Universität Zürich
Institut für Informatik

Dynamic and Distributed
Information Systems

Evaluation und Evolution von Pattern-Matching-Algorithmen zur Betrugserkennung

Am Beispiel des gegebenen Algorithmus *ChainFinder*

Diplomarbeit im Fach Informatik

Vorgelegt von

- Stefan Amstein Uster, Schweiz, Matrikelnummer: 03-710-860

Betreut von

- Jonas Luell

Angefertigt am

- Institut für Informatik der Universität Zürich, Binzmühlestrasse 14, CH-8050 Zürich, Schweiz
- Prof. A. Bernstein, PhD

Abgabe der Arbeit: 5. Mai 2009.

Abstract

Bei der Fahndung nach Betrugsfällen in grossen Datensammlungen von Unternehmen und Regierungsstellen kommen vermehrt Verfahren der künstlichen Intelligenz (wie Data Mining) zum Einsatz, aber auch Methoden zum Strukturvergleich (das sog. Graph-Pattern-Matching).

In dieser Arbeit wird ein gegebener Graph-Pattern-Matching-Algorithmus („ChainFinder"), der nach Transaktionsketten in Finanzdaten sucht, welche auf betrügerisches Verhalten hinweisen können, evaluiert und evolviert. Für die Evaluation werden Kennzahlen der Korrektheit und der Performance erarbeitet und Serien von Testläufen durchgeführt, wie auch deren Ergebnisse diskutiert. Im Verlauf der Evolution entstehen verschiedene Derivate des ChainFinders, welche ebenfalls evaluiert und vergleichend diskutiert werden.

Mithilfe der gewonnenen Erfahrungen wurde eine Applikation entwickelt, welche einem künftigen Nutzer die Evaluation eigener ähnlicher Algorithmen durch Automatismen und die eingeflossenen Konzepte erleichtern kann.

Inhalt

Abbildungen

1 Einleitung

Wird ein Betrug begangen, so verschafft sich der Täter gemäss Definition einen „unrechtmässigen Vermögensvorteil".[1] Das Delikt ist wahrscheinlich so alt wie die Menschheit selbst, hat jedoch durch die verschiedenen Epochen immer wieder neue Ausprägungen angenommen. Der steigende Einfluss der Informationstechnologie führte gerade in jüngster Zeit zu einer Digitalisierung von mehr und mehr Lebensbereichen. Jede Aktion im digitalen Raum hinterlässt potenziell eine Datenspur,[2] wobei die Menge an solchen Spuren stetig wächst. Interessant sind die neuen Möglichkeiten der Datenverarbeitung nicht nur für Forschung, Wirtschaft und die öffentliche Hand, sondern auch für Kriminelle. So entwickelten sich viele neue Betrugstechniken, während gleichzeitig aber auch neue Arten der Erkennung und Prävention betrügerischer Verhaltensweisen im Kontext einer mehr und mehr digitalisieren Gesellschaft entwickelt werden.

Typische für Betrugshandlungen anfällige Bereiche befinden sich sowohl im privaten, wie auch im öffentlichen Sektor. Demnach seien Wirtschaftszweige wie die Finanz- und Versicherungsindustrie, sowie Telekommunikationsunternehmen und IT-Dienstleister besonders hervorgehoben. Zunehmend kommen aber auch Privatpersonen in das Visier der Computerbetrüger. Im öffentlichen Sektor sei einmal das Gesundheitswesen genannt, oder auch Prozesse, die näher am Kern des politischen Systems liegen, wie beispielsweise Abstimmungen und Wahlen. So wurde im März dieses Jahres im U.S. Bundesstaat Kentucky die Verwundbarkeit von Wahlmaschinen bewiesen.[3]

Die betrügerische Handlung selbst hinterlässt in der Regel Datenspuren, welche zur Betrugserkennung herangezogen werden können. Sie unterscheiden sich in ihrer Form jedoch nicht von normalen Datenspuren. Bildlich gesprochen gilt es eine Nadel in einem Heuhaufen zu finden, wobei Nadel und Heu aus demselben Material bestehen.[4]

Die rein manuelle Fahndung nach Betrugsfällen ist wegen der schieren Grösse des Heuhaufens (sprich des Umfangs angefallenen Datenmengen) oftmals unpraktikabel. Deshalb wurden computergestützte Verfahren entwickelt, welche sich vor allem im Automatisierungsgrad und in der Effektivität unterscheiden. Oftmals kommen mehrere parallel oder in Folge zum Einsatz und es bestehen Eskalationswege hin zu weniger automatisierten Ansätzen, welche eine höhere Arbeitslast für menschliche Betrugsexperten darstellen und dadurch schnell hohe Kosten verursachen. In der Praxis muss also oft eine Abwägung gemacht werden, je nachdem, wie teuer die Aufdeckung eines Betrugsfalles ist und welchen Nutzen damit erreicht würde. Zugleich müssen auch gesetzliche Rahmenbedingungen wie Datenschutz-Normen oder Regulierungen, die den Finanzsektor betreffen, eingehalten werden.

Betrug ist eine Täuschung, die dem Täter einen ungerechtfertigten Vermögensvorteil verschafft.
Betrugserkennung umfasst die Gesamtheit der Massnahmen, die der möglichst zeitnahen Aufdeckung eines Betrugsfalles dienen, nachdem dieser begangen wurde.
Betrugsprävention setzt sich die Verhinderung von Betrugsfällen zum Ziel, bevor diese begangen werden.

Tabelle 1 Definitionen angelehnt an (Bolton & David, 2002)

[1] In der Schweiz unter anderem strafbar gemäss Art. 146 StGB, http://www.admin.ch/ch/d/sr/311_0/a146.html, zuletzt besucht am 22. März 2009.
[2] Fallen viele Spuren an, die weite Bereiche des Lebens umfassen, ohne dass ein effektiver Datenschutz gewährleistet ist, kann das wird das Individuum mehr und mehr transparent derjenigen Instanz gegenüber, welche Zugriff auf die betreffenden Daten hat.
[3] http://www.heise.de/tp/r4/artikel/30/30000/1.html, zuletzt besucht am 24. März 2009.
[4] Die Metapher ist aus (Bolton & David, 2002) entlehnt.

Die Betrugserkennung sollte demnach also möglichst effektiv und effizient sein und diese Merkmale auch bei steigender Menge an zu verarbeitenden Daten möglichst nicht verlieren (sprich sie soll *skalieren*).

Nach diesen grundsätzlichen Betrachtungen zur Betrugserkennung und -Prävention seien in den kommenden Abschnitten noch einige Branchen und deren Ansätze mit diesen Problemstellungen speziell erwähnt.

Computereinbruch

Computereinbruch und Computerkriminalität fallen nicht in allen Formen unter den Tatbestand des Betruges.[5] Momentan werden vermehrt Systeme mit dem Zweck angegriffen, sie dazu zu bringen, unerwünschte Werbung (sprich *SPAM*) zu versenden. Ebenfalls können Informationen ausgespäht oder die Ressourcen des Rechners für Angriffe auf Drittsysteme verwendet werden (sog. *Denial-of-Service* Angriffe). Unterdessen kann bereits von einem bedeutenden Mark für solche illegalen Dienstleistungen gesprochen werden, was deren Angebot für Kriminelle zu einem veritablen Geschäft macht.

Zur Bekämpfung solcher Angriffe existieren bereits einige kommerzielle Produkte auf dem Markt. Speziell herausgehoben seien solche zur sog. *Intrusion Prevention*. Diese analysieren in der Regel den Netzwerkverkehr (*Paket Inspection),* aber teilweise auch das Verhalten eines Rechners. Dabei kommen übliche Ansätze wie *Data Mining*, neuronale Netzwerke und Expertensysteme zum Einsatz. Auch *Model-Based Reasoning*, *State Transition Analysis* und genetische Algorithmen können dabei helfen, Angriffe aufzudecken. Das Interessante an dieser Domäne ist vielleicht eher die Herkunft der Daten und nicht unbedingt die Methoden. Als Grundlage der Analyse wird meistens der sog. *Audit Trail* des Betriebssystems benutzt, sprich also dessen integrierte Protokollierungsfunktionen. Weitergehende Informationen finden sich in (Bolton & David, 2002).

Mobilfunkindustrie

Das Ziel der Betrüger ist die Erschleichung von Dienstleistungen und/oder das Verschleiern der eigenen Identität. Um dies zu erreichen, identifiziert sich der Täter unter der Kennung eines legitimen Anschlussinhabers gegenüber dem Netzwerk des Mobilfunkanbieters durch die Verwendung einer geklonten SIM-Karte. Der Netzwerkbetreiber „sieht" also stets nur einen Account, welcher seine Dienste in Anspruch nimmt, auch wenn dieser von mehr als einer Person genutzt wird, also sowohl vom Anschlussinhaber, wie auch dem Betrüger.

Um den Rahmen dieser Arbeit nicht zu sprengen, seien hier nur noch einige Informationen zu den einschlägigen Techniken aufgelistet, welche bei der Betrugserkennung in der Mobilfunkbrache zur Anwendung kommen:

- **Verhalten:** Betrüger erkennt man möglicherweise an ihrem Telefonverhalten. Das reale soziale Netzwerk des Betrügers besteht auch dann weiter, wenn er unter im Aufdeckungsfall wechselnder Identifikation Leistungen beansprucht. (Bolton & David, 2002, p. 237)
- **Regelbasiert**: Es gibt physikalische Einschränkungen, die eine „Huckepack"-Nutzung eines Benutzerkontos verraten können. So ist es beispielsweise schwierig bis unmöglich für einen Menschen, sich mit mehr als etwa 1000 Kilometern pro Stunde fortzubewegen. Da der ungefähre Ort des Endgerätes bei der Tätigung eines Anrufes dem Provider bekannt ist, kann die Überprüfung der genannten Regel über mehrere Telefonate hinweg bei der Betrugsfahndung helfen.

[5] In der Schweiz geltende Tatbestände des Strafrechts sind unter anderem: Art. 143 (unbefugte Datenbeschaffung), Art. 143bis (unbefugtes Eindringen in ein Datenverarbeitungssystem), Art. 144bis (Datenbeschädigung), Art. 147 (betrügerischer Missbrauch einer Datenverarbeitungsanlage), Art. 150bis (sog. „Virentatbestand")

- **Weitere Verfahren:** Darüber hinaus gibt es Techniken der Datenvisualisierung und deren Auswertung durch Betrugsexperten sowie die Erstellung von Benutzerprofilen, neuronalen Netzwerken und die Analyse der eingegebenen Ziffern. Die analytische Leistung eines menschlichen Beobachters von sinnvoll visualisierten Daten kann unter Umständen der computergestützten Verarbeitung überlegen sein.

Finanzindustrie

Ein beträchtlicher Teil aller Ladendiebstähle im Einzelhandel wird vom Personal begangen. Auch bei Finanzinstitutionen haben die Mitarbeiter die Möglichkeit, sich durch das Ausführen von betrügerischen Transaktionen persönlich zu bereichern. Begehen die Mitarbeiter einen solchen Betrug, so sprechen wir von „*Internal Fraud*"[6]. Je geringer die technischen Chancen auf erfolgreiche Ermittlung sind, desto höher sind die Anreize für die Mitarbeiter, Delikte zu begehen.

Andere einschlägige Delikte in der Finanzindustrie sind die Geldwäscherei[7] und der Kreditkartenbetrug,[8] wobei diese Taten jedoch in der Regel nicht von den Mitarbeitern begangen werden.

Bei der Geldwäscherei wird versucht, die illegale Herkunft eines Vermögenswertes zu verschleiern. Bei der Suche nach Mustern in den Transaktionsdaten, welche auf Geldwäscherei hinweisen, muss man berücksichtigen, dass oftmals mehrere *Mittelsmänner* (das können natürliche oder juristische Personen sein) am Prozess beteiligt sind. Sie werden für ihre Dienste, sprich die Ausführung von Transaktionen, bezahlt. So ist davon auszugehen, dass am Ende weniger Geld in gewaschener Form übrig ist, als am Anfang in ungewaschener Form da war.

Geldwäscherei: „Einschleusung illegaler Erlöse aus Straftaten (zum Beispiel aus Drogenhandel) in den legalen Finanz- und Wirtschaftskreislauf".[9]

Internal Fraud: „the use of one's occupation for personal enrichment through the deliberate misuse of the employing organization's resources or assets". (Porter, 2003)

Betreffend Geldwäscherei sei noch eine spezielle Technik, das sog. *Smurfing*, hervorgehoben. Dabei wird ein grösserer Betrag auf mehrere kleinere Transaktionen aufgeteilt, welche jeweils andere Ziel-Konten haben. Durch die Verwendung kleinerer Beträge soll erreicht werden, den Betrugserkennungsmechanismen zu entgehen. Um Transaktionen von kleineren Beträgen durchzuführen, sind in der Regel weniger strenge Meldungs- und Überwachungsmechanismen vorgesehen.

Viele Firmen und Finanzinstitute halten sind sehr zurückhaltend bezüglich Informationen zu den Details der Betrugserkennung und –Prävention, welche sie durchführen. Das könnte darauf beruhen, dass sie Neueinsteigern keine detaillierte Anleitung für einen erfolgreicheren Betrug geben möchten. Inwieweit jedoch erfahrene Betrüger Anreize zur Verbreitung ihres Wissens haben, sei einmal dahingestellt. Klar ist, dass sich durch eine Geheimhaltung solcher Informationen gewisse „Markteintrittsbarrieren" für Neubetrüger begründen liessen.

Ausserdem sind die vorgehaltenen Daten in der Regel geheim, denn die Finanzinstitute sind vertraglich und gesetzlich zur Geheimhaltung verpflichtet. Mängel in der Datenintegrität oder -sicherheit haben potenziell verheerende Aus-

[6] In der Schweiz fallen solche Machenschaften möglicherweise unter den Tatbestand des „Computerbetrugs" i.S.v. Art. 147 StGB.
[7] Art. 305bis und 305ter StGB sowie die Konkretisierungen durch das Geldwäschereigesetz und dessen Verordnungen.
[8] Art. 148 StGB.
[9] http://de.wiktionary.org/wiki/Geldwäsche (letzter Zugriff im März 09)

wirkungen auf den Ruf des Unternehmens und damit den Unternehmenserfolg. Diese unternehmenspolitischen Schranken sind nicht zu unterschätzen, insbesondere wenn man bedenkt, dass die Entwicklung, die Implementation und der Betrieb einer systematischen Betrugsprävention und -Erkennung einen umfassenden Zugriff auf die Datenbestände benötigt.

