.

Table 6. Comparison based on Number of Nodes in Tree

Dataset	CanTree	CP-Tree	SPO-Tree
Mushroom	45704	27165	27021
Adult	71554	56242	56242
Soybean	8336	4405	4390
Accidents	1742760	1393793	1392590
Chess	52074	38609	38610
Connect-4	812529	359969	359292

Table 6 shows the number of nodes in each tree. Overall SPO-Tree does produce a more compact tree than CanTree and CP-Tree. As the bottleneck in all these algorithms is the reordering of the tree, SPO-Tree only carries out a reordering when necessary and a quick final resort before mining. It allows us to have a compact representation without incurring a large reordering overhead.

4.2 Synthetic Datasets

We used datasets generated by the program developed at IBM Almaden Research Center. We tried to measure the robustness of our algorithm using two different sets of experiments. In the first set of experiments we look at how the characteristics of the itemsets in the dataset affect the SPO-Tree. In the second set of experiments we look at how drift affects SPO-Tree.

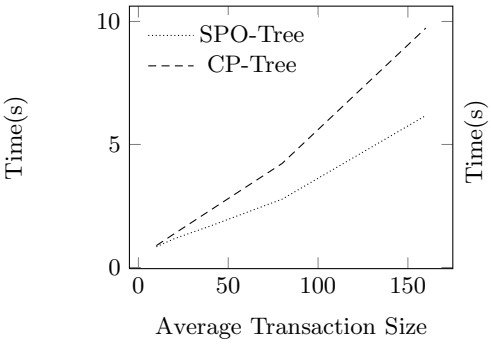


Fig. 13. Varying the average size of the transaction

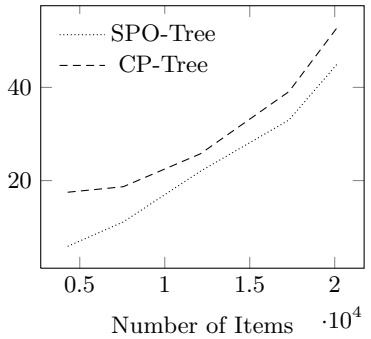


Fig. 14. Varying the number of items

Evaluating the Characteristics of Itemsets. In these experiments, we varied the (a) average length of the transactions and (2) number of unique items in the dataset. In both these experiments the CP-Tree block size was chosen, to replicate the number of organization phases of the SPO-Tree.

In the first experiment, we test the effect of varying the average length of the transactions. Figure 13 shows the total time taken by SPO-Tree and CP-Tree when the average transactions is varied. In this experiment we chose to use the minimum edit distance of 0.10, and the average block size of 2,000 for CP-Tree.

In the second experiment, we test the effect of varying the number of unique items. Figure 14 shows the total time taken by SPO-Tree and CP-Tree when the unique item is varied. In this experiment we use the minimum edit distance of 0.10, and the average of block size 500 for CP-Tree.

Evaluating the Effect of Drift. To evaluate the performance of our algorithm in capturing drift we modified the IBM data generator to inject known drift patterns to track drift detection. We divided our large itemsets into two groups. We select x random points in time which:

Introduce legitimate itemsets. The main motivation is to simulate emerging patterns. Given the original set of legitimate itemsets introduced, we hold off introducing a particular legitimate itemset until we reach a predetermined point.

Remove legitimate itemsets. The main motivation is to simulate disappearing patterns. When we reach a predetermined point, we reduce the occurrence of the legitimate itemset from the stream.

All the synthetic datasets using the same parameters where the average transaction length was 10, the number of frequent items was 100, the average length of a large item was 4 and the number of transaction was 1 million using 1000 unique items. Figure 15 shows the drift of the synthetic datasets. Each line within the graph represents the cumulative frequency of items of a group of large itemsets. The y-axis represents the cumulative frequency of the particular group. The fluctuations of the lines represent the changes of frequency of items within the group. The crossover of the two lines represents the drift occurring and possible changes of the order within the items based on their frequencies. From the figure we notice that Synthetic 1 to Synthetic 6 have a different number of change points. A change point occurs when there is a crossover of cumulative frequencies between the two lines. From the figure we, Synthetic 1 has 5 change points and Synthetic 3 has 12 change points. Synthetic 7 and 8 do not have change points between the groups, however there is still the possibility it may have an internal crossover within the group itself. This may happen as the individual frequencies of items within the group may fluctuate.

Table 7. Comparison based on Execution Time(s)

Dataset	SPO-Tree	CP-Tree			
		50K	100K	150K	200K
Synthetic 1	19.9	62.2	38.6	28.9	26.7
Synthetic 2	19.2	55.0	34.9	26.7	25.3
Synthetic 3	18.7	55.2	37.9	26.5	26.7
Synthetic 4	19.3	58.5	41.5	27.3	27.6
Synthetic 5	21.2	61.8	36.5	27.3	25.6
Synthetic 6	20.8	58.3	37.1	27.0	24.8
Synthetic 7	18.8	56.8	36.1	26.8	25.5
Synthetic 8	20.7	52.3	33.4	24.7	23.7

Table 7 shows that on average our approach was faster than CP-Tree by 22.7% (minimum 12.7% to maximum 30.0%) compared to CP-Tree with the interval of 200K. Overall the average number of times the reorganization phase was carried out by SPO-Tree was 7 times whereas CP-Tree was only 5 times. Despite having to reorganize the tree in using SPO-Tree, SPO-Tree is faster than CP-Tree.

During the implementation of both SPO-Tree and CP-Tree we cached the result of the comparison, so that sorting uses a faster comparison. When managing the header-list for the tree we kept a pointer to the list node, so that removing and adding elements from/to the list is $O(1)$. A copy of the program can be found at: `http://www.cs.auckland.ac.nz/~yunsing/SPO-Tree.html`.

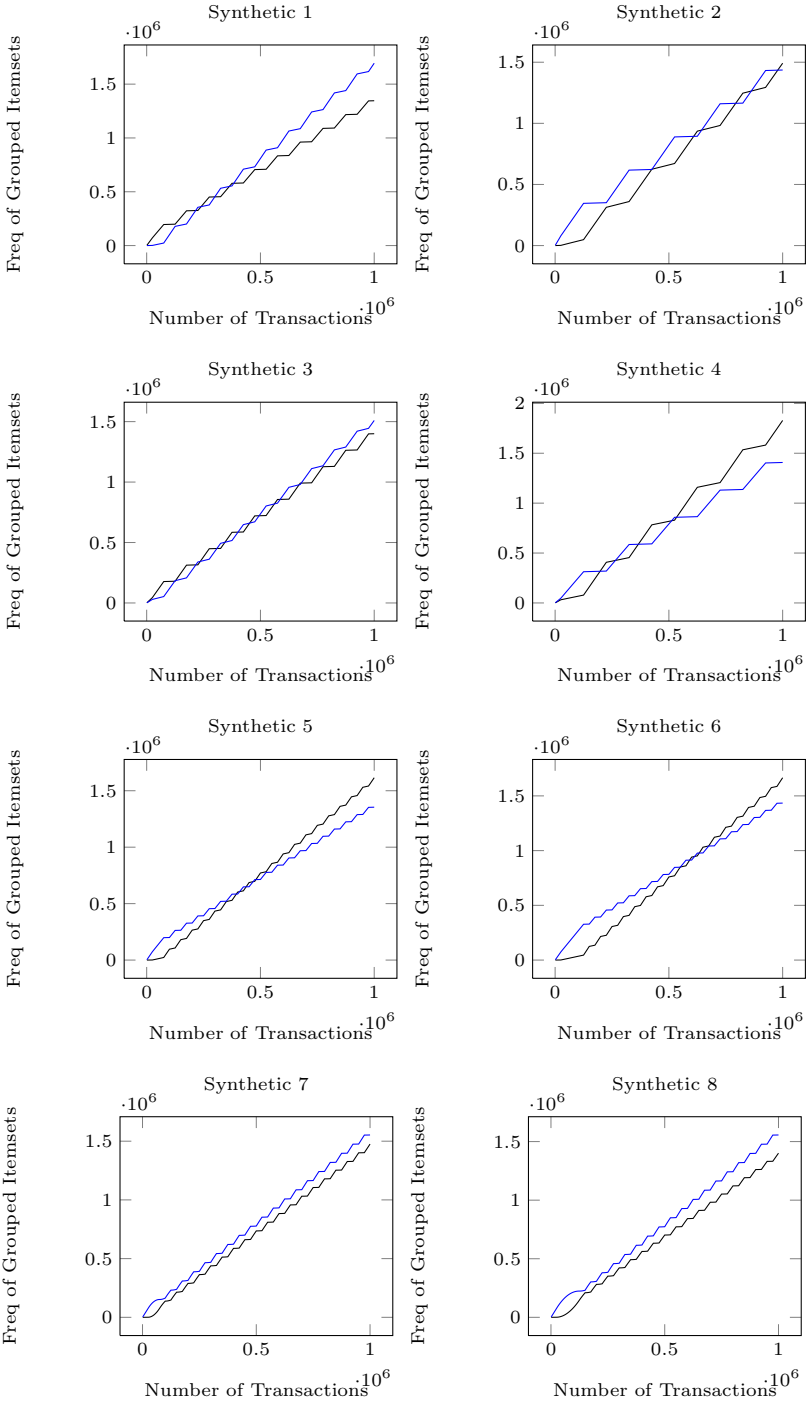


Fig. 15. Drift characteristics of the synthetic datasets

5 Conclusions and Future Work

A major contribution of SPO-Tree is that it builds an efficient single pass tree structure for FP-tree based incremental mining. The tree captures the content of the dataset and rearranges it into a more compact representation. We evaluated our algorithm and compared it with CP-Tree and CanTree. We have shown that SPO-Tree is faster because sometimes we have fewer reorganization. If there are more reorganization SPO-Tree remains faster as we are reorganizing the tree when it is optimal to do so. If a tree is not sorted when it is optimal to do so, the resorting of a tree becomes less efficient at the next round of sorting, as it becomes a large number of nodes/branches within the tree requires to be sorted. This will increase the time needed for the sorting to be carried out.

In our future work, we will also be adding an additional momentum parameter for each of the items, which will prevent resorting of the branches tree according to the current frequency as every transaction is processed, it will only resort the tree branches according to the current frequency once the momentum inertia threshold is surpassed.

References

1. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: On demand classification of data streams. In: Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2004, pp. 503–508. ACM, New York (2004)
2. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 207–216 (1993)
3. Cheng, H., Yan, X., Han, J.: Incspan: incremental mining of sequential patterns in large database. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2004, pp. 527–532. ACM, New York (2004)
4. Cheung, D.W.L., Han, J., Ng, V., Wong, C.Y.: Maintenance of discovered association rules in large databases: An incremental updating technique. In: Proceedings of the Twelfth International Conference on Data Engineering, ICDE 1996, pp. 106–114. IEEE Computer Society, Washington, DC (1996)
5. Cheung, W., Zaiane, O.: Incremental mining of frequent patterns without candidate generation or support constraint. In: Proceedings of the Seventh International Database Engineering and Applications Symposium, pp. 111–116 (2003)
6. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R.: Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowl. Inf. Syst. 10, 265–294 (2006)
7. Chi, Y., Wang, H., Yu, P., Muntz, R.: Moment: maintaining closed frequent itemsets over a stream sliding window. In: Fourth IEEE International Conference on Data Mining (ICDM 2004), pp. 59 –66 (2004)
8. Cuzzocrea, A.: CAMS: OLAPing Multidimensional Data Streams Efficiently. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 48–62. Springer, Heidelberg (2009)

9. Cuzzocrea, A.: Retrieving Accurate Estimates to OLAP Queries over Uncertain and Imprecise Multidimensional Data Streams. In: Bayard Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 575–576. Springer, Heidelberg (2011)
10. Cuzzocrea, A., Chakravarthy, S.: Event-based lossy compression for effective and efficient olap over data streams. Data Knowl. Eng. 69(7), 678–708 (2010)
11. Frank, A., Asuncion, A.: UCI machine learning repository (2010), http://archive.ics.uci.edu/ml/
12. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. SIGMOD Rec. 29, 1–12 (2000)
13. Koh, J.-L., Shieh, S.-F.: An Efficient Approach for Maintaining Association Rules Based on Adjusting FP-Tree Structures1. In: Lee, Y., Li, J., Whang, K.-Y., Lee, D. (eds.) DASFAA 2004. LNCS, vol. 2973, pp. 417–424. Springer, Heidelberg (2004)
14. Leung, C.K.S., Khan, Q.I., Li, Z., Hoque, T.: CanTree: A canonical-order tree for incremental frequent-pattern mining. Knowl. Inf. Syst. 11, 287–311 (2007)
15. Li, H.F., Lee, S.Y., Shan, M.K.: Online mining (recently) maximal frequent itemsets over data streams. In: Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, RIDE 2005, pp. 11–18. IEEE Computer Society, Washington, DC (2005)
16. Li, H.F., Shan, M.K., Lee, S.Y.: DSM-FI: An efficient algorithm for mining frequent itemsets in data streams. Knowledge and Information Systems 17, 79–97 (2008)
17. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB 2002, pp. 346–357. VLDB Endowment (2002)
18. O'Callaghan, L., Mishra, N., Meyerson, A., Guha, S., Motwani, R.: Streaming-data algorithms for high-quality clustering. In: Proceedings of the 18th International Conference on Data Engineering, pp. 685–694 (2002)
19. Sarda, N.L., Srinivas, N.V.: An adaptive algorithm for incremental mining of association rules. In: Proceedings of the 9th International Workshop on Database and Expert Systems Applications, DEXA 1998, pp. 240–245. IEEE Computer Society, Washington, DC (1998)
20. Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K.: CP-Tree: A Tree Structure for Single-Pass Frequent Pattern Mining. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 1022–1027. Springer, Heidelberg (2008)
21. Yu, J.X., Chong, Z., Lu, H., Zhang, Z., Zhou, A.: A false negative approach to mining frequent itemsets from high speed transactional data streams. Information Sciences 176(14), 1986–2015 (2006)
22. Zheng, Z., Kohavi, R., Mason, L.: Real world performance of association rule algorithms. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2001, pp. 401–406. ACM, New York (2001)

Finding Interesting Rare Association Rules Using Rare Pattern Tree

Sidney Tsang, Yun Sing Koh, and Gillian Dobbie

The University of Auckland
stsa027@aucklanduni.ac.nz,
{ykoh,gill}@cs.auckland.ac.nz

Abstract. Most association rule mining techniques concentrate on finding frequent rules. However, rare association rules are in some cases more interesting than frequent association rules since rare rules represent unexpected or unknown associations. All current algorithms for rare association rule mining use an Apriori level-wise approach which has computationally expensive candidate generation and pruning steps. We propose RP-Tree, a method for mining a subset of rare association rules using a tree structure, and an information gain component that helps to identify the more interesting association rules. Empirical evaluation using a range of real world datasets shows that RP-Tree itemset and rule generation is more time efficient than modified versions of FP-Growth and ARIMA, and discovers 92-100% of all the interesting rare association rules. Additional evaluation using synthetic datasets also shows that RP-Tree is more efficient, in addtion to showing how the execution time of RP-Tree changes with transaction length and rare-item itemset size.

Keywords: Rare Pattern Mining, FP-Growth, Information Gain.

1 Introduction

Association rule mining techniques are used to extract useful information from databases. The set of association rules that can be extracted from a database can be divided using a support threshold into frequent and rare association rules. Both frequent and rare association rules present different information about the database from which they are found, since frequent rules focus on patterns that occur frequently, while rare rules focus on patterns that occur infrequently. In many domains, events that occur frequently may be less interesting than events that occur rarely, since frequent patterns represent the known and expected while rare patterns may represent unexpected or previously unknown associations, which are useful to domain experts. For example, in the area of medicine, the expected, frequent responses to medications are less interesting than rare responses which may indicate unusual side-effects, adverse reactions or drug interactions. This is also true in other domains, such as the detection of fraudulent financial transactions and network intrusions [1].

