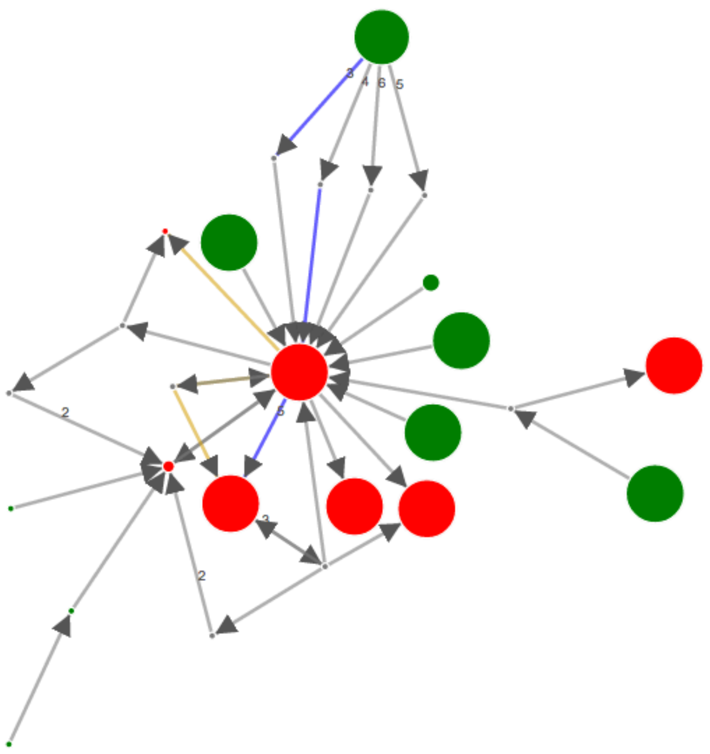
University of Zurich^UZH

Dynamic and Distributed Information Systems

Master Thesis Dec 31, 2013

Scalable Forensic Transaction Matching

And its application for detecting patterns of fraudulent financial transactions.



Daniel Strebel
of Zurich ZH, Switzerland

Student-ID: 07-908-072
strebel@ifi.uzh.ch

Advisor: **Philip Stutz**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First and foremost, I would like to thank Prof. Abraham Bernstein for introducing me to this very interesting topic and giving me the opportunity to closely collaborate with his Dynamic and Distributed Information Systems (DDIS) research group. I am deeply grateful to my advisor Philip Stutz for his critical but always constructive feedback and for being a mentor and friend for many years. I would like to thank the entire DDIS research group for countless inspiring conversations and playful working environment.

I would also like to express my thanks to Stefan Schurgast, Luzius Meisser and Stefan Amstein for providing my with valuable insights or data sets that contributed to this thesis.

Last but not least, I am very grateful to my friends and family for their support during all my years at the university.

Zusammenfassung

Die Erkennung von Betrugsmustern in grossen Datenmengen von finanziellen Transaktionsdaten stellt ein integraler Bestandteil forensischer Untersuchungen dar und findet Anwendung bei der Identifizierung von Geldwäscherei, internem Mitarbeiterbetrug und anderen illegalen Aktivitäten. Um diese grossen Datenmengen verarbeiten zu können, werden skalierbare und flexible Lösungen benötigt.
Diese Arbeit stellt ein neues Verfahren vor, bei dem Abhängigkeiten zwischen eingehenden und ausgehenden Transaktionen gesucht werden und die daraus entstehenden Komponenten zu grösseren Mustern verbunden werden. Die daraus resultierenden Muster können dann gefiltert und mithilfe einer speziellen graphischen Benutzeroberfläche untersucht werden.
Die Evaluation dieses Verfahrens zeigt, dass es eine kontinuierliche Überprüfung von realen Finanztransaktionen mit einem Durchsatz von über einer Million Transaktionen pro Minute ermöglicht. Zudem zeigen wir, dass unser System in der Lage ist, sechs komplexe Geldwäschereimuster auszudrücken und zu erkennen.
Das in dieser Arbeit vorgestellte System ist, nach userem Wissen, das erste welches in der Lage ist, skalierbar und aufgrund lokaler Informationen Finanzbetrugsmuster zu erkennen und dazu eine Abfragesprache bereitstellt, die eine breite Spanne an finanziellen Betrugsmuster ausdrücken kann.

Abstract

The detection of fraudulent patterns in large sets of financial transaction data is a crucial task in forensic investigations of money laundering, employee fraud and various other illegal activities. Scalable and flexible tools are needed to be able to analyze these large amounts of data and express the complex structures of the patterns that should be detected.

This thesis presents a novel approach of locally identifying associations between incoming and outgoing transactions for each participant of the transaction network and then aggregating these associations to larger patterns. The identified patters can be pruned and visualized in a graphical user interface to conduct further investigations.

The evaluation of our approach shows that it allows a stream-processing of real-world financial transactions with a throughput of more than one million transactions per minute. Furthermore we demonstrate the capability of our approach to express six sophisticated money laundering patterns, as reported by the Egmont group, and successfully retrieve components that correspond to these patterns.

To the best of our knowledge, this approach is the first to scalably identify dependent financial transactions based on local transaction matching, while providing a flexible query language to cover a broad range of financial fraud cases.

Table of Contents

1

Introduction

Graphs are known to provide an intuitive representation of otherwise complicated relationships and are therefore widely used to describe and analyze data in a broad range of application domains. The processing and analysis of static graphs, has seen a lot of research interest in recent times and has even been integrated in a number of industrial products. However, many interesting relationships cannot sufficiently be modelled as static graph structures, because the interaction between two individuals is dependent on a temporal dimension. The time attributes describes when an interaction took place or introduces a temporal ordering of relationships. This thesis focuses on the analysis of transactions, a subset of temporal relationships with additional properties. In addition to the temporal dimension, transactions also carry a payload, which could be information on a banking transaction, the message content of an email or many other forms of information. Based on this payload and the aforementioned temporal dimension, transactions are inherently well suited to express many interactions between individuals. Moreover by connecting transactions in a meaningful way, one can easily retrace flows of interactions between parties. Detecting these flows is essential for a broad range of application scenarios, some of which include the detection of money laundering schemas, threats to national security or to better understand word of mouth diffusion in social networks. This thesis presents a framework that enables the detection of interesting transaction patterns through fast and scalable transaction matching and a subsequent analysis of the reported components to assess their relevance for the use case at hand. To demonstrate the efficiency and expressiveness of the framework, we show how our approach is capable of capturing fraudulent financial transaction patterns in a data set of several million transactions. Even though the description in this thesis focuses on the task of finding fraudulent financial transactions, the underlying transaction matching approach is not limited to matching financial transactions and could easily be extended to be used in other application domains.

The problem addressed in this thesis is manifold and one of the most challenging factors is the sheer size of data that typically needs to be processed when investigating fraudulent transaction patterns. With current social networks consisting of over one billion users (Facebook Inc. 2013), which actively interact with each other, or publicly available payment graphs consisting of several million transactions (Quandl 2013), already the detection of relatively simple transaction patterns presents a persistent problem to both researchers and practitioners. It is evident that these data sets require novel algorithms and tools that are able to scale to such large amounts of transactions in order to extract relevant information. The size of graph structured data sets also suggests that there is currently a lot of information that can not be retrieved efficiently for the lack of appropriate tools, which are able to perform at these scales. The domain of graph mining has hence become a very active field of research with varying levels of maturity in different areas. A number of systems have been proposed to facilitate the processing of large graph-based data sources. Their focus, however, often lies on the computation of metrics of the graph structure rather than the detection of interesting subsets of the graph. Apart from the problem of size of the initial data set, the detection of transactional patterns presents an additional challenge because of the inherent properties of transactions themselves. While most graph mining approaches have looked at static properties of graphs, mining transactional graphs introduces additional complexity. A transactional graph not only consists of a set of vertices and a set of edges but also a temporal dimension, which can influence the states of all graph elements and possibly also their connectivity.

For detecting patterns in transactional data sets, the temporal dimension is a critical element in order to find associations between individual transactions. The approach described in the subsequent sections is able to not only consider the temporal dimension, when analyzing interactions of entities in the graph, but also exploits it in order to optimize its memory consumption and computational complexity. Based on the timestamps of the transactions and their payload the system decides whether or not transactions are dependent on each other and should be considered as a potential elements of a flow of transactions. For an optimal parallelization of the detection process the decision about which transactions should be matched is entirely done based on local information. The matching of the transactions is performed without any direct intervention of a user, however the system allows the user to specify the structure of interesting transactional patterns in advance. Once a transactional pattern is detected, the users conditions are applied in order to prune the patterns. The users conditions can involve simple queries about the reported pattern such as its size or require the execution of graph analysis algorithms to obtain more complex statistics about the pattern that was reported. The detection approach is designed to work on continuous data and report patterns, which were detected and considered relevant, immediately to the user.

1.1 Goal of this Thesis

The goal of this thesis is to demonstrate a novel approach of detecting patterns of transactions by locally matching dependent transactions and enforcing conditions on the resulting components. For the scope of this thesis we focus on transaction graphs in the environment of financial transactions, however the approach of how transactions are matched and the processing workflow are more generic and able to be applied to other use cases as well. For the purpose of parallelizing local computations, the system described in this thesis is built on top of Signal/Collect (Stutz, Bernstein, and Cohen 2010) a framework for parallel graph processing of large data sets. To allow for a broad range of computations on top of a given graph topology, Signal/Collect needs to be extended to support a change of vertex behaviour during an execution. Based on the associations of transactions that were detected at a local scope the system needs to detect larger connected components and assess whether the resulting patten is interesting. To evaluate the detection approach, we evaluate it on a set of 50 million Bitcoin transactions as well as a set of 1.8 million synthetic banking transactions. Because of the lack of ground truth in both of these data sets, the recall of the system is demonstrated by manually injecting known fraudulent patterns and retrieving them with the help of the system and a set of specified conditions on the pattern structures. The detection system should be designed to be used by domain experts with limited programming experience. The domain experts should therefore be able to express the detection process in a higher-level specification, where the conditions that the detected patterns have to fulfil and other execution parameters can be specified. In a final step, the system provides ad-hoc analysis component in the form of a graphical user interface. A specialized visualizer allows the investigator to analyze the patterns that were found and provides him with additional information.

1.2 Thesis Structure

The rest of this thesis is structured as follows: In the next chapter an overview over existing graph mining approaches and techniques is given. The third chapter gives a more specific description of selected areas within the domain of fraudulent transaction detection and introduces previous work in this area. In a second part it introduces the concept of matching transactions for the purpose of detecting financial fraud patterns and presents our matching approach at an implementation-independent level. The fourth chapter describes an end-to-end implementation of our transaction matching approach by explaining the different components of the detection system individually. The main focus hereby lies on the detection workflow that is responsible for matching transactions and detecting relevant connected components. In addition to that, it also describes the fundamental architecture components that allow for pluggable vertex implementations on top of the Signal/Collect programming abstraction. The chapter concludes with a description of the user facing features such as the execution description, which is used to specify an execution, and the different components that can be used to analyze the reported results. In the fifth chapter the detection system is evaluated for two different use cases within the domain of financial fraud detection. The evaluation analyzes the expressiveness and scalability of the proposed system as well as the recall of manually injected patterns. The thesis concludes with an outline of limiting factors and recommendations that have to be considered when building systems for scalable transaction matching and transactional pattern querying.

2

Related Work

Computer aided tools and techniques have been used to detect cases of fraud in the context of forensic investigations for many years. Given the large amounts of data that need to be considered in these kinds of activities, it is obvious that automated algorithmic procedures are advantageous compared to purely relying on human experts to detect and investigate suspicious cases. This section gives an overview over existing technologies that can be used in order to support the detection of suspicious transaction. In addition to that, we also describe a selection of techniques that do not directly target the application domain of matching fraudulent transactions, but provide lower-level foundations. They can therefore be viewed as the building blocks towards constructing a productive system. In addition to these low-level approaches, we will also show, how these approaches are currently used in order to detect fraudulent transactions and discuss their advantages and disadvantages.

2.1 Traditional Approaches

When trying to detect relevant information about transactions and their interdependence, two common ways are either to look at the individual transactions as distinct entries in a input data set or extract information based on their relationships and the resulting graph structures. Since graph structures are built on top of the transactional data, this section first describes traditional approaches of how to extract information solely based on the information on individual transactions.

2.1.1 Declarative Query Languages

Rules are often the most straightforward way to express suspicious factors with respect to whether a transaction has a high probability to be fraudulent. Despite of their simplistic nature, simple rules have been proven to be effective in many cases (Egmont Group 2000). Declarative query languages are a simple and widely used method to retrieve desired information based on existing rule conditions. SQL and similar declarative query languages are very efficient at handling relational data and allow investigators to retrieve information with relatively low effort (Blakeley 2008). For this reason it is obvious that many companies still rely on periodically querying their data bases for occurrences of transactions, which they suspect to be fraudulent (Chang et al. 2007).

Despite its popularity, pure SQL is poorly suited for applications that require an analysis of highly interlinked data. For this reason, SPARQL (Prud'hommeaux and Seaborne 2008) has emerged as the standard language for querying relational data in RDF format. By design SPARQL is inherently well suited for querying relationships among objects and even supports some basic functionality to define patterns that are of interest to the issuer of the query. Because of its relatively simple design and similarity to SQL, SPARQL is heavily used to query static knowledge bases or snapshots of knowledge that changes infrequently. Extensions such as C-SPARQL (Barbieri et al. 2009) have been proposed to extend SPARQL with the capability of processing continuous flows of information. C-SPARQL allows to query RDF data within time intervals and creates new RDF data streams that contain the query results.

One significant drawback that most declarative query languages have in common is the missing ability to compute non-trivial, additional information that is not present in the initial data base. However the computation of additional properties can be highly relevant for the detection of fraudulent transaction patterns. A desirable property of a detection system would therefore include the direct translation of simple rule conditions, which are similar to the ones found in declarative query languages, combined with the functionality to express more advanced computations that exploit the relational nature of the transaction data.

2.1.2 Traditional (non-relational) Data Mining

A common definition of data mining states "data mining is the application of specific algorithms for extracting patterns from data" (Fayyad, Piatetsky-Shapiro, and Smyth 1996). While this definition covers the problem space of finding fraudulent transaction patterns, it is also obvious that this is a very generic classification and a lot of different techniques and algorithms fall under this definition. In this thesis and in particular in the context of detecting fraudulent transactions, the concept of a pattern is used to describe dependent interactions between instances in a network. In data mining however, the term pattern traditionally refers to patterns within the attribute structure of some individual entity.

Even though their main focus relies on the assessment of individual transactions, traditional data mining techniques have been successfully applied in practice and research. Especially in domains where the input data consists of a large number of data entries with many attributes, finding good classifiers that are able to detect occurrences of fraud is a non-trivial task. For many data-intensive domains such as credit card purchases (Chan et al. 1999; Bhattacharyya et al. 2011) or insurance claims (Derrig 2002), data mining techniques therefore play a significant role in analyzing the individual entries. Supervised approaches, where transactions are labeled with a risk score based on previously detected fraud cases are especially valuable for the area of fraud detection. Usually these investigations are based on trainable learning algorithms such as neural networks or support vector machines but also Bayesian networks and decision trees have been used to predict whether some occurrence is likely to be fraudulent (Wang 2010; Ngai et al. 2011).

2.2 Graph Mining

Graphs are a particularly interesting way of organizing information because they are able to naturally represent the topology of a target domain. By mapping entities to nodes that are interconnected by edges one explicitly expresses relationships and dependencies within the data. Because of their wide applicability and their structural advantages over traditional data representations, graphs offer additional capabilities for knowledge discovery in many ways. For the scope of this thesis, graph representations are applied in order to model temporal interactions between participants of a network. More specifically, graphs are used in the context of forensic investigations to detect interesting patterns of interactions between users. In this section a short description of selected graph mining sub-domains is given. We also highlight some prominent implementations in every area and analyze them with respect to their suitability for discovering fraudulent patterns.

2.2.1 Graph Pattern Matching

Graph pattern matching is the task of finding occurrences of instances of an abstract pattern graph a larger data graph. This matching is typically defined in terms of *subgraph isomorphism* (Ullmann 1976). The NP-Complete matching of structural properties of graph elements is computationally expensive and therefore usually requires an upfront candidate selection (Gallagher 2006). To reduce the computational complexity many modifications of graph simulation (Henzinger, Henzinger, and Kopke 1995) were proposed, but most of them result in the detection of patterns that differ significantly from the structure found in the pattern query. Other approaches are based on the formulation of reachability constraints by extending graph simulations with regular expressions (Fan et al. 2011a) or by bounded simulation (Fan et al. 2011b) that limits the number of allowed hops. Both approaches can be very efficient when the characteristics of the graph can be expressed based on the labels on the vertices. In our case, where also transactions between homogeneous entities should be considered, this constraint seems to be infeasible to enforce.