Jeder nicht aufgedeckte Betrug bedeutet in der Regel einen finanziellen Verlust für die betreffende Unternehmung, oder zumindest einen potenziellen Verlust. Auf der anderen Seite verursacht aber auch die Betrugserkennung und –Prävention Kosten. Es kann auch sein, dass durch das Bekanntwerden der Betrugsanfälligkeit das Image des Unternehmensträgers, insbesondere in der Finanzindustrie, wo das entgegengebrachte Vertrauen der Kundschaft ein wesentlicher Teil des Unternehmenswertes ausmacht, in Mitleidenschaft gezogen wird. Die Kunden würden sich abgeschreckt fühlen, wenn sie erfahren würden, dass es bei der betreffenden Institution überhaupt möglich ist, zu betrügen. Solche Aspekte können sogar zum Entscheid führen, gar nicht erst eine systematische Betrugsprävention und –Erkennung in Erwägung zu ziehen, soweit nicht durch das Gesetz oder durch Vertrag eine solche Verpflichtung besteht.

Motivation für die vorliegende Arbeit

Für die automatisierte Betrugserkennung bieten sich verschiedene Ansätze an, welche im Kapitel 2.1 detailliert beschrieben sind. Der gegebene und an der Universität Zürich entwickelte Algorithmus „*ChainFinder*" nutzt ein Suchverfahren zur Mustererkennung in Transaktionsdaten (ein sog. *Pattern-Matching-Ansatz*). Dabei werden Ketten von Transaktionen innerhalb des Datenbestandes gesucht. Es handelt um eine Methode der Betrugserkennung, welche nur einen Teil der Betrugsdelikte erkennen kann. Er ist nicht fähig, beliebige Muster zu finden, und ausserdem ist davon auszugehen, dass nicht jeder Betrug durch das Finden von vordefinierten Mustern in den angefallenen Daten entdeckt werden kann. Denn dazu müssten die zu suchenden Muster bekannt sein. Und selbst wenn sie es wären, gäbe es wahrscheinlich immer noch Szenarien, in denen der Tatbestand des Betruges erfüllt sein kann, ohne dass eine verwertbare Datenspur entsteht.

Es soll überprüft werden, wie effizient und korrekt der *ChainFinder* seine Arbeit verrichtet. Des Weiteren sollen während dieses Prozesses Möglichkeiten zur Leistungsverbesserung des *ChainFinders* gefunden und implementiert werden. In einem letzten Schritt soll ein *Framework* gestaltet werden, das zur Evaluation von verschiedenen Algorithmen der Betrugserkennung genutzt werden kann. Die Hauptanforderung an dieses ist, einfach benutzbar und anpassbar zu sein und dadurch in der Praxis einen Mehrwert zu bieten.

2 Related Work

Die folgenden Abschnitte geben einige wichtige Ansätze von anderen Autoren wieder, beinhalten im Abschnitt über den *ChainFinder* allerdings auch die Ergebnisse von Analysen des Verfassers.

2.1 Suchverfahren

Zur Unterscheidung von möglicherweise betrügerischen Transaktionen von nicht betrügerischen Transaktionen (Hintergrundrauschen) gibt es verschiedenartige Ansätze, welche in diesem Unterkapitel genauer beleuchtete werden:

Abbildung 1: Verfahren zum Erkennen von potenziell betrügerischen Transaktionen

2.1.1 Plain SQL

Werden simple Strukturen gesucht, dann könnten möglicherweise noch reine SQL-Anfragen an das Data-Warehouse in vernünftiger Zeit durchlaufen. Allerdings dürfte die Effizienz recht schnell fallen, wenn die abgefragten Strukturen komplexer werden (viele Joins nötig). Der Code innerhalb der Query dürfte in einem solchen Fall auch ziemlich redundant sein, was deren Wartbarkeit senkt. Wird die Query dagegen dynamisch aus einem anderen Programm heraus generiert, könnte man eine höhere Flexibilität unter anderem durch die weniger umständliche Wartung erreichen, allerdings mit dem Nachteil einer steigenden Komplexität.

Der unbestrittene Vorteil eines solchen „*Plain-SQL*"-Ansatzes ist die potenziell sehr hohe Geschwindigkeit, welche sich entfalten kann, wenn man mit den Eigenheiten des jeweiligen Datenbankverwaltungssystems vertraut ist. Wenn es möglich sein sollte, das gesamte Evaluationssystem und auch den *ChainFinder* in SQL zu implementieren, fallen viele Herausforderungen weg, welche durch den Medienbruch DB2/Java entstanden sind.

Mehr dazu im Kapitel 3.3 wo im Rahmen der Problembeschreibung dieser spezifische technische Aspekt des dieser Arbeit zu Grunde liegenden Testsystems genauer erläutert wird.

2.1.2 Graph Pattern Matching

Graph-Pattern-Matching ist eine Generalisierung von *String-Pattern-Matching* (Valiente & Martinez, 1997) und lässt sich nach (Gallagher, 2006) formal folgendermassen definieren:

> *Der Daten-Graph ist ein Graph* (V, E) *mit den Knoten* V *und den Kanten* E*, wobei es für jedes Paar* $e \in E$ *mindestens ein Paar* (v_i, v_j) *gibt für das gilt* $(v_i, v_j) \in V$*. Die Knoten sind in unserem Fall nicht typisiert, doch die und die Kanten haben Attribute; Mehrfachkanten sind möglich.*
>
> *Ein Pattern Graph (oder Pattern Query)* $P = (V_p, E_p)$*, welches die strukturellen und semantischen Anforderungen beschreibt, welche ein Subgraph* $S_?$ *erfüllen muss, um den Pattern* P *zu erfüllen.*

Die Aufgabe besteht nun darin, ein Set T bestehend aus Subgraphen von D zu finden, welche zum Pattern P passen. Ein Graph $D' = (V', E')$ ist ein Subgraph von D genau dann, wenn sowohl $V' \subseteq V$, als auch $E' \subseteq E$. P muss ein einzelner, verbundener Graph sein, weswegen $t \in T$ ebenfalls verbunden ist. Ein Graph gilt als verbunden, wenn ein Pfad zwischen jedem Paar seiner Kanten existiert.[10]

Im Jahre 1979 wurde ein Algorithmus vorgestellt, der mit exponentieller Laufzeit entscheiden konnte, ob zwei Graphen *isomorph* sind, was in etwa so viel bedeutet, wie dass sie dieselbe Struktur haben. Dieser wandelte das Problem in einen Suchbaum um und expandierte diesen *naiv*. Daraus ergab sich das bereits angetönte Laufzeitverhalten; der Aufwand für einen jeden Test ist im schlimmsten Falle $O(m^n)$. Bereits in dieser Veröffentlichung (Ullmann, 1976) wurden Optimierungen vorgeschlagen, welche zwar mehr Speicherplatz erfordern, im Gegenzug aber die Laufzeit senken.

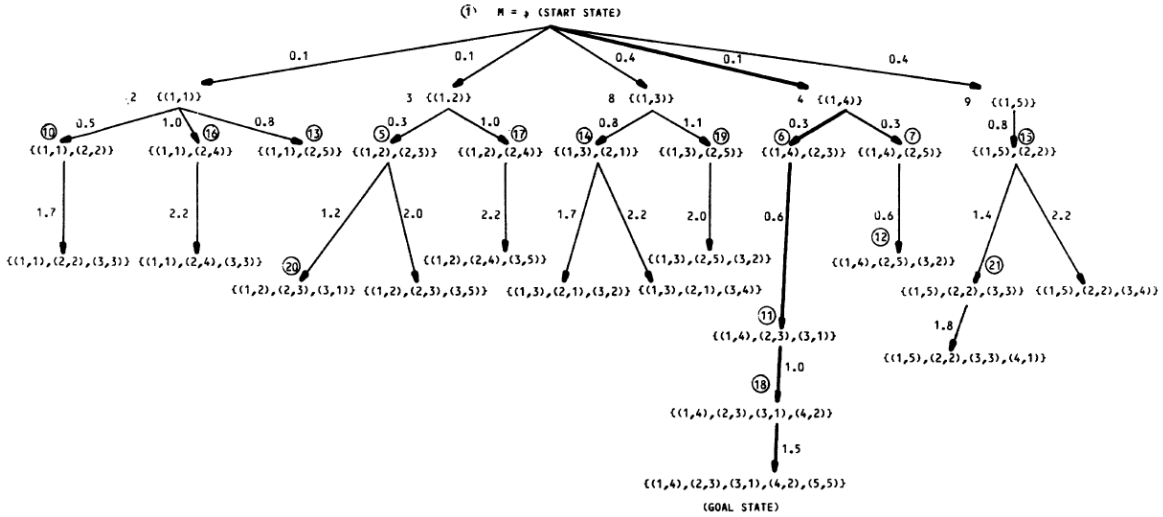


Fig. 7. State-space graph using uniform-cost search algorithm (circled numbers specify node expansion order).

Abbildung 2: Lösung eines Graph-Pattern-Matching Problems durch einen Suchbaum (Tsai & Fu, 1979)

Neuere Veröffentlichungen (Obler, Oning, & An, 1992) legen nahe, dass die Komplexität[11] des Graph-Isomorphismus-Problems eher in P liegt, als in NP, und somit die obere Schranke der Laufzeit nicht exponentiell zur Problemgrösse wächst. In der Praxis bedeutet dies in etwa soviel, als dass Pattern-Matcher skalieren *können,* wofür es durchaus auch schon Beispiele gibt; beispielsweise in (Tong, Faloutsos, Gallagher, & Eliassi-Rad, 2007).

Nicht universelle *Graph-Pattern-Matcher* können im Gegensatz zu universellen *Graph-Pattern-Matcher* entweder nur mit einer bestimmten Klasse von Graphen oder *Pattern-Queries* umgehen. Unter anderem durch eine solche Optimierung auf eine bestimmte Klasse von Problemstellungen sind Matching-Algorithmen möglich, welche Lösungen mit tieferem Aufwand als $O(m^n)$ finden. Eine Liste davon findet sich in (Valiente & Martinez, 1997). Neben der Einschränkung auf spezifische Teilprobleme wäre es andere Möglichkeit, den Suchvorgang durch die Verwendung von Heuristiken informierter zu gestalten und so weniger weite Teile des Suchraumes expandieren zu müssen, bis das Ergebnis feststeht (*pruning*).

[10] Wie bereits erwähnt, stammen die Definitionen aus dem angegebenen Paper. Die Namen der Mengen wurden aber angepasst und der Inhalt dahingehend verändert, dass nun Mehrfachkanten erlaubt sind, was für das Setting der Betrugserkennung im Finanzsektor typisch ist.

[11] Die Komplexitätstheorie in der Informatik beschäftigt mit der Fragestellung zur Lösbarkeit eines Problems, wobei oftmals auch auf die sogenannte „time-/*spacecomplexity*" Bezug genommen werden muss. Dabei geht es darum, wie viele Rechenschritte und wie viel Speicher für das Lösen einer Aufgabe benötigt werden.

Brain Galleger analysiert in seinem Paper *„Matching Structure and Semantics: A Survey on Graph-Based-Pattern-Matching“* (Gallagher, 2006) verschiedene weitere Publikationen und synthetisiert einen universellen Approach. Dieser hat zum Ziel, die Notwendigkeit von direkten Strukturvergleichen (also dem effektiven *Pattern-Matching*) zu minimieren und dadurch den Gesamtrechenaufwand zu senken (Verbesserung der *Time-Complexity*, bei tendenziell erhöhtem Speicherbedarf). Dieser umfasst die folgenden drei Schritte:

- **Datenanalyse und Metadatengenerierung:** Es werden sog. *Graph-Invarianten („graph invariants“)*[12] extrahiert. Bildlich gesprochen wären dies Fingerabdrücke oder Röntgenbildern, welche Charakteristiken des Graphs in anderer Form wiedergeben. Der Sinn dieses Preprocessings ist, die nachfolgenden Schritte im Pattern-Matching-Prozess *informierter* gestalten zu können.
- **Candidate Selection:** Um nicht auf alle Teile des Graphs oder der Subgraphen Strukturvergleiche (*Pattern-Matching*) machen zu müssen, werden anhand der generierten Metadaten die uninteressanteren Kandidaten vom Matching ausgeschlossen.
- **Pattern Matching i.e.S:** In dieser Phase werden die identifizierten Kandidaten durch Matching-Techniken überprüft. Die Algorithmen arbeiten typischerweise nach numerischen Verfahren, namentlich dann, wenn sie nur näherungsweise ihren Zweck erfüllen, vgl. (Tong et al., 2007). Such-basierte Ansätze hingegen liefern die optimale Lösungsmenge, also alle Treffer, vorausgesetzt es ist genügend Zeit vorhanden. Wie effizient das Pattern-Matching im i.e.S selbst ist, sei bei dieser Betrachtung noch dahingestellt.

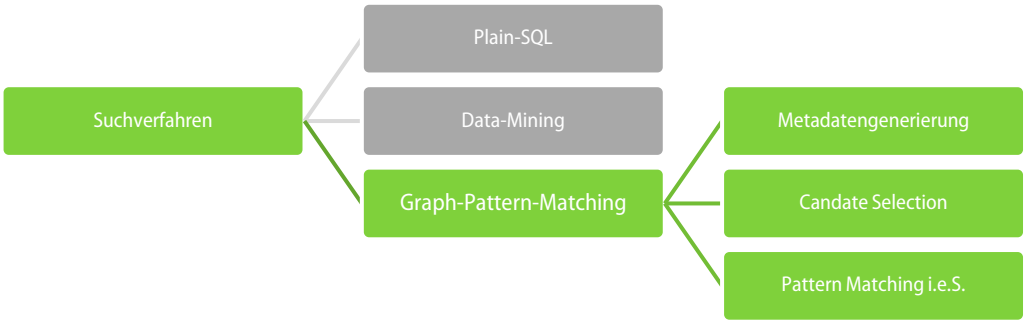


Abbildung 3: die Aufschlüsselung des Graph-Pattern-Matching Prozesses nach (Gallagher, 2006)

2.1.3 Data Mining

Das weite Forschungsfeld des Data Minings hätte durchaus Betrachtung verdient im Kontext der Betrugserkennung, da viele Ansätze auf solchen Methoden basieren. Dazu muss allerdings gesagt werden, dass der ChainFinder ein Pattern-Matching-Algorithmus ist und Betrachtungen des Data-Minings in diesem Kontext lediglich einen Exkurs darstellen.