Algorithms such as Apriori [2] can be used to find both frequent and rare association rules, but the latter requires the minimum support threshold to be set to a low value. However this may cause a combinatorial explosion of itemsets as the number of patterns

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 157–173, 2013.

that meet minimum support becomes insurmountable. Given n items, the number of possible itemsets approaches $2^n - 1$.

From this, we can see that frequent itemsets and association rules represent only a small fraction of that which can be generated from any non-trivial dataset. Using frequent association rule mining methods, it is not possible to identify and analyse the vast majority of itemsets and rules since they fall below the frequent threshold. As a result, many potentially useful rare association rules are overlooked. However, at the same time, it is not very helpful to simply generate all rare itemsets and rare association rules, since the vast majority of the huge number of rules will not be useful [3]. It is therefore helpful to be able to avoid generating rare itemsets which do not produce interesting association rules. We define an algorithm called RP-Tree that avoids generating a specific type of uninteresting rare itemset. We first define the types of rare itemsets.

There are three possible types of rare itemsets: first, itemsets which consist of rare items only; second, itemsets which consist of both rare and frequent items; and third, itemsets which consist of only frequent items which fall below the minimum support threshold. We refer to itemsets of the first and second types as *rare-item itemsets*. Rare-item itemsets are generally more interesting than itemsets of the third type, which we call *non-rare-item itemsets*. This is because frequent items occur commonly in the database, and there may be many non-rare-item itemsets that do not represent any interesting connection between items since the items only occurred together by chance. Empricial evidence for the claim that rare-item itemsets are more interesting than non-rare-item itemsets is given in Section 4. For now, we will illustrate this with the following simple example.

Suppose a database of patient symptoms contains the rare itemsets "1:{elevated heart rate, fever, skin bruises, low blood pressure}" and "2:{muscle pain, tinnitus, sneezing, heartburn}", where all items other than "low blood pressure" are frequent items. Itemset 1 is a rare-item itemset, and itemset 2 is a non-rare-item itemset. Itemset 1 contains a subset of the symptoms of sepsis, will produce a rule such as "{elevated heart rate, fever, skin bruises} $\rightarrow$ low blood pressure" that highlights the association between the different three former symptoms with low blood pressure, which is a symptom of severe sepsis. However, rules generated from itemset 2, such as "{muscle pain, tinnitus, heartburn} $\rightarrow$ sneezing" does not give any useful information, since all these symptoms are individually common, and have simply occured together by chance.

The key contribution of the paper is a novel algorithm called RP-Tree that finds rare-item itemsets using a tree structure. Unlike previous level-wise approaches, RP-Tree does not need to generate and test all plausible combinations of rare itemsets. We empirically show that RP-Tree finds rare itemsets and association rules more efficiently than existing algorithms, and identifies 92-100% of rare association rules that meet a confidence and lift threshold. The second contribution of this paper is an extension to RP-Tree that reduces the number of uninteresting association rules generated by excluding items that are poor at predicting the occurrence of rare items. To our knowledge, RP-Tree is the first rare association rule mining algorithm that uses a tree structure.

The paper is organized as follows. In Section 2 we look at previous work in the area of rare association rule mining. In Section 3 we present basic concepts and definitions

for rare association rule mining and discuss our novel RP-Tree approach. Section 4 describes the experimental results with UCI datasets. Section 5 describes the experimental results with synthetic datasets. Finally, Section 6 concludes the paper.

2 Related Work

Current rare itemset mining approaches are based on level-wise exploration of the search space similar to the Apriori algorithm [2]. In Apriori, k-itemsets (itemsets of cardinality k) are used to generate $k+1$-itemsets, which are then pruned using the downward closure property. Apriori terminates when there are no new $k+1$-itemsets remaining after pruning. MS-Apriori, Rarity, AfRIM, ARIMA and Apriori-Inverse are four algorithms that detect rare itemsets. They all use level-wise exploration similar to Apriori.

MS-Apriori [4] uses a bottom-up approach similar to Apriori. In MS-Apriori, each item can be assigned a different minimum item support value (MIS). Rare items can be assigned a low MIS, so that during candidate pruning, itemsets that include rare items are more likely to be retained and participate in rule generation. At the same time, the MIS for frequent items can remain high, which prevents the number of generated itemsets from exploding. This enables MS-Apriori to identify rules with high support, and rules with low support that includes one or more rare items. However, MS-Apriori is not able to identify rules which have low support and high confidence, and which only include frequent items. This is because the itemsets required to generate those rules will have been removed due to high MIS values.

Troiano et al. [5] notes that rare itemsets are at the top of the search space, so that bottom-up algorithms must first search through many layers of frequent itemsets. To avoid this, Troiano et al. proposed the Rarity algorithm that begins by identifying the longest transaction within the database and uses them to perform a top-down search for rare itemsets, thereby avoiding the lower layers that only contain frequent itemsets. In Rarity, potentially rare itemsets (candidates) are pruned in two different ways. Firstly, all k-itemset candidates that are the subset of any of the frequent $k+1$-itemsets are removed as a candidate, since they must be frequent according to the downward closure property. Secondly, the remaining candidates have their supports calculated, and only those that have a support below the threshold are used to generate the $k-1$-candidates. The candidates with supports above the threshold are used to prune $k-1$-candidates in the next iteration.

Adda et al. [1] proposed AfRIM that uses a top-down approach similar to Rarity. Rare itemset search in AfRIM begins with the itemset that contains all items found in the database. Candidate generation occurs by finding common k-itemset subsets between all combinations of rare $k+1$-itemset pairs in the previous level. Candidates are pruned in a similar way to the Rarity algorithm. Note that AfRIM examines itemsets that have zero support, which may be inefficient.

Szathmary et al. [6] proposed two algorithms that together can mine rare itemsets. As part of those two algorithms, Szathmary et al. defines three types of itemsets: minimal generators (MG), which are itemsets with a lower support than its subsets; minimal rare generators (MRG), which are itemsets with non-zero support and whose subsets are all

frequent; and minimal zero generators (MZG), which are itemsets with zero support and whose subsets all have non-zero support. The first algorithm, MRG-Exp, finds all MRG by using MGs for candidate generation in each layer in a bottom up fashion. The MRGs represent a border that separates the frequent and rare itemsets in the search space. All itemsets above this border must be rare according to the antimonotonic property. The second algorithm, ARIMA, uses these MRGs to generate the complete set of rare itemsets. This is done by merging two k-itemsets with $k-1$ items in common into a $k+1$-itemset. ARIMA stops the search for non-zero rare itemsets when the MZG border is reached, since above that there are only zero rare itemsets. There is previous work on the search and use of generators in itemset mining algorithms [7,8].

Apriori-Inverse [9] proposed by Koh et al. is used to mine perfectly rare itemsets, which are itemsets that only consist of items below a maximum support threshold (maxSup). Apriori-Inverse is similar to Apriori, except that at initialisation, only 1-itemsets that fall below maxSup are used for generating 2-itemsets. Since Apriori-Inverse inverts the downward-closure property of Apriori, all rare itemsets generated must have a support below maxSup. In addition, itemsets must also meet an absolute minimum support, for example 5, in order for them to be used for candidate generation. Since the set of perfectly rare-rules may only be a small subset of rare itemsets, Koh et al. also proposed several modifications that allow Apriori-Inverse to find near-perfect rare itemsets. The methods are based on increasing maxSup during itemset generation, but using the original maxSup during rule generation.

All of the above algorithms use the fundamental Apriori approach, which has potentially expensive candidate generation and pruning steps. In addition, these algorithms attempt to identify all rare itemsets, and as a result spend a significant amount of time searching for non-rare-item itemsets. However, we will show that these non-rare-item itemsets do not tend to give us interesting rare association rules.

The proposed RP-Tree algorithm is an improvement over these existing algorithms in three ways. Firstly, RP-Tree avoids the expensive itemset generation and pruning steps by using a tree data structure, based on FP-Tree, to find rare patterns. Secondly, RP-Tree focusses on rare-item itemsets which generate interesting rules and does not spend time looking for uninteresting non-rare-item itemsets. Thirdly, RP-Tree is based on FP-Growth, which is efficient at finding long patterns, since the task is divided into a series of searches for short patterns. This is especially beneficial since rare patterns tend to be longer than frequent patterns.

3 Rare Pattern Tree Mining

In this section we first discuss basic concepts and the definition of rare-item itemsets. We then describe our proposed RP-Tree algorithm and present a simple example. Finally we describe the modification to RP-Tree using an information gain threshold.

3.1 Basic Concept: Rare Itemsets

Let the set of items $\mathcal{I} = \{i_1, i_2, ...i_m\}$, and the transactional database $\mathcal{D} = \{t_1, t_2, ...t_n\}$ where every $t \subseteq \mathcal{I}$. An association rule is an implication $X \rightarrow Y$ such that $X \cup Y \subseteq \mathcal{I}$

and $X \cap Y = \emptyset$. X is the antecedent and Y is the consequent of the rule. The *support* of $X \rightarrow Y$ in $\mathcal{D}$ is the proportion of transactions in $\mathcal{D}$ that contains $X \cup Y$. The *confidence* of $X \rightarrow Y$ is the proportion of transactions in $\mathcal{D}$ containing X that also contains Y. The *lift* of $X \rightarrow Y$ is confidence($X \rightarrow Y$) / support (Y).

The minRareSup threshold is a noise filter, whereby items that are below this threshold are considered as noise. An itemset is a rare itemset if it has support less than the minimum frequent support threshold (minFreqSup) but above or equal to the minimum rare support threshold (minRareSup). As mentioned in Section 1, rare itemsets can be divided into types: *rare-item itemsets* which refer to itemsets that consist of only rare items and itemsets that consist of both rare and frequent items; and *non-rare-item itemsets* which consist of only frequent items which fall below the minimum support threshold.

For instance, suppose there were 4 items $\{a, b, c, x\}$ with supports a = 0.80, b = 0.30, c = 0.50, and x = 0.12, with minFreqSup = 0.15 and minRareSup = 0.05. If the itemset $\{a, b, c\}$ had a support of 0.09, then this itemset would be a non-rare-item itemset ((1) above) since all items are frequent, and its support lies between minFreqSup and minRareSup. The itemset $\{a, x\}$ would be a rare-item itemset ((2) above) assuming that the support of $\{a, x\} > 0.05$, since the itemset includes the rare item x.

Formally, an itemset X is a *rare itemset* iff

$$\text{support}(X) < \text{minFreqSup}, \text{support}(X) \geq \text{minRareSup}$$

An itemset X is a *non-rare-item itemset* iff X is a *rare itemset* and

$$\forall x \in X, \text{support}(x) \geq \text{minFreqSup}$$

An itemset X is a *rare-item itemset* iff X is a *rare itemset* and

$$\exists x \in X, \text{support}(x) < \text{minFreqSup}$$

The values of the minFreqSup and minRareSup thresholds are set depending on the characteristics of the dataset. However, the use of noise detection methods such as the χ^2 test can be helpful in selecting an appropriate minRareSup threshold.

3.2 RP-Tree Algorithm

FP-Growth, proposed by Han et al. [10] is a frequent itemset mining algorithm which uses a frequent-pattern tree (FP-Tree) to store a set of database transactions and reduces the required number of database scans to 2. The first scan is used to find the set of items in the database with support over the minimum frequent support threshold; the second is used to construct the initial FP-tree.

The RP-Tree algorithm, shown in Algorithm 1, is a modification of the FP-Growth algorithm. RP-Tree performs one database scan to count item support, similar to FP-Growth. During the second scan, RP-Tree uses only the transactions which include at least one rare item to build the initial tree, and prunes the others, since transactions that only have non-rare items cannot contribute to the support of any rare-item itemset. For example, if $\{x, y, z\}$ was the set of rare items for a given database, minRareSup and

minFreqSup, a transaction will have to contain at least one of x, y or z to avoid being pruned.

Note that the ordering of items in each transaction during insertion into the initial tree is according to the item frequency of the original database (and not the database with pruned transactions). This is because rare items in the reduced database may have higher supports than frequent items. If item frequencies of the reduced database were used for transaction item ordering, a frequent item may become the child of a rare item, which invalidates property 1 below.

Using this initial tree, RP-Tree constructs conditional pattern bases and conditional trees for each rare item only. Each conditional tree and the corresponding rare item are then used as arguments for FP-Growth (simplified version shown in Algorithm 2). The threshold used to prune items from the conditional trees is minRareSup. The union of the results from each of these calls to FP-Growth is a set of itemsets that each contain a rare-item, or rare-item itemsets.

The result of RP-Tree is the complete set of rare-item itemsets. This is because:

1. Rare-items will never be the ancestor of a non-rare item in the initial tree due to the tree construction process.
2. All itemsets that involve a particular item a can be found by examining all nodes of a and the nodes of all items that have a lower support than a in the initial tree.

Since RP-Tree examines all rare-item nodes in the initial tree, and all nodes that have a lower support than a rare-item are themselves rare items, RP-Tree must find all rare-item itemsets.

Algorithm 1. RP-Tree

Input: $\mathcal{D}, minRareSup, minFreqSup$;
Output: $results$; // Set of rare-item itemsets

Initialisation:
$allItems \leftarrow \{$all unique items in $\mathcal{D}\}$;
countSupport($allItems$); // First scan of database
$rareItems \leftarrow \{i \in allItems \mid i.supp \geq minRareSup \wedge i.supp < minFreqSup\}$;
$rareItemTrans \leftarrow \{t \in \mathcal{D} \mid \exists r \cdot r \in rareItems \wedge r \in t\}$;
$tree \leftarrow$ constructTree($rareItemTrans$); // Second scan of database

Mining:
$results = \emptyset$;
for item a in $tree$ **do**
  if $a \in rareItems$ **then**
    construct a's conditional pattern-base and then a's conditional FP-Tree $Tree_a$;
    $results \leftarrow results \cup$ FP-Growth($Tree_a, a$);
  end if
end for
return $results$;

Algorithm 2. FP-Growth (without single prefix path optimisation)

Input: $tree, \alpha$;
Output: $results$; // All itemsets generated from $tree$

$results \leftarrow \emptyset$;
for item a in $tree$ **do**
  generate pattern $\beta \leftarrow a \cup \alpha$ with $support = a.support$;
  $results \leftarrow results \cup \beta$
  construct βs conditional pattern-base and then β's conditional FP-tree $tree_\beta$;
  if $Tree_\beta \neq \emptyset$ **then**
    $results \leftarrow results \cup$ FP-Growth$(tree_\beta, \beta)$;
  end if
end for
return $results$;

RP-Tree Example. Applying RP-Tree to database $\mathcal{D}$ in Table 1, the support ordered list of all items is $\langle$(a:7), (i:6), (b:5), (c:4), (l:4), (d:3), (f:3), (e:2), (g:2), (h:1), (j:1), (k:1), (m:1)$\rangle$. Using minFreqSup = 4 and minRareSup = 1, only the items $\{d, f, e, g\}$ are rare, and included in $rareItems$.

During construction of the initial RP-Tree, only transactions 1, 3, 4, 5, and 6 are used, since the remaining transactions do not contain any rare items and cannot contribute to any of the result itemsets. In addition, since the support of items h, j, k and m falls below minRareSup, these items are ignored during RP-Tree construction. The initial tree constructed using FP-Growth, which only ignores items that fall below minRareSup, will use all transactions, as shown in Figure 1. This tree has 8 additional nodes compared to the tree built using RP-Tree from the reduced transaction set (shown in Figure 2(a)). The additional nodes are frequent items that correspond to transactions pruned by RP-Tree. To find the rare-item itemsets, the initial RP-Tree is used to build conditional pattern bases and conditional RP-Trees for each rare item $\{d, f, e, g\}$. The conditional tree for item g is shown in Figure 2(b). Each of the conditional RP-Trees and the conditional item are then used as parameters for the FP-Growth algorithm, for example, FP-Growth($Tree_g, g$).