2.2.2 Unsupervised Graph Mining

While the methods described above required the user to explicitly specify what patterns to extract from the graph storage, the domain of unsupervised graph mining operates in an opposite direction. The problem at hand is much less specific and the goal is usually to gain new insights about the structure of the information as well as to find anomalies (Chandola, Banerjee, and Kumar 2009), which could lead to the detection of interesting patterns. Noble et al. presented an unsupervised system (Noble and Cook 2003) for detecting anomalies within graphs based on the information theoretic value of sub-graphs. This approach for detecting interesting patterns seems to perform particularly well in cases where the portion of anomalies within the graph is very rare and their structure is well distinguishable from regular occurrences. In our use cases, the graph structures of fraudulent transactions have a very high resemblance to regular transactions. Anomaly-based detection strategies are therefore not particularly well suited for our use case, because they would most likely lead to higher error rates of both types. However, unsupervised graph mining could be used complementary to the approach presented in this thesis in order to learn previously unknown patterns. These new patterns could be integrated in the detection work-flow to better assess the relevance of the reported components or find new types of fraud.

2.2.3 Graph Navigation Languages

Graph navigation is another approach that has proven to be useful for accessing information stored in graph representations. Compared to declarative query languages as well as the previously presented unsupervised graph mining approaches, graph navigation is placed at a much lower level of abstraction. While for the declarative query languages the underlying storage structure is transparent, graph navigation directly makes use of the graph structure of the data. Because expensive joins are avoided, this can lead to more efficient query processing in some use cases (Holzschuher and Peinl 2013), but the concept is less generally applicable. Usually, traversals are performed by using a piping mechanism, which feeds the output of one operation as an input to subsequent operations. The concept of low level traversal has already been used for XML traversals in the form of XPath[1]. The XPath language provides primitives to traverse an XML tree and retrieve one or more nodes or attributes. Similar approaches have been proposed to extract information from more general graph structures. With Gremlin[2] and Trails (Kröni and Schweizer 2013) two systems were developed to extend the Neo4j[3] graph database with low level graph traversal functionality. Gremlin allows users to define external storage structures that can be accessed and modified during graph traversals. Compared to Neo4j's declarative query language Cypher[4], Gremlin was found less intuitive to formulate more sophisticated graph queries (Holzschuher and Peinl 2013) but performed slightly better when operating on direct relationships between nodes in the graph. A significant drawback of current low level graph traversal languages with respect to our use case scenarios is the missing support for for querying based on their temporal dimension. In addition to that, the exact structure of the graph might be unknown and an unguided traversal through the graph, based on graph traversal functionality, would not yield any additional benefit. Graph navigation languages might however be a very promising approach to manually explore the vicinity of entities based on a reported component of suspicious transactions and could be applied in a subsequent investigation.

2.3 Graph Computation Frameworks

Specialized graph algorithms, e.g. PageRank, have been proven to scale nicely on top of general purpose parallelization infrastructures such as MapReduce or more specialized frameworks for graph processing such Singal/Collect (Stutz, Bernstein, and Cohen 2010) GraphLab (Low et al. 2010; Low et al. 2012), Pregel (Malewicz et al. 2010) or Giraph [5]. However the purpose of most of these systems is to show the parallelization of graph

[1]http://www.w3.org/TR/xpath20/ (visited on 11/04/2013)
[2]https://github.com/tinkerpop/gremlin (visited on 11/05/2013)
[3]http://www.neo4j.org/ (visited on 11/05/2013)
[4]http://docs.neo4j.org/chunked/stable/cypher-query-lang.html (visited on 11/05/2013)
[5]http://giraph.apache.org (visited on 12/24/2013)

computations and they provide relatively little insight about the information inside the graph apart from its structural properties. Based in the insights of the aforementioned parallelization infrastructure, more specialized systems have been presented that allow the user to query relational data in the form of RDF data graphs. Both TripleRush (Stutz et al. 2013) and Trinity (Shao, Wang, and Li 2013) have been proven to process pattern queries on large graphs but are limited with regard to their expressiveness of the query language. To the best of our knowledge, none of these systems allow to query for transactional patterns among entities in large graphs. Also none of these systems directly support relational queries with a temporal dimension. However such queries are of significant interest because a purely static analysis of a data graph is often insufficient for many application scenarios. Which opens the space for the focus of this thesis.

3

Algorithmic Transactional Fraud Detection

Detecting previously unknown and potentially illegal activities within large reports of transactions presents an ongoing challenge for many public and private organizations. A number of tool have been developed to allow domain experts to conduct investigative operations. Some early applications (Senator et al. 1995) provided a simple graphic user interface that allowed users to monitor and investigate transactions. However, these tools required the experts to issue precise queries in order to find the information that they were looking for. Over time it has become apparent that looking at individual data records only provides limited insight about actual events and that a more holistic view is needed, in order to detect more complex occurrences. One of the first productive systems to detect such more elaborate patterns in large amounts of data is the National Association of Securities Dealers' Regulation Advanced Detection System (ADS). The ADS was introduced in 1997 to monitor trades and quotations in the NASDAQ stock market (Kirkland et al. 1999). Cortes (Cortes, Pregibon, and Volinsky 2001) looked at transactional data within a telecommunication network and proposed an algorithm to identify so-called "Communities of Interest" which are likely to identify fraudulent participants. A manual analysis in such use cases would most often be too complex and not feasible for most application scenarios. To capture these occurrences of connected transactions, which can be associated to build patterns, tools have to go beyond looking at the individual transactions and also have to analyze the interdependence between them. In addition to the mere detection, the tools also have to be able decide whether a pattern is in some way violating policies or at least has a high probability to be considered fraudulent. This chapter first describes a selection of use cases where our detection approach could be applied in order to find fraudulent transaction patterns and then introduces the general idea in an implementation-independent way.

3.1 Application Domains and Previous Approaches

Financial transaction networks provide many areas that have needs for the deployment of algorithmic fraud detection tools. With credit card transactions being the area, which received the most attention (Wang 2010) within research but also in the industry, fraud detection in the financial domain has become a persistently active field of interest. Some algorithms in this field have proven to be generalizable enough to be applied to a number of problem domains. Fraud detection is therefore also ubiquitous in preventing money laundering or insurance fraud schemas.

Financial fraud can span a variety of different forms and includes cases of document forgery, unauthorized actions performed on managed bank assets but also transaction based fraud cases, where the delinquents are trying to obscure the true origin of a money stream (Egmont Group 2000). The motivations behind these patterns can have many origins and might allow the coordinating instance to perform some sort of money laundering or other forms of illegal payout schemas. The exact definition of these fraud cases requires a lot of explicit and implicit knowledge about the target domain and the category of fraud that should be investigated. However many fraud schemas that work by concatenating seemingly unsuspicious money transactions to achieve a higher-level goal. Even more interestingly these transaction patterns are often composed of a small number of relatively simple building blocks, that are combined to more complex smurfing schemas (Luell 2010). This property can be exploited in order to detect an initial underlying structure with regard to how transactions are dependent on each other.

3.1.1 Internal Financial Fraud Detection

Luell (Luell 2010) looked at company internal fraud cased that were reported by an undisclosed financial institution located in Switzerland. The study lists transaction chains and a number of smurfing patterns as central indicators for the occurrence of internal fraud cases. The proposed workflow for detecting suspicious transaction patterns is constructed by combining multiple processing steps. In a first step a so-called $BaseMatcher$ is applied to prune the search space and find an initial set of suspicious sub-graphs that confirm to a specified pattern. One implementation of such a $BaseMatcher$ was implemented in the form of a so-called $ChainFinder$ (Luell 2010; Amstein 2009). The $ChainFinder$ is a specialized form of tree search that allows the detection of tempo-relational patterns in a graph. Such patterns can include some form of chain transaction where money is forwarded from one account to its subsequent node in the chain or more general and complex smurfing patterns. As an intermediary result of the $BaseMatcher$, a so-called *AlertGraph* is generated that consists of all identified suspicious sub-patterns. In order to find more meaningful patterns, this *AlertGraph* is then further refined by expanding the patterns in the alert graph and performing some structural analysis. One proposed solution to perform these kinds of structural analysis is a System called $TMatch$ (Luell 2010; Moll 2009). $TMatch$ is itself a multi-step process and starts with a set of initial nodes which could be the result of the previously introduced $ChainFinder$. In a first step all the nodes are iteratively expanded if they fulfill all the required conditions for expanding and have not yet been expanded. In a second step the expanded structures are analyzed using a $ComponentFinder$ which identifies connected components and filters out structures that are not of interest. In a last step a number of scoring algorithms can be run on the remaining connected components, which results in a score of each component for each scoring aspect. These scores are then reported to the investigator of the inquiry together with the connected components. Unfortunately the neither the original data set nor concrete examples of specific fraud patterns were published by the author because of an non disclosure agreement with the participating financial institution. However we were given access to a data set of simulated transactions that were generated within the context of this work (Galliker 2008) in order to test the capabilities of our system to recognize patterns in a similar use case. Compared to the implementation in (Luell 2010) the system described in this thesis is also more generic in a way that it does not require the specification of suspicious accounts but analyzes all transactions in the input data. In addition to that, our system is also able to detect more generic layering schemas and is not restricted to chains and sum-chains.

3.1.2 Bitcoin Transactions

Another domain that can be analyzed by using our implementation is the network of Bitcoin transactions. Bitcoin Nakamoto 2008 is a peer-to-peer cryptocurrency that is able to preform and authorize financial transactions without the involvement of a central trusted institution. The system works under the assumption that the majority of the participating nodes in the network play behave according to the protocol and any potentially malicious node would therefore have no incentive not to comply with the protocol. Because the Bitcoin network is designed to work without any central coordination, it is necessary to make all transactions public, i.e. visible to every participating node in the network, by keeping them in the so-called block chain. In order to maintain anonymity of the participants despite the public transaction logs, users are encouraged to use newly generated public addresses for every transaction that they are involved in. As reported in a study on the anonymity in the Bitcoin system (Reid and Harrigan 2013) this advice is not followed by all users of the network. With very little effort common owners of Bitcoin public addresses can be inferred by observing when addresses appear as joint inputs in a transaction. By the design of the Bitcoin protocol, addresses that are jointly used as inputs of a transaction must be controlled by the same owner. Using this property the transaction network as contained the Bitcoin block chain, can be transformed into a user network as shown in figure 3.1.

The user graph is characterized as a clustering of public addresses that are controlled by the same user. However the clustering is not guaranteed to be complete and there

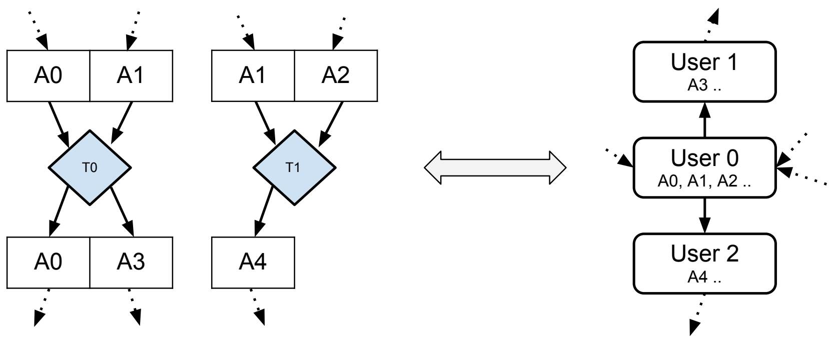


Figure 3.1: Transaction- vs. user network

might be several clusters of public addresses, which are controlled by the same user but can not be linked solely based on the graph structure. Consequently the user network as seen on the right in figure 3.1 is an approximation of the true money flow between the actual participants of the network. In order to further improve the clustering of public addresses, (Meiklejohn et al. 2013) used the transaction graph structure together with external data sources. However even this approach is still far from a complete resolving of the ownership of all addresses.

The topic of anonymity in the Bitcoin network is especially interesting and has received a lot if attention recent years, because some exponents in politics, fear that the Bitcoin currency might help users to circumvent regulatory policies and that tax evasion, money laundering or illegal transactions would be impossible to trace. While the level of anonymity for transactions within the Bitcoin network is relatively high, the owner of an address has to unveil his or her true identity when exchanging Bitcoins for a traditional currency such as USD. For this reason Bitcoin exchanges such as Mt. Gox [1] are of special interest because they are currently one of the only points that would be suitable for enforcing regulations. However, if the acceptance of Bitcoins would rise in a way that valuable goods could be bought using this currency, it would open new ways for laundering Bitcoins outside of the closed Bitcoin system. In (Stokes 2012) the authors analyzed the potential of virtual currencies such as the Linden Dollar (Second life) and

[1]https://www.mtgox.com/ (visited on 10/13/2013)

Bitcoin to be used for money laundering purposes. They point out the difficulty of monitoring transactions within the virtual financial network, because no financial institution in involved as an intermediary. Another challenge that regulatory institutions have to face, is, that in the peer to peer case of the Bitcoin network monitoring to prevent or detect money laundering presents itself more difficult than in the more centralized Linden Dollar case. However, the money laundering potential of virtual currencies is lower than it appears at first sight. Taking into account the currently still small capitalization of virtual currencies and still much lower transaction volumes would require much higher efforts for laundering even moderately large sums of money.

Apart from the legal aspects involved with the Bitcoin network, it represents also a very interesting data set to perform forensic investigations and try to find occurrences of smurfing schemas as described in (Luell 2010). While the data set used in the detection of possible internal fraud cases was highly confidential and only covered transactions within accounts of one financial institution, the Bitcoin block chain is publicly accessible and contains every valid transaction that was ever performed. Because of their similar nature and function as financial transaction network, the Bitcoin transaction graph shows similar structures of transactions as reported in the internal fraud cases. For instance, a number of famous Bitcoin theft cases were shown to be built on a very similar set of basic building blocks for obscuring the traces that lead from the victim to the theft (Meiklejohn et al. 2013). As in the case of company internal fraud, chain transactions also played an important role in Bitcoin thefts. However the chain transactions in the Bitcoin case formed what the authors called a "peeling chain" which is a chain of subsequent transactions where at each transaction a small fraction of the amount is transferred to a separate account outside of the transaction chain. This pattern is displayed in figure 3.2a). In addition to peeling chains the authors also reported the splitting, aggregation and folding of transactions as the fundamental building blocks that are involved in almost all theft cases. Figures 3.2b), 3.2c) and 3.2d) show an illustration of their structures. Since these three building blocks are sufficient to build other, more complex patterns, such as the smurfing patterns reported for company internal fraud, they are used at the core of our system in order to detect interdependence between transactions and trough that, allow for the detection of a wider range of transaction patterns.

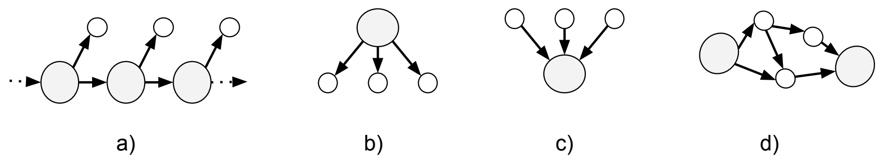


Figure 3.2: Building blocks of financial fraud patterns.

3.1.3 Other Applications Domains

Even though the system described in this thesis focuses on matching transactions in the domain of fraudulent financial transactions, the underlying concepts of our approach could also be applied to other domains. For instance, another form of transactional flows, which is relevant within the field of transaction based forensic investigations, is the flow of information between participants of a network. With the increasing amount of information that is shared in social networks such as Facebook, Google+ or Twitter this form of information analysis has become very relevant. While many applications mainly focus on the detection of influential nodes within a network, e.g. to promote new products or learn about emerging trends, the analysis of the flow of information can also be crucial in forensic investigations. As seen in recent history of so-called whistle blower cases, where individuals shared sensitive information about governmental operations (Nayar 2010) or on clients of financial institutions, communication networks are used to spread information of dubious origin. This has raised the need for a systematic pattern analysis of information flow networks. Just as the analysis of money transfers described in the previous section, analyzing information flows requires a temporal analysis of the graph structure. As described in (John Tang Mirco Musolesi and Latora 2010) considering transactions with respect to the time at which the interaction took place yields more accurate results because it the time ordering is preserved. For analyzing the spread of information, the time ordering of events is crucial to the detection of informational flow patterns because it is immediately obvious that information has to be received first before it can be spread to other network participants.

For the scope of this thesis we will focus on the detection of patterns of financial transaction. The general architecture of the system presented is however independent of the origin of the data and could match any transactional data set where an interdependence between transactions can be defined as described in the next section.

3.2 (Financial-) Transaction Matching

Before explaining the concrete implementation of the transaction matching system in the next chapter, we would like to introduce our approach in a more abstract and implementation-independent form. As already mentioned, the goal of our approach is to scalably detect patterns of interesting transactions in large data sets. For this reason we propose an approach of matching transactions based on local information and then aggregate the associations that were found to connected components. As transactions can be thought of as a set of interactions between entities that have a temporal annotation as well as a payload, they could represent many different forms of interactions and are not limited to financial transactions. This section therefore describes our transaction matching approach in a generic and use-case-independent way.