Wichtige Anwendungsgebiete für Verfahren dieser Kategorie sind die Erstellung von Nutzerprofilen, wie sie beispielsweise im Marketing, der staatlichen Überwachung oder eben bei der Betrugserkennung hilfreich sein können. Die Techniken des Data Mining sind meist statistisch-mathematisch begründet und zeichnen sich in der Regel durch eine gute Skalierbarkeit aus, was sie für die Verarbeitung von grossen Datenmengen interessant macht.

[12] “An invariant is a quantity used to characterize a graph” (Washio & Motoda, 2003) in (Gallagher, 2006).

Ihr Ziel ist die Mustererkennung, sprich die Extraktion von *Information* aus Daten. So lassen sich darin Anomalien, Strukturen und Verhaltensänderungen finden. Werden solche in Finanztransaktionen entdeckt, muss dazu jedoch gesagt werden, dass nicht alles, was andersartig ist, auch betrügerisch sein muss.

Die Data-Mining-Verfahren lassen können weiter in *supervised Data-Mining*[13] oder *unsupervised Data-Mining* unterteilt werden.

2.2 Der gegebene Algorithmus „ChainFinder“

Der gegebene Algorithmus *ChainFinder* findet die längsten Ketten („Chains“) von Transaktionen durch ein spezielles Suchverfahren ausgehend von einer Menge von „verdächtigen“ Konten („Roots“). Jede der angegebenen Roots wird sequenziell abgearbeitet. Es findet keine parallele Suche nach den Ketten statt, wenn mehr als ein Root im Rootset ist. Die Verarbeitung der Daten dauert deshalb entsprechende lange, wenn das Rootset viele Elemente enthält.

Der *ChainFinder* wurde in einer Java-Implementation zur Verfügung gestellt. Durch die Verwendung eines Suchverfahrens sollten alle Lösungen gefunden werden. Ob dies wirklich geschieht, ist zu prüfen. Die konkrete Implementation sieht allerdings gewisse Abbruchbedingungen vor, welche bestimmte Treffer ausschliessen können, obwohl sie eigentlich die Charakteristiken einer Kette haben.

Die Resultate können gespeichert werden oder auch nicht. Es steht die Ausgabe in eine Textdatei oder in eine Datenbank zur Verfügung, wobei erwartet wird, dass die dazu nötigen Tabellen bereits angelegt sind. In seiner ursprünglichen Version wurde der *ChainFinder* durch Konstanten konfiguriert. So musste schon beim Start die Datenbanktabelle angegeben werden, in welcher gesucht werden sollte und diese liess sich während der Ausführung nicht ändern. Mehreren Datenbankstandards waren kompatibel und die Treffermenge konnte über eine andere Datenbankverbindung als diejenige, die ihm den Zugang zu den zu durchsuchenden Transaktionsdaten ermöglichte ausgeben werden. In der folgenden Abbildung ist der Fall dargestellt, in dem diese beiden Verbindungen gleich sind.

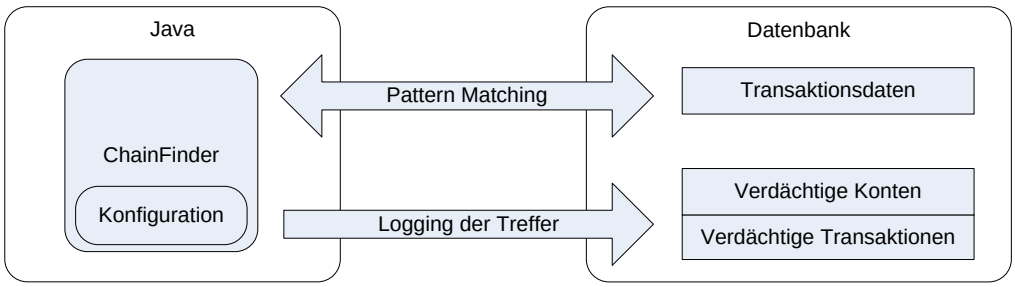


Abbildung 4: Der *ChainFinder* und dessen Umfeld

Beim Start jedes Durchlaufs wird jeweils eine Menge von Konten („Rootset“) angegeben, welches mehrere Elemente enthalten kann, die dann sequenziell nach dem gleichen Verfahren abgearbeitet werden. Die zu suchenden Patterns sind Transaktionsketten von verschiedener Länge, aber mit ähnlichen Beträgen und ähnlichen Ausführungsdaten. Folgende wichtige Parameter können bei einem Durchlauf gesetzt werden:

- **Rootset:** Diese Konten sollen darauf überprüft werden, ob sie Ausgangspunkt von Transaktionsketten sind.

[13] Ein bekannter Algorithmus für diesen Fall ist SUBDUE (Holder, Cook, & Djoko, 1994).

- **Zeitfenster, in Tagen:** Gibt die maximale Datumsabweichung innerhalb der Kette an. Die einzelnen Transaktionen dürfen bei wachsender Position innerhalb der Kette nicht weiter in der Vergangenheit liegen.
- **Unschärfe-Koeffizient[14]:** Bezeichnet die zugelassene Abweichung der Beträge im Vergleich zur vorhergehenden Transaktion.
- **Zyklen zugelassen:** Falls inaktiv, müssen nicht alle Konten innerhalb der Kette unterschiedlich sein.
- **Minimale Kettenlänge:** Es werden nur Ketten gefunden, welche mindestens die angegebene Zahl an Transaktionen umfassen. So verläuft eine Kette mit Länge 2 über 3 Konten.
- **Richtung:** Gibt an, ob die Elemente des Rootsets Ausgangspunkt oder Ende einer Kette sind, oder nur innerhalb einer solchen vorkommen müssen.

2.3 Die Evaluation von Pattern Matchers

In (Gallagher, 2006) wird hervorgehoben, dass der Vergleich von Pattern-Matching-Techniken über verschiedene Domänen hinweg schwierig ist und zudem die Praxisrelevanz der Resultate der Evaluation schwer zu ermessen ist. Der Grund liege darin verschiedene Algorithmen verschiedene Ziele haben und die Datensets, auf denen sie evaluiert werden, stark variieren und meist synthetischer Natur seien. Je nach Forschungsgruppe seien Graphen zum Einsatz gekommen, welche sich in ihrer Grösse und Struktur stark unterscheiden.

Eine wichtige Unterscheidung dabei ist, ob das Matching auf einem einzigen Graphen oder wie im Falle von Transaktionsdaten im Finanzsektor auf einer grossen Menge von autonomen Teilgraphen, also auf einem so genannten *Graph-Transaction-Setting* geschieht.

Graph-Transaction Setting und der ChainFinder[15]: In unserem Setting besteht der Daten-Graph aus vielen autonomen Teilgraphen und ist *sparse,* was bedeutet wie dass die Zahl der effektiv verbundenen Knoten weit unter der maximal möglichen liegt. Einige Definitionen von *sparse* vergleichen nur die effektive *Kantenzahl* mit der maximal möglichen. Dies macht jedoch wenig Sinn, wenn Mehrfachkanten zugelassen sind wie bei den Bankdaten.

Ausserdem unterscheiden sich die eingesetzten Evaluations-Metriken bedeutend bei inexakten und exakten Pattern-Matcher. Letztere sind in der Regel als Suchverfahren implementiert und zeichnen sich dadurch aus, dass sie genau das finden, was sie sollen, wenn ihnen genug Zeit zur Verfügung steht. Erstere sind in der Regel darauf spezialisiert, mit der ihnen zur Verfügung gestellten Zeit möglichst gut umzugehen. Sie arbeiten oft iterativ und liefern in einer ersten Iteration eine Treffermenge, welche nur eine vergleichsweise grobe Ähnlichkeit mit der wirklichen Treffermenge hat. Diese Ähnlichkeit verbessern sie aber dann meist, wenn ihnen zusätzliche Iterationen gestattet werden und somit die verfügbare Rechenzeit für die Lösung des Problems erhöht wird.

Wenn die Ergebnisse nicht exakt sind, sind Masseinheiten zur Beurteilung von deren Qualität wichtig. Zwei davon sind im Kapitel 2.4 kurz eingeführt, nämlich *Precision and Recall* und die *Slot-Error-Rate*. Sie sind im Falle des *ChainFinders* von begrenzter Relevanz, eben genau, weil er zu den exakten Pattern-Matcher gehört. Das heisst zwar nicht, dass er wirklich in jedem Fall die erwarteten Treffer liefert, da es Fehler in der Implementation oder falsch gesetzte Parameter beim ChainFinders oder im System, das ihn evaluiert, geben kann. Aber durch die Arbeit mit ihm ist aufgefallen, dass

[14] Ist in der konkreten Implementation über 2 Parameter definiert (zugelassene Abweichung nach oben und nach unten).
[15] Angelehnt an: http://en.wikipedia.org/wiki/Sparse_graph (März 2009).

es nicht ganz so wichtig ist, wie „falsch“ oder „richtig“ die Ergebnisse sind. Es reichte in der Regel zu wissen, ob sie komplett korrekt waren oder nicht.

Bei diesen Betrachtungen viel vielleicht auf, dass kaum andere Veröffentlichungen zitiert wurden oder nicht wirklich versucht wurde, einen konkreteren Überblick über das Forschungsfeld der Graph-Pattern-Matching-Evaluation zu zeichnen. Der Grund dafür liegt zum Teil darin, dass das Thema Evaluation in meisten Veröffentlichungen eher stiefmütterlich behandelt wird, insbesondere werden die benutzten Methoden oftmals sehr unzureichend beschrieben. Ein noch wichtigerer Grund ist aber, dass sich in (Gallagher, 2006) genau mit dem gleichen Thema befasst, wie dieses Unterkapitel dieser Diplomarbeit, nämlich einem Überblick über das *Related Work* zum Thema „Evaluation von Graph-Based-Pattern-Matcher“. Die hier gemachten Aussagen überschneiden sich zum Teil mit denjenigen aus dem genannten Paper und nicht jede davon ist explizit gekennzeichnet.

In (Luo & Hancock, 2001) wird ein Algorithmus für inexaktes Graph-Pattern-Matching vorgeschlagen, der anhand von Anwendungsbeispielen aus der Computergrafik evaluiert wird. Die folgende Tabelle enthält noch drei weitere Analysen von Evaluations-Teilen aus Veröffentlichungen, welche Graph-Pattern-Matching-Algorithmen vorstellten. Sie zeigt, dass die verwendeten Performance-Kennzahlen zumindest in den ausgewählten 4 Papers unterschiedlich sind im Falle von der Verwendung von exaktem, respektive inexaktem Matching.

Jahr	Matching-Typ	Typisierte Knoten	Atrribuierte Kanten	Mehrfachkanten	Error Rate	Error/Iterations	Runtime, absolut	Runtime/Dataset size	Comp. to other Strategies	Comparision to Variants	Max. Edge Count	Referenz
2001	inexakt	-	X	-	X	X	X	-	X	-	?	(Luo & Hancock, 2001)
2003	inexakt	-	X	-	X	X	X	-	X	-	?	(Park, K. M. Lee, S. U. Lee, & J. H. Lee, 2003)
2008	exakt	X	X	-	-	-	X	X	-	X	2M	(Jiefeng Cheng, J. Yu, Bolin Ding, P. Yu, & Haixun Wang, 2008)
2007	inexakt	X	-	-	X	x	X	X	-	X	2M	(Tong et al., 2007)

2.4 Masseinheiten zur Güte der Treffermenge

Der *ChainFinder* arbeitet im Prinzip als exakter Pattern-Matcher, der genau das findet, was dem Query-Pattern entspricht. Je nach der verwendeten Konfiguration wird jedoch nicht die exakte Treffermenge geliefert, sondern eine davon abweichende. In diesem Moment macht es Sinn, Masseinheiten zur Beurteilung der Qualität einer Treffermenge zu definieren. Eben diese Frage nach der Güte der Resultate bei einer *Recherche* ist in der Domäne des *Information Retrieval* wichtig.

2.4.1 Trefferquote und Genauigkeit

Wird auch *Precision and Recall* genannt und sei hier als erstes Konzept aufgeführt. Die Trefferquote misst die Wahrscheinlichkeit, mit der ein relevantes Dokument gefunden wird. Die Genauigkeit misst die Wahrscheinlichkeit, mit der

ein gefundenes Dokument relevant ist. Bei der Verwendung dieser Masseinheit zur Güte der Treffermenge entsteht sowohl ein Wert für die Trefferquote, als auch für die Genauigkeit. Das folgende Konzept kommt mit einer einzigen Kennzahl aus.

2.4.2 Slot-Error-Rate:

Die *Slot-Error-Rate* (SER) definiert einen „Slot" als eine Art Platzhalter für jeden gültigen Treffer, der in der Resultatsmenge auftauchen müsste (Makhoul, Kubala, Schwartz, & Weischedel, 1999). Wird jeder von diesen durch einen korrekten Treffer gefüllt, beträgt nach unserer eigenen Definition die SER 0%. Gibt es mehr Ergebnisse als Slots, müssen einige davon zwingend falsch sein. Die Slot-Error-Ratio hätte dann einen Wert von mehr als null Prozent.