Table 1. Transaction database $\mathcal{D}$

TID	Transactions	TID	Transactions	TID	Transactions
1	{a, b, c, d, f}	5	{a, c, e, f}	9	{i, l, k}
2	{a, c}	6	{a, b, d, g}	10	{i, b, l}
3	{a, c, d, f, g}	7	{i, b, c, l, j}	11	{i, m}
4	{a, b, e, h}	8	{a, i, l}	12	{i}

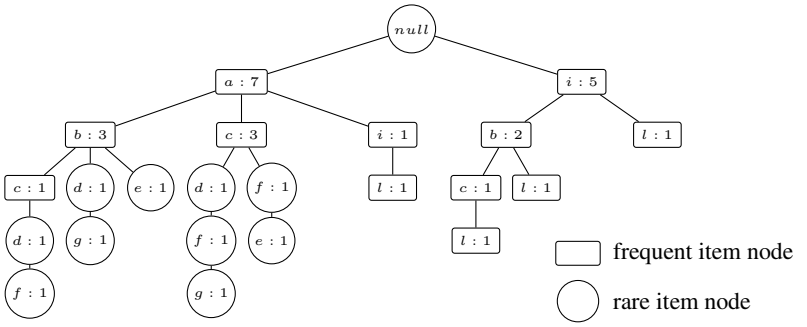
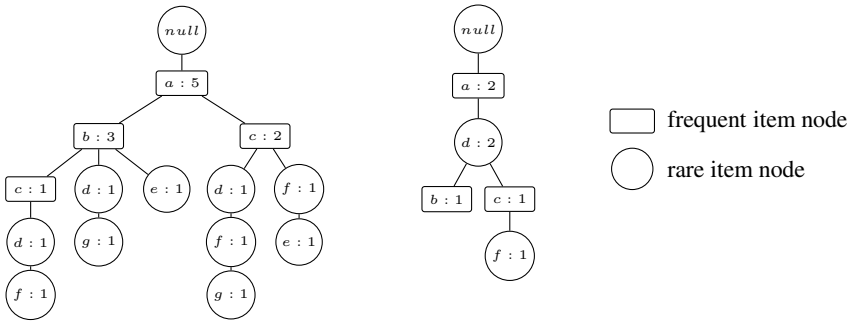


Fig. 1. Pattern tree constructed from database $\mathcal{D}$ using FP-Tree



(a) Initial tree constructed from $\mathcal{D}$ (b) Conditional tree, $Tree_g$

Fig. 2. Pattern trees constructed from database $\mathcal{D}$ using RP-Tree

3.3 RP-Tree with Information Gain

Rare rules that predict the occurrence of rare-items are more interesting than rules that predict the occurrence of frequent items. To more selectively generate this type of rule, RP-Tree has been extended using an information gain component (RP-Tree-IG). Information gain is commonly used in classification to measure how well an attribute predicts the occurence of particular classes. In RP-Tree-IG, information gain is used to ignore frequent items that are poor predictors of rare items, and itemsets containing these frequent items are not generated by RP-Tree-IG.

Transactions that contain more than one rare item (and class) are converted into multiple transactions during the information gain calculation so that each transaction contains only 1 rare item. For example, $\{a, b, d, e\}$, where d and e are rare, is split into $\{a, b, d\}$ and $\{a, b, e\}$.

Information gain [11] is calculated as: $\text{IG}(X) = \text{Entropy}(Y) - \text{Entropy}(Y \mid X)$ where Y is the set of rare items, and X is a frequent item. Frequent items that do not have an information gain higher than a pre-defined threshold are not used for itemset generation. Specifically, line 11 in Algorithm 1 and line 4 in Algorithm 2 become:

$$\text{for item } a \in tree \text{ where } IG(a) \geq minIG$$

where $minIG$ is the minimum information gain threshold an item must meet to be used for generating itemsets. Using the previous example, the classes Y are $\{d, f, e, g\}$, and attributes X are $\{\{a\}, \{b\}, \{c\}\}$. The information gain of c is $IG(c) = 1.971 - 1.842 = 0.129\, bits$. If, for example, $minIG = 0.1$, then c would be used for generating itemsets.

4 Experimental Results with UCI Datasets

In our experiments we compared the performance of ARIMA, FP-Growth, and RP-Tree with and without the information gain component. We also generated rules from these itemsets and compared the quality of these rules using the seven interest measures examined in [12]: χ^2, lift, confidence (all and max), coherence, cosine and Kulczynski (abbreviated to kulc). The equations for calculating these measures are shown in Table 2. All algorithms were implemented in Java and executed on an Intel Core 2 Duo 2.33 GHz machine with 4GB of RAM running Windows 7.

In these experiments ARIMA and FP-Growth were modified in order to obtain comparable results within a reasonable time. The ARIMA algorithm was modified in two ways. Firstly, the absolute minimum support (corresponding to minRareSup in RP-Tree) may now be greater than 1. An itemset must meet this support threshold to be included in the result. This is necessary to allow the removal of noisy itemsets, and to allow ARIMA to finish in a reasonable time without setting minFreqSup to an extremely low value. Secondly, candidate support count is done by building a tree structure with candidates from each level, which is more efficient than iterating through each itemset for each transaction. The FP-Growth algorithm was modified to find rare itemsets by generating all itemsets that meet minRareSup, then removing all itemsets that exceed minFreqSup.

Experiments comparing RP-Tree and RP-Tree-IG with Rarity and AfRIM have been omitted since they use a level-wise approach similar to ARIMA, and their performance compared to ARIMA has already been reported in detail in [5] and [1]. It is sufficient to note that AfRIM and Rarity can perform several orders of magnitude faster than ARIMA under specific conditions, such as a low number of rare itemsets, or when there is a small number of items. However, in general, AfRIM performs 2-3 times faster, while Rarity performs about 30 times faster than ARIMA.

Nine datasets from UCI Repository [13] were used in the experiments: Connect-4, Congressional Voting Records (Voting), Primary Tumor (Tumor), Zoo, Teaching Assistant Evaluation (Teaching), Flags, Adult, Dermatology and Soybean Large (Soybean).

4.1 Itemset Generation Results

In this section we compare the number of itemsets generated, and the execution time for ARIMA, FP-Growth, RP-Tree and RP-Tree-IG. We use the same minFreqSup and minRareSup threshold across all experiments.

Figure 3 shows the number of itemsets generated for each algorithm and dataset. The differences in the number of itemsets are due to the types of rare itemsets each algorithm

Table 2. Rule Interest Measures [12]

Measure	Definition
χ^2	$\sum \frac{(observed-expected)^2}{expected}$
Lift$(X \rightarrow Y)$	$\frac{sup(X \cup Y)}{sup(X)sup(Y)}$
AllConf$(X \rightarrow Y)$	$\frac{sup(X \cup Y)}{max(sup(X), sup(Y))}$
MaxConf$(X \rightarrow Y)$	$max\{\frac{sup(X \cup Y)}{sup(X)}, \frac{sup(X \cup Y)}{sup(Y)}\}$
Coherence$(X \rightarrow Y)$	$\frac{sup(X \cup Y)}{sup(X)+sup(Y)-sup(X \cup Y)}$
Cosine$(X \rightarrow Y)$	$\frac{sup(X \cup Y)}{\sqrt{sup(X)sup(Y)}}$
Kulc$(X \rightarrow Y)$	$\frac{sup(X \cup Y)}{2}(\frac{1}{sup(X)} + \frac{1}{sup(Y)})$

can generate. ARIMA and FP-Growth both generate the complete set of rare itemsets, RP-Tree generates only rare-item itemsets, and RP-Tree-IG generates rare-item itemsets using items that meet an information gain threshold. The number of itemsets found by RP-Tree tends to be several times less than that found by FP-Growth, except for the Connect-4 and Adult dataset, for which RP-Tree found about 80% of all rare itemsets.

Figure 4 shows the time taken for itemset generation for each itemset and algorithm. The runtime for ARIMA is more than 32 times longer than FP-Growth in all datasets except Teaching. This is due to the small size of the Teaching dataset, where overhead takes up a large proportion of the time taken. Realistically, most real world datasets would be of a reasonable size and the overhead is negligible. We also see that time taken for RP-Tree-IG is consistently less than that of RP-Tree, which is in turn less than FP-Growth. The time taken for ARIMA is significantly longer than FP-Tree due to the computationally expensive candidate generation and pruning steps. The differences in time taken for RP-Tree and RP-Tree-IG are the result of pruning transactions without any rare items, and pruning itemsets without any rare items above the minIG threshold, respectively. Transaction pruning reduces the size of the initial tree generated, and reduces the amount of computation required and the number of rare itemsets found. Note that the results for ARIMA for the Dermatology and Soybean datasets have been excluded from Figures 3 and 4 since execution did not complete within 2 hours.

4.2 Rule Generation Results

Rules were generated from the itemsets found using ARIMA/FP-Growth (both of which generate the complete set of rare itemsets), RP-Tree, and RP-Tree-IG. For all algorithms, parameters are: minFreqSup = 15% and minRareSup = 5. For RP-Tree, information gain minIG is set to 0.25.

In addition, we intentionally exclude rules do not meet the minimum confidence of 0.9 and lift of 1.0. Rules that have low confidence, by definition, are not reliable, and rules that have lift of less than 1.0, do not predict the occurrence of items (i.e., the consequent Y). By excluding these rules, the number of useful rules that are generated by each of the algorithms can be compared, and we can evaluate whether the omission of some rare-itemsets by RP-Tree affects the number of useful association rules found.

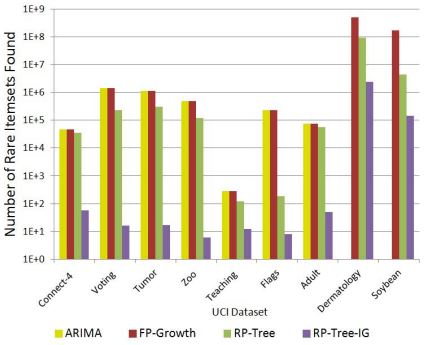


Fig. 3. Number of itemsets generated

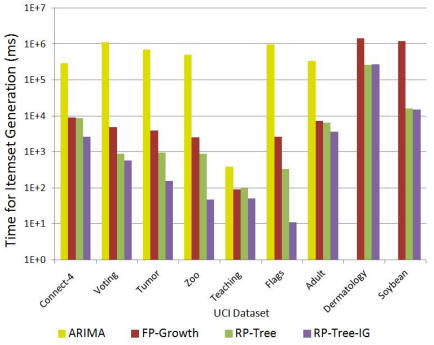


Fig. 4. Time taken for itemset generation

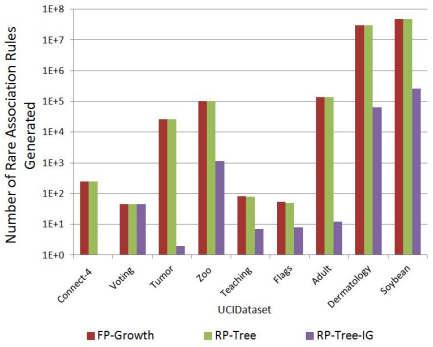


Fig. 5. Number of rules generated that meet confidence and lift thresholds

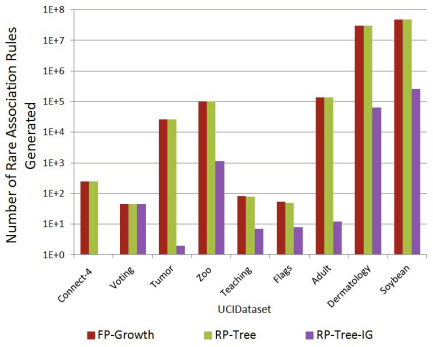


Fig. 6. Time taken for rule generation relative to FP-Growth

For the Connect-4 and Adult datasets, RP-Tree found around 80% of the itemsets found by FP-Growth. For the remaining seven datasets, RP-Tree generated significantly fewer itemsets compared to FP-Growth, ranging from 42.0% for Adult to 2.60% for Soybean, as shown in Figure 3. However, results in Figure 5 shows that the number of rules that met the confidence and lift thresholds were either identical or very similar. This shows that the set of rare itemsets that are ignored by RP-Tree does not tend to generate rules that meet both the confidence and lift thresholds, which supports the idea that the non-rare-item itemsets are not interesting can be safely omitted by RP-Tree. Since fewer itemsets are generated by RP-Tree compared to FP-Growth, the time required for rule generation is also reduced, as seen in Figure 6. For example, time taken for rule generation for RP-Tree for the soybean dataset is reduced to 1.8% of that for FP-Growth, while the number of association rules retained is 99.7% of FP-Growth. Overall the time taken for RP-Tree is lower than FP-Growth.

The Information Gain component for RP-Tree results in far fewer rules than RP-Tree for all datasets except Voting, with no change, and tends to generate rules that are

Table 3. Real World Datasets

Dataset	Algorithm	Support	Conf	χ^2	Lift	AllConf	Coherence	Cosine	Kulc	MaxConf
Connect-4	FP-Growth	19.374	0.999	911.416	49.296	0.014	0.014	0.097	0.507	0.999
	RP-Tree	19.374	0.999	911.416	49.296	0.014	0.014	0.097	0.507	0.999
	RP-Tree-IG	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Voting	FP-Growth	56.935	0.918	56.094	1.918	0.273	0.267	0.5	0.596	0.918
	RP-Tree	56.935	0.918	56.094	1.918	0.273	0.267	0.5	0.596	0.918
	RP-Tree-IG	56.935	0.918	56.094	1.918	0.273	0.267	0.5	0.596	0.918
Tumor	FP-Growth	18.099	0.939	12.643	1.686	0.097	0.096	0.281	0.518	0.939
	RP-Tree	18.099	0.939	12.643	1.686	0.097	0.096	0.281	0.518	0.939
	RP-Tree-IG	9	1	15.487	2.748	0.071	0.071	0.266	0.536	1
Zoo	FP-Growth	11.497	0.951	16.157	2.283	0.272	0.266	0.483	0.611	0.951
	RP-Tree	11.497	0.951	16.157	2.283	0.272	0.266	0.483	0.611	0.951
	RP-Tree-IG	12.039	0.96	24.611	2.808	0.348	0.342	0.551	0.654	0.96
Teaching	FP-Growth	6.849	0.992	19.608	3.408	0.17	0.17	0.347	0.581	0.992
	RP-Tree	6.673	1	20.319	3.506	0.172	0.172	0.347	0.586	1
	RP-Tree-IG	7.125	1	58.796	8.138	0.416	0.416	0.591	0.708	1
Flags	FP-Growth	6.546	0.997	4.064	1.619	0.054	0.054	0.224	0.526	0.997
	RP-Tree	6.533	0.997	4.061	1.619	0.054	0.054	0.223	0.526	0.997
	RP-Tree-IG	11.75	0.993	41.297	4.245	0.261	0.259	0.49	0.627	0.993
Adult	FP-Growth	1822.12	0.944	2427.051	4.345	0.09	0.088	0.204	0.517	0.944
	RP-Tree	1623.95	0.945	2513.977	4.469	0.087	0.085	0.197	0.516	0.945
	RP-Tree-IG	1151.286	0.958	1070.297	2.029	0.047	0.047	0.184	0.502	0.958
Dermatology	FP-Growth	34.352	0.938	10.964	1.375	0.132	0.131	0.343	0.535	0.938
	RP-Tree	34.34	0.938	10.966	1.376	0.132	0.131	0.343	0.535	0.938
	RP-Tree-IG	39.003	0.961	13.887	1.336	0.148	0.147	0.37	0.554	0.961
Soybean	FP-Growth	40.013	1	21.678	1.471	0.192	0.192	0.435	0.596	1
	RP-Tree	40	1	21.704	1.472	0.192	0.192	0.435	0.596	1
	RP-Tree-IG	40	1	28.51	1.62	0.211	0.211	0.457	0.606	1

of higher quality. For five datasets (Zoo, Teaching, Flags, Soybean and Dermatology) there are increases in most of the seven interest measures. However, for one dataset (Adult), the measures decreased. There were no rules generated at all for Connect-4. The reduction in the number of rules is due to the minIG threshold reducing the number of items that participate in itemsets.