3.2.1 General Idea

The general idea behind our approach comes from the observation of the detection systems and fraud case scenarios reported in the previous section. While the identification of suspicious individual entities in a transaction network can often be detected with relatively straightforward flag conditions or classification mechanisms, the detection of interaction patterns that are composed of multiple entities is more challenging. We also described that most financial fraud cases used some form of layering or other mechanism to obfuscate money streams. Money laundering schemas typically span multiple accounts and generate seemingly complex transaction structures between them. However, at the level of the individual accounts the patterns are composed of a small number of simple building blocks. Figure 3.3 shows three basic building blocks from an accounts perspective. Each account is represented as an entity with incoming and outgoing transactions that are represented as circles at the left and right side of each entity. The dotted lines that connect the incoming and outgoing transactions within the entities show money streams that are either forwarded (as in entity 1), split (entity 2) or aggregated (entity 3) and through that associate sets of transactions. Our approach exploits these simple building blocks by proactively detecting their existence at the local level of an account. Of course an existence of such a building block does not yet imply, that an account has to be considered to be suspicious. For this reason our approach processes the building blocks and tries to connect them if they share transactions. In our example case in figure 3.3 the three building blocks would be connected because they share the two transactions that are represented by black arrows. The resulting connected component would then have to be assessed with regard to its relevance for the specific use case.

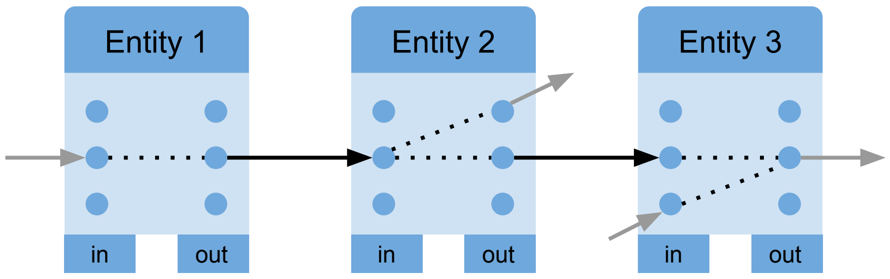


Figure 3.3: General transaction matching approach.

3.2.2 Matching Conditions

As described above, the matching of transactions between entities is not specific to one particular use case, and only requires that a dependency condition between transactions can be defined. In order to understand the matching of transactions and use it in the system implementation in the next chapter, we first have to define the properties of dependent transactions. As an initial condition, dependent transactions have to be connected to the same entity e.g. the same account acts as the receiver of sender of a transaction that participates in an association. We denote the set of received transactions of an entity $x \in E$ as T_x^{in} and the set of sent transactions as T_x^{out}. Since we are only interested in transactions that connect different entities of the graph we require that the same entity never acts as the sender and receiver of a single transaction:

$$T_x^{in} \cap T_x^{out} = \varnothing, \forall x \tag{3.1}$$

The goal of the matching process is to find pairs of sets of transactions that match each other. These pairs consist of a set of input transactions M_x^{in} and a set output transactions M_x^{out} at the same entity. In order to be regarded as a successful match, the two sets have to meet all of the following restrictions:

Both match sets have to be subsets of the respective sets of transactions.

$$M_x^{in} \subset T_x^{in} \wedge M_x^{out} \subset T_x^{out} \tag{3.2}$$

Both match sets must not be empty.

$$M_x^{in} \neq \varnothing \wedge M_y^{out} \neq \varnothing \tag{3.3}$$

All members of the input match set must happen before the earliest member of the output match set.

$$\max_t M_x^{in} < \min_t M_x^{out} \tag{3.4}$$

Because the transactions contain a domain specific payload, a successful match requires, that the aggregated payload of the input transactions must match the aggregated payload of the output transactions. The aggregation functions $aggr_{in}$ and $aggr_{out}$ hereby depend on the specific use case and the nature of the transactions.

$$aggr_{in}(M_x^{in}) \simeq aggr_{out}(M_x^{out}) \tag{3.5}$$

In the specific case of matching financial transactions, the association could be based on the sum of transaction values while allowing for a small difference in values of ε.

$$\sum M_x^{in} = \sum M_x^{out} \pm \varepsilon \tag{3.6}$$

This specification of how transactions can be matched based on their pair-wise interdependence can be considered as a generalization of the sum-chains or chains described in (Luell 2010) and presents the fundamental mechanism of the transaction matching system. The next chapter will show, how transactions that comply with these dependency conditions can be detected in large sets of financial transactions and outlines an end-to-end implementation of the entire matching process.

4

System Design and Implementation

This chapter describes the architecture and implementation of a system that is able to detect relevant connected components within a stream of transactions. In addition to that, the system offers a front-end and a domain specific language to facilitate usage in investigative scenarios. The description of the system is separated into three parts and starts with the specification of the central transaction matching functionality, where transactions are loaded and interesting connected components are detected. In a second step, a number of tools are presented that are designed to help an investigator analyze the suspicious patterns, which are returned by the system. In a final step we show how the different modules of the matching system can easily be orchestrated using a domain specific execution description language.

4.1 Transaction Matcher

On a conceptual level, the tasks of the transaction matcher module can be separated into a workflow that consists of four distinct steps. In an initial phase, a set of transactions need to be loaded into the system from some database or source input file. In a second step, the system attempts to find connected components, where subsets of the transactions are pairwise dependent on each other. The third step associates the previously connected transactions by following transitive dependencies. In a last step, the connected components are separated from the unmatched transactions and tested against a set of conditions. These conditions can be specified by the user and allow him to express his interest in specific occurrences or patterns. The result of this process is a set of connected components that are of interest to the user. The four steps of the transaction matcher are summarized in the visualization in figure 4.1.

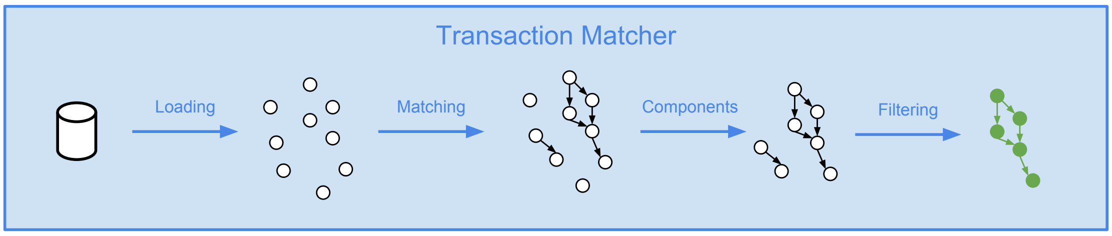


Figure 4.1: High-level view of the transaction matcher module.

4.1.1 Hypothetical Batch-Processing Model

A first naive approach to implement this workflow would be, to sort the input data by the timestamps of the transactions and then partition it. These partitions can the be run through the four steps described above. This version of batch-processing imposes a number of restrictions on the resulting connected components that can be detected. One of the most difficult aspects of this form of processing is the appropriate selection of the right partition size. The algorithm in such a batch-processing schema can only detect patterns that are contained within a partition. While larger partitions are more likely to contain entire patterns, they also require significantly more memory. Another problem is, that a non-overlapping partitioning of the input data is not able to capture connected components that span two or more partitions. For this reason we need to ensure that all the components are at least entirely contained once within a partition. This could be achieved by applying overlapping partitions as shown in figure 4.2. Depending on the overlap properties, one can optimize for either a minimal size of the partition or a small number of repeated executions per element in the data set. It can easily be proved, that in our concrete case this means, that in order to capture a connected component of duration d we need partitions of size $2d$ that are shifted by d to reduce the number of repeated executions for each transaction. Another disadvantage of this system is, that it can only consider information that is available within the processed partition of the graph and can therefore not accumulate knowledge over time. On the other hand the batch-processing is embarrassingly parallel because the processing of one portion is not dependent on any other execution. As a consequence all portions of the graph could be computed in parallel on a distributed infrastructure.

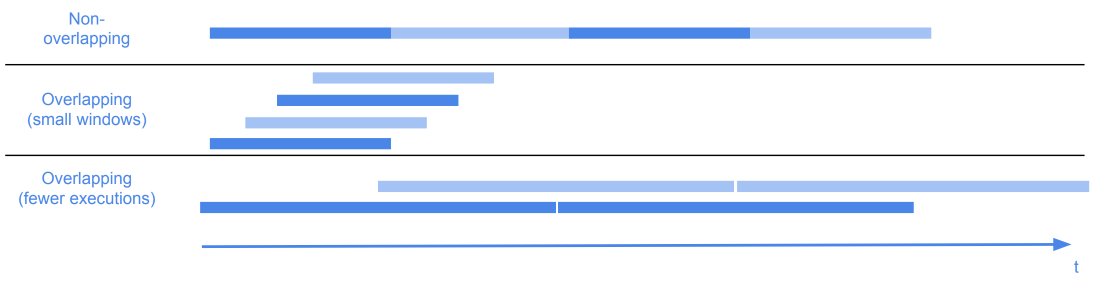


Figure 4.2: Overlap strategies for batch-processing.

4.1.2 Continuous-Processing Model

Considering the fact, that in a batch-processing system a lot of the patterns would be detected multiple times and therefore a significant portion of the effort would be redundant, we decided to also explore the possibility of streamed processing of the input data. The main steps that are needed in order to detect interesting patterns remain the same as shown in figure 4.1. However, the streaming mode holds transactions that can each be in any step of the processing workflow. This is in contrast to the batch-processing mode, where the entire data set is pushed through the steps of the workflow and only one workflow step is active at a specific point in time. Figure 4.3 shows an exemplary snapshot of a streamed execution and the different phases that the transactions are in.

As time flows from left to right, new transactions are added to the right and automatically enter the matching phase. The newly added transactions are now to be associated with other transactions. After a specified amount of time transactions are no longer able to receive associations from subsequent transactions. If the transaction has not yet been matched, it is automatically discarded and removed from the process. If the transac-

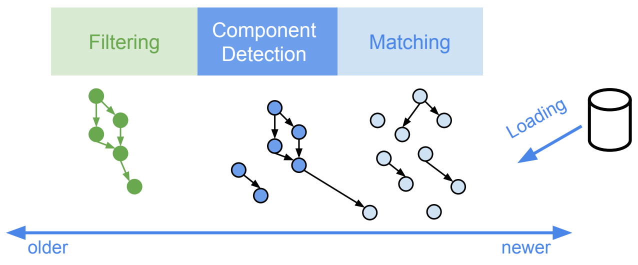


Figure 4.3: Exemplary snapshot of different processing phases in a streamed execution.

tion was successfully matched it enters the component detection phase. In this phase the transaction gathers information about the members of its connected component and waits for all component members to also enter the component detection phase. At the point where the entire connected component is in the component detection phase, the component is reported and a number of filters are applied to assess whether or not the component is of interest to the user. After successfully passing the filtering step the components are reported and removed.

The following sections will explain each of the processing steps in detail. However before going through the processing model, we explain how transactions are represented as graph structures and how the basic graph elements are composed to allow different behavior depending on the processing phase that they are in.

4.1.3 Graph Representation

Transactions can be modeled in various forms as graphs structures. A very intuitive representation is a graph that consists of set of entity vertices V and a set of transactions as edges E between them. A systematic visual representation of this is shown in figure 4.4a). Despite its intuitiveness, this representation comes at the cost of several limitations when the temporal dimension of transactions is considered. The edges would only be valid at a single point in time and a static graph $G = (V, E)$ would imply reachabilities that are not valid, when the temporal order of the transactions is considered. Additionally the stated goal of finding suspicious flows of transactions would be more difficult if a flow visited the same vertex more than once i.e. the static graph G contains cycles of repeatedly visited accounts. Of course these cycles could be unfolded by considering the time stamps on the transactions, but this would again introduce additional complexity to the system. For these reasons it is beneficial to represent both, the entities and the transactions, as vertices as shown in figure 4.4b). The upper vertices in b), which represent the transactions, are hereby semantically equivalent to the edges in a). The transaction vertices have exactly one incoming and one outgoing edge, whereas the vertices that are located below them and represent the entities that issue transactions can have multiple incoming and outgoing edges.

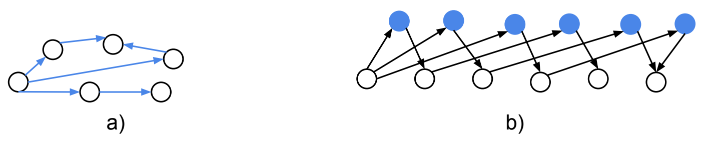


Figure 4.4: Alternative graph representations for transaction networks.

In the matching system, the graph structure shown in figure 4.4b) is constructed at loading time. In addition to the association of a transaction with its source and target entities it can build new connections to other transactions if they are associated with each other. For instance if a transaction is dependent on another transaction that happened earlier in time they will be bi-connected to represent this pair-wise relationship. Obviously this connectivity between transactions will cause the graph structure of transactions and entities to lose its bipartite graph property that existed after loading.

4.1.4 Pluggable Vertex Algorithms

As described above, the process of identifying related transactions and aggregating them to transaction patterns involves a series of steps that a transaction would have to step through until it can be reported as part of a component. As the processing system is built on top of Signal/Collect (Stutz, Bernstein, and Cohen 2010) we had to come up with an extension to Signal/Collect that allowed different vertex behaviors based on the step in a workflow that a vertex is currently in. This change in behavior however should happen based on local decision factors and be transparent for other components of the graph. In addition to that the neighborhood of a vertex should not be altered when a vertex changes its behavior. For this reason a special vertex implementation called *RepeatedAnalysisVertex* was created that allows for pluggable algorithm implementations. Pluggable algorithms implement the *VertexAlgorithm* interface and can be swapped during run time.

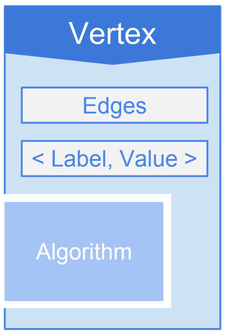


Figure 4.5: RepeatedAnalysisVertex with pluggable vertex algorithm.

The *RepeatedAnalysisVertex* as shown in 4.5 is essentially an adaption of the state pattern (Gamma et al. 1995) of object oriented design. The enclosing vertex implementation delegates all algorithm dependent calls to the enclosed *VertexAlgorithm* but is responsible for the vertex's connectivity within the graph. In addition to that the *RepeatedAnalysisVertex* is capable of storing arbitrary values in a dictionary data structure. This dictionary used for two purposes. On the one hand, it allows to easily add new named attributes as field values without sub-classing and adding the fields at compile time. These field values can later be used in the specific algorithm. Additionally the dictionary can also retain intermediary results, that were obtained by executing algorithms on the vertices. This allows the user to reuse results in subsequent algorithm executions by retrieving the stored intermediary result. If we look at a transaction vertex in our financial transactions use case, the dictionary is initially loaded with entries for the value of the transaction, the time stamp and similar values. During the execution of the work-flow additional entries, such as information about the component that it belongs to, can be added to the dictionary and reused by algorithm implementations.

Field	Type	Description
Id	Integer	A globally unique identifier
Time	Long	Unix timestamp (sec. since 1/1/1970)
Src	Integer	The Id of the source entity
Target	Integer	The Id of the target entity

Table 4.1: Mandatory fields for transaction vertices.

4.1.5 Loading

As described in the overview section on the transaction matching implementation, the loading step represents the initial phase of the entire detection process. The goal of this phase is to generate the required graph elements for each entry in the input data an add them to the graph for further processing. Because the behavior of the graph elements changes over the process of the execution, all vertices of the graph are directly loaded in the pluggable form as described in the preceding section. As each entry in the input data is assumed to represent a transaction between two entities, the loading of such an entry results in three vertex instances. Two of the instantiated vertices represent the source and target vertex of the transaction and the third vertex represents the transaction itself. The source and target entities are loaded with no additional state information apart from their globally unique Ids and are directly added to the graph. If the graph already contains an entity with the same Id no additional entity will be added. The entities initially have no reference to the transactions that it issues but will accumulate this knowledge during the matching process. The transactions on the other hand are loaded with more state information and include all the relevant information that is available as part of the input data. The number of fields that can be included in the transaction vertex is not limited, however for the transaction matching phase a number of fields are required. Those fields are listed in table 4.1. In addition to the mandatory fields some information about the payload of the transaction might also have to be included in order to be able to meaningfully associate the transactions. After storing all the fields the transaction is added to the graph and proceeds to the matching phase.