In unserem Kontext sind jedoch nur die *korrekten Treffer* (C), die *falsch Positiven* (I) und die *falsch Negativen* (D) interessant. Die Formel aus dem Paper hat aber noch einen zusätzlichen Parameter:

$$SER = \frac{S+D+I}{C+S+D}$$ (Original aus dem Paper)

S ist in unserem Kontext nicht relevant und wird deshalb 0 gesetzt. Für eine Erklärung, was dieser Parameter genau tut und in welchem Setting er relevant ist, sei hier noch einmal auf (Makhoul et al., 1999) verwiesen.

$$SER = \frac{D+I}{C+D}$$ (Eigene Abwandlung)

Nehmen wir an, zu einer Anfrage existieren 100 passende Dokumente (sprich „Pattern" in der Datenbasis). Findet der *ChainFinder* nur 90 davon, aber liefert keine inkorrekten Elemente, dann ist die SER 10%

$$SER = \frac{D+I}{C+D} \quad \rightarrow \quad \frac{10+0}{90+10} = 10\%$$

Nehmen wir an, zu einer Anfrage existieren 10 passende Patterns. Findet der *ChainFinder* nur eines davon, aber dafür noch 7 inkorrekte Elemente, dann ist die SER 10%.

$$SER = \frac{D+I}{C+D} \quad \rightarrow \quad \frac{9+7}{1+9} = 160\%$$

Der Vorteil bei der Verwendung der Slot-Error-Ratio zur Beurteilung der Qualität einer Treffermenge liegt darin, dass es reicht, nur einen Wert zu betrachten. Als Nachteil könnte die fehlende Normierung des Wertebereichs angesehen werden.

Das folgende Kapitel 3 beschreibt die Details der Problemstellung und geht unter anderem auf einige Besonderheiten des Testsystems ein.

3 Problembeschreibung

Die Evaluation und Evolution von Algorithmen setzt Masseinheiten zu deren Leistungsfähigkeit (Performance) und der Güte ihrer Ergebnisse (Korrektheit) voraus. In diesem Kapitel werden einige relevante Begriffe und Konzepte definiert respektive synthetisiert.

Der gegebene Algorithmus *ChainFinder* wurde bereits im vorherigen Kapitel vorgestellt. Die hier aufgestellte Problemstellung ist bewusst abstrakt gehalten. Dieses Kapitel basiert zwar auf den Erfahrungen aus dem konkreten Umgang mit dem *ChainFinder*, fixiert sich aber nicht vollends auf diesen.

3.1 Grundlagen und Setting

Der *ChainFinder-Algorithmus* dient wie oben erläutert zur Betrugserkennung und wurde in Zusammenarbeit mit einem grossen schweizerischen Finanzinstitut einwickelt. Er liegt in einer Referenzimplementation in Java vor und er ist mit mehreren Datenbankstandards kompatibel, wobei in unseren Test nur DB2 zum Einsatz kam, welches auch im betreffenden Finanzinstitut eingesetzt wird. Sein Ziel ist die Aufdeckung von Mustern (Ketten) in den Transaktionsdaten von Finanzunternehmen, die auf einen Betrugsfall durch die eigenen Angestellten hinweisen. Dieser Tatbestand wird auch als „*Internal Fraud*“ bezeichnet. Eine der grössten Herausforderungen ist es, eine angemessene Skalierbarkeit zu erreichen. Optimierungen in diesem Bereich sind besonders essenziell, da im Data-Warehouse des betreffenden Finanzinstituts eine sehr grosse Anzahl von Transaktionen vorgehalten wird und diese Zahl täglich um etwa eine Million steigt.

Betrachten wir nun die angesprochenen Transaktionsdaten etwas genauer: Jede Transaktion enthält unter anderem Angaben zu *Absender* und *Empfänger*, zum *Betrag*, zum *Ausführungsdatum* und zur *Kennung des Sachbearbeiters*, der sie erfasst hat. Bezeichnen wir jede Transaktion als ein Element der Menge *D* (der Buchstabe ist abgeleitet vom Begriff „Datenbasis“). Im Live-Betrieb stellt sich für die Evaluation des Algorithmus das Problem, dass die Input-Daten für einen Suchlauf wohl immer eine Teilmenge oder die Gesamtheit der Echt-Daten (Realdaten) sind. Es ist im Vornherein nicht bekannt, welche Transaktionen mit Betrugsfällen in Verbindung stehen. Deshalb ist es sinnvoll, zur Evaluation und Evolution von Algorithmen in unterschiedlichen Szenarien mit unterschiedlichen Arten von Datenbeständen zu arbeiten (z. B. rein synthetische Daten oder simulierte Realdaten, wo künstliche Transaktionsmuster, die auf Betrug hindeuten (*Patterns*) eingeschleust werden). So können gewisse Eigenschaften der Datenbasis genauer kontrolliert werden sowie die Ergebnisse des Suchlaufs besser evaluiert werden.

Formal kann eine Menge von Transaktionsdaten als gerichteter Graph, welcher sowohl Mehrfachkanten wie auch Zyklen zulässt, betrachtet werden. Die *Knoten* (oder *Nodes*) repräsentieren dabei Konten (*Accounts*), die *Kanten* stehen jeweils für eine Transaktion. Da die Transaktion selbst Attribute wie den Betrag oder das Datum enthält, zeichnet sich der Graph durch *attribuierte* Kanten aus. Ein Konto existiert genau dann, wenn es an mindestens einer Transaktion beteiligt ist. Diese Annahme a) begründet sich mit einfacherer Datenhandhabung in einem relationalen Datenbanksystem und der fehlenden Relevanz von „toten Konten“ beim *ChainFinder*. So liesse sich im Fall a) der ganze Graph einfach in einer einzigen Tabelle speichern. Würden Konten gemäss Annahme b) ohne Bewegungen auch erfasst, müsste diese Informationen in einer separaten Tabelle gespeichert werden, was die Abfrage aufwändiger machen würde. Da die Problemstellung lautet, *Transaktions*muster zu suchen, die betrügerisches Verhalten implizieren könnten, sind Konten ohne Bewegungen nicht von Belang. Dies ist daher auch sinnvoll, da solche Konten per Definition nicht Bestandteil einer Transaktion sein können, und somit sinnvollerweise aus der Problemstellung aus-

gegrenzt wurden. Ein im Beobachtungsraum totes Konto ist von Betrugshandlungen ohnehin nicht betroffen, da innerhalb dieses bestimmten Zeitraumes keine Gelder zu und weggeflossen sind.

Unterscheiden wir nun zwischen den beiden Fällen, wie eine Kante gerichtet sein kann. Handelt es sich um eine abgehende Kante, befindet sich das betreffende Konto in der Rolle des *Absenders* („*Originator*"), im umgekehrten Falle in derjenigen des *Begünstigten* („*Beneficiary*"). Jeder verbundene Subgraph $S_?$ von D kann „betrügerischen" Charakter haben, oder auch nicht. Diese Einteilung geschieht aufgrund von gewissen Kriterien K. Genügt $S_?$ den Matching-Kriterien, qualifiziert ihn das als *Pattern*. Die Menge aller *potenziellen Patterns* Q ist unabhängig von der Datenbasis D.

Wird ein *Pattern* bei einem Suchlauf gefunden, sprechen wir von einem *Match* (oder *Treffer*), wobei die Vereinigung aller Treffer die *Treffermenge* T repräsentiert.

3.2 Strategie vs. Implementation

Eine *Strategie* A (auch *Algorithmus*) ist eine „genau definierte Handlungsvorschrift zum Lösen eines Problems[16]". Wird diese in Programmcode umgesetzt (implementiert), sprechen wir von der Implementation von A oder kurz I_A. Der Prozess der Umsetzung bezeichnen wird formal als $A \rightarrow I_A$.

Aus diesen Betrachtungen lassen sich einige Fragen ableiten, welche zur Evaluation des Gesamtkonzepts auf verschiedenen Ebenen helfen:

1. Ist die Menge der potenziellen Treffer Q überhaupt geeignet, real begangene Betrugshandlungen zu indizieren? Diese Frage ist jedoch nicht Gegenstand der vorliegenden Arbeit.
2. Ist die Strategie A überhaupt fähig, alle Patterns zu finden? Wird es Duplikate oder falschpositive Elemente in der Treffermenge geben?
3. Ist ein Fehler bei der Implementierung $A \rightarrow I_A$ gemacht worden? Dies würde sich in einem Programmabbruch äussern oder zu einer Treffermenge führen, welche sich nicht mit derjenigen deckt, die von A zu erwarten wäre.
4. Des weiteren könnten auch Informationen über die Performance Rückschlüsse auf Fehler in der Implementation I_A zulassen .

A (Strategie)	Handlungsvorschrift zum Lösen eines Problems, „Algorithmus"
I_A (Implementation)	Konkrete Implementation einer Strategie

Die universelle Betrachtung aller Algorithmen und deren Evaluation und Evolution sind nicht Schwerpunkt dieser Arbeit, da sie in unserem Setting nur von begrenzter Relevanz sind. Trotzdem sei folgender Exkurs dargestellt, um eine weiterführende Perspektive auf das Thema zu ermöglichen:

Ein *Algorithmus* (oder *„Strategie"*) ist ein Element A der Menge aller Suchalgorithmen $\mathcal{A}_s$. Er kann eine oder mehrere konkrete Implementationen I_A haben. I_A ist demnach die Repräsentation einer gewissen Strategie A. Jeder Suchlauf (= *Inspektion*) basiert auf für diesen Suchlauf speziell definierten Parametern P von I_A und dem Datenset D. Die Ergebnismenge T einer Inspektion ist abhängig von der Korrektheit der Implementation $A \rightarrow I_A$ und von den gewählten Parameter. Diese Mengen, welche mit einem Grossbuchstaben benannt sind, repräsentieren alle gültigen

[16] Definition angelehnt an http://en.wikipedia.org/wiki/Algorithm (letzter Zugriff im März 09).

Daten ($\mathcal{D}$), Implementationen ($\mathcal{I}$), Strategien ($\mathcal{A}$), respektive Parameter ($\mathcal{P}$). $\mathcal{P}$ ist abhängig von I_A. $\mathcal{I}$ ist anhängig von A. Eine Strategie könnte also verschiedene konkrete Implementationen haben, wobei es interessant ist deren Korrektheit und Performance miteinander zu vergleichen und ihr Verhalten unter gewissen Parametern und auf verschiedenen Datensets zu beobachten. Es wird davon ausgegangen, dass P während eines Durchlaufs konstant bleibt.

Q_{real} sei die potenzielle Treffermenge abhängig von der Implementation I_A, und deren gesetzten Parameter P. Auf der anderen Seite ist Q_{ideal} nur abhängig von den Kriterien K.

Falls kein unerwarteter Programmabbruch auftritt, liefert I_A die Treffermenge T abhängig von P und D zurück. Folgende Konsistenzbedingungen sind in diesem Fall zu prüfen:

- Entspricht Q_{ideal} Q_{real}?
  - Das kann indirekt via Analyse der Treffermenge T überprüft werden, in dem künstlich Patterns (also Elemente aus Q_{ideal}) in eine Datenbasis D eingeschleust werden.
  - Für die weitere Analyse von Bedeutung ist es festzustellen, wie die beiden Mengen zueinander in Relation stehen (sprich wo sind Überlagerungen und wo nicht).
- Repräsentiert die Implementation I_A die Strategie A?
  - Dies kann durch ein Code-Review beantwortet werden (was nicht Thema dieser Arbeit ist).
- Sind alle Elemente der Treffermenge ($t_i \in T$) auch Element von Q_{real}?
  - Kann über mehrere Inspektionen hinweg überprüft werden, mit P und D konstant.
  - Ist die Zuordnung bei einer bestimmten Inspektion nicht mehr gegeben, liegt wahrscheinlich ein Programmabbruch vor, da I_A konstant bleibt.
- Sind alle $t_i \in T$ auch $\in Q_{ideal}$?
  - Falsche Konfiguration möglich (Lösung: andere Parameter prüfen).
  - Fehlerhafte Implementation (Parameter konstant halten und D ändern).
  - Fehler oder Instabilität im (verteilten) System.

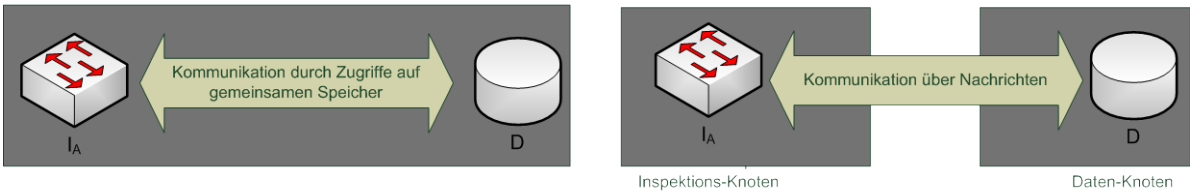
Diese Überlegungen betreffen die Korrektheit der Treffermenge T und der Implementation I_A. Eine feingranulare Performance-Evaluation und das Auffinden von Optimierungsmöglichkeiten, sprich die Evolution von I_A, sind ohne der Architektur des Gesamtsystems Beachtung zu schenken schwierig. Das folgende Unterkapitel betrachtet ein verteiltes System im Kontext der Evaluation und Evolution des *ChainFinders* oder generell von Algorithmen zur Betrugserkennung.

3.3 Evaluation im verteilten System

Wir gehen davon aus, dass die Implementation I_A des zu evaluierenden Algorithmus A auf Daten zugreift, welche nicht bereits beim Start vollständig im Speicher liegen, sondern von einem anderen Prozess bezogen werden, der kein Zugriff auf gemeinsame Speicherbereiche hat. Explizit ausgeschlossen ist also der Fall, dass die Datenbasis bereits „hardcoded" in der I_A vorliegen, da dies im Kontext der Betrugserkennung nicht besonders realistisch ist. Der *ChainFinder* bezieht seine Daten über *JDBC*, die *Treffermenge* kann entweder in einer Textdatei oder in einer Datenbank geschehen. Es wird also mindestens einmal eine Prozessgrenze überschritten (beim Lesen der Daten). In diesem Falle kann man schon von einem *verteilten System* sprechen. Möchte man nun seine Performance verbessern, müsste die Interna des Datenbankverwaltungssystems in die Betrachtungen miteinbezogen werden. Wenn dieses nicht auf

derselben Maschine läuft, wie der Algorithmus, dann müsste der Netzwerktopologie auch noch Beachtung geschenkt werden.