Table 3 shows, for each dataset, the average support and confidence, and the average of each of the seven measures listed in Table 2, for each algorithm. The rule quality of FP-Growth compared to RP-Tree, according to the seven interest measures, is very similar since the set of generated rules is almost identical. RP-Tree with Information Gain tends to have higher values by selectively retaining rules that are more interesting.

Case Study. From the Teaching dataset, FP-Growth, RP-Tree and RP-Tree-IG generated 53 rules, 49 rules, and 8 rules respectively. The interest measures of the 4 additional rules generated by FP-Growth from non-rare-item itemsets are shown in Table 4. Rules 1 and 2 have lower than average values for confidence and all interest measures. Rules 3 and 4 have lower values for confidence, lift, kulc and maxConf; and higher values for χ^2, allConf, coherence and cosine. 8 association rules were generated using RP-Tree-IG. Of the 8 rules, 6 had higher than average values for confidence and all 7 interest measures compared to FP-Growth, while the remaining 2 had lower values.

From the Adult dataset, FP-Growth, RP-Tree and RP-Tree-IG generated 83 rules, 80 rules and 7 rules respectively. The 3 additional rules generated by FP-Growth had lower than average values for confidence, χ^2, lift and maxConf; and higher values for allConf, coherence, cosine and kulc, as shown in Table 4 . The 7 rules generated all had several measures that were lower than the average compared to FP-Growth.

The omission of non-rare-item itemsets by RP-Tree only has a small effect on the number and quality of association rules generated compared to FP-Growth, since the additional rules are of average quality and are few in number compared to the overall number of rules generated.

Table 4. Non-rare-item itemsets generated by FP-Growth

Dataset	Rule ID	Confidence	χ^2	Lift	AllConf	Coherence	Cosine	Kulc	MaxConf
Teaching	1	0.900	0.585	1.114	0.074	0.073	0.258	0.487	0.900
	2	0.900	0.227	1.062	0.070	0.070	0.252	0.485	0.900
	3	0.900	21.385	3.315	0.220	0.214	0.444	0.560	0.900
	4	0.900	21.385	3.315	0.220	0.214	0.444	0.560	0.900
Adult	1	0.928	88.662	1.034	0.160	0.158	0.386	0.544	0.928
	2	0.908	208.679	1.062	0.170	0.168	0.393	0.539	0.908
	3	0.915	29.765	1.019	0.164	0.161	0.387	0.539	0.915

5 Experimental Results with Synthetic Data

Synthetic test data was generated using a modified version of the IBM synthetic dataset generator described in [2] which allows the number of rare-item itemsets to be controlled. To control the number of rare-item itemsets, we removed all rare items after transaction generation using the original algorithm. We then selectively re-inserted rare items into the transactions to control the number of rare-item itemsets. Using synthetic data, we compare the precision, recall and execution time of RP-Tree and FP-Growth for finding rare-item itemsets, and the change in execution time for RP-Tree as the average transaction size and average rare-item-itemset size changes. Thirty sets of synthetic data are generated for each set of parameters.

5.1 Varying the Number of Rare-Item Itemsets

We generated synthetic datasets with 500,000 transactions, 80 unique items, and an average of 9 items per transaction, while varying the number of rare-item itemsets the dataset contains. The number of rare-item itemsets varied from 50 to 400 in increments of 50, and for each, a sample of 30 synthetic datasets were generated.

Precision and Recall. As expected, both FP-Growth and RP-Tree found all of the inserted rare-item itemsets for all synthetic datasets, giving both algorithms a recall of 1. For RP-Tree, the algorithm found all inserted rare-item itemsets, and no non-rare-item itemsets; giving RP-Tree a precision of 1. For FP-Growth, precision depends on the number of non-rare-item itemsets that are found. As shown in Figure 7, if the number

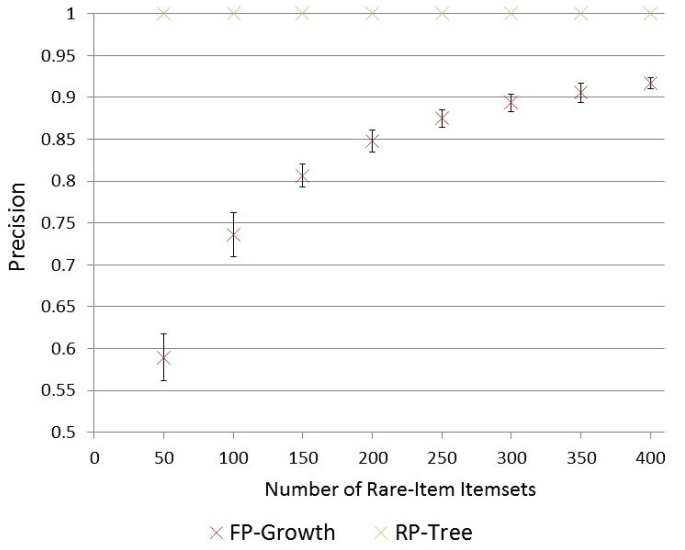


Fig. 7. Precision of FP-Growth and RP-Tree for finding rare-item itemsets (T9L50I80D500K)

of non-rare-item itemsets is roughly constant as the number of rare-item itemsets increases, the precision of FP-Growth for finding only rare-item itemsets increases. The error bars in Figure 7 represents one standard deviation of uncertainty.

Performance. The execution time for RP-Tree is consistently less than that of FP-Growth. In Figure 8, the cross shows the average ratio of execution times between RP-Tree and FP-Growth, calculated as $RP\text{-}Tree_{Time}/FP\text{-}Growth_{Time}$, and the error bars represent one standard deviation of uncertainty. The results show that RP-Tree completes execution in, on average, 33% of the time taken by FP-Growth, and the ratio of 1, which indicates no difference in performance, is always beyond two standard deviations for each number of rare-item itemsets. These performance results are consistent with the execution times when using the UCI datasets.

5.2 Varying Average Transaction Size

We generated synthetic data sets that had varying average transaction sizes the effect on execution time. Each synthetic dataset had 500k transactions, an average of 80 unique items and the average transaction size was between 7 and 13. As seen in Figure 9, as transaction size increases, execution time increases superlinearly. This trend similar to that of FP-Growth, where the size of the frequent pattern tree, and execution time, increases more rapidly than transaction size. Each of the execution times are the average of execution times of 30 synthetic datasets. The error bars are not included since the distribution of times are not gaussian but are instead mostly low with a few very long execution times. For example, for the datasets where the average transaction size is

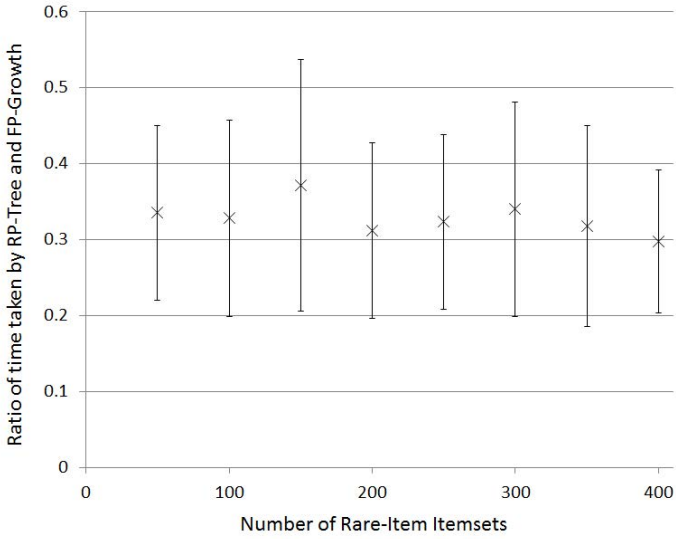


Fig. 8. Ratio of execution times for RP-Tree and FP-Growth (T9L50I80D500K)

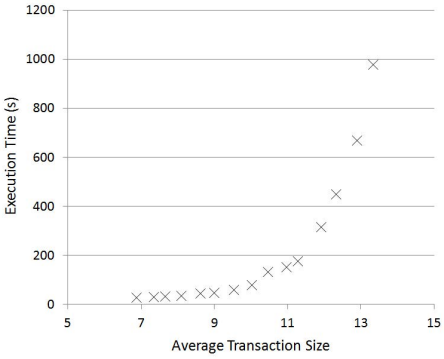


Fig. 9. Execution times for RP-Tree for different transaction sizes (L50I80D500K)

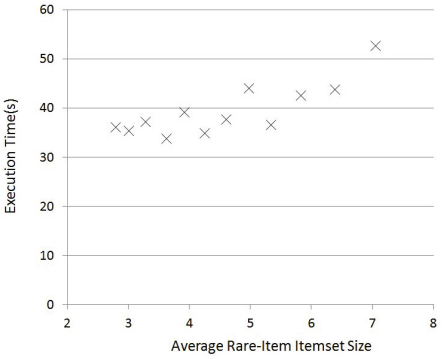


Fig. 10. Execution times for RP-Tree for different large itemset sizes (T9L50I80D500K)

12, the mean execution time is 315 seconds, while the median 31 seconds. This is most likely caused by RP-Tree pruning very few transactions for some datasets, which results in much longer execution times.

5.3 Varying Average Rare-Item Itemset Size

We examined the effect the average size of rare-item itemsets has on the execution time of RP-Tree. The synthetic datasets generated had 500k transactions, an average of 80

unique items, transaction size of 9 and average rare-item itemset size between 3 and 7 items. As the average rare-item itemset size increases, execution time also increases. Once again, the distribution of execution times is not gaussian, but are mostly low, about 30 seconds, with some datasets taking much longer, for example 198 seconds for a dataset with an average rare-item itemset size of 7.

6 Conclusions and Future Work

We present a new method for finding rare association rules in large databases. To our knowledge, this is the first algorithm that uses a tree structure to mine rare itemsets. Our algorithm finds a subset of all rare itemsets, which we call rare-item itemsets. We evaluated our method by comparing the quality of association rules generated against those generated using the FP-Growth algorithm from 9 datasets. We found that, in the majority of cases, RP-Tree generated far fewer itemsets for some datasets compared to FP-Growth. This meant that rule generation took much less time for RP-Tree than FP-Growth. However, at the same time, there was very little reduction in the number of rules that met the minimum confidence and lift thresholds. This shows that rare-item itemsets are more interesting since they contribute to almost all the rules that pass the thresholds, and the omission of non-rare-item itemsets by RP-Tree does not reduce rule quality, and in most cases, improves the overall rule quality in the set. We also generated multiple sets of synthetic data to examine the behaviour of execution time when the average transaction length and rare-item itemset length in the dataset changes, and to verify the performance increase of RP-Tree over FP-Growth for finding rare-item itemsets.

In our future work, we intend to find other ways of focusing on more potentially interesting association rules, such as rules that contain only rare items as the consequent. In addition, we intend to investigate the effect of the minRareSup on the quality of rules generated by RP-Tree, and to find ways of dealing with noise and removing coincidental non-rare-item itemsets.

References

1. Adda, M., Wu, L., Feng, Y.: Rare itemset mining. In: Proceedings of the Sixth International Conference on Machine Learning and Applications, ICMLA 2007, pp. 73–80. IEEE Computer Society, Washington, DC (2007)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, Santiago, Chile, pp. 487–499 (1994)
3. Sotiris, K., Dimitris, K.: Association rules mining: A recent overview. GESTS International Transactions on Computer Science and Engineering 32 (1), 71–82 (2006)
4. Liu, B., Hsu, W., Ma, Y.: Mining association rules with multiple minimum supports. In: Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 337–341 (1999)
5. Troiano, L., Scibelli, G., Birtolo, C.: A fast algorithm for mining rare itemsets. In: Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications, ISDA 2009, pp. 1149–1155. IEEE Computer Society, Washington, DC (2009)

6. Szathmary, L., Napoli, A., Valtchev, P.: Towards rare itemset mining. In: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2007, vol. 01, pp. 305–312. IEEE Computer Society, Washington, DC (2007)
7. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: 3rd Intl. Conf. on Knowledge Discovery and Data Mining, pp. 283–286. AAAI Press (1997)
8. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328 (1996)
9. Koh, Y.S., Rountree, N.: Finding Sporadic Rules Using Apriori-Inverse. In: Ho, T.-B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, pp. 97–106. Springer, Heidelberg (2005)
10. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD 2000, pp. 1–12. ACM, New York (2000)
11. Mitchell, T.M.: Machine Learning, pp. 57–60. McGraw-Hill (1997)
12. Wu, T., Chen, Y., Han, J.: Association Mining in Large Databases: A Re-examination of Its Measures. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS (LNAI), vol. 4702, pp. 621–628. Springer, Heidelberg (2007)
13. Frank, A., Asuncion, A.: UCI machine learning repository (2010), `http://archive.ics.uci.edu/ml`

Discovering Frequent Patterns from Uncertain Data Streams with Time-Fading and Landmark Models

Carson Kai-Sang Leung[1], Alfredo Cuzzocrea[2], and Fan Jiang[1]

[1] University of Manitoba, Winnipeg, MB, Canada
{kleung,umjian29}@cs.umanitoba.ca
[2] ICAR-CNR and University of Calabria, Rende (CS), Italy
cuzzocrea@si.deis.unical.it

Abstract. Streams of data can be continuously generated by sensors in various real-life applications such as environment surveillance. Partially due to the inherited limitation of the sensors, data in these streams can be uncertain. To discover useful knowledge in the form of frequent patterns from streams of uncertain data, a few algorithms have been developed. They mostly use the sliding window model for processing and mining data streams. However, for some applications, other stream processing models such as the *time-fading model* and the *landmark model* are more appropriate. In this paper, we propose mining algorithms that use (i) the time-fading model and (ii) the landmark model to discover frequent patterns from streams of uncertain data.

Keywords: Knowledge discovery, data mining techniques, data streams, frequent itemsets, probabilistic data.

1 Introduction

Frequent pattern mining [2] helps discover implicit, previously unknown, and potentially useful knowledge in the form of frequently occurring sets of items that are embedded in the data. For example, it finds from shopping market basket data those sets of popular merchandise items, which in turn helps reveal shopper behaviour.

Nowadays, the automation of measurements and data collection is producing tremendously huge volumes of data. For instance, the development and increasing use of a large number of sensors (e.g., acoustic, chemical, electromagnetic, mechanical, optical radiation and thermal sensors) for various real-life applications (e.g., environment surveillance, security, manufacture systems) have led to *data streams* [5,25]. To discover useful knowledge from these streaming data, several mining algorithms [4,6,13] have been proposed. In general, mining frequent patterns from dynamic data streams [15,17,26] is more challenging than mining from traditional static transaction databases due to the following characteristics of data streams:

A. Hameurlain et al. (Eds.): TLDKS VIII , LNCS 7790, pp. 174–196, 2013.

1. *Data streams are continuous and unbounded.* As such, we no longer have the luxury to scan the streams multiple times. Once the streams flow through, we lose them. We need some techniques to capture important contents of the streams. For instance, *sliding windows* capture the contents of a fixed number (w) of batches (i.e., w most recent batches) in the streams. Alternatively, *landmark windows* capture contents of all batches after the landmark (i.e., sizes of windows keep increasing with the number of batches). Similarly, *time-fading windows* also capture contents of all the batches but weight recent data heavier than older data (i.e., monotonically decreasing weights from recent to older data).
2. *Data in the streams are not necessarily uniformly distributed.* As such, a currently infrequent pattern may become frequent in the future and vice versa. We have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as frequencies of some patterns (as it is impossible to recall those pruned patterns).