4.1.6 Matching

After the new transactions are loaded and added to the graph, they have to be matched with other transactions in order to find sets of transactions that are dependent on each other. As described the high level overview, the matching phase provides the initial structure which underlies the suspicious patterns and is therefore essential to the recall of the entire process. The goal of the matching process is hereby to find transaction patterns that are dependent on each other with regard to the conditions described in the general description of the matching process in section 3.2.2. The matcher therefore has to consider all the conditions in (3.1)-(3.5) in order to find successful matches among the transactions. For the purpose of maintaining a use case independent infrastructure, the matching process can be divided in a generic and an application-domain specific part. As seen in the more formal description above, the matching of transactions is based on the local view of the entities that participate in these transactions. Regardless of the specific application domain, an entity is responsible for maintaining its transactions and detecting possible associations among them. Because the entities are loaded as separate instances and exist independently of the transactions the transactions have to announce themselves at both their sending and receiving entity by sending them a subset of their transaction attributes. In turn the entities hold lists of the incoming and outgoing transactions and try to find matches among them. The process of matching these lists of transactions can be a computationally expensive task and accounts for a significant amount of the effort of the entire processing workflow. In order to reduce this effort the system has to avoid repeated computations of matches over the same set of transactions. Therefore the matching process is only performed once new options, i.e. transactions, are available for matching. Moreover, since transaction are assumed to be loaded in the order of their time stamps, an entity only starts a new matching attempt at the arrival of new outgoing transactions. Incoming transactions will be added to the list of input candidates, because condition 3.4 does not allow that it would be matched with transactions, which happened before this transaction. At the arrival of outgoing transactions the entity will try to match the new outgoing transaction with all the incoming and outgoing transactions that are available at that point in time and eligible given the conditions stated above. If the matching could not detect any associations among the transactions the new output is added to the list of the potential matches for further executions. In case the attempt resulted in a successful match, all the transactions that took part in this match are connected with a pair of edges. These denote their pairwise dependence in both directions.

Figure 4.6 illustrates the steps of the matching process for financial transactions, where the matching is based on the sums of the transaction amounts as described in condition 3.6. For the simplicity of this illustration we assume that the matcher only considers exact matches by setting $\varepsilon = 0$. The first step shows an entity with two incoming transactions and one outgoing transaction. Even though both incoming transactions have a smaller time stamp than the outgoing transaction, they cannot be matched, because no combination of the input values matches the output. After a new outgoing transaction to the entity, it is now able to identify a possible match. By summing up both the values of both input transactions, the aggregation matches the most recent output. Subsequently the successfully matched transactions learn about their related transactions and build up bidirectional connections between the inputs and the outputs. The user of the system can also decide if transactions should only participate once in a match or be available for multiple matching associations. The fist approach yields unambiguous results but could potentially miss out on matching interesting patterns. If transactions can participate in multiple matching associations all matching possibilities are considered at the cost of potentially capturing larger components than intended. Depending on the use case and the user's preference function he can therefore choose either option, by setting the appropriate flag in the execution description.

The matching of transactions is temporally constrained to a sliding window, which means that transactions that could not successfully be matched within a predefined duration are automatically removed from the graph. Consequently, the transaction will also be removed from the lists of matching candidates at its respective source and target entities. The sliding window is achieved by periodically sending a timeout message to

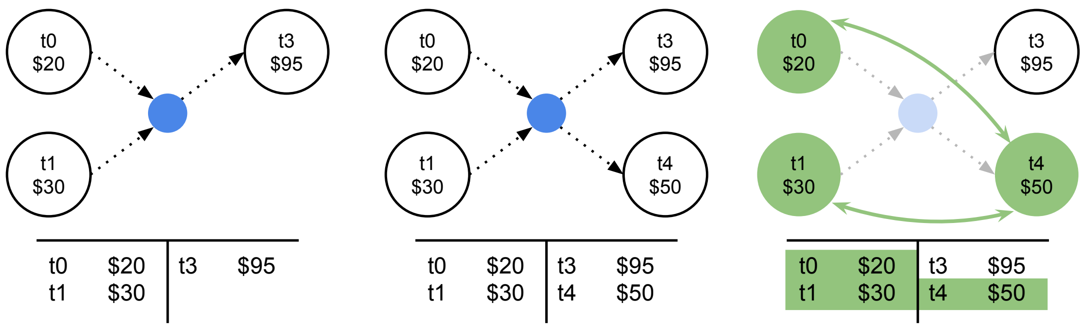


Figure 4.6: Matching of transactions at an entity.

all active vertices in the graph. Upon receipt of a timeout message the transactions compares the time in the message with its own time stamp. If its time stamp is older than the timeout it received, the transaction will no longer be eligible for matching. These transaction then either proceed to the component detection phase or are removed from the graph in case they have not been matched. Entities on the other hand use the same timeout messages to remove intermediary matching structures that they built up based on the transaction information and have now timed out. When an entity has removed all its transactions, it removes itself from the graph as well. The broadcasting of the timeout messages is a compromise that ensures consistency with regard to the order of removal of transactions without having a central instance that keeps track of the transactions and their timestamps.

4.1.7 Component Detection

Transactions are transferred to the component detection phase, once they were successfully matched with other transactions by the means of the previously described conditions and are no longer available for matching. The goal of the component detection phase is to build larger constructs from the previously detected associations between transactions. The constructs are built by concatenating the associations from the previous phase in a way that an output transaction participates as an input transaction in another association. Because the graph elements allow for pluggable vertex behavior, the component detection is initiated by switching out the algorithm on the matched transactions and replacing it with the component detection algorithm. The component detection is based on a simple algorithm to find consensus about which transaction the component master should be. The id of the component master will then serve as the component identifier. The component master is determined by broadcasting the transaction signature of that transaction that is believed to be the master of the component. This signature contains the id of that transaction and its time stamp. Every transaction starts by propagating its own signature and will switch to the signature that it has received from one of its neighbors, if the signature represents a more recent transaction. In case the timestamps are equal, the signature will be switched, if the ID contained in the received signature is greater than the one currently held. If a transaction decided to switch to a new signature it broadcasts that new signature along its associations. This aggregation of signatures deterministically converges towards the label of the most recent transaction in the graph. Since the component detection causes all transactions contained within a component to learn about the e most recent transaction, that transaction will later act as the master of this component.

The component detection phase ends when the most recent transaction of the component cannot be associated with new transactions anymore. This means that for all transactions in the component, the difference in time to a new transaction would be longer than the matching specification allowed for. For data sets that are composed of highly connected transactions the user also has the choice of specifying an optional maximum duration of a connected component to ensure that components are reported and do not get stuck in the component detection phase.

4.1.8 Filtering

The component detection phase described in the previous section results in a number of connected components which each consist of a set of component members and one component master. However the transactions within a component only know about their local neighborhood and the existence of the component master. In the next step the system needs to decide whether a component is relevant to the user. For this purpose the information about the component needs to be centralized and reported. Because the number of components can be large, this process is based on a hierarchical organization, where components are represented by their component masters. In a first step the component master therefore has to learn about all the members of the component that he represents. In order to do that, every component member sends information about itself to the component master. The master will in turn aggregate all these messages and build a list of all its members. As soon as the master learned about all its members he reports the component to an external component handler.

The component handler itself is responsible for identifying the interesting components within the set of connected components and report them to the result handlers (see section 4.2). For this purpose the component handler can be configured with a filtering workflow. This workflow is essentially represented as an ordered list of conditions, which the components have to fulfill in order to be considered relevant to the user. The component handler is responsible for distributing this workflow to the components and receives all components that successfully passed the entire workflow. In the implementation this means, that whenever a new component is reported at the component handler, an instance of the workflow is sent to the component master for evaluation. Figure 4.7 shows a component workflow consisting of 3 conditions and an acceptance state that is executed on two different components. The components are associated with a step in the workflow which means that the components at least fulfill the conditions of the previous steps. The green component has reached the acceptance step and can be reported as a successful match.

The conditions in the different workflow steps are required to evaluate to a Boolean value that determines whether the component passes this step of the workflow or not. This list of conditions allow the user to express any condition that is required in order to assess the relevance of a component in the context of a specific use case. Each condition is composed of information about the component and a comparison function to evaluate

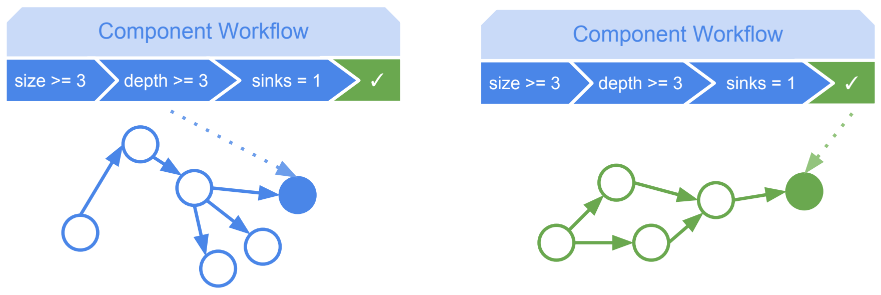


Figure 4.7: Component workflow execution on different components.

the returned result. The information about the component can either directly be answered by the component master, e.g. the size of the component, or require the execution of an algorithm on the component members. If an algorithm execution is required, the results of the algorithm execution need to be aggregated by the component master to obtain the information that needs to be checked against the comparison function. This execution of arbitrary algorithms and especially graph algorithms on the component structure allows for complex attribute queries that the component master could answer purely based on the list of component members and contributes to the expressiveness of the overall system. The different queries and algorithms that can be used to obtain information about the components are extensible and can be directly registered at the component handler. They are then loaded via the workflow steps on the component's vertices. An exemplary request is shown in listing 4.1 were the request consists of a function on the component master. The resulting size would then be evaluated against the evaluation condition and the master would proceed to the next step in the workflow

```
// Queries the master for the size of its component
val SizeQuery = ComponentMasterQuery(master => master.members.size)
```

Listing 4.1: Simple component query example

Queries that require the execution of an algorithm on the component structure are slightly more complicated because they consist of two parts. The first argument is the algorithm that should be loaded on all members of the component and the second argument describes the aggregation function that the component master needs to apply to compute the return value. Listing 4.2 shows such an algorithm execution, which runs a depth exploration on the component graph which will label each vertex of the component with its maximal depth based on the component structure. The aggregation function is relatively simple and computes the maximum value of all results returned by the components. By implementing the same set of interfaces one can define various other graph algorithms to extract relevant information from the component.

```
// Queries the component for the max depth i.e. the longest path from
    any source to a sink transaction
val depthAlgorithm = ComponentAlgorithmExecution(depthMemberAlgorithm,
    maxInt)

// Returns the depth of a transaction in the graph.
val depthMemberAlgorithm = ComponentMemberAlgorithm(vertex => new
    PatternDepthAnalyzer(vertex))

// Get the max integer of all component member replies
val maxInt: (Iterable[ComponentMemberMessage], ComponentMaster) => Any
    = {
  (repliesFromMembers, master) =>
    {
      val replies =
          repliesFromMembers.asInstanceOf[ArrayBuffer[ComponentMemberResponse]]
      replies.map(_.response.get.asInstanceOf[Int]).max
    }
}
```

Listing 4.2: Component member algorithm example

After passing the last workflow condition, the component master composes a serialized representation of its component and passes this to the component handler. The component handler then forwards these serialized components to the result handlers that are registered with it. The result handler's functionality as well as a selection of different result handler implementations will be described in the following section. An example of a complete filtering process consisting of two conditions, of which one is a simple query on the component master and the other one requires the execution of the previously described *depthAlgorithm*, is visualized in figure 4.8. As the figure shows, the component handler only takes part in the initialization and the final phase of the workflow whereas the workflow is processed independently on the component master and its members. This not only reduces the load on the component handler but also avoids unnecessary communications between the component handler and the master. For this reason the current implementation uses only one component handler but the architecture would allow for replicated component handlers to balance the workload if that would ever become necessary.

4.2 Result Handling

In order to make meaningful use of the patterns that were found by the matching and filtering process that was described in the previous section, the system also provides a number of tools that allow the user to investigate the reported results or persistently store them in some other location. As described in the previous section the component handler receives a serialized version of the entire component from the component master and then broadcasts this information to all registered result handlers. Because the reports need to be provided in an exchangeable format that can be used by a variety of different tools we decided to use the JSON serialization format to represent the serialized components. An example of such a serialized component in JSON format can be found

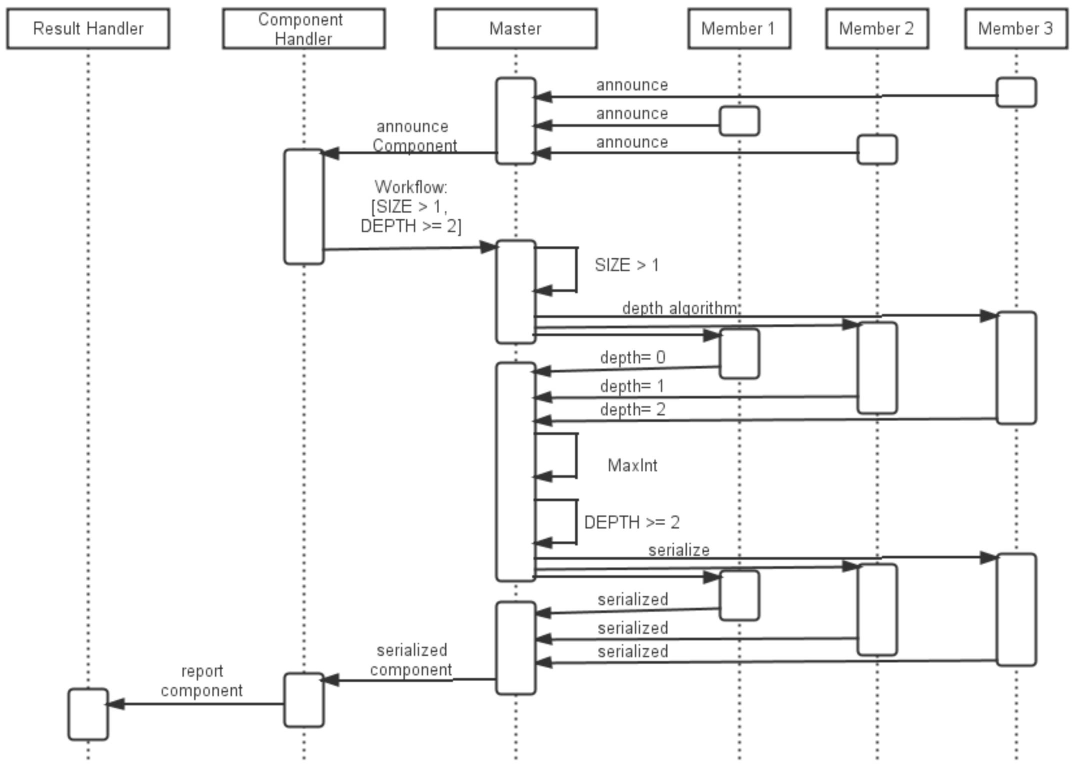


Figure 4.8: Workflow execution on a component with three members.

in Appendix A.1. The result handlers that are described in the rest of this section are all able to accept such serialized components and process them according to their implemented functionality.

4.2.1 Command Line Debugger

The command line debugger is the most basic result handler. It accepts the reported component and forwards its character string representation to the standard output. This result handler is mainly used for debugging or logging purposes. However, despite of its simplicity this mode allows other command line tools to further process the results and hence contributes to the extensibility of the system.

4.2.2 Visualizer

The visualizer is the most elaborate result handler and allows the user to inspect the reported components in a visual graph representation. The visualizer result handler is composed of a server and a client part. The server component's tasks can be separated into two logically separated components. On the one hand, the server is responsible for serving the client libraries and the static visualizer page that builds the foundation of the client's web app. In addition to its file serving functionality, the server also acts as the receiver of the serialized components from the component handler and sends them to the connected clients. Clients are connected to the server via HMTL5 web sockets, which allows for two-way communications between the server and the clients. The two-way communication channel allows the server to send information directly to the client application without the delay and overhead of manual refreshes. In addition to that, the client can connect to the web socket endpoint over a network connection and run on a separate machine. A high-level overview over the different visualizer components is shown in figure 4.9.