Abbildung 5: Unterscheidung "verteiltes System" und "einzelner Prozess"



Szenario 1: Nur ein Prozess

Szenario 2: Ein verteiltes System

Wir gehen davon aus, dass I_A nur einen Prozess umfasst und bezeichnen diesen Teil des verteilten Systems als *Inspektions-Knoten*. Der Prozess oder die Prozesse, welche den Zugriff auf die Daten ermöglichen und sich auch um deren physische Speicherung kümmern, nennen wir *Daten-Knoten*. Die Begriffe wären etwas irreführend, falls die Datenbank und I_A auf der gleichen physischen Maschine laufen würden (von verschiedenen „Knoten" zu sprechen impliziert üblicherweise getrennte Hardware). Wir gehen davon aus, dass in der Praxis getrennte Hardware eingesetzt wird, wie auch auf unserem Testsystem. Erfolgt die Evaluation auf einem verteilten System, stellen sich vielfältige neue Herausforderungen, zum Beispiel:

- **Syntax und Semantik**: Die Kenntnis mehrerer Programmiersprachen wird nötig, was das Verständnis ihrer Grundkonzepte voraussetzt. Das wird umso komplexer, je enger die Verzahnung sein muss, sprich je breiter die Interfaces sein *müssen*.
- **Vorhersehbarkeit**: Bei Performance-Messungen gibt es eine grössere Unsicherheit (Einflüsse durch Prozessorlast von unbekannter Quelle innerhalb der Systeme oder Veränderungen der Netzwerkleistung). Für eine akkurate Evaluation der Performance sind diese Faktoren einzubeziehen, beziehungsweise zu kompensieren.
- **Technische Aspekte**: Die Administration wird komplexer, da die Performance des Gesamtsystems von bedeutend mehr Parametern abhängt, als wenn nur ein Rechner involviert wäre. Es müssen daher mehr als ein Betriebssystem, mindestens eine Datenbank und Entwicklungsumgebungen für mehrere Programmiersprachen verwaltet werden.

Daten-Knoten	Prozesse zur Datenverwaltung und deren Ressourcen.
Inspektions-Knoten	Der Prozess, in welchem der Algorithmus läuft.

Das Problem der „kleinen" Queries: In einem verteilten System nimmt die reine Anzahl der Queries, die abgesetzt werden, unter Umständen eine wichtige Rolle ein. Durch den nötigen Nachrichtentausch entsteht unter anderem durch die längeren Signalwege und Kapselung des Daten- und Kontrollflusses in andere Protokolle Overhead. Auf unseren Testrechnern ist diese Kosten-Konstante durchschnittlich im Bereich von etwa zwei bis drei Millisekunden (pro Query) anzusiedeln[17]. Zur Ermittlung dieser Werte wurden Anfragen verwendet, die nur einen skalaren Wert

[17] Somit würde sich für eine Million Anfragen nur schon Overhead in der Grössenordnung einer halben Stunde anfallen.

zurückgaben und sich bei der Auswahl des Datensatzes auf ein einziges Feld bezogen, dass alleiniger Primärschlüssel ist. Die Verbindung zum Datenbank-Server erfolgte über eine lokale Netzwerkverbindung (LAN).

3.4 Evaluation von Pattern-Matcher

Ein System zur Evaluation einer I_A kann diese auf Korrektheit und/oder Performance prüfen und muss dem Umstand Rechnung tragen, dass die I_A über mehrere Knoten verteilt ist. Nennen wir ein solches Evaluationssystem E, respektive I_E, wenn explizit seine Implementation gemeint ist. Wenn die I_A ein verteiltes System ist, wird wahrscheinlich auch eine I_E nötig sein, die sich über alle Knoten der I_A erstreckt.

Für die Überprüfung der Korrektheit einer I_A betrachten wir die Treffermenge T, welche bei der Inspektion einer gewissen Datenbasis D unter gewissen Parametern P resultiert. Die Konfigurationsparameter P teilen sich in die Gruppen $P_{restrict} \cup P_{queries} \cup P_{context} \cong P$ auf. Diese Aufteilung basiert auf den Erfahrungen mit dem *ChainFinder*, lassen sich aber evtl. auch auf andere Algorithmen übertragen.

- $P_{queries}$ umfasst alle diejenigen Parameter, welche sich konzeptionell eher der Beschreibung der zu suchenden Muster zuordnen lassen.
- $P_{restrict}$ enthält Einstellungen, welche sich zwar potenziell auf die Treffermenge auswirken, aber konzeptionell wenig mit Beschreibung des zu suchenden Musters zu tun haben.
- $P_{context}$ enthält Einstellungen, welche primär die Schnittstellen zu anderen Programmteilen regelt, speziell der Datenbank. Darunter sind Angaben zur Datenstruktur und –Semantik zu verstehen.[18] Ein Teil dieser Information müsste in irgendeiner Form und Art der I_E übergeben werden muss, damit sie ebenfalls Operationen auf der Datenbasis vornehmen kann.

Es gibt auch noch Parameter, die sich nicht auf die Treffermenge auswirken und nichts mit dem Kontext der Datenbankanbindung zu tun haben. Diese werden hier vernachlässigt, weshalb die Formel nicht von exakter Gleichheit der vereinigten Teilmengen und P spricht. $P_{queries}$ und $P_{restrict}$ haben keinen direkten Einfluss auf den Evaluationsvorgang.

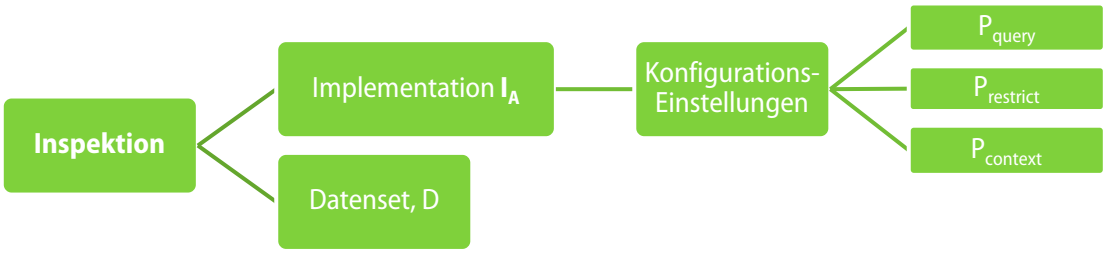


Abbildung 6: Ein abstrakter Evaluator

Diese Betrachtungen gewinnen im Kontext des *ChainFinders* und dessen Evaluation an Bedeutung: So hat er einige Parameter, welche ihn vor zu komplexen Operationen bewahren sollen. Sie sollen verhindern, dass es zu einem zu hohen Speicherverbrauch kommt oder dass zu viele CPU-Zyklen gebraucht werden. Auch eine zu grosse Treffermenge von Ketten, die ihren Ursprung bei ein und demselben Konto haben, soll verhindert werden. Der Grund dafür liegt darin, dass es bei realen Daten einzelne Konten gibt, von denen um Grössenordnungen mehr Transaktionen ausgehen

[18] Ein konkretes Bespiel dafür wären Vorlagen für daraus resultierende SQL-Queries.

als beim Durchschnitt. Von solchen Konten erscheinen in der Treffermenge proportional mehr falsch-positive Resultate, die bei Nachforschung durch menschliche Betrugsexperten vermeidbare Kosten verursachen.[19] Alle diese Einstellungsmöglichkeiten gehören gemäss der vorherigen Definition in die Menge $P_{restrict}$.

Im Vergleich zu anderen Ansätzen ist der *ChainFinder* kein universeller Pattern-Matcher (Siehe oben, Kapitel 2.1.2), das heisst, er kann nicht mit beliebigen *Pattern-Queries* umgehen, sondern ist auf das Finden von Transaktionsketten mit hoher Effizienz optimiert. Es handelt sich um einen Pattern-Matcher, der grundsätzlich exakt arbeitet. Das heisst die Muster, welche den Pattern-Queries genügen, werden auch gefunden. Diese Exaktheit ist jedoch abhängig von den gesetzten Werten in $P_{restrict}$.

Der *ChainFinder* kann Transaktionsketten von beliebiger Länge finden, wobei jedoch solche, die sich über mehr als elf Transaktionen erstrecken, nicht wirklich relevant sind, da sie kaum vorkommen.

Der *ChainFinder* geht von einem Rootset aus, also einer Menge von Konten, welche je nach Konfiguration entweder der Anfang oder das Ende einer Kette in der Treffermenge sind. Die Betrachtungen zur Korrektheit sind relativ simpel, wenn man nur einen Root hat. Dann findet nur die längsten Ketten und keine Duplikate, jedoch innerhalb der Grenzen von $P_{restrict}$ vielleicht nicht alle davon.

Um beim *ChainFinder* die Korrektheit der Ergebnisse zu überprüfen, müsste es eine Funktionalität geben, die ein Datenset generiert oder lädt und dieses dann eventuell noch präpariert. Denn die Menge und Charakteristiken der vorkommenden Patterns sollten definierbare Eigenschaften haben. Diese Präparation wird im Folgenden auch Injektion genannt und ist Bestandteil der I_E, welcher diese Aufgabe zukommt (‚auch „*Injektor*" genannt).

3.5 ChainFinder und Performance

Die Programmlogik, welche die Evaluation durchführt („*Evaluator*", I_E), kann entweder nur in der Daten-Plattform ablaufen oder beide Plattformen benutzen. Beim *ChainFinder* erstreckt sich die I_A über beide Plattformen hinweg, zumindest bei der Suchfunktionalität.

Interessant wäre es zu sehen, wie viel Zeit für welche Teilaufgaben benötigt wird. Rein intuitiv stellt sich beispielsweise schnell die Frage, wie lange die Datenbank-Zugriffe dauern und wie viele einzelne Queries abgesetzt werden.

Wie lange ein Lauf dauert ist abhängig von den Charakteristiken der Datenbasis D und einigen Parametern aus $P_{restrict}$. <Tabelle 2: Performance-Kennzahlen und deren Relevanz> betrachtet unter anderem die drei Stufen „*Metadatengenerierung*", „*Candidate Selection*" und „*Pattern-Matching*" (Gallagher, 2006) im Kontext der Performance-Evaluation und einer möglichen Evolution des *ChainFinders*.

[19] Speziell betrachtet seien Parameter `CLONE_MAX`, `MAX_CLONES_TRACK` und `MAX_ROWS_FETCHED`.

Performance-Kennzahl	Relevanz
Metadatengenerierung (Laufzeit)	***Abstrakt:*** *Handelt sich dabei um Indizes auf Datenbanktabellen, ist die Evaluation nicht trivial. Es reicht nicht aus, einfach nur die Geschwindigkeitsgewinne zu betrachten, sondern der Verwaltungs-Overhead müsste noch abgezogen werden. Eine genauere Messung wäre sicher dann möglich, wenn die Metadatengenerierung durch I_A selbst geschieht.*
	ChainFinder: *Es werden keine persistenten Metadaten generiert. Die intern generierten Partikel gehören eher zum Matching-Prozess selbst. Es gibt aber sehr wohl Indizes in der Datenbank. Daraus folgt, dass es interessant wäre, Optimierungen an der Datenhaltung vorzunehmen.*
Candidate Selection (Laufzeit / eventuell Anzahl abgesetzte Queries)	***Abstrakt:*** *Betrachtet man die Laufzeiten mit aktivierten und deaktivierten Optimierungen bei der Candidate Selection, lassen sich Performance-Verbesserungen belegen.* *Der **ChainFinder** selektiert in der gegebenen Version selbst keine Kandidaten, sondern übernimmt einfach das angegebene Rootset.*
Pattern-Matching	*Es wäre sicher interessant zu sehen, wie viel Zeit die Datenbankzugriffe in Anspruch nehmen und wie viele einzelne Queries abgesetzt werden. Die benötigte Zeit für interne Berechnungen (sprich die Ausführung von Java Code im Falle des ChainFinders) wäre ebenfalls interessant. Es muss dann aber sichergestellt werden, dass der Aufwand für die Protokollierung sauber abgezählt wird.*
Logging (Laufzeit / Anzahl Queries)	*Die Zeit und gegebenenfalls die Anzahl von Queries, welche für die Protokollierung der Resultate anfallen, werden geloggt. Sollte die Korrektheit der Treffermenge geprüft werden, ist das Logging genau dann optimal, wenn genau die richtigen Resultate in der Treffermenge vorhanden sind.*
Total Runtime (Laufzeit je nach Eigenschaften von $\boldsymbol{D}$ und $\boldsymbol{P}$)	*Sagt für sich selbst wenig aus. Macht man sie aber abhängig von Eigenschaften des Datensets und verändert diese stufenweise, lassen sich Rückschlüsse darauf machen, wie gut eine I_A skaliert betreffend dieser Eigenschaften.*

Tabelle 2: Performance-Kennzahlen und deren Relevanz.

3.6 Implikationen der Evaluation

Die Evaluation passiert über einen oder mehrere Läufe mit den gleichen oder unterschiedlichen Datensets ($D \in \mathcal{D}$) und Konfigurationseinstellungen $P \in \mathcal{P}(I_A)$, wobei ein Durchlauf hier auch als „Inspektion“ bezeichnet wird. Eine Abfolge von Inspektionen sei ein „Szenario“. Das Datenset wird dabei typischerweise generiert oder aus bestehender Quelle geladen und meist vor dem Durchlauf noch präpariert.

Ablauf: In der ersten Phase werden die Daten vorbereitet und der Algorithmus wird gestartet. Während seines Laufs muss er je nach Anforderungen an die Performance-Evaluation selbst Profiling-Informationen erheben und in geeigneter Form abspeichern. In der letzen Phase werden die Resultate geprüft.

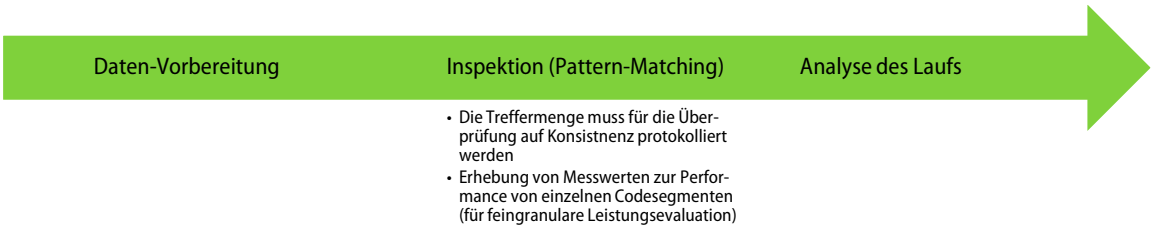


Tabelle 3: Zeitlicher Ablauf eines einzelnen kompletten Evaluations-Durchlaufs.