Many existing mining algorithms discover frequent patterns from *precise* data (in either static databases [12,16,23] or dynamic data streams [14,27]), in which users definitely know whether an item is present in, or absent from, a transaction in the data. An interesting and related aspect is represented by the problem of supporting *OLAP over data streams* (e.g., [7,9]) as computing *multidimensional aggregates* over data streams can be considered as computing a specialized class of patterns over such streams. This also poses *computational overhead issues*, which very often are addressed by means of high-performance infrastructures (e.g., [11,10]). However, there are situations in which users are uncertain about the presence or absence of items. For example, due to dynamic errors (e.g., inherited measurement inaccuracies, sampling frequency of the sensors, deviation caused by a rapid change of the measured property over time such as drift or noise, wireless transmission errors, network latencies), streaming data collected by sensors may be uncertain. As such, users may highly suspect but cannot guarantee that an item x is present in a transaction t_i. The uncertainty of such suspicion can be expressed in terms of *existential probability* $P(x, t_i) \in (0, 1]$, which indicates the likelihood of x being present in t_i in probabilistic data. With this notion, every item in t_i in (static databases or dynamic streams of) precise data can be viewed as an item with a 100% likelihood of being present in t_i. A challenge of handling these uncertain data is the huge number of "possible worlds" (e.g., there are two "possible worlds" for an item x in t_i: (i) $x \in t_i$ and (ii) $x \notin t_i$). Given q independent items in all transactions, there are $O(2^q)$ "possible worlds" [18]. Similar problems still arise in the multidimensional case as well (e.g., [8]).

In past few years, several mining algorithms have been proposed to discover frequent patterns from uncertain data. However, most of them (e.g., UF-growth [20], CUF-growth [22], UH-Mine [1], U-Eclat [3], UV-Eclat [21], U-VIPER [24]) mine frequent patterns from *static databases*—but *not* dynamic streams—of uncertain data. For the algorithms that mine from data streams (e.g., UF-streaming [19]), they use *sliding windows*. While the use of sliding windows is useful for situations where users are interested in discovering frequent patterns from a fixed-size time

window (e.g., frequent patterns observed in the last 24 hours), there are also other situations where users are interested in a variable-size time window capturing all historical data (with or without stronger preference on recent data than older one). In these situations, other window models (e.g., time-fading model or landmark model) are needed. Hence, a logical question is: How to discover frequent patterns from dynamic streams of uncertain data when using the time-fading model or the landmark model?

In response to this question, we propose algorithms for discovering useful knowledge from streams of uncertain data using the time-fading and landmark models. Our key contributions are: (i) the proposal and maintenance of a tree structure to capture the frequent patterns discovered from batches of transactions in dynamic streams when using the time-fading model, (ii) the design of a family of tree-based stream mining algorithms that use such a tree structure for discovering and storing frequent patterns—especially the algorithm that does *not* require the traversal and update of *all* tree nodes, (iii) extension of the aforementioned tree structure and stream mining algorithms for discovering frequent patterns using the landmark model, (iv) analytical evaluation of these algorithms, as well as (v) experimental evaluation of these algorithms.

As the current paper is an extension and enhancement of our DaWaK 2011 paper [25], additional contributions beyond the basic framework include the following: (i) handling the situation where batches of data streams arrive out of order—say, certain batches arrive before the ones preceding them (Section 4), (ii) extending the mining of frequent patterns with the time-fading model to that with the landmark model (Section 5), (iii) providing analytical results for these extensions (Section 6), and (iv) running additional experiments (Section 7).

This paper is organized as follows. The next section gives some background information that is relevant to the remainder of this paper. In Section 3, we introduce our tree-based mining algorithms that use the *time-fading model* to discover frequent patterns from streams of uncertain data. Section 4 discusses how we handle the out-of-order batches. In Section 5, we extend our algorithms to use the *landmark model* to discover frequent patterns from streams of uncertain data. Analytical and experimental results are shown in Sections 6 and 7. Finally, Section 8 presents the conclusions.

2 Background and Related Work

In this section, we provide background information about mining frequent patterns from static databases of uncertain data and using the sliding window model to mine frequent patterns from dynamic streams of uncertain data.

2.1 UF-growth: Mining from Static Databases of Uncertain Data

Among the algorithms that mine frequent patterns from static databases of uncertain data (e.g., UF-growth [20], CUF-growth [22], UH-Mine [1], U-Eclat [3], UV-Eclat [21], U-VIPER [24]), the tree-based *UF-growth* algorithm is used in UF-streaming for stream mining (Section 2.2). To discover frequent patterns,

UF-growth constructs a UF-tree to capture contents of uncertain data. Each tree node keeps an item x, its existential probability $P(x, t_i)$, and its occurrence count. The UF-tree is constructed in a similar fashion to that of the FP-tree [16] except that nodes in the UF-tree are merged and shared only if they represent the same x and $P(x, t_i)$. Once the UF-tree is constructed, UF-growth extracts appropriate tree paths to mine frequent patterns using the "possible world" interpretation [18]. A pattern is *frequent* if its expected support $\geq$ user-specified *minsup* threshold. When items within a pattern X are independent, the *expected support* of X in the database DB can be computed by summing (over all transactions $t_1, ..., t_{|DB|}$) the product of existential probabilities of items within X [18]:

$$expSup(X, DB) = \sum_{i=1}^{|DB|} \left(\prod_{x \in X} P(x, t_i) \right). \quad (1)$$

Note that, while UF-growth discovers frequent patterns from uncertain data, it mines from static databases (instead of dynamic data streams).

2.2 UF-streaming: Mining from Uncertain Data Streams with Sliding Windows

Unlike UF-growth [20] (which does not handle data streams) or FP-streaming [14] (which does not handle uncertain data), the *UF-streaming* algorithm [19] mines frequent patterns from uncertain data streams by using a fixed-size sliding window of w recent batches. UF-streaming first calls UF-growth (Section 2.1) to find "frequent" patterns from the current batch of transactions in the streams (using *preMinsup* as the threshold). A pattern is "frequent" (more precisely, subfrequent) if its expected support $\geq$ *preMinsup*. Note that, although users are interested in truly frequent patterns (i.e., patterns with expected support $\geq$ *minsup* $>$ *preMinsup*), *preMinsup* is used in attempt to avoid pruning a pattern too early. This is important because data in the continuous streams are not necessarily uniformly distributed.

UF-streaming then stores the mined "frequent" patterns and their expected support values in a tree structure, in which each tree node X keeps a list of w support values. When a new batch flows in, the window slides and support values shift so that the "frequent" patterns (and their expected support values) mined from the newest batch are inserted into the window and those representing the oldest batch in the window are deleted. This process is repeated for each batch in the stream. The expected support of any frequent pattern X can be computed by summing all w expected supports of X (one for each batch in the sliding window). Let $expSup(X, B_i)$ denote the expected support of X in Batch B_i. Then, at time T, the expected support of X in the current sliding window containing w batches of uncertain data in Batches $B_{T-w+1}, ..., B_T$ inclusive can be computed as follows:

$$expSup\left(X, \cup_{i=T-w+1}^{T} B_i\right) = \sum_{i=T-w+1}^{T} expSup(X, B_i). \quad (2)$$

3 Our Proposed TUF-streaming Algorithms

Among the three commonly used models for processing streams (i.e., sliding window, landmark, and time-fading models), the landmark window keeps all batches after the landmark (i.e., keeps an increasing number of batches). Similarly, the time-fading window also keeps an increasing number of batches, but it weights older data lighter than recent data (i.e., monotonically decreasing weights from current to older data).

In this section, we propose our family of algorithms—called **TUF-streaming**—that use the ***t***ime-fading model in an ***u***ncertain data environment to mine "***f***requent" patterns from ***streaming*** data.

3.1 TUF-streaming(Naive): A Naive Algorithm

Inspired by the UF-streaming algorithm (which uses sliding windows), we propose **TUF-streaming(Naive)** that uses *time-fading windows*. The key steps of the algorithm can be described as follows. First, for each batch B_i of uncertain data in the stream, our naive algorithm applies UF-growth with *preMinsup* to find "frequent" patterns (i.e., patterns with expected support $\geq$ *preMinsup* from a batch). Then, it stores the mined "frequent" patterns and their expected support values in a tree structure called ***UF-stream***, in which each tree node corresponding to a pattern X keeps a list of support values. Note that the time-fading window does not slide. Instead, it grows. This mining process and the UF-stream insertion process are repeated for each batch in the stream of uncertain data. Let $expSup(X, B_i)$ denote the expected support of X in B_i. Then, at time T, the expected support of X mined from the time-fading model can be computed by summing over all batches the expected supports of X (weighted by the *time-fading factor* α, where $0 \leq \alpha \leq 1$):

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \sum_{i=1}^{T} \left(expSup(X, B_i) \times \alpha^{T-i}\right), \quad (3)$$

See Fig. 1 for a skeleton of our TUF-streaming(Naive) algorithm.

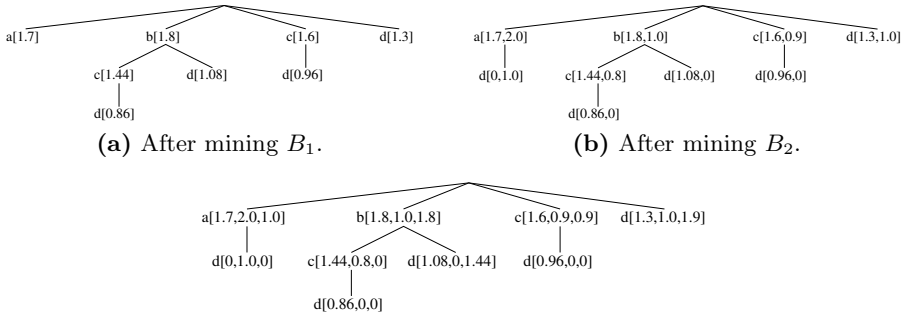
Example 1. Consider the following stream of uncertain data:

Batches	Transactions	Contents
B_1	t_1	{a:0.7, d:0.1, e:0.4}
	t_2	{a:1.0, b:0.9, c:0.8, d:0.6}
	t_3	{b:0.9, c:0.8, d:0.6}
B_2	t_4	{a:1.0, c:0.1, d:0.7}
	t_5	{a:1.0, d:0.3, e:0.1}
	t_6	{b:1.0, c:0.8}
B_3	t_7	{a:1.0, c:0.9, d:0.3}
	t_8	{b:0.9, d:0.8}
	t_9	{b:0.9, d:0.8, e:0.7}

Algorithm TUF-streaming(Naive)

1. For each batch B_i do
   2. Apply UF-growth with *preMinsup* to B_i to find "frequent" patterns.
   3. Insert each mined "frequent" pattern X and its expected support value into the UF-stream:
      3a. If the nodes corresponding to X do not exist in the UF-stream, then create new tree nodes (each which keeps a list of support values) for X.
      3b. The i-th position of the tree nodes corresponding to X stores $expSup(X, B_i)$.
   4. The i-th position of all unvisited nodes stores 0.
5. At time T, use Equation (3) to compute $expSup(X, \cup_{i=1}^{T} B_i) = \sum_{i=1}^{T} \big(expSup(X, B_i) \times \alpha^{T-i}\big)$, which sums over all the support values (weighted by α^{T-i}) in the list of support values for X.

Fig. 1. A skeleton of the TUF-streaming(Naive) algorithm



(a) After mining B_1. **(b)** After mining B_2.

(c) After mining B_3.

Fig. 2. The UF-stream structures for the TUF-streaming(Naive) algorithm

Here, each transaction contains items and their corresponding existential probabilities, e.g., $P(a, t_1)$=0.7. Let the user-specified *minsup* threshold be 1.0. When using the time-fading model, TUF-streaming(Naive) applies UF-growth to B_1 in the uncertain data stream using *preMinsup* < *minsup* (say, *preMinsup*=0.8) and finds "frequent" patterns $\{a\}, \{b\}, \{b, c\}$, $\{b, c, d\}, \{b, d\}, \{c\}, \{c, d\}$ & $\{d\}$ with their corresponding expected support of 1.7, 1.8, 1.44, 0.86, 1.08, 1.6, 0.96 & 1.3. These patterns and their expected support values are then stored in the UF-stream structure as shown in Fig. 2(a). Each node in the UF-stream keeps an item and a *list* of expected support values. So far, each list is of length 1 (e.g., c:[1.44] on the branch $\langle b[1.8], c[1.44], d$:[0.86]$\rangle$ represents "frequent" pattern $\{b, c\}$ with an expected support of 1.44).

Next, when the second batch B_2 arrives, TUF-streaming(Naive) applies a similar procedure: Call UF-growth to find "frequent" patterns $\{a\}, \{a, d\}, \{b\}$, $\{b, c\}, \{c\}$ & $\{d\}$ with expected support values of 2.0, 1.0, 1.0, 0.8, 0.9 & 1.0, respectively. Then, the algorithm appends each expected support value to the

list of the appropriate tree node in UF-stream. The resulting UF-stream, as shown in Fig. 2(b), consists of nine nodes (due to the addition of the node d[0,1.0] representing the new pattern $\{a, d\}$ having an expected support of 1.0 in B_2 but infrequent in B_1). Note that the list in each node now consists of two expected support values. Expected support of any pattern X can be computed using Equation (3) based on the expected support values stored in the list of X. For instance, let the time-fading factor α be 0.9, then $expSup(\{b, c\}, B_1 \cup B_2) = 1.44\alpha + 0.8 \approx 2.10$.

Similarly, when subsequent batches arrive, TUF-streaming(Naive) applies a similar procedure. Fig. 2(c) shows the resulting UF-stream structure after processing B_3. At that time, $expSup(\{b, c\}, B_1 \cup B_2 \cup B_3) = 1.44\alpha^2 + 0.8\alpha + 0 \approx 1.89$. □

3.2 TUF-streaming(Space): A Space-Saving Algorithm

Although TUF-streaming(Naive) finds all "frequent" patterns, it may require a large amount of space. As data streams are continuous and unbounded, storing the expected support value of X for each batch in the streams can be impractical because it could lead to a potentially infinite list for each node. A careful analysis on Equation (3) reveals that the expected support of X is the sum of weighted expected support of X over all batches. Unlike the sliding window model (which requires the deletion of the oldest batch), the fading-time model does not require the deletion of any old batches. Instead, it assigns lighter weights to old batches than recent batches. Hence, we propose a space-saving algorithm called **TUF-streaming(Space)**, which does not need to keep track of the details for each batch. We rewrite Equation (3) in a recursive form as follows:

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \left[expSup\left(X, \cup_{i=1}^{T-1} B_i\right) \times \alpha\right] + expSup(X, B_T). \quad (4)$$

By doing so, the algorithm keeps only a *single* value—i.e., $expSup(X, \cup_{i=1}^{T} B_i)$—instead of a potentially infinite list of $expSup(X, B_i)$. See Fig. 3 for a skeleton of our TUF-streaming(Space) algorithm.