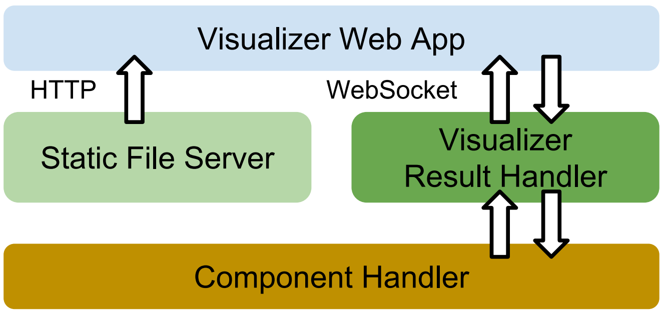
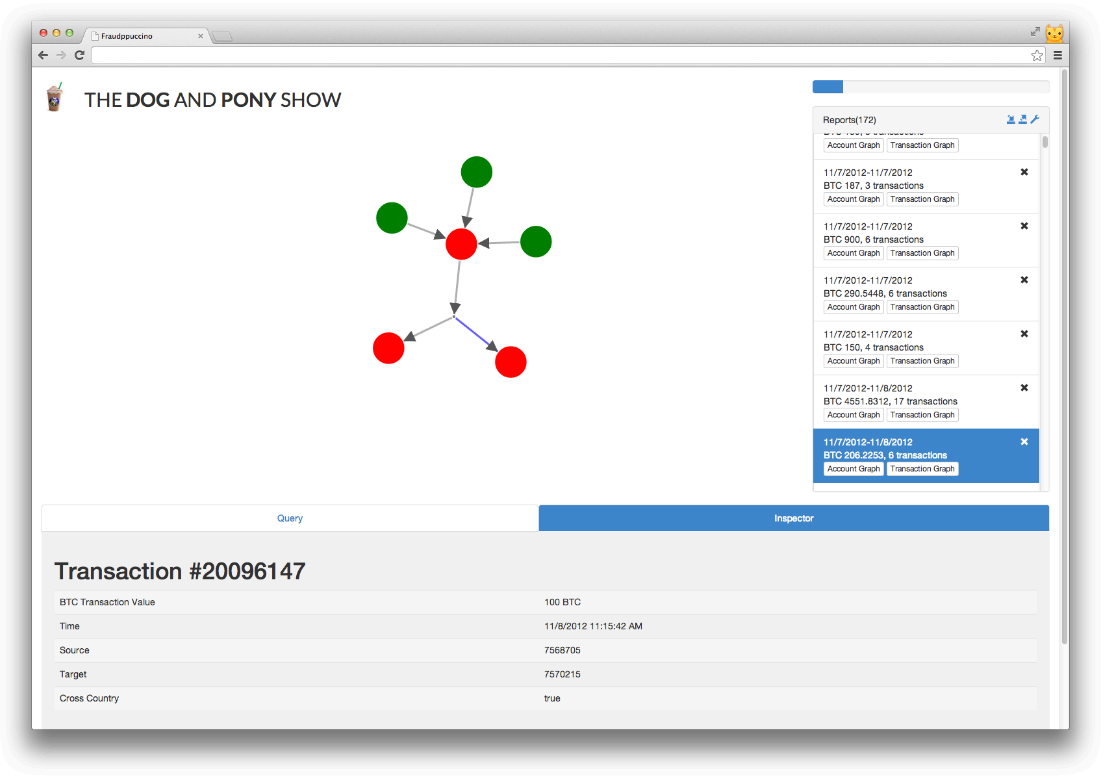


Figure 4.9: Visualizer architecture overview.

The client side of the visualizer is implemented as a browser-based web application as shown in the screen shot in figure 4.10. The client application is composed of three areas that each serve a different purpose. In the upper right part of the application a number of reported components are listed. The components in the list are represented by some key indicators about it. The user can select any component that he is interested in and have it been drawn on the canvas in the middle of the screen. Depending on whether the user decides to display the component as an account or a transactions graph the vertices in the visualizations represent accounts or transactions. New transactions that were sent from the server are automatically appended to the list of reported components. The application also provides basic import and export functionality to interact with the components in the reports list and a progress bar that shows the current state of processing. The upper left side of the client application consists of a HTML5 canvas where the graph visualization is drawn. The current implementation includes a series of different coloring and sizing schemas to semantically highlight the properties of the underlying graph but their exact properties go beyond the scope of this description. The lower section of the application contains an inspector tab, which displays information about graph elements if the user clicks on transactions or accounts.



11/7/2012-11/7/2012
BTC 187, 3 transactions

11/7/2012-11/7/2012
BTC 900, 6 transactions

11/7/2012-11/7/2012
BTC 290.5448, 6 transactions

11/7/2012-11/7/2012
BTC 150, 4 transactions

11/7/2012-11/8/2012
BTC 4551.8312, 17 transactions

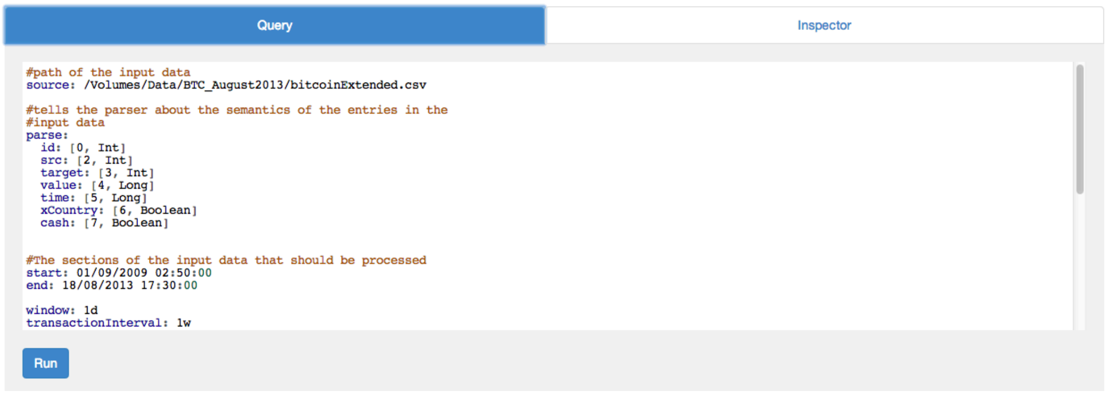
11/7/2012-11/8/2012
BTC 206.2253, 6 transactions

Transaction #20096147

BTC Transaction Value	100 BTC
Time	11/8/2012 11:15:42 AM
Source	7568705
Target	7570215
Cross Country	true

Figure 4.10: Display of a reported component in the visualizer.

Apart from just being a result handler that processes the reported results, the visualization component also provides an editor that allows the user to specify an execution and start it from the web application. The editor can be accessed by clicking on the tab left to the inspector tab and is shown in figure 4.11. A detailed description of the syntax of the execution description is given in the next section.



```
#path of the input data
source: /Volumes/Data/BTC_August2013/bitcoinExtended.csv

#tells the parser about the semantics of the entries in the
#input data
parse:
  id: [0, Int]
  src: [2, Int]
  target: [3, Int]
  value: [4, Long]
  time: [5, Long]
  xCountry: [6, Boolean]
  cash: [7, Boolean]


#The sections of the input data that should be processed
start: 01/09/2009 02:50:00
end: 18/08/2013 17:30:00

window: 1d
transactionInterval: 1w
```

Figure 4.11: Execution description editor in the user interface.

4.2.3 Database Interface

The visualization handler described above is targeted towards online visualization of patterns that were reported. Depending on the use case and the volume of reported transactions this form of processing the results might not be appropriate or overwhelming for the user. Additionally the visualizer only visualizes the reported components but does not automatically store the results that were found. If a reported result needs to be analyzed at a later point in time the user would have to manually export and re-import the reported components. To simplify this process we decided that the system should also support a persistent storage option where the results can be written to a database. As the system supplies the results in JSON format, mapping the results to a relational database schema would require additional information and schema definitions. It is intuitively clear that a schema less database would be a better fit for our requirements than the more strict SQL based database. Consequently we decided to implement a MongoDB backed result handler that allows the storing of the reported components in

JSON format without further processing or mapping effort. MongoDB is a document-oriented database, which means that no explicit schema is required. The MongoDB result handler automatically creates a separate collection for each execution of the detection system. During the execution of the detection system the detected components are continuously added to the current collection as a separate *DBObject*. Storing the reported components in the document oriented database also allows for querying of the results e.g. to test whether a transaction was reported as a member of an interesting component. The MongoDB driver is currently configured to store the components on a MongoDB instance running on the local machine but it can also be parametrized to use externally hosted or sharded MongoDB instances.

4.3 Execution Description Language

The transaction matching system was designed with an emphasis on creating an implementation that is generic enough to be able to match transactions in various use case scenarios. Such a generic system, however, comes at the cost of more parameterization, because the implementation has to be configured in order to perform as required. Apart from the system being extensible and generic, another desirable property is, that the system should be open to a broad range of users. This means that the system should be designed in a way that it does not require advanced programming expertise or a deep understanding of the matching process. This is achieved by extracting the fundamental configuration steps into a separate set of instructions that act as the execution description (ED). Because the ED contains a number of instructions that are specific to orchestrating the components of our system, we decided to use YAML as the underlying format of the ED rather than trying to map it to an extended syntax of an existing query language such as SQL or SPARQL. Other benefits of YAML are its concise syntax while maintaining a high level of readability.

Listing 4.3 shows an example of an ED that contains all the necessary entries to perform a matching execution and report the results that were found. The ED can be logically split into three distinct sections. The first section (in lines 1-7) specifies the loading of transactions from an input file. The *source* value specifies the path of the source file on the server that contains the transactions. The *parse* entry explains, how the entries in the source file should be represented on the transaction vertex. For example, line 7 instructs the parser to parse the fifth entry of the input as a long value integer and store it as the transactions time stamp. The parser hereby allows the specification of arbitrarily named fields of primitive type. As already described in table 4.1 a number of entries are mandatory and need to be listed with their respective value. In our case we added an optional entry *value* to the loading script, which will be used in this scenario for matching financial transactions. As this example illustrates, the parse statement also allows the exclusion of entries in the input file, e.g. here we excluded the entry at index one by not listing that index in the parse list.

The second set of instructions (lines 9-14) describes the matching process itself. The *matcher* statement instructs the system to use the matcher for financial transactions as illustrated in figure 4.6. For other matching strategies one could register other matcher implementations and use their name within the ED. The *start* and *end* entries specify the interval that should be processed by the system. All transactions that have a time stamp, which lies within this interval, are eventually processed by the matcher. The *window* entry tells the system how many transactions should be loaded before the loader stops and awaits the convergence of the computation. After the computation has converged, the loading is resumed. As described in the matcher description, transactions are only accepting associations with other transactions for a certain amount of time. This maximum interval between transactions that can be associated with one another is specified by the *transactionInterval* entry. The last entry of the matcher specification section describes all the conditions that have to be met by connected components in order to be considered as significant to the user. The *filters* entry therefore consists of a list of conditions of the form (Algorithm, Boolean Comparator, Value) for algorithms that are evaluated against a statically compiled value, e.g. an integer, or (Algorithm, Boolean Comparator, Algorithms) for two algorithms that are evaluated against each other.All algorithms that are used within a filter need to be previously registered with the system or already be present as one of the default implementations.

The ED ends with a description of the result *handlers* that are activated to process the reported components. As for the matcher or the filter entries, the handlers also need to be registered with the system if additional result handlers are being used. By default all three result handlers described in the previous section are already registered with the system and can be used with the respective entry in the handlers list.

```yaml
source: /path/to/transactions.csv
parse:
  id: [0,Int]
  src: [2,Int]
  target: [3,Int]
  value: [4,Long]
  time: [5,Long]

matcher: financial
start: 01/09/2009 02:50:00
end: 18/08/2013 17:30:00
window: 1d
transactionInterval: 1w
filters: [SIZE > 5,SINKS = 1,SINKVALUE > 10000000000,DEPTH > 3,
    COUNTRYHOPS > 2]

handlers: [WEBSERVER, CONSOLE]
```

Listing 4.3: Example of an execution description in YAML

5

Evaluation

This chapter evaluates our implementation with regard to its capabilities of expressing and retrieving interesting financial fraud patterns along with key performance measurements of the computation. For the lack of availability of real-world financial data sets, which include ground truth with regard to whether a transaction or a set of connected transactions should be considered fraudulent, this evaluation is performed on two unlabeled data sets. One data set analyzed in this section was synthetically constructed to resemble the typical structure of account interactions within banks. The other data set consists of real-world transactions that were obtained by reconstructing the transactions found in the Bitcoin Blockchain. Because the Bitcoin use case is is based on publicly available data and provides a larger set of transactions, it is used as the main data set for this evaluation. The synthetic bank transactions are used in a second step in order to validate the general applicability of the results obtained from the Bitcoin evaluation.

To construct the data set for our Bitcoin evaluation we used the Bitcoin Blockchain and deanonymized the public addresses, as described in section 3.1.2. By merging common input addresses we were able to obtain a transaction log between users of the Bitcoin system that consists of 50 million transactions. According to a study presented by the Bank of America (Chang et al. 2007), this size is comparable to the amount of wire transfers that are handled by large American banks on a daily basis. Because of the increasing popularity of the Bitcoin currency, the number of transactions per day has steadily grown during the analyzed time period between January 9, 2009 and July 10,

2013. As shown in figure 5.1, the number of transactions per day differs by three orders of magnitude during this time period. This non-uniform distribution of transactions provides the opportunity to analyze the detection system's performance on varying scales of input data rates. Because the data set does not provide pre-labeled fraud patterns and not enough fraud cases are reported to manually label the transactions, the goal of this evaluation section is not to detect historic fraud cases in the Bitcoin transaction graph. Instead we show that the system is able to detect patterns, which correspond to patterns reported as part of real world financial fraud cases, within this data set. The detection procedure as well as the patterns used in this evaluation are therefore not specific to a Bitcoin use case but rather designed to enable the detection of more general financial transaction patterns. The capability of the system to capture structures of real-world fraud patterns is evaluated in the first section of this chapter. The subsequent sections will then analyze the performance of the system with respect to the matching complexity as well as the matching duration used in the execution.

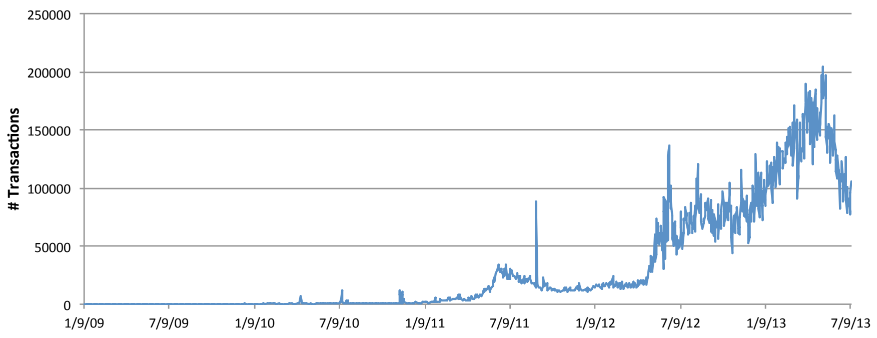


Figure 5.1: Bitcoin transactions per day.

5.1 Expressiveness of the System

To support the claim that the described process of matching transactions is an appropriate method for detecting meaningful transaction patterns, this section focuses on the expressiveness of the system. The desired expressiveness can be formulated as two requirements that the system has to fulfill. In a first step, the system has to be able to let the user formulate his interest in the structure of transaction patterns. The second requirement concerns the recall of the system and requires that the system returns all transaction patterns that fulfill the conditions that the user expressed.

5.1.1 Expressiveness of the Query Language

To evaluate the expressiveness of the query language, we first have to define a set of classes of fraud cases that we should be able to describe with the help of the query language. As previously described, our data set of Bitcoin transaction provides no labeled fraud patterns. Apart from a number of reported thefts there is also no external information available that could be used as ground truth for such an analysis. Additionally, we require the system to be able to detect transaction patterns independently from the Bitcoin scenario. For this reason, our definition of interesting fraud cases needs to be of a more general nature as well. To gather information about different classes of financial fraud, we analyzed a report (Egmont Group 2000) of a hundred reported money laundering and financial fraud cases. This report was published by the Egmont group, which is a global network of financial intelligence units (FIU). In this report the different FIUs published anonymized case studies of various illegal financial activities that they had detected. Since this report contains a variety of different fraud cases ranging from simple theft cases to very complex money laundering schemas, we first classified the reported cases with respect to their detection criterion in order to assess their suitability for applying the proposed detection system.

Figure 5.2 shows how the case studies can be split in four classes. The visualization shows each class together with the portion of the cases that fall in each of them. Approximately a third of the reported cases were either only detectable using additional information, i.e. information outside the transaction logs, or were not described in enough detail. A use case is described in not enough detail if it is not possible to decide what detection criterion could have let to the detection of this case. In addition to that, a significant portion of the reported cases was detected using simple flag conditions.

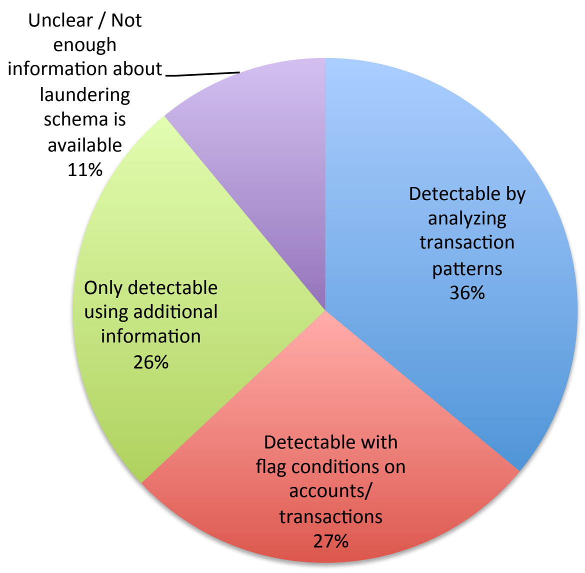


Figure 5.2: Classification 100 Egmont fraud cases by detection criterion.