Es bestehlt also eine Abhängigkeit zwischen der I_E und D, wobei sich die I_E wiederum auf Parametern aus $P_{context}$ beziehen muss und deshalb ebenfalls abhängig ist von Teilen der I_A. Ein universeller kontextunabhängiger Evaluator kann deswegen schon aus Prinzip nicht existieren. Es sind nun die Fragen zu klären, ob es eine Architektur gibt, die geschickt mit diesen Abhängigkeiten umgeht und nicht mehr Komplexität schafft, als sie verbirgt. Zugleich sollte sie nicht überangepasst an eine gewisse I_A sein. Die Schnittstelle zwischen I_E und I_A sollte sowohl universell, als auch schmal sein.

Die Parameter können dabei alle Teile des Evaluationssystems betreffen, also Einfluss auf die Generatoren, Injektoren oder das Verhalten des Algorithmus haben. Dies wiederum bedeutet, dass die im Szenario festgesetzten Parameter teilweise sowohl dem Algorithmus als auch dem Evaluationssystem bekannt sein sollen. Etwas einfacher ausgedrückt läuft dies darauf hinaus, dass zuerst der *ChainFinder* evaluiert wird und danach ein Framework erstellt wird, welches mit verschiedenen Algorithmen umgehen kann. Diese kleine Vorwegnahme der Lösung macht die vorangegangenen eher abstrakten Ausführungen hoffentlich etwas verständlicher.

In den vorhergehenden Abschnitten wurde gezeigt, welche Informationen für die Evaluation der Performance und Korrektheit nötig sind, und in welchen Teilen unseres konzeptionellen Evaluationssystems diese vorhanden sind. Es entstehen also Abhängigkeiten und weitere Implikationen. Wie diese genau im Hinblick auf Instruktions- und Datenfluss-, respektive der nötigen Source-Code-Anpassungen in I_A und nachrangig I_E aussehen zeigt folgende Tabelle im Detail auf.

Funktion	Einfluss auf Performance des Laufs	Informationstransfer zwischen I_A und I_E nötig	I_A –Anpassungen nötig (source code)
Generierung	Nein	**Ja**	Nicht zwingend
Injektion	Nein	**Ja**	Nicht zwingend
Konsistenzprüfungen	Nein	**Ja**	Nicht zwingend
Performancemessungen	**Ja**	Nein	**Ja**

Tabelle 4: Nötige Anpassungen der I_A im Kontext der Evaluation.

3.7 Weitere Betrachtungen

Imperatives SQL, Stored Procedures: SQL ist in seiner ursprünglichen Form eine deklarative Programmiersprache, was in etwa so viel bedeutet, als dass Charakteristiken der Lösung beschrieben werden. Diese Charakteristiken werden in *SQL-Statements* umgesetzt, welche jedoch keine Aussage zum einzuschlagenden Weg geben. Intern im Datenbank-

verwaltungssystem findet dann gezwungenermassen eine Umsetzung im imperativen „Schritt-für-Schritt"-Ansatz statt.

Die Frage ist nun, wie nahe der generierte Algorithmus an der optimalen Problemlösungsstrategie ist. Die praktische Umsetzung dieses Ansatzes und einen Vergleich des *ChainFinders* und eines SQL-Derivates („*Pure-SQL-ChainFinder*") sei im Folgenden erläutert.

SQL in seiner ursprünglichen Form hat wegen seiner oben erwähnten deklarativen Natur beträchtliche Limitierungen, wenn es darum geht, komplexere Logik in einer Abfrage (*Query*) auszudrücken. Es existieren jedoch Erweiterungen des SQL-Standards, welche die Verwendung von imperativen Konstrukten wie Verzweigungen und Schleifen zulassen (sog. *prozedurales SLQ*, auch PL/SQL). Ist also eine Teilaufgabe innerhalb der I_E sehr datenbanklastig und lässt sich diese gut isolieren, bietet sich die Auslagerung in eine *Stored Procedure* an. Man muss jedoch mit Einbussen in der Flexibilität rechnen und diverse Eigenheiten des Datenbanksystems kennen.

So war es auf unserem Testsystem beispielsweise möglich Tabelle mit einer Million Transaktionen innerhalb von nur einer Minute zu generieren. Dieser Performance-Gewinn lässt sich dadurch erklären, dass die *Stored-Procedures* auf dieselben Speicherbereiche zugreifen können wie der Datenbank-Kernel. Der genannte Overhead für die Kapselung und die Kommunikation anfällt, somit auch die fixen Kosten pro Anfrage. Für Aufgaben mit komplexer Logik oder grosser Abhängigkeit von externen Bibliotheken oder Prozessen ist prozedurales SQL eher ungeeignet (da es Wiederverwendung von Code-Fragmenten nicht gut unterstützt, was die Redundanz erhöht und damit die Wartbarkeit senkt).

3.7.1 Hypothese: Pure SQL-Evaluationssystem

Von der Form her bietet sich eine *Stored Procedure* oder auch nur eine einzelne *UPDATE-Anweisung* an, aber auch ein *Trigger*. Bei den ersten beiden Optionen muss der Algorithmus selbst angepasst werden, damit er die entsprechende Procedure aufruft oder die betreffende Query nach einem vollendeten Durchlauf ausführt. Dadurch würde die I_E nur fast ausschliesslich in der Datenbank ablaufen.

Der innere Zustand des Algorithmus bleibt einem *puren SQL-Evaluator* verborgen, da er die Systemgrenze zur Host-Plattform nicht überschreitet (er „sieht" nur die Daten und kann eventuell noch auf UPDATES aktiv reagieren). Möchte man nun trotzdem Performance-Messungen vornehmen, gibt es eine explizite und eine implizite Methode dies zu tun. Erstere geht davon aus, dass die I_A so erweitert wird, dass sie die relevanten Messwerte selbst erfasst und diese der Datenbank *explizit* übergibt. Bei der zweiten Methode gibt es keine Veränderungen am Algorithmus, sie basiert rein auf der Analyse des Outputs, welcher sowieso in der Datenbank landet. Dies müsste aktiv über *Trigger* oder nachträglich über sowieso vorhandene Zeitstempel geschehen.

Eine weitere Methode wäre die Analyse der Aktivitätsprotokolle der Datenbank. Es ist aber nicht davon auszugehen, dass diese Informationen in der Praxis einfach zugänglich sind. Und ausserdem würde dann schon wieder ein neuer Medienbruch eingeführt, da diese Informationen kaum mit den gleichen Mitteln abgerufen und verarbeitet werden können wie Datenbankinhalte.

3.7.2 Hypothese: Evaluationssystem fast nur in Java

Es wäre recht einfach gewesen, fast die ganze Logik des Evaluationssystems in Java zu implementieren und dann einfach eine relativ grosse Menge von Queries mit tiefer Komplexität an die Datenbank zu senden. Wenn aber eine grosse Menge Daten über einzelne Queries generiert werden sollen, ist das teuer im Setting unseres verteilten

Systems, sprich vor allem die Vorbereitung der Datenbasis hätte eine schlechte Performance gehabt. Die Portierbarkeit wäre aber wahrscheinlich höher gewesen, weil die einzelnen Queries dann nur sehr einfache Konstrukte des SQL-Sprachschatzes nutzen würden, welche wahrscheinlich sogar über verschiedene DBMS' hinweg in ähnlicher Weise unterstützt werden. Bei OLAP-Funktionen wie *row_number()* ist das etwas anders. Ausserdem wäre es dann relativ einfach, wenn der Code durch verschiedene Hände gehen würde, da die Anforderungen an das Fachwissen über eine bestimmte Datenbanktechnologie sich noch im Rahmen halten würden. Dieser Ansatz hat seine Vorteile und seine Nachteile.

3.8 Zusammenfassung

Aus den aufgezeigten Überlegungen stellen sich nun einige Fragen:

- **ChainFinder-Evaluation**: Wie effizient und korrekt arbeitet der gegebene *ChainFinder-Algorithmus* im Moment in verschiedenen Szenarien? Nach welchen Kriterien ist dies zu beurteilen? Wohin und in welcher Form werden die Evaluationsdaten gespeichert? Mit welchen Techniken soll das Evaluationssystem umgesetzt werden? Über welche Bereiche des verteilten Systems soll es sich erstrecken?
- **ChainFinder-Evolution**: Wie kann der *ChainFinder* gesamthaft schneller gemacht werden?
- **Der Übergang zu einem Evaluations-Framework**: Wie genau werden die obigen Analysen mit mehreren Algorithmen und Szenarien durchgeführt? Wo wäre ein guter Kompromiss zwischen Einfachheit und Flexibilität anzusiedeln und was wäre eine geschickte Implementation dafür? Wie streng soll das Framework seine Prozesse erzwingen? Soll es bewusst Möglichkeiten geben, mit den vorgegebenen Konventionen zu brechen? Wie lässt sich eine Flexibilität in Bezug auf den zu evaluierenden Algorithmus erreichen?

4 Lösungsansatz

In diesem Kapitel wird die Entwicklung des Lösungsansatzes (Framework) über verschiedene Stufen hinweg dargestellt. Nicht jeder einzelne Irrweg wird im Detail besprochen. Es folgt ein kurzer Überblick zu den einzelnen Unterkapiteln, um dem Leser eine bessere Übersicht zu ermöglichen.

- Analysen und erste Tests des *ChainFinders*.
- Ein Evaluator in purem SQL implementiert, der vor allem die Datengenerierung und die Injektion von Patterns beherrschte, aber die Ergebnisse und die Performance noch nicht so genau prüfte.
- Der Sprung zu einer Template-Engine. Die genannten Nachteile liessen sich nicht so leicht nur in DB2 ausbügeln, darum generierte nun Java die nötigen SQL-Anweisungen.
- Der Evaluator bekam ein simples Frontend, dieses war gegen Ende recht interaktiv und auf Usability bedacht, was aber später wieder etwas zurückgenommen wurde.
- Übergang zu einer Workbench (Framework), welche potenziell mehrere Algorithmen in mehreren Szenarien testen kann.

4.1 Ausgangslage – nur der ChainFinder

Bevor mit der eigentlichen Evaluation und Evolution des *ChainFinders* begonnen werden konnte, musste zunächst die Umgebung aufgesetzt werden. Nach der Installation der Java-IDE, des DB2-Servers und den Entwicklungstools für diese Umgebung, folge eine kurze Analyse der gegebenen Implementation des *ChainFinders*.

Dazu wurde relativ früh ein kleines Datenset mit 40'000 realitätsnahen Transaktionen für erste Tests zur Verfügung gestellt. Dieses Datenset diente als Grundlage eines ersten Funktionstest, welcher sich konzeptionell wie folgt darstellen lässt.

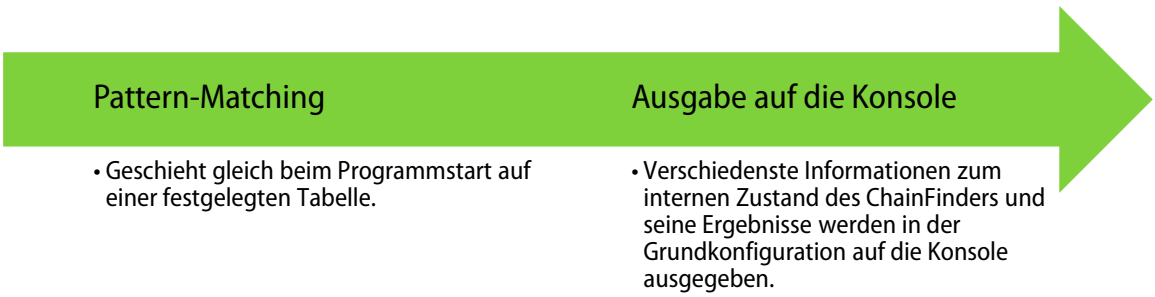


Abbildung 7: Ausganslage.

4.2 Entwicklungsstufen des Evaluationssystems

4.2.1 Experimentierphase mit DB2

Die Aufgabenstellung erforderte zunächst die Arbeit am Transaktionsdatensimulator von Simon Galliker und eventuell die Integration desselben ins Evaluationssystem. Nach einigen Refactorings am Code der betreffenden Bachelorarbeit und der Auslagerung von vielen Aufgaben in pures SQL[20] wurde deutlich, dass sich durch die Verwendung von *Stored Procedures*[21] einiges an Performance gewinnen liess. Allerdings waren dazu sehr gute Kenntnisse der spezifischen Aspekte des Datenbanksystems und den DB2-spezifischen SQL-Sprachschatz erforderlich, die sich durch etliche Experimente gewinnen liessen. In einer nächsten Stufe in dieser Experimentierphase mit DB2 entstand ein eher wenig flexibles Evaluationssystem mit beschränktem Funktionsumfang, das fast nur in SQL implementiert war. Dazu wurde prozedurales SQL inklusive vieler *OLAP-Funktionen* und *imperatives SQL* verwendet, wie auch die SQL-Sprachelemente, welche zur DBMS-Verwaltung dienen.

Im Zuge dieser Analysen stellte sich die Frage, ob ein reiner SQL-Evaluator sinnvoll sein könnte. Ist es möglich eine I_E so zu gestalten, dass diese weit gehend in der Datenbank läuft? Oder anders gefragt: Ist es möglich, einen Plain-SQL-Evaluator zu gestalten? Was sind die Einschränkungen und welche Vorteile könnte ein solcher Ansatz bieten?

Ermittelte Anforderungen:

- Der Vorbereitung der Datenbasis und die Analyse der Ergebnisse sollte ausreichend schnell geschehen.
- Die versteckten Ketten sollten die relevanten Charakteristiken von echten Ketten haben.[22]

Details zur Implementation: Zuerst werden verschachtelte Sichten (*Views*) und einige OLAP-Funktionen verwendet, vornehmlich jedoch noch während den abgebrochenen Optimierungen am Transaktionsdatensimulator. Etwas später besteht das Evaluationssystem aus einer oder mehreren *Stored Procedures*, welche zur Vorbereitung der Arbeitstabelle[23] und auch zur Analyse der Ergebnisse dienen. Innerhalb von Java sind neben der Implementation des ChainFinders nur zwei SQL-Aufrufe nötig. Einer davon veranlasst die Vorbereitung der Datenbasis und der andere startete die Evaluation des Laufs, wobei die Statistiken des Laufs in eine Tabelle (die Evaluationstabelle) geschrieben werden.