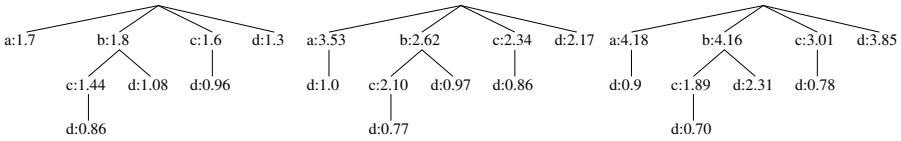
Example 2. Let us revisit Example 1. When using the time-fading model, our TUF-streaming(Space) algorithm uses Equation (4) to compute expected support values. For instance, Fig. 4(a) shows the expected support values stored in the UF-stream after mining "frequent" patterns from B_1. They are identical to those shown in Fig. 2(a).

Afterwards (say, after mining B_i for $i \geq 2$), instead of appending the expected support values of "frequent" patterns mined from B_i, the algorithm modified the stored value. For instance, after mining B_2, instead of storing [1.44, 0.8] for $\{b, c\}$ as in Fig. 2(b) for TUF-streaming(Naive), TUF-streaming(Space) stores their sum $1.44\alpha+0.8 \approx 2.10$ as $expSup(\{b, c\}, B_1 \cup B_2)$ in Fig. 4(b). Similarly, after mining B_3, instead of storing [1.44, 0.8, 0] as in Fig. 2(c) for TUF-streaming(Naive), TUF-streaming(Space) stores their sum $(1.44\alpha + 0.8)\alpha + 0 \approx 1.89$ as $expSup(\{b, c\}, B_1 \cup B_2 \cup B_3)$ in Fig. 4(c). □

Algorithm TUF-streaming(Space)

1. For each batch B_i do
   2. Apply UF-growth with *preMinsup* to B_i to find "frequent" patterns.
   3. Insert each mined "frequent" pattern X into the UF-stream and update its expected support value by using Equation (4):
      3a. For each tree node in the UF-stream, fade its expected support value by α.
      3b. If the nodes corresponding to X do not exist in the UF-stream, then create new tree nodes (each which keeps only a single support value) for X.
      3c. Else, add its expected support value in B_i (i.e., $expSup(X, B_i)$) to the faded expected support value.
4. At time T, look up the updated expected support value for X to give $expSup(X, \cup_{i=1}^{T} B_i)$.

Fig. 3. A skeleton of the TUF-streaming(Space) algorithm



(a) After mining B_1. **(b)** After mining B_2. **(c)** After mining B_3.

Fig. 4. The UF-stream structures for the TUF-streaming(Space) algorithm

3.3 TUF-streaming(Time): A Time-Saving Algorithm

TUF-streaming(Space) greatly reduces the amount of space required from a potentially infinite list of expected support values to a much more realistic and practical requirement of storing only a single expected support value in each node in the UF-stream. However, as observed from the recursive formula shown in Equation (4), the expected support of any pattern X up to time T directly depends on the expected support of X up to time $T-1$. As such, TUF-streaming(Space) needs to visit every node in the UF-stream after mining each batch (even if the corresponding pattern is not "frequent" in that batch) in order to compute expected support for "frequent" patterns. On the one hand, such a requirement may incur a long runtime. On the other hand, if one were to skip nodes in some batches, then the resulting expected support may not be correct. See Example 3.

Example 3. Let us revisit Example 2. Fig. 4 shows that $\{b, d\}$ is "frequent" with $expSup(\{b, d\}, B_1) = 1.08$. We know that $\{b, d\}$ does not appear in B_2, but it is "frequent" in B_3 with $expSup(\{b, d\}, B_3) = 1.44$. If after mining B_2, we decide to skip the node corresponding to $\{b, d\}$, then the expected support value stored in the UF-stream would remain unchanged (i.e., at 1.08). Then, after mining B_3, we decide to visit and update the node for $\{b, d\}$, then the expected support value stored in the UF-stream would become $1.08\alpha + 1.44 = 2.52$ (cf. the correct expected support of 2.31). The problem was caused by skipping this node after

Algorithm TUF-streaming(Time)

1. For each batch B_i do
   2. Apply UF-growth with *preMinsup* to B_i to find "frequent" patterns.
   3. Insert each mined "frequent" pattern X into the UF-stream and update its expected support value by using Equation (5):
      3a. If the nodes corresponding to X do not exist in the UF-stream, then create new tree nodes (each which keeps a single support value and the "last visit" field) for X.
      3b. Else, fade the expected support value by α^{T-LV} and add its current expected support value in B_i (i.e., $expSup(X, B_i)$) to it. Set LV to be the current time.
4. At time T, if $LV = T$ (i.e., nodes just visited), then look up the updated expected support value for X to give $expSup(X, \cup_{i=1}^{T} B_i)$. Else, fade the old values by α^{T-LV} and add its expected support value in B_i.

Fig. 5. A skeleton of the TUF-streaming(Time) algorithm

mining B_2 (even though $\{b, d\}$ is not "frequent"). The skip led to the missing multiplication of α. □

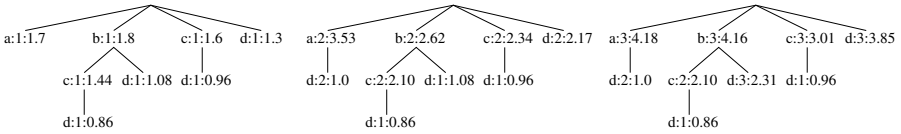
To solve the above problem, we propose a time-saving algorithm called **TUF-streaming(Time)**, which does not need to visit every node in the UF-stream after mining each batch. With this algorithm, the number of nodes visited at each batch is *proportional to* the number of "frequent" patterns mined from that batch. In other words, it visits only those nodes representing patterns that are "frequent" in that batch. It is possible due to our analytical results, which reveal that Equation (4) can be written as follows:

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \left[expSup\left(X, \cup_{i=1}^{LV} B_i\right) \times \alpha^{T-LV}\right] + expSup(X, B_T), \quad (5)$$

where LV is an additional field stored in each node of the UF-stream to indicate the batch number of last visit of the node. See Fig. 5 for a skeleton of our TUF-streaming(Time) algorithm.

Example 4. Let us revisit Example 2. When using TUF-streaming(Time), although each node in the UF-stream contains an additional "last visit" field (i.e., requires slightly more space), we no longer need to visit every node in UF-stream after mining a batch (i.e., takes less time). This is a space-time tradeoff.

Our TUF-streaming(Time) uses the "last visit" field in Equation (5) for computing expected support values to be stored in the UF-stream for the time-fading model. Unlike the TUF-streaming(Space) algorithm that visits every node, TUF-streaming(Time) visits only "frequent" nodes. After mining B_1, the algorithm visits and stores eight expected support values (i.e., 1.7, 1.8, 1.44, 0.86, 1.08, 1.6, 0.96 & 1.3 for "frequent" patterns $\{a\}, \{b\}, \{b, c\}, \{b, c, d\}, \{b, d\}, \{c\}$, $\{c, d\}$ & $\{d\}$, respectively) in the UF-stream. The "last visit" fields for all these eight nodes are "1", indicating that they were last visited in Batch B_1. For example, the node d:1:0.86 shown in Fig. 6(a) indicates that $\{b, c, d\}$ with expected support of 0.86 was last visited in Batch B_1.



(a) After mining B_1. **(b)** After mining B_2. **(c)** After mining B_3.

Fig. 6. The UF-stream structures for the TUF-streaming(Time) algorithm

Then, for B_2, only six patterns are "frequent": $\{a\}, \{a, d\}$, $\{b\}, \{b, c\}, \{c\}$ & $\{d\}$. The TUF-streaming(Time) algorithm only visits and updates these six nodes. For instance, it visits the node for $\{b, c\}$ and updates its expected support (to 2.10) by multiplying the old $expSup(\{b, c\}, B_1)$=1.44 by α and then adding $expSup(\{b, c\}, B_2)$ =0.8 to the product: $1.44\alpha + 0.8 \approx 2.10$. It results in c:2:2.10 as shown in Fig. 6(b). For patterns that are not "frequent" (e.g., $expSup(\{b, d\}, B_2)$=0), the algorithm delays the visit to that node. This explains why the node $\{b, d\}$ was not visited in Batch B_2. It is represented as d:1:1.08, which means $\{b, d\}$ with expected support of 1.08 was last visited & updated at Batch B_1.

After mining B_3, TUF-streaming(Time) visits and updates four nodes, including $\{b, d\}$. As shown in Fig. 6(c), the node is represented as d:3:2.31, which indicates that $expSup(\{b, d\}, B_1 \cup B_2 \cup B_3) = 1.08 \times \alpha^{3-1} + expSup(\{b, d\}, B_3)$ $= 1.08\alpha^2 + 1.44 \approx 2.31$. If the user requests for all "frequent" patterns at this time, the algorithm visits all nodes to compute their expected support. Otherwise, TUF-streaming(Time) visits nodes only for "frequent" patterns mined in a batch and delays the visit to other nodes. Hence, it requires shorter runtime than TUF-streaming(Space), which visits every node in the UF-stream. □

4 Discussion: Handling Out-of-Order Batches of Uncertain Streaming Data

Note that we do not require each batch of uncertain streaming data to be of the same size or contain the same number of transactions. In the previous section, we only assumed that batches of streaming data come according to an ordered sequence. In other words, Batch B_i arrives before Batch B_j for any $i < j$. As the contents of each batch may vary, it is possible that a later batch may arrive before an earlier batch. If it happens, how could we mine frequent patterns from these out-of-order batches? Without loss of generality, let us assume that each batch can be identified by a sequential number i (say, consecutive batch number). In the remainder of this section, we discuss how the three TUF-streaming algorithms handle the out-of-order batches.

For *TUF-streaming*(*Naive*), when B_i arrives and "frequent" patterns are found from B_i, the expected support of each of these patterns is then put in the i-th position of the list of expected support values for the pattern. Then, at time

T, the expected support of a pattern X from B_1 to B_T can be computed using Equation (3), which sums the weighted expected support values of each batch. If there is any delay in the arrival of some batches (i.e., on the dual side, there are some early arrivals), the algorithm stores the relevant information (e.g., each "frequent" pattern X and its expected support value $expSup(X, B_e)$ found in the early-arrival batch B_e) in the TUF-stream. However, the algorithm delays the computation of expected support at time T $exupSup(X, \cup_{i=1}^{T} B_i)$ until all batches before B_T arrive. See the following example.

Example 5. Let us revisit Example 1. If batch B_3 arrives before B_1 and B_2, TUF-streaming(Naive) (i) processes B_3, (ii) computes the expected support values of all "frequent" patterns in B_3, and (iii) stores these values in the 3rd position of the list of expected support values in the tree node of UF-stream. Then, when batch B_1 arrives, the algorithm (i) processes B_1, (ii) computes the expected support values of all "frequent" patterns in B_1, and stores (iii) these values in the 1st position of the list of expected support values in a similar fashion. The algorithm keeps track of which batches have been processed. When all of them have arrived, the algorithm computes expected support solely based on the expected supports stored in the list by using Equation (3). □

TUF-streaming(*Space*) computes the expected support of X by Equation (4), which is a recursive function. On the surface, it may appear to be impossible to handle out-of-order batches. A careful analysis of the equation reveals that it is feasible to handle out-of-order batches. If there is any delay in the arrival of some batches, the algorithm updates the stored expected support value by multiplying appropriate weights. See the following example.

Example 6. Let us revisit Example 2. If batch B_3 arrives before B_1 and B_2, TUF-streaming(Space) (i) processes B_3, (ii) computes the expected supports of all "frequent" patterns in B_3, and (iii) stores the expected support values. Then, when batch B_1 arrives, the algorithm (i) processes B_1, (ii) computes the expected supports of all "frequent" patterns in B_1, and (iii) "updates" the expected support values. The algorithm keeps track of which batches have been processed. This is possible because of the following:

$$
\begin{aligned}
& expSup\left(X, \cup_{i=1}^{3} B_i\right) \\
= \; & expSup\left(X, \cup_{i=1}^{2} B_i\right) \times \alpha + expSup(X, B_3) \\
= \; & \left[expSup\left(X, \cup_{i=1}^{1} B_i\right) \times \alpha + expSup(X, B_2)\right] \times \alpha + expSup(X, B_3) \\
= \; & \left[expSup\left(X, B_1\right) \times \alpha + expSup(X, B_2)\right] \times \alpha + expSup(X, B_3) \\
\\
= \; & expSup(X, B_3) + \\
& \left[expSup(X, B_1) \times \alpha^{3-1}\right] + \\
& \left[expSup(X, B_2) \times \alpha^{3-2}\right].
\end{aligned}
$$

When all batches have arrived (i.e., no gap), the expected support values stored in the node are accurate; prior to that, the stored expected support values may be lower than the real/accurate one. □

TUF-streaming(*Time*) computes the expected support of X by Equation (5), which is also a recursive function. To handle out-of-order batches, the algorithm applies similar procedure to that of TUF-streaming(Space) except that TUF-streaming(Time) only visits nodes representing "frequent" patterns (cf. all patterns as in TUF-streaming(Space)). See the following example.

Example 7. Let us revisit Example 4. If batch B_3 arrives before B_1 and B_2, TUF-streaming(Time) (i) processes B_3, (ii) computes the expected supports of all "frequent" patterns in B_3, and (iii) stores the expected support values. Then, when batch B_1 arrives, the algorithm (i) processes B_1, (ii) computes the expected supports of all "frequent" patterns in B_1. If a node corresponding to a "frequent" pattern found in B_1, the algorithm updates its expected support value; otherwise, the algorithm skips the node. The algorithm also keeps track of which batches have been processed. When all batches have arrived (i.e., no gap), the expected support values stored in the nodes are accurate; prior to that, the stored expected support values are lower than the real/accurate one. □

5 Extended TUF-streaming Algorithms for Discovering Frequent Patterns with the Landmark Model

Recall that, besides the time-fading model, the landmark model is another commonly used model for processing streams. In general, both time-fading and landmark models keep an increasing number of batches. However, with the time-fading model, historical data are weighted lighter than recent data (i.e., monotonically decreasing weights from current to historical data). In Section 3, we proposed a family of three TUF-streaming algorithms that use the *time-fading model* to discover frequent patterns from uncertain data streams. In this section, we extend this family of algorithms to discover frequent patterns from uncertain data streams when using the *landmark model*. We call the new family of algorithms **XTUF-streaming**, i.e., the family of the e***x***tended ***TUF-streaming*** algorithms.