In these cases simple heuristics were used to raise suspicion upon unusual behavior on individual accounts or transactions. The last group of cases consists of reports, where fraud was detectable by looking at more sophisticated structures that involved multiple accounts and transactions. Naturally, this last group will be in the focus of this expressiveness evaluation. This class is particularly interesting, because it allows us to assess the capabilities of the system and the query language with respect to their ability to capture patterns with a complex transaction structure. Based on the characteristic properties of the transaction patterns of the use cases in that group, the cases were further classified into six distinct scenarios. A list of the scenarios, together with the number of reports that fall into each one of them, can be found in table 5.1. A detailed classification for all Egmont use cases can be found in appendix A.3.

As described in the implementation section in chapter 4, the specification of the connected components that should be returned to the user can be specified as a filter condition in the execution description. This filter allows the user to express his interest in the returned components based on a concatenation of desired component properties.

Use Case	# Reports
UC 1: Cash in, X-Country	11
UC 2: Single Transaction Channeling	7
UC 3: Split Equal	3
UC 4: Account Cycle	5
UC 5: Cash Washer	3
UC 6: Multi Country Hops	1
Not Supported	6

Table 5.1: Classification of Egmont transaction patterns in use cases.

All properties that are used to specify the components that correspond to the identified Egmont use cases are explained in table 5.2. These properties can either be directly available to the master of a connected component or require an execution of an algorithm on that component.

Table 5.2 lists all the properties that can be used to assemble the six filter conditions that are able to capture the use cases derived from the Egmont report. The six filter conditions are listed in listing 5.1 together with a short comment, which describes the nature of the components that this filter is intended to find. For all use cases we required the total flow of the pattern to be above 100 Bitcoin. Because of the lack of ground truth the execution of these filter conditions can not be evaluated by the usual means of precision, recall and accuracy. However they demonstrate how the system and the complementary execution description language are able to express different patterns of relevant structure. Since the set of properties, which can be used in a filter condition, is extensible and can be tailored to suit a use case or data set, filter conditions provide a powerful and flexible way to assess the relevance of a component.

```
# Use Case 1: Cash input is split across multiple countries
filters: [SOURCETRANSACTIONS=CASHSOURCES,
    COUNTRYHOPS>=2,SINKVALUE>10000000000]
# Use Case 2: Large portion of the component's input value is
    channelled through a single transaction
filters: [SOURCETRANSACTIONS>1, MAXTRANSACTIONVALUE ~=10% SOURCEVALUE,
```

Attribute	Description
CASHSOURCES	Number of source transactions that were executed in cash.
CYCLEMEMBERS	Number of accounts participating in a circular transaction path.
COUNTRYHOPS	Number of transactions within a path spanning multiple countries.
DEPTH	Longest transaction path between a source and a sink transaction
FAIRSPLITS	Number of transactions that were split in equal parts.
MAXTRANSACTIONVALUE	Largest value of all transactions in the component.
SAMEDAYSPLITS	Number of transactions that were split within 24h
SINKVALUE	Aggregated value of sink transactions.
SIZE	Number of transactions within a component.
SOURCETRANSACTIONS	Number of transactions that are sources for the components.
SOURCEVALUE	Aggregated value of source transactions.

Table 5.2: Explanation of filter attributes used in the Egmont use case.

```
    SINKVALUE>10000000000]
# Use Case 3: Transaction is split into 2 or more splits of
    approximately equal size.
filters: [FAIRSPLITS>0, SINKVALUE>10000000000]
# Use Case 4: The list of accounts in a transaction flow path contains
    cycles.
filters: [CYCLEMEMBERS>0, SINKVALUE>10000000000]
# Use Case 5: Cash input is split within 24h.
filters: [SOURCETRANSACTIONS=1, CASHSOURCES=1, SINKVALUE>10000000000,
    SAMEDAYSPLITS>0]
# Use Case 6: A transaction path contains a large number of cross
    country transactions.
filters: [COUNTRYHOPS>2, SINKVALUE>10000000000]
```

Listing 5.1: Filter conditions for all evaluated Egmont use cases.

5.1.2 Recall of Injected Components

As mentioned above, our Bitcoin data set lacks ground truth with respect to whether it contains patterns, which correspond to any of the previously defined six use cases. In order to be able to evaluate whether the filter conditions are capable of identifying relevant components, we first have to ensure that they are contained in the input data. For this reason, we manually injected five patterns for each use case in a set of one million transactions and tried to retrieve them by executing a detection algorithm with the corresponding filter condition. In addition to injecting the patterns, our Bitcoin data set also had to be modified in order to contain additional information on the transactions, which was used by the different filter conditions. Therefore we labeled transactions as cash or cross-country with a probability of 10 percent each for each property. Figure 5.3 shows the components that were returned by issuing the filter condition for use case 1. The use case is characterized by components, where assets that originated from cash transactions were subsequently transferred to foreign jurisdictions. The cash transactions are displayed as a yellow lines whereas the cross-country transactions are visualized in blue. As shown at the top of the illustration, the system correctly returned the five injected components. However, the lower part of this illustration shows that the result also includes patterns, which already existed in the data set and fulfilled the filter conditions.

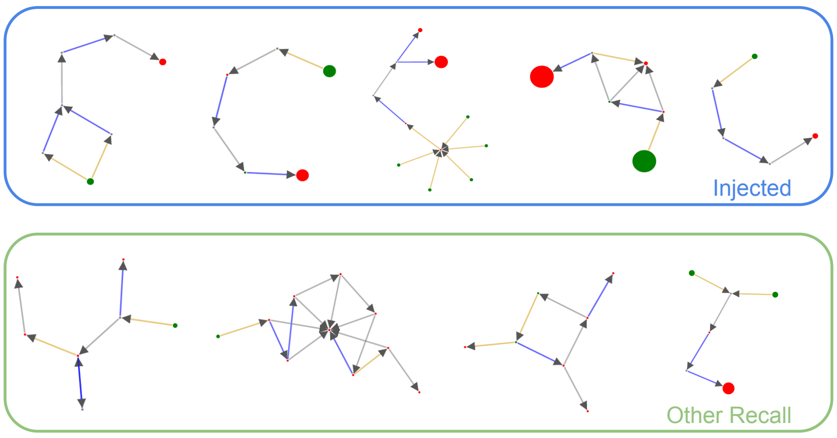


Figure 5.3: Account graphs of the retrieved components for use case 1.

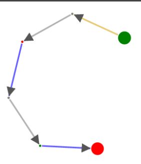
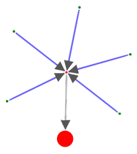
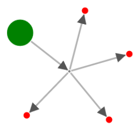
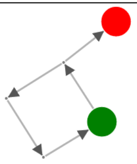
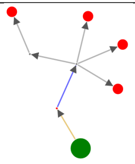
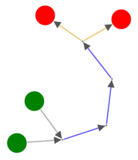
Use Case	1	2	3	4	5	6
Example						
Reports	9	172	12	145	7	30
Recall Inj.	100%	100%	100%	100%	100%	100%

Table 5.3: Recall of injected components.

Table 5.3 shows, that for all six use cases, the system was able to return the complete set of the five injected components together with a number of other, already existing, components. The amount of additional components that were retrieved varies significantly between the use cases. Because no ground truth is available to classify these patters as true or false positives, a direct correlation with the precision of the returned results can not be made based on these numbers.

5.2 Performance depending on Matching Complexity

In a first step towards assessing the performance of the detection system, this section analyzes its performance with respect to the matching complexity. We show that already a small change in matching complexity affects the performance of the system in all three analyzed dimensions. As already described in the implementation description in section 4.1.6, the matching of dependent transactions is limited in its complexity i.e. the number of input and output transactions that will be matched against each other. This number of transactions that take part on each side of the matching process is represented by the matching complexity factor. Depending on the use case and the available infrastructure, the matching complexity of the system can be specified in the execution description, as shown in listing 5.2. The maximum matching duration that is allowed between dependant transactions is fixed at a value of one week. This means that only transactions, which happened within a week after each other, are considered by the matcher. The effect of a variable matching duration will be analyzed in the next section of this chapter. The complete execution description configuration, which was used for this evaluation and the evaluation of the matching duration in the next section, is listed for reproducibility in appendix A.2. Both evaluations were performed on the previously data set consisting of 50 million transaction records from the Bitcoin block chain. All performance evaluations were run on a single machine with two twelve-core AMD Opteron^{TM} 6174 processors and 66 GB RAM. All executions were repeated five times for each parameter setting and reported as the arithmetic mean over all five runs.

```
matchingComplexity: 10
```

Listing 5.2: Matching complexity specified in the execution description.

5.2.1 Execution Time

In a first step, we analyze the effect of a varying matching complexity on the running time of a pattern detection execution. As an increase in matching complexity increases the search space from which the sets of matching transactions are selected, the running time naturally increases with an increased matching complexity. As shown in figure 5.4, the total running time of an execution only increases by a relatively small factor for matching complexities between 4 and 10. However, the running time almost triples when the matching complexity is increased from 10 to 12. As indicated by the lower line in the graph, the effort that is spent in the garbage collection of the Java Virtual Machine (JVM) shows an increase with a similar factor and increases its share on the total running time with higher matching complexities. For a matching complexity of 12 already half the running time is spent in garbage collection.

The increasing influence of garbage collection time on the running time can also be seen in the running time of the individual streaming windows, as shown in figure 5.5. While for matching complexities below 12 the running times of the individual streaming windows only vary very little for neighboring intervals, the running times for the executions with a matching complexity of 12 have a large variance. Based on garbage collection

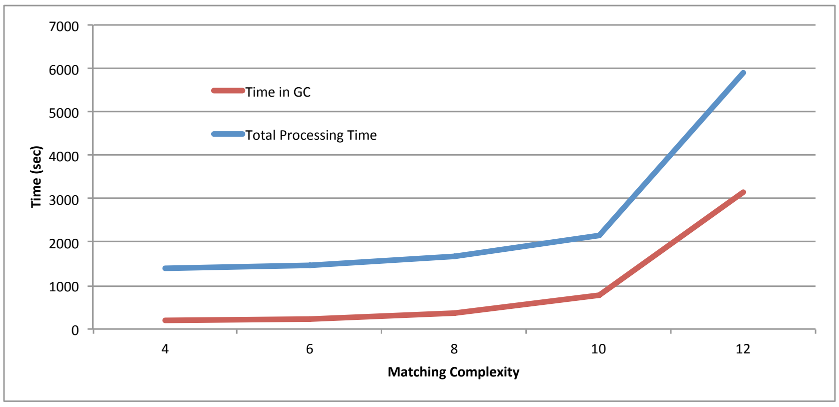


Figure 5.4: Execution time for different matching complexities.

statistics from the JVM it can be shown that the collection is accountable for most of this increased variance in the execution times for the loading windows. In addition to that the matching complexities also require more comparisons in the transaction matching process and through that, increases the computational effort.

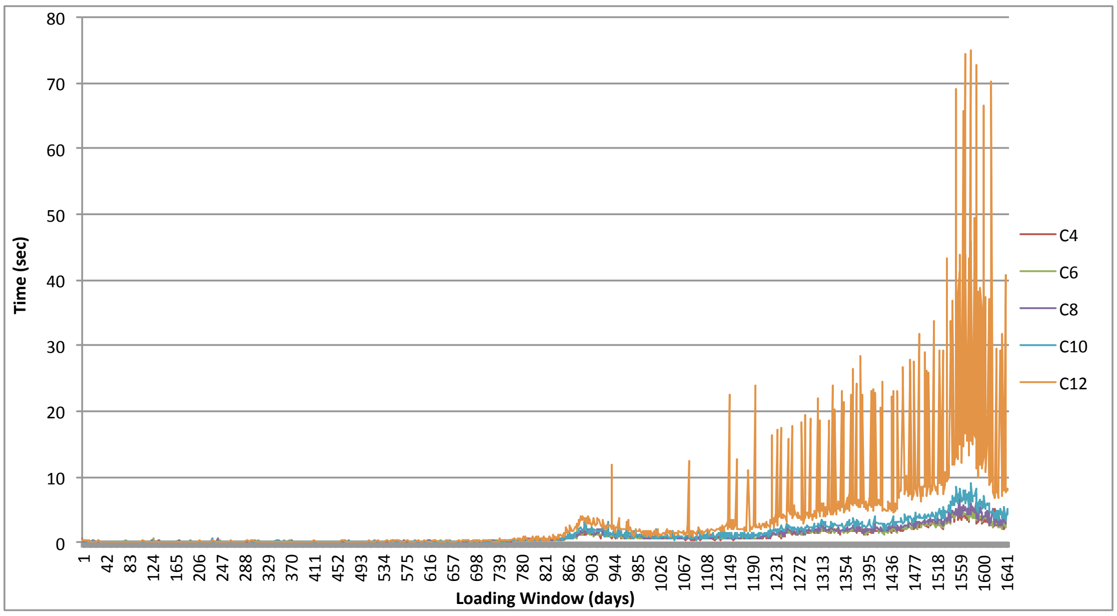


Figure 5.5: Running time per loading window for different matching complexities.

5.2.2 Memory Usage

As observed in the analysis of the running time, the time used in garbage collection overtakes the computation time with increasing matching complexity. The cause for this increase in garbage collection time is shown in figure 5.6. The steep increase in memory usage around day 1000 shows the effect of the space complexity of $O(2^c)$ of the data structure being used to hold the partial matches of transactions at the accounts. Since the memory is measured as the current heap usage and no garbage collection was manually enforced, the memory usage of the different complexities can not directly be compared to each other because they are highly influenced by when garbage collections were performed.

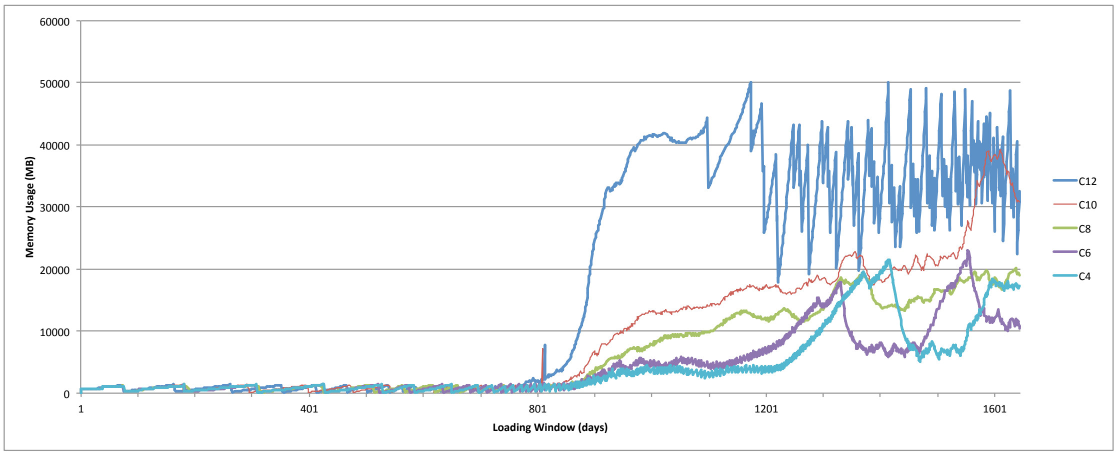


Figure 5.6: Memory usage for varying matching complexities.

5.2.3 Recall

The last factor that is evaluated with respect to its dependence on the matching complexity, is the recall i.e. how many components are reported for a certain query. Interestingly, the evaluation runs show a decrease in reported components, when running the artificial and very generic query shown in listing 5.3 on the set of Bitcoin transactions. Figure 5.7 shows how the number of reported components can also be reduced with an increase in matching complexity. This finding appears counter intuitive at first, but an analysis of the reported components confirms, that while the number of pairwise associated transactions still increases with a higher matching complexity factor, the number of components that are reported decreases. The absolute number of components is decreased, because new connections are found that associate smaller components with each other. This finding could, however, depend on the fact that the Bitcoin system contains a very high number of matchable transactions because of its structural difference to traditional bank transactions, as described in section 3.1.2. The Bitcoin system issues transactions by splitting and aggregating previous transactions, whereas a traditional banking system transactions do not require the reference of previous transactions in order to be executed. For sufficiently large matching complexities and matching durations our system would therefore connect all dependent transactions in the Bitcoin block chain.

```
filters: [SIZE > 5,SINKVALUE > 10000000000,DEPTH > 3,COUNTRYHOPS > 2]
```

Listing 5.3: Filter condition used to measure the number of reports for varying matching complexities.