Der *ChainFinder* (siehe auch Kapitel 2.2) wird so konfiguriert, dass er seine Ergebnisse in zwei Datenbanktabellen speichert, welche der erwähnten Stored-Procedure als Grundlage für die Evaluation des Laufs auf Korrektheit dienen.

Benefits

- Das Interface konnte schmal gehalten werden, sprich es waren nicht viele Änderungen an der Implementation des *ChainFinders* nötig. Für diesen relativ simplen Evaluator ist es auch nicht zwingend notwendig, weite Teile des *ChainFinder*-Codes zu verstehen, sondern es reicht aus, seine Interaktionen mit der Datenbank zu betrachten.

[20] Es gab noch einige Experimente mit mySQL, da zuerst noch nicht klar war, welche Datenbanktechnologie eingesetzt werden soll. Später wurde nur noch DB2 verwendet.
[21] Der Verfasser entwickelte ebenfalls eine DB2-Stored-Procedure, welche es ermöglichte, Stern- und Diamant-Patterns, die Smurfing verwendeten, performant in die Daten zu injizieren. Diese wären relevant gewesen, wenn der Funktionsumfang des ChainFinders auf den Bereich des Anti-Money-Launderings ausgeweitet worden wäre. Allerdings wurde dieser Teil der ursprünglichen Aufgabenstellung zugunsten der Entwicklung eines Evaluations-Framework für beliebige Algorithmen aufgegeben.
[22] Im Detail: Die Beträge sollten die genau bis zur maximal eingestellten Fuzzy Ratio abweichen und das Transaktions-Datum sollten genau innerhalb des Windows liegen.
[23] Das mit einer Arbeitstabelle vergleichbare Konzept *Stage* wird später in diesem Kapitel eingeführt.

- Die *Stored Procedures* zur Vorbereitung der Datenbasis haben ausserdem eine gute Performance.
- Die *OLAP-Funktionen* und zum Teil auch die Konstrukte in imperativem SQL sind sehr viel mächtiger als vergleichbare Ansätze in Java.

Limitations

- Durch die Verwendung von komplexeren und vor allem DB-Plattform-spezifischeren Konstrukten reduzierte sich die Portierbarkeit auf andere Datenbanksysteme.
- Alle *Stored-Procedures* müssen zwingend in einem konsistenten Zustand gehalten werden. So liess es DB2 beispielsweise nicht zu, eine *Stored Procedure* zu schreiben, die auf eine Tabelle die noch nicht existierte oder eine andere Datenstruktur als erwartet hatte.
- Die Wartbarkeit des Codes ist tief. Das liegt daran, dass es innerhalb von komplexeren *SQL-Queries* und *Stored-Procedures* kaum Wege gibt, Redundanzen zu vermeiden.
- Informationen über die Datenstruktur und –Semantik mussten immer manuell zwischen den beiden Welten „Java“ und „DB2“ abgeglichen werden.
- Innerhalb einer *Stored-Procedure* konnten Befehle zum Anlegen und Löschen von Tabellen oder anderer Stored-Procedures sowie auch die Erstellung und Verwaltung von Indizes nicht oder nur umständlich implementiert werden.
- Es gibt noch keine GUI; jeder Batch-Job,[24] wie auch generell fast jede Änderung auch nur unwichtigerer Parameter, erforderte zwingend Änderungen am Quelltext und meist auch das Ersetzen von weiten Teilen des Evaluationssystems.

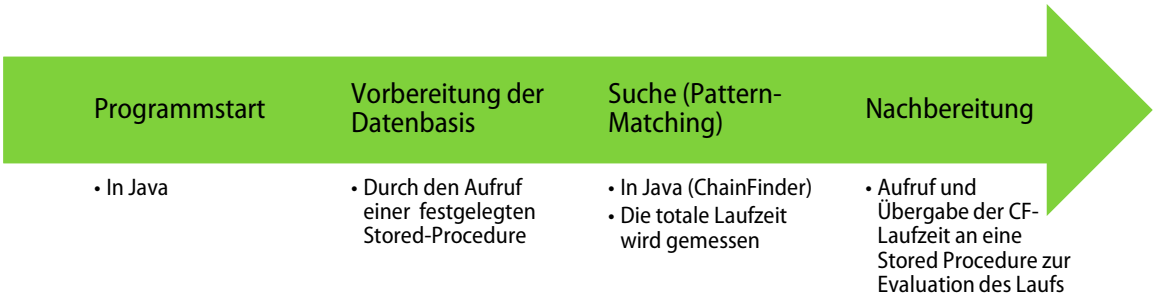


Abbildung 8: Ablauf einer (fast) reinen SQL-Evaluation.

Diskussion und Motivation für die nächstfolgende Stufe: Das „Lock-In“ auf die Plattform DB2 ist noch zu verkraften, da davon auszugehen ist, dass die künftigen zu testenden Algorithmen beim selben Finanzinstitut laufen und somit wahrscheinlich auch DB2 nutzen werden. Die schlechte Wartbarkeit, die geringe Flexibilität und der eingeschränkte Funktionsumfang sind weniger leicht hinzunehmen. Allerdings ist die Betrachtungsweise des *ChainFinders* als eine „Blackbox“[25] im Hinblick auf die Entwicklung von Konzepten für ein Evaluations-Framework für verschiedene Algorithmen in der Betrugserkennung ein hilfreicher Ausgangspunkt.

Soll ein Lauf einer feingranularen Zeitmessung unterliegen, taugt dieses Denken jedoch wenig, da solche Daten erst einmal erhoben werden müssen und dies nur innerhalb der *ChainFinders*, respektive des künfitigen zu evaluierenden

[24] Später wird das Konzept eines *Szenarios* eingeführt, was gewisse Ähnlichkeit mit einem „Batch-Job“ hat.
[25] Dies ergibt sich daraus, dass die Evaluation ohne das Wissen über die konkrete Implementation des ChainFinders vorgenommen werden kann.

Algorithmus, möglich ist. Eine „Blackbox"-Betrachtungsweise ist deshalb unter diesem Blickwinkel des Profiling, also der Evalution auf Performance, untauglich. Es liessen sich aber grobe Zeitmessungen machen und durch das geschickte Setzen von Indizes liess sich die Leistung des ChainFinders bezüglich der totalen Laufzeit schon steigern. Auf diese Thematik sei aber nicht mehr weiter eingegangen, denn es ist davon auszugehen, dass dies in der Praxis die Arbeit der Datenbankadministratoren im Rechenzentrum ist und dass ihm diese die bestmögliche Arbeitsumgebung zur Verfügung stellen.

4.2.2 Ein Evaluator basierend auf einer Template-Engine

Ein Grossteil der Einschränkungen des vorher skizzierten Systems lässt sich durch die Einführung eines Template-Systems umgehen. Diese würde sogar Kontrollstrukturen innerhalb von Templates ermöglichen. In meinen Augen wäre es aber nicht sehr sinnvoll, eine Logik in einer dritten Programmiersprache (neben Java und PL/SQL) zuzulassen, weshalb für die vorliegende Implementation darauf verzichtet wird.[26]

Die Applikation bekommt eine GUI, welche nach und nach immer mehr Funktionen bietet, wie zum Beispiel einen Template-Viewer mit Syntax-Highlighting, Voreinstellungen für die Parameter und in Echtzeit aktualisierte Statusinformationen zur momentanen Iteration innerhalb einer Stapelverarbeitung von mehreren Läufen.

Details zur Implementation:

- Die Velocity-Template Engine des Apache Projektes wird eingebunden.
- Die Resultatsmenge wird nun in die Datenbank gespeichert und kann so leichter analysiert werden.
- Die Architektur wird auf MVC umgestellt. Jedoch sind viele Klassen als Singletons implementiert, um Brüche mit der Architektur zuzulassen. So können Änderungen schnell ad-hoc von einer beliebigen Stelle des Programmes aus geschehen.
- Die Entwicklung der GUI geschieht mithilfe des GUI-Builders „Matisse". Die Tatsache, dass nun eine GUI existiert erfordert die Verwendung von Multithreading, damit keine Bedienelemente „einfrieren".
- Grundsätzlich gibt es für jedes Szenario eine Klasse, welche die Stapelverarbeitungsparameter enthält und eine Evaluatoren-Klasse, welche die ganze Logik beinhaltet. Letztere ist für alle Aufgaben innerhalb eines Laufs verantwortlich. Darunter fallen sowohl solche für das Präparieren der Arbeitstabelle und der Daten, wie auch für die Kommunikation mit der Implementation des ChainFinders und solche für die Analyse und Speicherung der Resultate.

Benefits:

- Es ist nun möglich, vollkommen dynamisch generierte *Stored Procedures* in die Datenbank zu schreiben, was einen höheren Automatisierungsgrad und schnellere Entwicklungszyklen ermöglichte.
- Die GUI ermöglicht eine einfachere Bedienung.

Limitations:

[26] Es sei allerdings angemerkt, dass diese Funktionalitäten nicht komplett deaktiviert wurden, sondern, dass es einfach eine Design-Entscheidung war, diese nicht zu benutzen.

- Die Implementation der GUI machte die Anpassung des Codes schwieriger. So war unter anderem die Verwendung des Observer-Patterns nötig und an einigen Stellen eine wurden vermehrte Übergabe von Variablen benötigt, was die Komplexität erhöhte[27] und der Isolation etwas schadete.
- Es entstand eine zusätzliche Schicht an Komplexität durch die Verwendung von Variablen innerhalb von Java und deren Übergabe in die Template-Engine. Dieses Problem wurde nur teilweise durch die Design-Entscheidung entschärft, dass die Template-Variabeln aus Sicht des Java-Codes „write-only" und aus Sicht der Template-„Logik" nur „read-only" waren.

Motivation für die nächste Stufe: Die Workbench war schwierig anpassbar und es gibt viele Abhängigkeiten auf den Ebenen GUI-Builder/Model, Implementation des ChainFinders/Model, Model/Templating-Subsystem. Das System war sowohl eng gekoppelt, lässt aber auch bewusst Brüche mit der vorgegebenen Architektur zu. Dies ist beim GUI-Code nicht anders. Dieser erstreckte sich nur über eine einzige Klasse, was ebenfalls dazu führte, dass diese in ihrem inneren Aufbau und in ihren Abhängigkeiten zu weiteren Klassen grösstenteils verstanden werden muss.

Es war relativ einfach, neue Funktionen ad-hoc zu implementieren aber nur genau dann, wenn Wissen über die Eigenheiten des ganzen Systems vorhanden war. Die Programmierung „auf Vorrat" und die Implementierung einiger nicht essenziellen Features und die damit einhergehende Notwendigkeit zur Verwendung von zusätzlichen Design-Patterns verschärft diese Problematik zusätzlich.

4.2.3 „Plug-and-Play" Evaluationssystem

Alle nicht essenziellen Features und Design-Patterns, wie auch jeglicher Code, welcher „auf Vorrat" produziert wurde, wurde entfernt. Das bisherige Vorgehen lässt sich wahrscheinlich recht gut mit dem Begriff des „explorativen Prototypings" umschreiben. Gleichzeitig wurde bereits in Kapitel 3.6, welches das Problem der Evaluation von Algorithmen zur Betrugserkennung auf einer abstrakteren Ebene betrachtet, impliziert, dass generell viele Abhängigkeiten zwischen einzelnen Teilen des Systems zu bändigen sind. Folgende wichtige Anforderung an diese Entwicklungsstufe des Evaluationssystem wurden speziell beachtet:

- Das Aufwand/Nutzen-Verhältnis sollte bei der Evaluation eines erweiterten Frameworks besser sein, als die ad-hoc Entwicklung eines eigenen Evaluators für einen neuen Algorithmus, ohne die Verwendung des vorliegenden Frameworks.
- Die Resultate sollten sich einfach auswerten lassen.

Details zur Implementation

Alle nicht essenziellen Features wurden entfernt, um die Anpassung und Weiterverwendung durch einen neuen Entwickler zu vereinfachen. Es gab keine tiefen Klassenhierarchien mehr und vom Design her setzt das Framework stark auf „Composition" und nur wenig auf Vererbung. Die Architektur im Detail wird in Kapitel 4.6 erläutert.

Benefits

- Es gibt nun einen Editor mit angepasstem Syntax-Highlighting, sowie einige Tools für das Experimentieren und den vereinfachten Umgang mit SQL-Templates. Eine genaue Anleitung zum praktischen Umgang mit diesen Funktionen findet sich im Kapitel 4.7.

[27] Dieses Design-Pattern kommt nicht ohne die Verwendung von Vererbung aus, was zumindest potenziell die Flexibilität beeinträchtigte – „Vererbung ist teuer".

- ❖ Jedes Szenario kann eine Eingabemaske definieren, die sich automatisch in die GUI einklinkt.
- ❖ Neben dem *ChainFinder* wurde noch der Dummy-Algorithmus „MiniMouse“ und mit einem rudimentären Evaluationssystem eingebaut. Dieses einfache Beispiel, wie das Framework erweitert werden kann, sollte einem künftigen Entwickler die Einarbeitung erleichtern.
- ❖ Es gibt eine einfache Maske zum Absetzen von ad-hoc Anfragen über die momentane Datenbankverbindung. Am Ende jedes Szenarios werden über diese Maske automatisch die Statistiken über die Läufe angezeigt.

Limitations

- ❖ Es wird kein eigener Automatismus geboten, um GUI-Code zu erzeugen. Allerdings kann dieses Problem durch die Verwendung eines GUI-Builders umgangen werden.