5.1 XTUF-Streaming(Naive): Extended TUF-Streaming(Naive) for the Landmark Model

When using the *landmark model*, transactions in each batch (regardless of whether they are historical data or recent data) are treated equally. As such, all batches (regardless of whether they are old or recent) are assigned the same weights. Hence, we extend our proposed TUF-streaming(Naive) algorithm by assigning the time fading factor α to 1. As a result, the extended TUF-streaming(Naive) algorithm—called **XTUF-streaming(Naive)**—simplifies Equation (3) to become the following, which computes the expected support of X with the *landmark model* by summing all expected supports of X (after the landmark B_1):

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \sum_{i=1}^{T} \left(expSup(X, B_i) \times \alpha^{T-i}\right)$$
$$= \sum_{i=1}^{T} expSup(X, B_i). \quad (6)$$

Example 8. Let us revisit Example 1, but let us use the *landmark model* here. Then, the XTUF-streaming(Naive) algorithm uses Equation (6) to compute expected support. As the contents (e.g., every "frequent" pattern and its expected support value $expSup(X, B_i)$ for each batch B_i stored in the i-th position of the list of support values) of the UF-stream are identical to those shown in Fig. 2, the expected support values of "frequent" patterns after mining B_1 are the same for both landmark and time-fading models (see Example 1). For subsequent batches B_i (for $i \geq 2$), the expected support values of "frequent" patterns after mining B_1 are different because XTUF-streaming(Naive) uses Equation (6) to compute the expected support. For instance, $expSup(\{b, c\}, B_1 \cup B_2) = 1.44 + 0.8 = 2.24$ (cf. 2.10 in Example 1), and $expSup(\{b, c\}, B_1 \cup B_2 \cup B_3) = 1.44 + 0.8 + 0 = 2.24$ (cf. 1.89). □

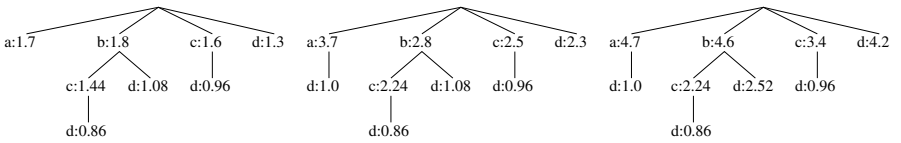
5.2 XTUF-Streaming(Space): Extended TUF-Streaming(Space) for the Landmark Model

In the previous section, we showed how we extended the TUF-streaming(Naive) algorithm into XTUF-streaming(Naive) to discover "frequent" patterns from uncertain data streams using the landmark model. A logical question is how to extend the other two algorithms in the TUF-streaming family to discover "frequent" patterns from uncertain data streams using the landmark model? Recall that, when using the time-fading model to find "frequent" patterns, TUF-streaming(Naive) stores the expected support value of X for each batch in the streams. Due to the continuous and unbounded nature of data streams, it may end up storing a potentially infinite list in each node. Hence, it is better to apply the space-saving algorithm. To extend the TUF-streaming(Space) algorithm for the landmark model, we again set the time-fading factor α to 1. As a result, the extended TUF-streaming(Space) algorithm—called **XTUF-streaming(Space)**—simplifies the recursive Equation (4) into the following:

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \left[expSup\left(X, \cup_{i=1}^{T-1} B_i\right) \times \alpha\right] + expSup(X, B_T)$$
$$= expSup\left(X, \cup_{i=1}^{T-1} B_i\right) + expSup(X, B_T). \quad (7)$$

By doing so, the XTUF-streaming(Space) algorithm keeps only a *single* value—i.e., $expSup(X, \cup_{i=1}^{T} B_i)$ —instead of a potentially infinite list of $expSup(X, B_i)$.

Example 9. Let us revisit Example 2, but let us use the *landmark model* here. Then, the XTUF-streaming(Space) algorithm uses Equation (7) to compute expected support values. As shown in Fig. 7(a), the expected support values stored



(a) After mining B_1. **(b)** After mining B_2. **(c)** After mining B_3.

Fig. 7. The UF-stream structures for the XTUF-streaming(Space) algorithm

in the resulting UF-stream after mining "frequent" patterns from B_1 are identical to those shown in Fig. 2(a) and 4(a).

Afterwards (say, after mining B_i for $i \geq 2$), instead of appending the expected support values of "frequent" patterns mined from B_i, XTUF-streaming(Space) updates the stored value. For instance, after mining B_2, instead of storing [1.44, 0.8] for $\{b, c\}$ as in Fig. 2(b) for both the TUF-streaming(Naive) and XTUF-streaming(Naive) algorithms, XTUF-streaming(Space) stores their sum 1.44+0.8 = 2.24 as $expSup(\{b, c\}, B_1 \cup B_2)$ in Fig. 7(b). Similarly, after mining B_3, instead of storing [1.44, 0.8, 0] as shown in Fig. 2(c) for both TUF-streaming(Naive) and XTUF-streaming(Naive), the XTUF-streaming(Space) algorithm stores their sum $(1.44 + 0.8) + 0 = 2.24$ as $expSup(\{b, c\}, B_1 \cup B_2 \cup B_3)$ in Fig. 7(c). □

5.3 XTUF-Streaming(Time): Extended TUF-Streaming(Time) for the Landmark Model

Recall that the TUF-streaming(Space) and XTUF-streaming(Space) algorithms greatly reduce the amount of space required when mining "frequent" patterns from uncertain data streams with the time-fading and landmark models, respectively. However, the two algorithms need to visit every node in the UF-stream after mining each batch. When using landmark model, if a pattern X does not appear in a particular batch (say, B_i), its expected support $expSup(X, B_i)$ would be 0. It would be a waste of time to traverse this node and update its expected support by adding 0. Hence, it is desirable to have an algorithm that requires less time for mining. Recall that the TUF-streaming(Time) algorithm does not need to visit every node in the UF-stream after mining each batch. Consequently, the number of nodes visited at each batch is *proportional to* the number of "frequent" patterns mined from that batch. In other words, the algorithm visits only those nodes representing patterns that are "frequent" in that batch. However, it keeps an extra field (i.e., the "last visit" field to indicate when was the last time a particular node is visited), which is needed to adjust the appropriate time fading factor α. In other words, with the time-fading model, TUF-streaming(Space) requires less space, but it needs to visit every node in the UF-stream. Conversely, TUF-streaming(Time) requires less time, but it needs an additional field for every node in the UF-stream. It is a space-time tradeoff.

Fortunately, when using the landmark model (i.e., α=1), we can get the benefits of both worlds. The reason is that, when using the time-fading model, the extra field LV is used to compute the appropriate power for the time fading factor α (i.e., α^{T-LV}, where T is the current time). However, when using the landmark model, $\alpha = 1$ implies that $\alpha^{T-LV} = 1$ (regardless of the value of LV). Specifically, Equation (5) becomes the following:

$$\begin{aligned} expSup\left(X, \cup_{i=1}^{T} B_i\right) &= \left[expSup\left(X, \cup_{i=1}^{LV} B_i\right) \times \alpha^{T-LV}\right] + expSup(X, B_T) \\ &= expSup\left(X, \cup_{i=1}^{LV} B_i\right) + expSup(X, B_T), \end{aligned} \quad (8)$$

where LV is the batch number in which X was last visited (i.e., when X was "frequent"). Note that, between T and the last visit LV of X, the expected support of X remains unchanged, i.e., $expSup\left(X, \cup_{i=1}^{LV} B_i\right) = ... = expSup\left(X, \cup_{i=1}^{T-1} B_i\right)$ because $expSup(X, B_{LV+1}) = ... = expSup(X, B_{T-1}) = 0$. The extended TUF-streaming(Time) for the landmark model—called **XTUF-streaming(Time)**—only needs to visit those nodes corresponding to "frequent" patterns mined from each batch, and it does not need to keep track of when they were last visited.

Example 10. Let us revisit Example 4. When using XTUF-streaming(Time), we do *not* need to store any additional field such as "last visit" (as we did in TUF-streaming(Time)) in the nodes in the UF-stream. Moreover, we no longer need to visit every node in the UF-stream after mining a batch (as we did in XTUF-streaming(Space)). This is a reduction in both space and time.

Our XTUF-streaming(Time) algorithm uses Equation (8) to compute expected support values to be stored in the UF-stream for the landmark model. Note that the algorithm visits only "frequent" nodes. For instance, after mining B_1, the algorithm visits and stores eight expected support values (i.e., 1.7, 1.8, 1.44, 0.86, 1.08, 1.6, 0.96 & 1.3 for "frequent" patterns $\{a\}, \{b\}, \{b, c\}$, $\{b, c, d\}, \{b, d\}, \{c\}, \{c, d\}$ & $\{d\}$, respectively) in the UF-stream. The contents (i.e., "frequent" patterns and their expected support values) of this UF-stream are identical to those shown in Fig. 7(a).

Then, for B_2, only six patterns are "frequent": $\{a\}, \{a, d\}$, $\{b\}, \{b, c\}, \{c\}$ & $\{d\}$. XTUF-streaming(Time) only visits and updates these six nodes. For instance, it visits the node for $\{b, c\}$ and updates its expected support (to 2.24) by adding $expSup(\{b, c\}, B_2)$=0.8 to the old value 1.44. For patterns that are not "frequent" (e.g., $expSup(\{b, d\}, B_2)$=0), the algorithm delays the visit to those nodes.

After mining B_3, the XTUF-streaming(Time) algorithm visits and updates four nodes, including $\{b, d\}$. For instance, the node is represented as d:2.52, which indicates that $expSup(\{b, d\}, B_1 \cup B_2 \cup B_3) = 2.52$. Here, XTUF-streaming(Time) visits nodes only for "frequent" patterns mined in a batch. It requires shorter runtime than XTUF-streaming(Space), which visits every node in the UF-stream. Again, the contents of this UF-stream are identical to those shown in Fig. 7(c). □

6 Analytical Evaluation

In this paper, we proposed (i) a family of three TUF-streaming algorithms for discovering "frequent" patterns from streams of uncertain data when using the time-fading model and (ii) another family of three TUF-streaming algorithms for the landmark model. In this section, we analyze these algorithms.

6.1 Memory Consumption

With respect to the *memory requirement*, let $|FP_i|$ denote the number of "frequent" patterns mined from Batch B_i. When using the *time-fading model*, the TUF-streaming(Naive) algorithm requires the largest amount of space as it requires $T \times |\cup_i FP_i|$ expected support values to be stored in the UF-stream (where T is the number of batches mined so far at time T). In contrast, the TUF-streaming(Space) algorithm requires the least amount of space because each node only stores a single value (i.e., a total of $|\cup_i FP_i|$ values), whereas TUF-streaming(Time) requires slightly more space than TUF-streaming(Space) because each node needs to keep the "last visit" field in addition to the usual expected support value for each "frequent" pattern (i.e., a total of $|\cup_i 2FP_i|$ values). However, it is bounded (cf. an unbounded or potentially infinite list of expected support values in TUF-streaming(Naive)). Moreover, such a slight increase in space usually pays off as it reduces the runtime.

Similar comments apply to the XTUF-streaming algorithms for mining "frequent" patterns with the *landmark model*, except that the amount of space required by the XTUF-streaming(Time) algorithm is identical to that required by the XTUF-streaming(Space) algorithm. Specifically, XTUF-streaming(Naive) stores $T \times |\cup_i FP_i|$ expected support values in the UF-stream (where T is the number of batches mined so far at time T), whereas XTUF-streaming(Space) only stores a single value for each node for a total of $|\cup_i FP_i|$ up-to-date expected support values. Unlike TUF-streaming(Time) (which stores both the expected support value and the "last visit" field in each node for a total of $|\cup_i 2FP_i|$ values), XTUF-streaming(Time) does not need to store the "last visit" field. Hence, it stores only $|\cup_i FP_i|$ values, which is identical to the memory requirement for XTUF-streaming(Space).

6.2 Runtime

As for the *runtime* for the *time-fading model*, both TUF-streaming(Naive) and TUF-streaming(Space) algorithms are required to visit every node in the UF-stream regardless of whether or not the corresponding pattern is "frequent", i.e., visit $|\cup_i FP_i|$ nodes for each of T batches for a total of $T \times |\cup_i FP_i|$ visits (e.g., visited 26 nodes for Fig. 2 or Fig. 4). In contrast, TUF-streaming(Time) only visits those nodes corresponding to "frequent" patterns, and it requires $\sum_i |FP_i|$ visits (e.g., visited 19 nodes for Fig. 6).

Again, similar comments apply to the XTUF-streaming algorithms when mining "frequent" patterns with the *landmark model*, except that these algorithms

require shorter runtime than those with the time-fading model due to simpler calculations. Similar to the simplification of Equation (3) to Equation (6), computation of expected support values become simpler when α=1 (for the landmark model) because the terms $\left[expSup\left(X, \cup_{i=1}^{T-1} B_i\right) \times \alpha\right]$ in Equation (4) and $\left[expSup\left(X, \cup_{i=1}^{LV} B_i\right) \times \alpha^{T-LV}\right]$ in Equation (5) can be simplified to become $expSup\left(X, \cup_{i=1}^{T-1} B_i\right)$ and $expSup\left(X, \cup_{i=1}^{LV} B_i\right)$, respectively. Moreover, as XTUF-streaming(Time) only visits those nodes corresponding to "frequent" patterns, it requires shorter runtime than TUF-streaming(Time) due to the absence of the "last visit" field.

7 Experimental Evaluation

Managing streaming data originated by intermittent sources (like sensors) is still an open problem for database and data mining research (e.g., [28,29]). Hence, assessing and reliably evaluating performance plays a critical role for any stream mining algorithm. Following this idea, we conducted an extensive experimental campaign, whose results are presented in this section.

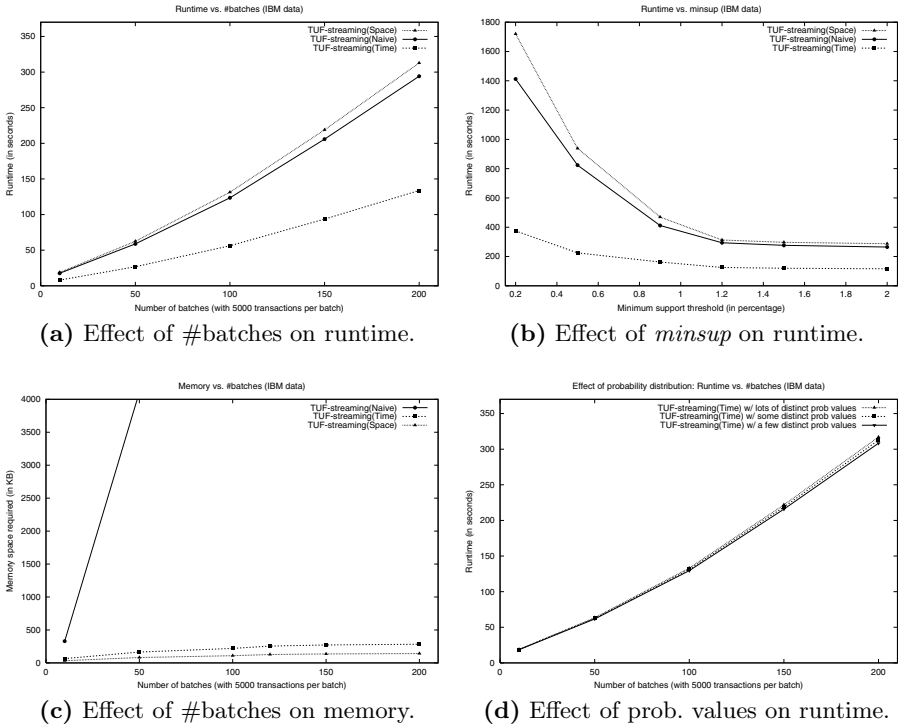
Different datasets, which included IBM synthetic data and UCI real data, were used for experimental evaluation. The reported results are based on the average of multiple runs in a time-sharing environment using an 800 MHz machine. Runtime includes CPU and I/Os for mining of "frequent" patterns and maintenance of the UF-stream structure. We evaluated different aspects of our proposed algorithms, which were implemented in C.

7.1 Evaluating TUF-streaming with IBM Synthetic Data

Among different datasets, we used an IBM synthetic data with 1M records with an average transaction length of 10 items and a domain of 1,000 items. We assigned an existential probability from the range (0,1] to every item in each transaction. We set each batch to contain 5,000 transactions (for a maximum of w=200 batches).

First, we evaluated the functionality of our proposed algorithms. When using the time-fading model, TUF-streaming(Naive), TUF-streaming(Space) and TUF-streaming(Time) all gave the same collection of frequent patterns. Similarly, when using the landmark model, the three XTUF-streaming algorithms—i.e., XTUF-streaming(Naive), XTUF-streaming(Space) as well as XTUF-streaming(Time)—all gave the same collection of frequent patterns. Moreover, when α=1, all six algorithms produced the same collection of frequent patterns, but their runtime varied.