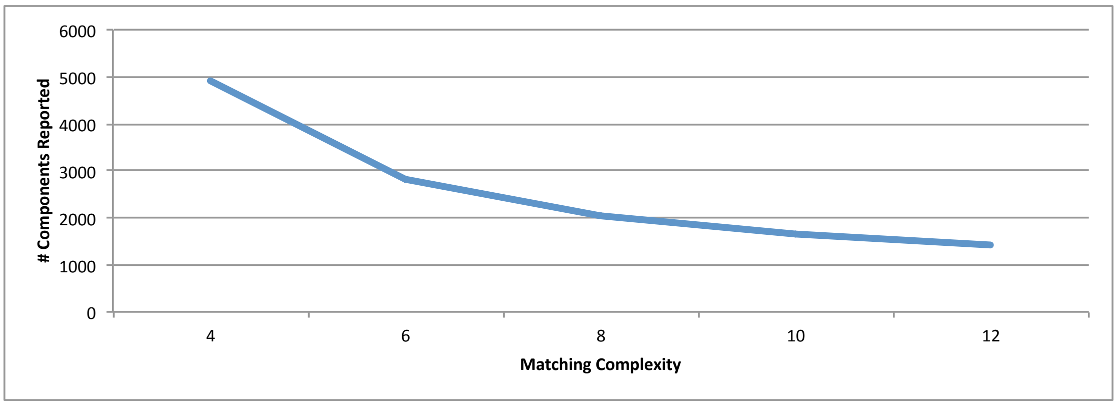


Figure 5.7: Number of reported components for varying matching complexities.

5.3 Performance depending on Matching Duration

After having shown that an increase in matching complexity can have a significant impact on the memory that is being used to store partial matches of transaction, we also analyze the effect of varying the matching durations. A matching duration can be informally described as the maximal time that is allowed between two transactions to be associated with each other. Listing 5.4 shows, how also this property can directly be specified in the execution description language. Apart from the matching complexity being fixed at a value of 10 transactions, the execution parameters as well as the infrastructure remain unchanged compared to the previous complexity evaluation. In the following we highlight the effect of different matching durations on the execution time as well as on the memory usage. Because a longer matching duration also means that components take longer to time out and are therefore also reported later, a comparison of recall would yield time shifted results and is therefore not reported here.

```
transactionInterval: 4w
```

Listing 5.4: Matching duration specified in the execution description.

5.3.1 Execution Time

As seen in figure 5.8 a change in duration only has a very moderate effect on the total execution time. The lower line also suggests, that the total garbage collection times are approximately equally low over all analyzed matching durations. This makes sense, since the amount of data resulting from timed out intermediary data structures is dependent on the matching complexity but not on the matching duration. Even though the garbage collection effort remains unchanged for all matching durations, the total processing time still by 59% when the matching duration is changed from 1 week to 12 weeks.

The reason for this increase in the total processing time on longer matching windows, can be seen in figure 5.9. As seen in the graph the base lines of all matching durations are approximately equal, however, the number of processing windows with a processing time that is considerably higher than the base line increases both in frequency and in duration. As the garbage collection times remained almost unchanged for analyzed matching durations, this increase in processing time must have resulted from additional effort spent in the matching of transactions.

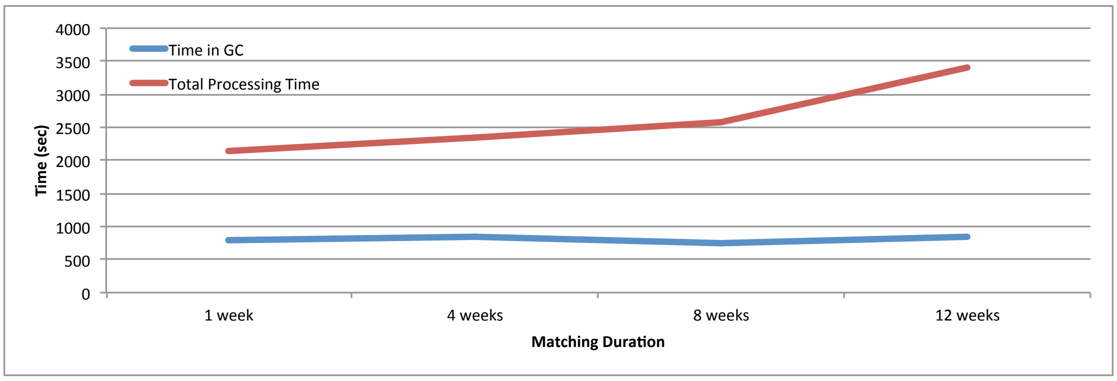


Figure 5.8: Execution time for different matching durations.

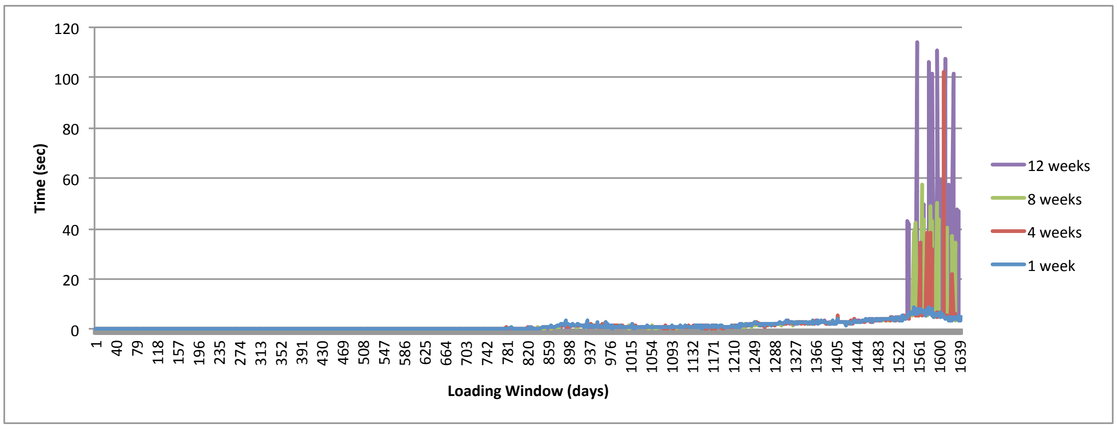


Figure 5.9: Running time per loading window for different matching durations.

5.3.2 Memory Usage

Keeping transactions available for matching for a longer amount of time directly requires that more transactions are kept in memory at every point in time. Figure 5.9 shows, how the memory usage of the system developed over the analyzed period of streaming windows. The graph shows nicely how a longer matching duration requires more memory and that the memory usage develops approximately linearly with the matching duration.

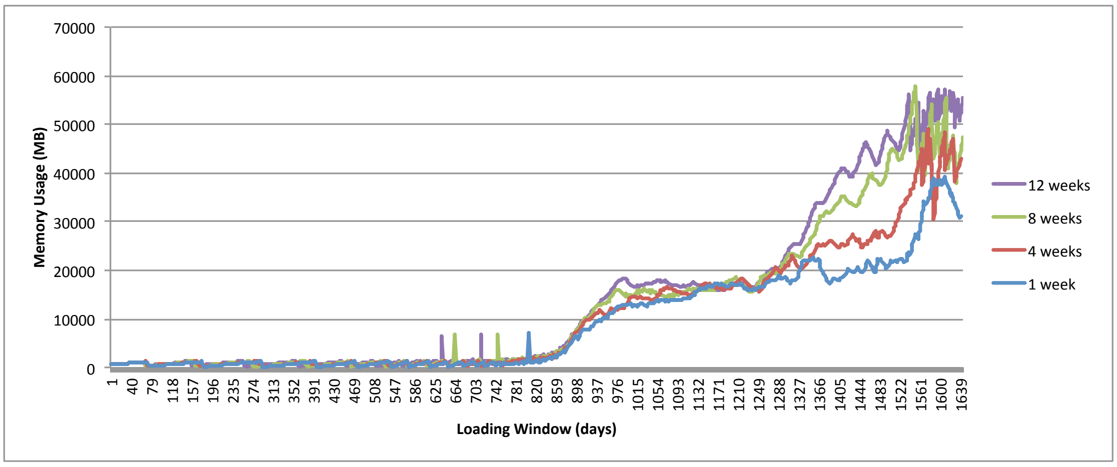


Figure 5.10: Memory usage for varying matching durations.

5.4 Generated Synthetic Banking Transactions.

In order to demonstrate that the detection of interesting components can also be applied to other data sets, than the previously evaluated Bitcoin use case, and to validate our assess the generalizability of our findings, we also analyzed the behavior of the system when applied to a synthetically generated set of banking transactions. The generated set of transactions is constructed to closely resemble the structure of actual financial transactions without the concealment of sensitive information or linking to actually existing accounts. A detailed description of how the data set was generated can be found in (Galliker 2008). In essence, the transactions were generated by gathering information about the transaction volumes as well as the structure of how accounts are interlinked and then constructing an arbitrarily large data set with similar properties. The data set used for this evaluation consists of 346,000 accounts and 1.8 million transactions that are exponentially distributed amongst them. Figure 5.11 shows that, in contrast to the previously analyzed Bitcoin transactions, the generated transactions are uniformly distributed within the time period.

The previous evaluation on the Bitcoin data set suggested a limitation of the system with regard to handling matching complexities above 10 transactions. By analyzing the same dimension on the synthetic data set, we want to test the hypothesis, that this limit

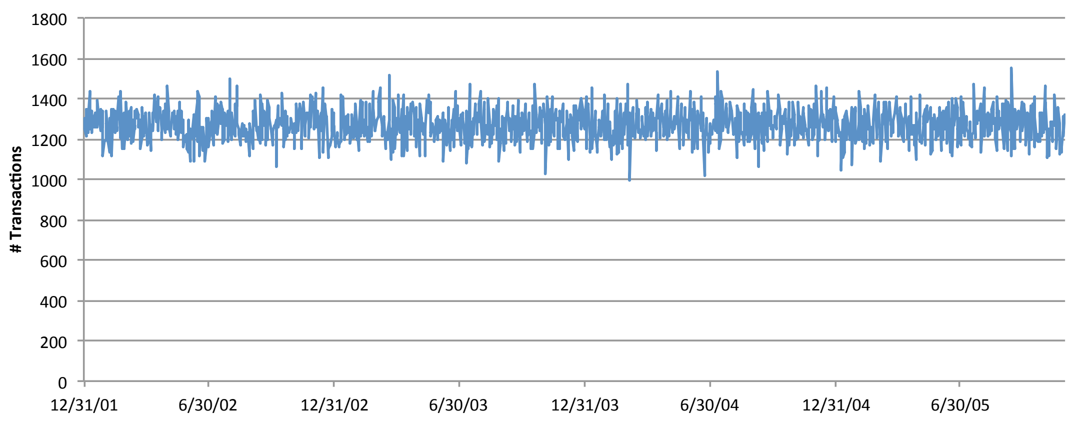


Figure 5.11: Generated banking transactions per day.

is not a direct result of the implementation of the matching module, but is also influenced by the transaction structure in the input data. To evaluate the influence of an increased matching complexity on the synthetic data set, we run the execution description as listed in listing A.3 with the matching duration fixed at twelve weeks. As shown in the graph in figure 5.12, we were able to process the simulated data set with a four times higher matching complexity than the Bitcoin data set allowed for. However, the graph also shows almost no increase in running time, when the matching complexity was increased from 30 to 40 transactions. This leads to the conclusion, that, with a matching complexity of 30, almost all transactions were captured by the matcher. Therefore the complexity increase to 40 transactions did not require any additional matching effort. The indicated garbage collection times show that the amount of timed out objects is again dependent on the complexity factor. However because the maximum amount of transactions per day is much lower than in the Bitcoin evaluation, the total running time remains relatively low. The claim that a matching complexity of 30 is already sufficient to capture all the transactions that were available for matching is also supported by looking at the memory usage of the different matching complexities in figure 5.13. Since the complexity increase from 30 to 40 did not require more memory, we can conclude, that an approximately equal amount of intermediary matching structures were created in both cases.

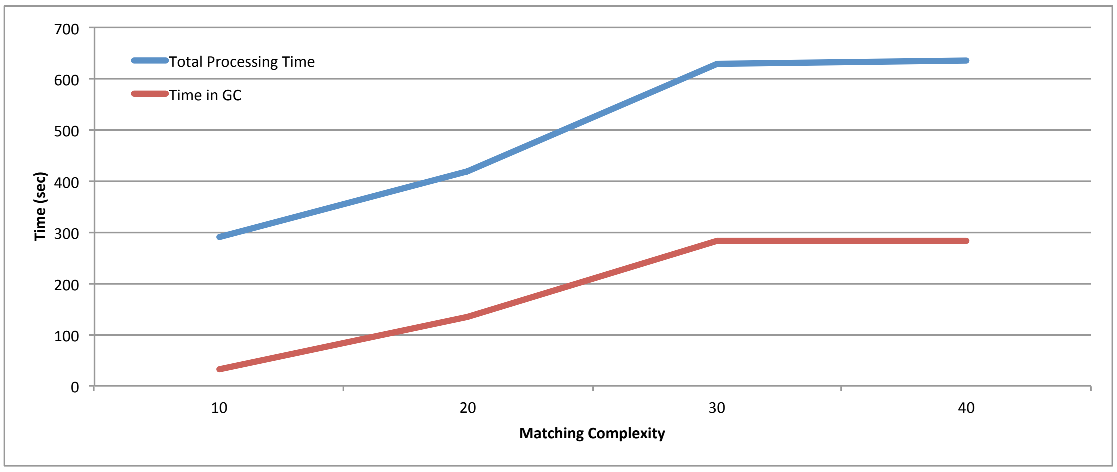


Figure 5.12: Execution time for different matching complexities.

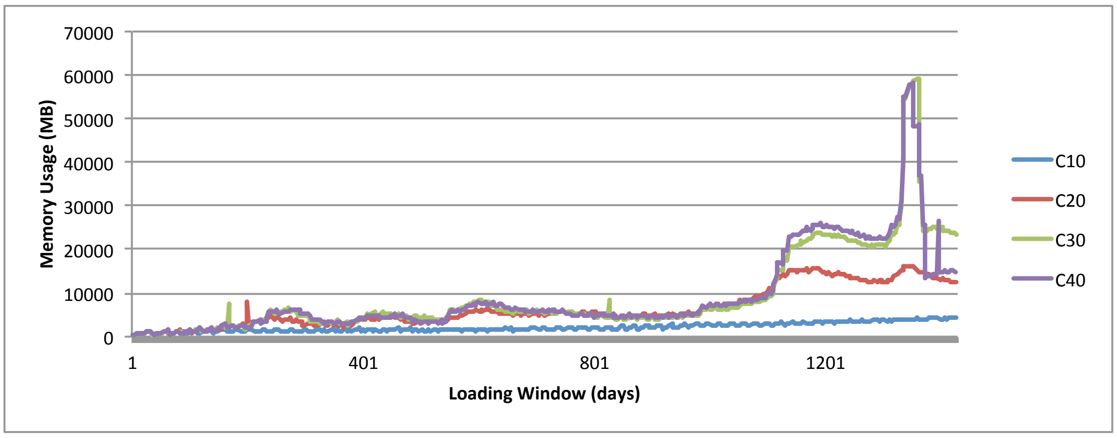


Figure 5.13: Memory usage for different matching complexities.

Based on the observations on this synthetic data set we conclude that higher matching complexities than the limit observed in the Bitcoin evaluation are possible depending on the data set. A smaller number of transactions that are active within the matching window is evidently beneficial to reduce overall the matching effort and therefore allows for higher matching complexities. In a practical application this could be directly exploited by pruning the set of transactions that are considered by the matcher. Apart from the pure number of transactions in the data set, the structure of the transactions in our synthetic data set also differs significantly from the Bitcoin transactions. Because our synthetic transactions were generated in order to closely resemble real-world banking transactions, a dependence of transactions on previous transactions is not required as in the Bitcoin data set and therefore a lot less components and especially much smaller components are retrieved. In addition to that, the Bitcoin data set contains many associations where one incoming transactions is split amongst a large number of outgoing transactions where most of the outgoing transactions where only a fraction of a cent. Despite their unimportance they still have to be considered in the matcher when the matching complexity is chosen high enough. Because our synthetic transactions model real-world transactions where such micro payments would not be economical such enormous splits are a lot less frequent. For these reasons the synthetic data set allowed us to operate at a significantly higher matching complexity than in the previous evaluation.

6

Limitations and Future Work

The evaluation has shown how the current implementation is capable of identifying a variety of patterns within the domain of financial transactions and demonstrated the performance of the system with respect to two parameters of the matching algorithm. However despite these very positive results the system also presents a number of limitations that shall be addressed in this section together with a suggestion of future work of how they could be overcome.

6.1 Memory Consumption vs. CPU Time

The first limitation that became apparent when analyzing the memory consumption with respect to the matching complexity is the relatively large memory footprint of the matching process. The intermediary matching structures that store combinations of transactions to be used in the matching process require a significant amount of memory. On the other hand, they allow the matcher to rely on pre-computed combinations, which reduces the computational effort when a combination is evaluated repeatedly. The decision of sacrificing memory for less CPU time also increased the time the system spent in garbage collection time which adds a third dimension that could be optimized. In order to reduce the running time of evaluations with a high computational complexity all three dimensions offer potential for optimizations. The most straightforward optimization targets the JVM garbage collector that could be tuned to reduce the very expensive collections of the full heap during the computations. Another way of reducing the garbage collection time would be to reduce the memory footprint in the first place by allocating less objects or reducing their size. For this reason the intermediary results could be stored as combinations of subsets instead of storing the subsets redundantly for each combination where the subset is contained.

6.2 Distributed Computation

The distribution of the computation presents another solution to overcome the memory limitations of a single machine. Because the entire matching process is based on the Signal/Collect framework and its underlying actor system, the distribution of the computation is relatively straightforward. The transaction and entity vertices can be loaded on multiple machines without changing the current implementation. However the information that is exchanged needs to be serialized and no objects may be shared between instances that reside on different machines. The current implementation already does not use shared states between computation instances, however the serialization of the messages exchanged in the computation would have to be addressed. The distribution of the computation could further help to increase the overall throughput in scenarios where large fractions of the transactions are only exchanged within small subgroups of entities. By partitioning the transaction graph in a way that all the members of a subgroup reside on the same machine, the matching process would benefit from the additional memory and CPU strength while keeping the network communication low.

6.3 Recall

As described in the evaluation section we were not able to measure the absolute recall of suspicious components in the graph because of the lack of ground truth. Even though all manually injected components had been retrieved, there are still scenarios where the detection could miss out on reporting a component. The limitation of the matching complexity and matching durations as described in the evaluation could cause the matcher to not detect dependencies amongst transactions, which lie outside of these boundaries. For some use cases, associations of transactions that lie outside of this restrictions might be regarded as either too complex or too far from each other, while in other cases they exclude relevant matches. In order to reduce the number of cases that fall under this second case, the performance of the system needs to be tuned further to allow for a broader range of matching durations and complexities. Another limitation of the matching process is introduced because the system allows transactions to participate in multiple matches with other transactions. While this allows capturing a larger amount of matches between the transactions, it also increases the size and structure of their connected component. Because the filtering process currently only operates at the level of the entire component this expansion of the component potentially also influences the precision and recall of the entire system. A solution for this problem would be, to evaluate the filters on all sub-graphs of a connected component while unfolding the ambiguity introduced from transactions that were matched multiple times.

6.4 Low Latency Execution

The last limitation, that is addressed in this section, results from the design choice that the system waits for all members of a component to time out, i.e. to no longer be available for matching. In the current implementation components are only checked against filter conditions and reported once all its members have timed out. For very large components, this means, that the reporting of sub-components that participate early on in a component is potentially delayed for a long amount of time. In the worst case, where the component will continuously be extended with new transactions, the component could never be reported. The current implementation prevents these infinitely growing components by enforcing a maximum duration on the components. Even though this limitation only had to be applied on the atypical transaction structure of the Bitcoin use case, it introduces the possibility of again missing out on components that were cut off by this duration limit. A solution the problem would be to continuously evaluate components and report them even when some of its members are still available for matching. This would require that the transaction vertices would run the algorithms for matching, component detection and filtering concurrently instead of sequentially. The current implementation already supports the nesting of algorithms, however the transaction vertex would have to be adapted to be able to rout the messages it received to the appropriate algorithm implementation. Together with the validation of sub-graphs of the connected components, as mentioned above, this would result in an architecture that is capable of directly identifying all suspicious components that a transaction participates in, at the time that transaction is added to the graph.

7 Conclusions

This thesis introduced a novel approach for identifying interesting patterns in transactional data series. We presented a bottom-up graph algorithm based on matching dependent transactions at a local level and using these associations as building blocks for constructing larger components. We described how this approach can be implemented on top of a generic graph processing framework and used in forensic investigations. Our implementation targets the domain of financial fraud detection and showed that our solution is capable of detecting complex money laundering patterns within millions of transactions. Even though the data sets in our evaluation lacked ground truth, we were able to show that the matching approach and the flexible filter conditions enable the retrieval of patterns that correspond to real-world occurrences of financial fraud. We were also demonstrated the capability of our system with respect to the goal of being able to process large amounts of transactional data. The continuous processing of incoming transaction streams allows us to process more than one million transactions per minute in the Bitcoin evaluation and over 350,000 transactions in the evaluation of simulated banking transactions. Despite of these impressive performance benchmarks, our implementation is designed as a flexible and modular architecture that is open and extensible to be used in other use case scenarios. With the higher-level execution description language and the graphical user interface, to analyze the reported patterns, we also showed that our matching approach can be integrated in an end-to-end forensic analysis and used by domain experts.

Bibliography

Amstein, Stefan (2009). “Evaluation und Evolution von Pattern-Matching-Algorithmen zur Betrugserkennung”. MA thesis. University of Zurich.

Barbieri, Davide Francesco et al. (2009). “C-SPARQL: SPARQL for continuous querying”. In: *Proceedings of the 18th international conference on World wide web*. ACM, pp. 1061–1062.

Bhattacharyya, Siddhartha et al. (2011). “Data mining for credit card fraud: A comparative study”. In: *Decision Support Systems* 50.3, pp. 602–613.

Blakeley, Mike (2008). “SQL as an Audit Tool”. In: *EDPAC: The EDP Audit, Control, and Security Newsletter* 37.6, pp. 1–16.

Chan, Philip K et al. (1999). “Distributed data mining in credit card fraud detection”. In: *Intelligent Systems and their Applications, IEEE* 14.6, pp. 67–74.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar (2009). “Anomaly detection: A survey”. In: *ACM Computing Surveys (CSUR)* 41.3, p. 15.

Chang, Remco et al. (2007). “WireVis: Visualization of categorical, time-varying data from financial transactions”. In: *Visual Analytics Science and Technology, 2007. VAST 2007. IEEE Symposium on.* IEEE, pp. 155–162.

Cortes, Corinna, Daryl Pregibon, and Chris Volinsky (2001). *Communities of interest.* Springer.

Derrig, Richard A (2002). “Insurance fraud”. In: *Journal of Risk and Insurance* 69.3, pp. 271–287.

Egmont Group (2000). *100 Cases from the Egmont Group.* URL: *http://www.egmontgroup.org/library/download/21* (visited on 11/05/2013).

Facebook Inc. (2013). *Facebook, Key Facts.* URL: *http://newsroom.fb.com/Key-Facts* (visited on 10/29/2013).

Fan, Wenfei et al. (2011a). “Adding regular expressions to graph reachability and pattern queries”. In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on.* IEEE, pp. 39–50.

Fan, Wenfei et al. (2011b). “Incremental graph pattern matching”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data.* ACM, pp. 925–936.

Fayyad, Usama, Gregory Piatetsky-Shapiro, and Padhraic Smyth (1996). “From data mining to knowledge discovery in databases”. In: *AI magazine* 17.3, p. 37.

Gallagher, Brian (2006). “Matching structure and semantics: A survey on graph-based pattern matching”. In: *AAAI FS* 6, pp. 45–53.

Galliker, Samuel (2008). “Generierung von synthetischen Banktransaktionsdaten”. BA thesis. University of Zurich.

Gamma, Erich et al. (1995). *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Henzinger, Monika Rauch, Thomas A Henzinger, and Peter W Kopke (1995). “Computing simulations on finite and infinite graphs”. In: *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on.* IEEE, pp. 453–462.

Holzschuher, Florian and René Peinl (2013). “Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j”. In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. EDBT ’13. ACM, pp. 195–204.

John Tang Mirco Musolesi, Cecilia Mascolo and Vito Latora (Jan. 2010). “Characterising Temporal Distance and Reachability in Mobile and Online Social Networks”. In: *ACM SIGCOMM Computer Communication Review* 40.1, pp. 118–124.

Kirkland, J Dale et al. (1999). “The NASD Regulation advanced-detection system (ADS)”. In: *AI Magazine* 20.1, p. 55.

Kröni, Daniel and Raphael Schweizer (2013). “Parsing graphs: applying parser combinators to graph traversals”. In: *Proceedings of the 4th Workshop on Scala*. SCALA ’13. ACM, 7:1–7:4.

Low, Yucheng et al. (2010). “GraphLab: A New Framework For Parallel Machine Learning”. In: *UAI*, pp. 340–349.

Low, Yucheng et al. (2012). “Distributed GraphLab: A framework for machine learning and data mining in the cloud”. In: *Proceedings of the VLDB Endowment* 5.8, pp. 716–727.

Luell, Jonas (2010). “Employee Fraud Detection under Real World Conditions”. PhD thesis. University of Zurich, p. 183.

Malewicz, Grzegorz et al. (2010). “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10, pp. 135–146.

Meiklejohn, Sarah et al. (2013). “A Fistful of Bitcoins: Characterizing Payments Among Men with No Names”. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC ’13, pp. 127–140.

Moll, Linard (2009). “Anti Money Laundering under real world conditions - Finding relevant patterns”. MA thesis. University of Zurich.

Nakamoto, Satoshi (2008). *Bitcoin: A peer-to-peer electronic cash system.*

Nayar, Pramod K (2010). “WikiLeaks, the New Information Cultures, and Digital Parrhesia”. In: *Economic and Political Weekly* 45.52, pp. 27–30.

Ngai, EWT et al. (2011). “The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature”. In: *Decision Support Systems* 50.3, pp. 559–569.

Noble, Caleb C and Diane J Cook (2003). “Graph-based anomaly detection”. In: *Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, pp. 631–636.

Prud'hommeaux, Eric and Andy Seaborne (2008). "SPARQL Query Language for RDF". In: *W3C Recommendation.* Working Draft 2009.January, pp. 1–106.

Quandl (2013). *Bitcoin Total Number of Transactions.* URL: *http://www.quandl.com/BCHAIN-Blockchain/NTRAT-Bitcoin-Total-Number-of-Transactions* (visited on 10/29/2013).

Reid, Fergal and Martin Harrigan (2013). "An analysis of anonymity in the bitcoin system". In: *Security and Privacy in Social Networks.* Springer, pp. 197–223.

Senator, Ted E et al. (1995). "Financial Crimes Enforcement Network AI System (FAIS) Identifying Potential Money Laundering from Reports of Large Cash Transactions". In: *AI magazine* 16.4, p. 21.

Shao, Bin, Haixun Wang, and Yatao Li (2013). "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* SIGMOD '13. ACM, pp. 505–516.

Stokes, Robert (Oct. 2012). "Virtual money laundering: the case of Bitcoin and the Linden dollar". In: *Inf. Commun. Technol. Law* 21.3, pp. 221–236.

Stutz, Philip, Abraham Bernstein, and William W. Cohen (2010). "Signal/Collect: Graph Algorithms for the (Semantic) Web." In: *International Semantic Web Conference.* Vol. 6496. Lecture Notes in Computer Science. Springer, pp. 764–780.

Stutz, Philip et al. (2013). "TripleRush: a fast and scalable triple store". In: *9th International Workshop on Scalable Semantic Web Knowledge Base Systems.* CEUR Workshop Proceedings.

Ullmann, Julian R (1976). "An algorithm for subgraph isomorphism". In: *Journal of the ACM (JACM)* 23.1, pp. 31–42.

Wang, Shiguo (2010). "A Comprehensive Survey of Data Mining-Based Accounting-Fraud Detection Research". In: *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on.* Vol. 1.

A

Appendix

A.1 Result Exchange Format

```
{
    "14": {
        "id": -10000003,
        "start": 1352267497000,
        "end": 1352712643000,
        "flow": 25000000000,
        "members": [
            {
                "id": -10000001,
                "cash": false,
                "value": 25000000000,
                "target": 11110009,
                "time": 1352407188,
                "src": 11110029,
                "component": -10000003,
                "xCountry": true,
                "successor": [
                    -10000002
                ]
            },
            {
                "id": -10000003,
                "cash": false,
                "value": 25000000000,
                "target": 11110020,
                "time": 1352712643,
                "src": 11110019,
                "component": -10000003,
                "xCountry": false,
                "successor": []
            },
            {
                "id": -10000000,
                "cash": true,
                "value": 25000000000,
                "target": 11110029,
                "time": 1352267497,
                "src": 11110000,
                "component": -10000003,
                "xCountry": false,
                "successor": [
                    -10000001
                ]
            },
            {
                "id": -10000002,
                "cash": false,
                "value": 25000000000,
                "target": 11110019,
                "time": 1352641705,
                "src": 11110009,
                "component": -10000003,
                "xCountry": true,
                "successor": [
                    -10000003
                ]
            }
        ]
    }
}
```

Listing A.1: JSON serialized reported component.

A.2 Execution Descriptions used in Performance Evaluations

```
#path of the input data
source: <<INPUT_FILE_PATH>>

#tells the parser about the semantics of the entries in the
```

```
#input data
parse:
  id: [0,Int]
  src: [2,Int]
  target: [3,Int]
  value: [4,Long]
  time: [5,Long]

#interval of the input data that should be processed
start: 01/09/2009 02:50:00
end: 08/18/2013 17:30:00

#streaming window duration
window: 1d

#set to the max amount of time between two associated transactions
transactionInterval: <<INTERVAL>>

#set to true if the matcher should follow all possible matching
    possibilities
exhaustiveMatching: true

#set the max number of inputs and outputs that are matched against each
    other
matchingComplexity: <<COMPLEXITY>>

#sets the max duration of components to prevent them from lasting for
    the entire streaming period.
maxComponentDuration: 8w

#conditions that a component has to fulfil to be reported
filters: [SIZE > 5,SINKVALUE > 10000000000,DEPTH > 3,COUNTRYHOPS > 2]

#handlers that receive the reported components
#e.g. WEBSERVER, CONSOLE, MONGODB
handlers: [WEBSERVER, COUNTING]
```

Listing A.2: Execution description used in the performance evaluation on Bitcoin data.

```
#path of the input data
source: <<INPUT_FILE_PATH>>

#tells the parser about the semantics of the entries in the
```

```
#input data
parse:
  id: [0,Int]
  src: [1,Int]
  target: [2,Int]
  value: [4,Long]
  time: [3,Long]

#interval of the input data that should be processed
start: 12/31/2001 22:00:00
end: 18/08/2006 17:30:00

#streaming window duration
window: 1d

#set the max amount of time between two associated transactions
transactionInterval: 12w

#set to true if the matcher should follow all possible matching
    possibilities
exhaustiveMatching: true

#set the max number of inputs and outputs that are matched against each
    other
matchingComplexity: <<COMPLEXITY>>

#sets the max duration of components to prevent them from lasting for
    the entire streaming period.
#maxComponentDuration: 8w

#conditions that a component has to fulfil to be reported
filters: [SIZE > 5,SINKVALUE > 10000,SINKTRANSACTIONS>SINKACCOUNTS,
    DEPTH>3]

#handlers that receive the reported components
#e.g. WEBSERVER, CONSOLE, MONGODB
handlers: [WEBSERVER]
```

Listing A.3: Execution description used in the performance evaluation on synthetic bank transactions.

A.3 Classification of Egmont Cases

Case #	**Use Case**	**Only detectable using information outside the transaction graph**	**Flag condition on single transaction/account is sufficient**	**Can be modelled as a suspicious component but is not supported**	**unclear/not enough information**
1		1			
2			1		
3		1			
4	1				
5	4				
6	2				
7	1				
8	2				
9				1	
10		1			
11	1				
12			1		
13	2				
14			1		
15			1		
16			1		
17		1			
18	3				
19		1			
20	4				
21		1			
22		1			
23			1		
24				1	
25	3				
26	1				
27	1				
28					1
29			1		
30			1		
31		1			
32					1
33		1			
34	2				

Case #	Use Case	Only detectable using information outside the transaction graph	Flag condition on single transaction/account is sufficient	Can be modelled as a suspicious component but is not supported	unclear/not enough information
35			1		
36			1		
37				1	
38					1
39					1
40	4				
41		1			
42		1			
43			1		
44		1			
45			1		
46	5				
47				1	
48			1		
49				1	
50		1			
51	5				
52					1
53	5				
54			1		
55	2				
56	3				
57	1				
58			1		
59	6				
60	2				
61	4				
62					1
63		1			
64		1			
65			1		
66			1		

Case #	**Use Case**	**Only detectable using information outside the transaction graph**	**Flag condition on single transaction/account is sufficient**	**Can be modelled as a suspicious component but is not supported**	**unclear/not enough information**
67	2				
68			1		
69	4				
70		1			
71		1			
72				1	
73			1		
74	1				
75			1		
76			1		
77		1			
78					1
79			1		
80	1				
81			1		
82			1		
83			1		
84					1
85	1				
86		1			
87	1				
88	1				
89			1		
90		1			
91			1		
92					1
93		1			
94		1			
95					1
96		1			
97		1			
98		1			
99		1			
100					1

Table A.1: Complete classification of Egmont case studies

List of Figures

List of Tables

List of Code Snippets