In terms of runtime, when the number of batches (w) increased, the runtime increased. See Fig. 8(a) for the time-fading model. Among the three algorithms, TUF-streaming(Naive) and TUF-streaming(Space) took almost the same amount of time. The former appended the expected support values of "frequent" patterns discovered from a new batch whenever the batch was processed and mined, whereas the latter took slightly more time to update the expected support due to multiplication and addition. Both algorithms visited all nodes in the UF-stream. In



(a) Effect of #batches on runtime. **(b)** Effect of *minsup* on runtime.

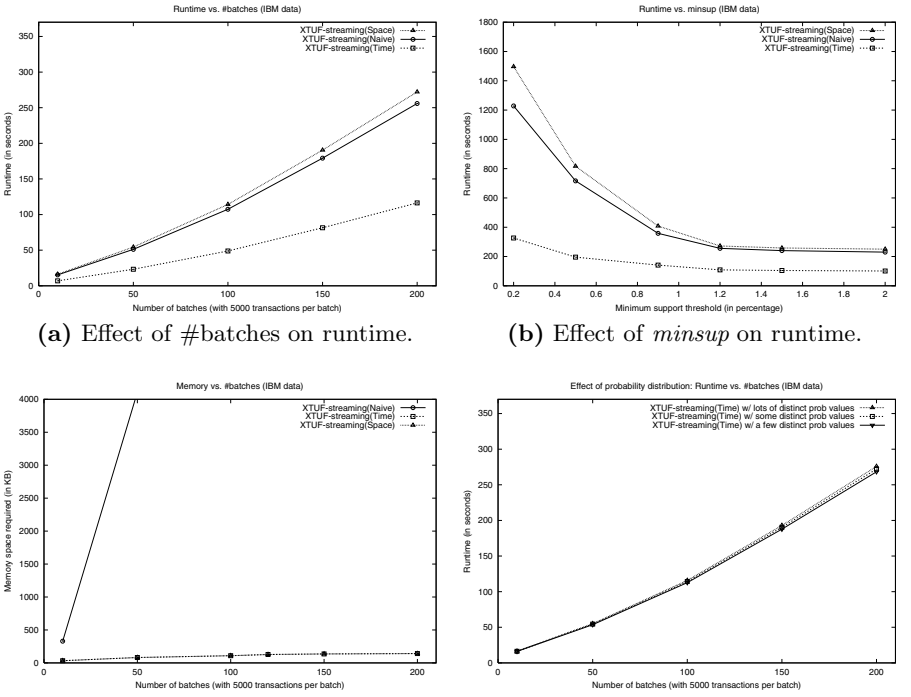
(c) Effect of #batches on memory. **(d)** Effect of prob. values on runtime.

Fig. 8. Experimental results of TUF-streaming algorithms on IBM synthetic data

contrast, after mining each batch, TUF-streaming(Time) took shorter runtime because the algorithm visited only nodes corresponding to the patterns discovered from that batch. For example, TUF-streaming(Naive) and TUF-streaming(Space) visited an average of about 22K nodes per batch, whereas TUF-streaming(Time) visited about 35K nodes for the entire mining process.

We also varied *minsup* values. Fig. 8(b) shows that, when *minsup* increased, the number of expected support values stored in the UF-stream structure decreased for all algorithms because the number of "frequent" patterns mined from the stream decreased.

Next, we evaluated the memory consumption of our algorithms. Fig. 8(c) shows that, when the number of batches increased, the number of expected support values stored in the UF-stream increased. For TUF-streaming(Naive), the increase was almost linear as the list of expected support values in each node increased proportional to the number of batches. Moreover, as data are not necessarily uniformly distributed, different patterns can be discovered from different batches. These add a few patterns to the collection of patterns to be kept in the UF-stream. In contrast, as TUF-streaming(Space) only kept a single value for each node, its memory consumption was independent of the number of batches.



(a) Effect of #batches on runtime. **(b)** Effect of *minsup* on runtime.

(c) Effect of #batches on memory. **(d)** Effect of prob. values on runtime.

Fig. 9. Experimental results of XTUF-streaming algorithms on IBM synthetic data

TUF-streaming(Time) occupied twice the amount of space when compared with TUF-streaming(Space) due to the extra "last visit" field in each node.

Furthermore, we also tested the effect of the distribution of item existential probability. When items took on a few distinct existential probability values, UF-trees used in UF-growth became smaller. Regardless of the size of UF-trees, the number of "frequent" patterns returned by UF-growth (i.e., the number of nodes kept in the UF-stream structure) was *not* different significantly. Hence, as shown in Fig. 8(d), the runtime for the algorithms was very similar.

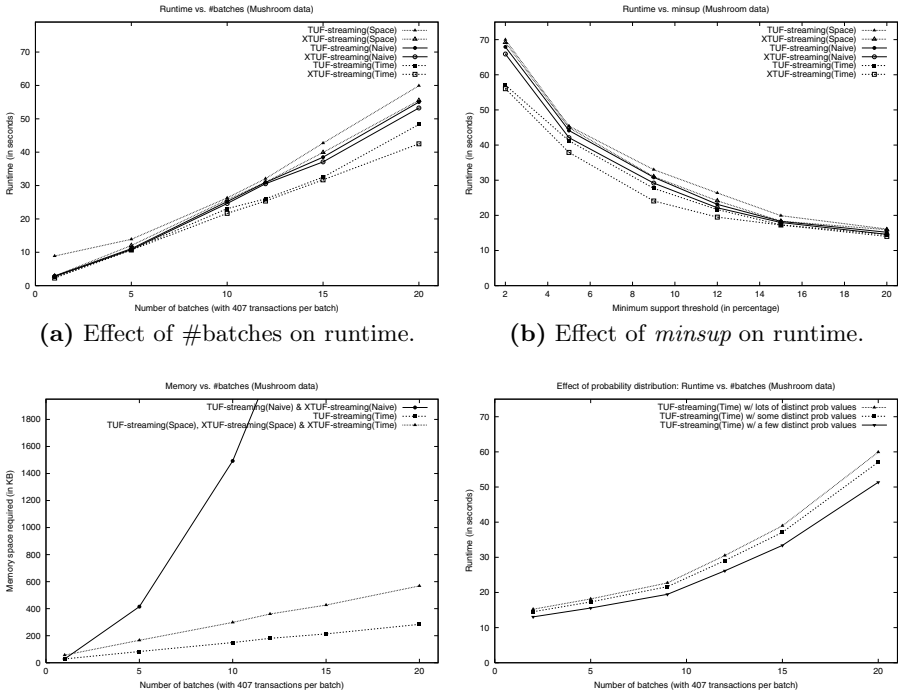
7.2 Evaluating XTUF-streaming with IBM Synthetic Data

Next, we repeated the above experiments when conducting experiments with the family of three XTUF-streaming algorithms. For the landmark model, the trends were very similar. One difference was that runtime of the algorithms using the landmark models was slightly shorter due to the simplicity of the computation (e.g., computation did not involve the multiplication of α). As evidenced by Fig. 9(a), when the number of batches (w) increased, the runtime increased. The ranking for the three algorithms was same as the ranking for their correspond-

ing counterparts in the TUF-streaming family. XTUF-streaming(Naive) and XTUF-streaming(Space) took almost the same amount of time due to the absence of multiplication (i.e., only addition) operations. XTUF-streaming(Time) took the shortest runtime because the algorithm visited only the nodes corresponding to the patterns discovered from that batch.

As shown in Fig. 9(b), when *minsup* increased, the number of expected support values stored in the UF-stream structure decreased for all three XTUF-streaming algorithms because the number of "frequent" patterns mined from the stream decreased.

Fig. 9(c) shows that, when the number of batches increased, the number of expected support values stored in the UF-stream also increased. As expected, XTUF-streaming(Naive) increased almost linearly because the list of expected support values in each node increased proportional to the number of batches. In contrast, as XTUF-streaming(Space) only kept a single value for each node, its memory consumption was independent of the number of batches. Unlike the TUF-streaming(Time) algorithm (which occupied twice the amount of space as required TUF-streaming(Space)), the XTUF-streaming(Time) algorithm occupied the same amount of space as XTUF-streaming(Space) due to the absence of "last visit" field from each node.



(a) Effect of #batches on runtime. **(b)** Effect of *minsup* on runtime.

(c) Effect of #batches on memory. **(d)** Effect of prob. values on runtime.

Fig. 10. Experimental results on the UCI Mushroom data

The distribution of item existential probability did not play a significant role in making significant changes in runtime. As shown in Fig. 9(d), the runtime for XTUF-streaming to handle the three different distributions was very similar.

7.3 Evaluating with UCI Real Mushroom Data

In addition to conducting experiments with the IBM synthetic data, we also evaluated our algorithms using UCI real data. For instance, we used the mushroom data, which consists of 8,124 transactions with an average transaction length of 23 items and a domain of 119 distinct items. The experimental results, as shown in Fig. 10, are consistent with those experimented with the IBM synthetic data.

8 Conclusions

In this paper, we proposed tree-based mining algorithms that mine frequent patterns from dynamic streams of uncertain data with both time-fading and landmark models. All algorithms apply UF-growth with *preMinsup* to find "frequent" patterns. The mined patterns are then stored in the UF-stream structure together with their expected support values. Then, when the next batch of streaming transactions flows in, the algorithms update the UF-stream structure accordingly. Differences among the proposed algorithms are that the naive algorithms—i.e., TUF-streaming(Naive) and XTUF-streaming(Naive)—keep a potentially infinite list of expected support values for each node in UF-stream. The space-saving algorithms—i.e., TUF-streaming(Space) and XTUF-streaming(Space)—reduce the memory consumption by keeping only a single value for each node. In contrast, the time-saving algorithms—i.e., TUF-streaming(Time) and XTUF-streaming(Time)—visit only those nodes corresponding to "frequent" patterns. The family of TUF-streaming algorithms mines "frequent" patterns with the time-fading model, whereas the family of XTUF-streaming algorithms (i.e., the extended TUF-streaming algorithms) mines with the landmark model. We also discussed how to handle situations where batches of data streams do not arrive in sequential order. Analytical and experimental results showed the space and time effectiveness of our proposed algorithms when using the time-fading and landmark models for mining frequent patterns from streams of uncertain data.

Acknowledgements. This project is partially supported by NSERC (Canada) and University of Manitoba.

References

1. Aggarwal, C.C., Li, Y., Wang, J., Wang, J.: Frequent pattern mining with uncertain data. In: ACM KDD, pp. 29–37 (2009)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB, pp. 487–499. Morgan Kaufmann (1994)

3. Calders, T., Garboni, C., Goethals, B.: Efficient Pattern Mining of Uncertain Data with Sampling. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part I. LNCS, vol. 6118, pp. 480–487. Springer, Heidelberg (2010)
4. Cao, F., Ester, M., Qian, W., Zhou, A.: Density-based clustering over an evolving data stream with noise. In: SDM, pp. 328–339. SIAM (2006)
5. Castellanos, M., Gupta, C., Wang, S., Dayal, U.: Leveraging web streams for contractual situational awareness in operational BI. In: EDBT/ICDT Workshops, article 7. ACM (2010)
6. Chen, Y., Nascimento, M.A., Ooi, B.C., Tung, A.K.H.: SpADe: on shape-based pattern detection in streaming time series. In: IEEE ICDE, pp. 786–795 (2007)
7. Cuzzocrea, A.: CAMS: OLAPing Multidimensional Data Streams Efficiently. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 48–62. Springer, Heidelberg (2009)
8. Cuzzocrea, A.: Retrieving Accurate Estimates to OLAP Queries over Uncertain and Imprecise Multidimensional Data Streams. In: Cushing, J.B., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 575–576. Springer, Heidelberg (2011)
9. Cuzzocrea, A., Chakravarthy, S.: Event-based lossy compression for effective and efficient OLAP over data streams. Data & Knowledge Engineering 69(7), 678–708 (2010)
10. Cuzzocrea, A., Leung, C.K.-S.: Frequent itemset mining of distributed uncertain data under user-defined constraints. In: SEBD, pp. 243–250 (2012)
11. Cuzzocrea, A., Furfaro, F., Mazzeo, G.M., Saccá, D.: A Grid Framework for Approximate Aggregate Query Answering on Summarized Sensor Network Readings. In: Meersman, R., Tari, Z., Corsaro, A. (eds.) OTM Workshops 2004. LNCS, vol. 3292, pp. 144–153. Springer, Heidelberg (2004)
12. Ezeife, C.I., Zhang, D.: TidFP: Mining Frequent Patterns in Different Databases with Transaction ID. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 125–137. Springer, Heidelberg (2009)
13. Gaber, M.M., Zaslavsky, A.B., Krishnaswamy, S.: Mining data streams: a review. SIGMOD Record 34(2), 18–26 (2005)
14. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities. In: Data Mining: Next Generation Challenges and Future Directions, pp. 105–124. AAAI/MIT Press (2004)
15. Gupta, A., Bhatnagar, V., Kumar, N.: Mining Closed Itemsets in Data Stream Using Formal Concept Analysis. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2010. LNCS, vol. 6263, pp. 285–296. Springer, Heidelberg (2010)
16. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM SIGMOD, pp. 1–12 (2000)
17. Jiang, N., Gruenwald, L.: Research issues in data stream association rule mining. SIGMOD Record 35(1), 14–19 (2006)
18. Leung, C.K.-S.: Mining uncertain data. WIREs Data Mining and Knowledge Discover 1(4), 316–329 (2011)
19. Leung, C.K.-S., Hao, B.: Mining of frequent itemsets from streams of uncertain data. In: IEEE ICDE, pp. 1663–1670 (2009)
20. Leung, C.K.-S., Mateo, M.A.F., Brajczuk, D.A.: A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 653–661. Springer, Heidelberg (2008)
21. Leung, C.K.-S., Sun, L.: Equivalence class transformation based mining of frequent itemsets from uncertain data. In: ACM SAC, pp. 983–984 (2011)

22. Leung, C.K.-S., Tanbeer, S.K.: Fast Tree-Based Mining of Frequent Itemsets from Uncertain Data. In: Lee, S.-g., Peng, Z., Zhou, X., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012, Part I. LNCS, vol. 7238, pp. 272–287. Springer, Heidelberg (2012)
23. Leung, C.K.-S., Tanbeer, S.K.: Mining Popular Patterns from Transactional Databases. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2012. LNCS, vol. 7448, pp. 291–302. Springer, Heidelberg (2012)
24. Leung, C.K.-S., Tanbeer, S.K., Budhia, B.P., Zacharias, L.C.: Mining probabilistic datasets vertically. In: IDEAS 2012, pp. 199–204. ACM (2012)
25. Leung, C.K.-S., Jiang, F.: Frequent Pattern Mining from Time-Fading Streams of Uncertain Data. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 252–264. Springer, Heidelberg (2011)
26. Ng, W., Dash, M.: Discovery of Frequent Patterns in Transactional Data Streams. In: Hameurlain, A., Küng, J., Wagner, R., Pedersen, T.B., Tjoa, A.M. (eds.) TLDKS II. LNCS, vol. 6380, pp. 1–30. Springer, Heidelberg (2010)
27. Yu, J.X., Chong, X., Lu, H., Zhou, A.: False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: VLDB, pp. 204–215. Morgan Kaufmann (2004)
28. Yu, B., Cuzzocrea, A., Jeong, D.H., Maydebura, S.: On managing very large sensor-network data using Bigtable. In: IEEE/ACM CCGrid, pp. 918–922 (2012)
29. Yu, B., Cuzzocrea, A., Jeong, D., Maydebura, S.: A Bigtable/MapReduce-Based Cloud Infrastructure for Effectively and Efficiently Managing Large-Scale Sensor Networks. In: Hameurlain, A., Hussain, F.K., Morvan, F., Tjoa, A.M. (eds.) Globe 2012. LNCS, vol. 7450, pp. 25–36. Springer, Heidelberg (2012)

Author Index