4.3 Entwickelte Konzepte im Detail

Folgende Konzepte wurden im Laufe der Erarbeitung entwickelt:

- **Ein Szenario** ist die Batch-Informationen zur Verarbeitung von mehreren Durchläufen eines Algorithmus. Es definiert, welche Generatoren/Loader, Stages, Profiler und Checker eingesetzt werden.
- **Die Stage** ist die Arbeitstabelle für einen gewissen Durchlauf.
- **Ein Generator** kann synthetische Daten generieren, die am Schluss in die Stage (Arbeitstabelle) geschrieben werden.
- **Ein Loader** ist eine Spezialform eines Generators. Seine Funktionalität besteht darin, vorhandene Einträge aus einer Datenquelle in die Arbeitstabelle zu kopieren.
- **Ein Checker** überprüft die Integrität der Ergebnisse eines Durchlaufs.
- **Ein Profiler** ist für die Messung der Performance verantwortlich.
- **Evaluationstabelle:** In der Regel existiert eine solche Tabelle pro Szenario. Sie speichert die Statistiken der einzelnen Läufe (welche vom Checker und/oder Profiler kommen). Zusätzlich enthält sie typischerweise Angaben zu den Parametern des Generators/Loaders, des verwendeten Injektors oder der verwendeten Injektoren, des Szenarios und der verwendeten Parameter der Implementation des Algorithmus.
- **Der Proxy:** Jeder Algorithmus, der mit dem Framework getestet werden soll, wird nur indirekt über einen Proxy angesprochen. Er regelt die Änderung an dessen Konfiguration und stellt Template-Variabeln zur Verfügung, welche sehr oft gebraucht werden in den vorher genannten Einheiten (wie Checker, Stage, Profiler, Szenario und Generator/Loader). Diese sind gemäss der Definition in der Problemstellung dann typischerweise genau die Elemente der Menge $P_{contex\ t}$.

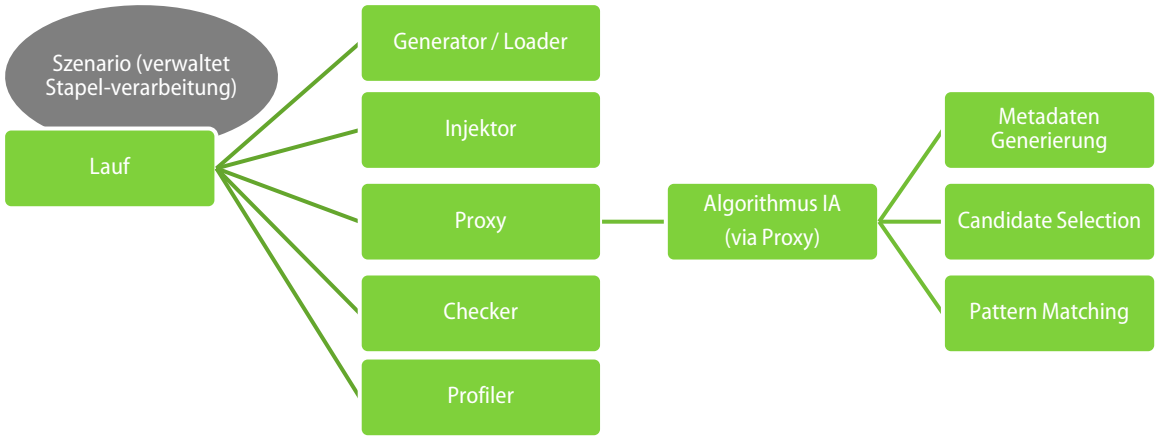


Abbildung 9: Komponenten des entwickelten Evaluationssystems.

4.3.1 Stage

Wir gehen davon aus, dass wir eine Bühne oder Stage haben („Arbeitstabelle"), auf der sich das Datenset für den momentanen Durchlauf befindet.

Vorteile im Vergleich zum direkten Zugriff auf die vorgegebenen Tabellen: Nehmen wir an, wir hätten 4 verschiedene Datensets, welche sich in ihrer Grösse unterscheiden und das Einzige, was uns interessiert, sei die Laufzeit. Wir könnten nun direkt auf die vier Tabellen zugreifen, oder wir könnten den Inhalt in eine weitere Tabelle kopieren. Dies würde den Vorteil mit sich bringen, dass wir nicht schauen müssen, ob die genauen Datentypen und die Indizes bei allen 4 Tabellen gleich sind. Die Performance wäre nun also nicht mehr abhängig von den Indizes etc., sondern nur noch von derjenigen der Stage. Das Konzept der Stage oder auch „Staging Area" stammt aus dem Bereich des Data Warehousings.

4.3.2 Generator/Loader

Ein Generator ist dazu da, ein gewisses Datenset gemäss bestimmten Kriterien zu generieren. Dies geschieht meist einmal pro Szenario, also einmal für mehrere Läufe. Wenn ein Generator nur die Daten aus einer anderen Tabelle lädt und in die Stage kopiert, kann er auch als Loader bezeichnet werden.

Um funktionieren zu können, muss er gewisse Informationen über die Datenstruktur und deren Semantik haben. Bei den Daten handelt es sich um Transaktionsdaten, was eine Abbildung von einer komplexeren Datenstruktur in einzelne Transaktionen bedingt. Die Struktur des generierten Graphen baut sich aus den gewählten Einträgen in den Feldern Absender und Begünstigter auf. Sie werden durch die Kanten repräsentiert. Die anderen Einträge wie Datum oder Sachbearbeiter sind dann nur Attribute der betreffenden Kante, sagen aber nicht über die Struktur aus.

Die Performance des Generators hat keinen Einfluss auf die Performance-Evaluation der I_A, da er seine Arbeit vor dem Aufruf des Algorithmus verrichtet. In der Praxis könnte es mühsam sein, mit Generatoren zu arbeiten, welche sehr lange brauchen.

Wir haben das Problem der einzelnen kleine Queries im Kontext eines verteilten Systems betrachtet (siehe oben, Kapitel 3.3). Dieses käme dann zum Tragen, wenn ein Generator seine Arbeit über viele einzelne kleine SQL-Statements hinweg machen würde, womit wegen den einzelnen Queries zu viel Overhead anfallen würde. Dies ist

typisch für den Fall, wenn wir Transaktionen in komplexeren Datenstrukturen in Java repräsentieren und dann als simple Datensätze an die Datenbank übergeben müssen. Ein anderer Weg wäre es, komplexere SQL-Statements zu generieren, womit man das Problem beschränken könnte. Es fallen aber immer noch Aufwände für die Umwandlung der proprietären Datentypen in SQL-Statements, der Transport und das Verpacken der Queries und der Resultssets, welche bei der Verwendung von *Stored Procedures* entfallen.

4.3.3 Injektor

Ein Injektor wird üblicherweise für jeden Lauf einmal ausgeführt. Er sollte deswegen mit Vorteil wissen, wie er die Hintergrunddaten von den injizierten Patterns unterscheiden kann. Das gibt ihm die Möglichkeit, seine Änderungen zurückzunehmen, ohne gleich die ganze Tabelle zu löschen.

Die Injektoren arbeiten mit gewissen Parameter und einer gewissen Strategie, um ein Datenset so zu ändern, dass neue Patterns darin enthalten sind. Sie bräuchten zumindest eine Teilmenge der Struktur und Semantik-Informationen, die der Generator hat.

Am Anfang werden typischerweise die bereits vom vorherigen Lauf her noch bestehenden Patterns gelöscht. Vor dem ersten Lauf in synthetischen Realdaten findet keine Löschung statt, da noch keine künstlichen Patterns hinzugefügt worden sind.

4.3.4 Evaluations-Tabelle

Um die Ergebnisse der Konsistenz-Prüfungen und der Zeitmessungen zu speichern, wird eine für das Szenario dedizierte Datenbanktabelle erstellt, die Evaluations-Tabelle, wobei jeder Lauf eine Zeile darstellt. Die momentane Implementation des Frameworks verwendet dazu aus Templates generierte SQL-Statements[28].

4.3.5 Checker

Die Konsistenzprüfer analysieren die Ergebnisse des Laufs (Treffermenge) auf Korrektheit durch Vergleich der Resultatsmenge mit den durch den Injektor in die Arbeitstabelle eingefügten Patterns, wobei die Statistiken dieser Analysen in die Evaluationstabelle geschrieben werden.

Die Checker sind abhängig von den Parametern $P_{context}$, P_{query} und $P_{restrict}$, jedoch nicht von der Implementation der Strategie selbst, sondern nur von der Strategie. Die Idee hinter den Checkern ist es, Fehler in der I_A zu finden, also in der Umsetzung einer Strategie in Code. Gleichzeitig könnten die Fehler aber auch aus falsch funktionierenden Generatoren oder Injektoren stammen. Streng genommen müsste man deren Ergebnisse auch noch automatisiert auf Korrektheit überprüfen.

4.3.6 Szenario

Ein Szenario definiert, welche Läufe mit welchen Parametern durchgeführt werden, und wie die Evaluation dieser Läufe stattfindet. Es liefert also die Informationen, wie die Stapelverarbeitung ablaufen soll.

Nachdem die Evaluationstabelle erstellt wurde, wird festgelegt, welcher Generator oder welche Generatoren zum Einsatz kommen, mit welchen Injektoren und ihren entsprechenden Parametern die Patterns geschaffen werden, und schliesslich, welche Checker und Profiler zum Einsatz kommen.

[28] Die Verwendung von Hibernate hätte sich vielleicht angeboten, sie wurde nicht geprüft, weil dann ein zusätzlicher Export-Filter nötig gewesen wäre, um die Daten in einem Aufbereitungsprogramm wie Excel weiter zu verwenden.

4.4 Die Szenarien im Detail

4.4.1 Das Hierarchical-Tree-Szenario

Um den ChainFinder unter Extrembedingungen zu testen, wurde das Datenset so ausgestaltet, dass es ein einzelner hierarchischer Baum ist. Dieser hat definitionsgemäss genau eine Wurzel. Auf jeder Hierarchiestufe verzweigt sich der Baum um einen definierten Faktor, bis der Faktor *„Maximale Tiefe"* erreicht wird. Das vorliegende Szenario stellt den *ChainFinders* so ein, dass die Wurzel als einziges Element des Rootsets genommen wird.

Das Ziel dieses Szenarios ist es, genau die injizierten Transaktionen zu finden. Dazu muss sichergestellt werden, dass keine Zufallstreffer möglich sind oder dass diese wenigstens sehr unwahrscheinlich sind.

Stage („HierTreeStage"): Die Tabelle enthält die üblichen Felder *Transaktions-ID*, *Absender*, *Begünstigter*, *Sachbearbeiter* und *Betrag*. Der Injektor und der Checker brauchen zusätzlich noch die Felder *Ketten-ID* und *Position in der Kette* sowie *Tiefe*.

Generator („HierTreeGen"): Zwei der Parameter des Generators können über das Frontend eingestellt werden (diese sind im Folgenden fett gedruckt). Die anderen sind weniger wichtig und können nur in der entsprechenden Szenario-Klasse im Java-Code geändert werden:

- **Anzahl Transaktionen pro Multiedge** *(„branchTrx")*: Die ist ein einfacher Weg, um die Datenmenge etwas grösser zu machen. Je grösser dieser Wert, desto mehr Transaktionen finden zwischen den verbundenen Konten statt.
- **Branch-Factor** *(„branchNodes")*: Dieser Wert gibt an, auf wie viele Konten der nächst tieferen Stufe sich ein Konto verzweigt. Eine Erhöhung dieses Parameters lässt die Datenbasis exponentiell anschwellen.

Weiter lässt sich das tiefst mögliche Datum der Transaktionen einstellen. Es werden Transaktionen generiert, deren Datum innerhalb des Zeitraums zwischen dem tiefst möglichen Datum und einer definierbaren Zeitspanne in Tagen ab dem tiefstmöglichem Datum liegt. Im Moment ist diese Zeitspanne sehr gross gewählt, damit die später injizierten Ketten nicht wegen zufälligen Überschneidungen mit den Hintergrunddaten länger werden bzw. sich deren Anzahl durch „Zufallstreffer" erhöht.

Zudem lassen sich die maximale Anzahl der zu generierenden Transaktionen und die maximale Tiefe des Baums festlegen.[29] Letztere sollte nicht verändert werden, da sonst manuelle Anpassungen am Injektor nötig sind.

Dank der Verwendung einer *Stored-Procedure* zur Generierung der Datenbasis lässt sich auf unserem Testsystem eine Leistung von etwa einer Million Transaktionen pro Minute erreichen. Diese Geschwindigkeit ist gleich hoch, auch wenn eine langsame Netzwerkverbindung zum Einsatz kommt, aber um Grössenordnungen höher, als wenn jede einzelne Transaktion durch eine einzelne Query erzeugt werden würde.

Injektor („HierTreeInjector")**:** Es werden eine bestimmte Anzahl Ketten der Länge 2 bis 10 in die Datenbasis injiziert, die alle bei der Wurzel des Baumes starten. Der Wert des einzigen wählbaren Parameters *„multiplicator"* mal neun entspricht der Anzahl total injizierten Ketten, da jeweils eine Kette der Länge 2 bis10 injiziert wird.

[29] Unsere Datenbank hatte Mühe, sobald die Tabelle mehr als ungefähr 10 Millionen Transaktionen enthielt.

Checker („HierTreeChecker"): Wie bereits erwähnt, soll die Konfiguration des Generators, des Injektors und des *ChainFinders* in diesem Szenario dazu führen, dass nur genau diejenigen Ketten gefunden werden, die auch injiziert werden. Der Checker ist so eingerichtet, dass er einen Fehler meldet, wenn zu viele Ketten, Transaktionen oder Konten in der Ergebnismenge vorliegen, sowohl wenn deren Zahl zu tief ist. Anders als beim *SimData-Szenario* muss im vorliegenden Szenario die Treffermenge exakt den injizierten Pattern entsprechen, da die Hintergrunddaten wie bereits erwähnt so gewählt wurden, dass keine „Zufallstreffer" entstehen.

Profiler („CFProfiler"): Der Profiler misst unter anderem die Zeit, die für die Speicherung der Treffermenge (*Runtime Logging*), für die Ausführung des Java-Codes (*Runtime CFJava*) und für die Datenbankanfragen (*Runtime CFQuery*)[30] während des Pattern-Matchings aufgewendet wurde. Er ist bei allen Szenarien für den *ChainFinder* der Gleiche.

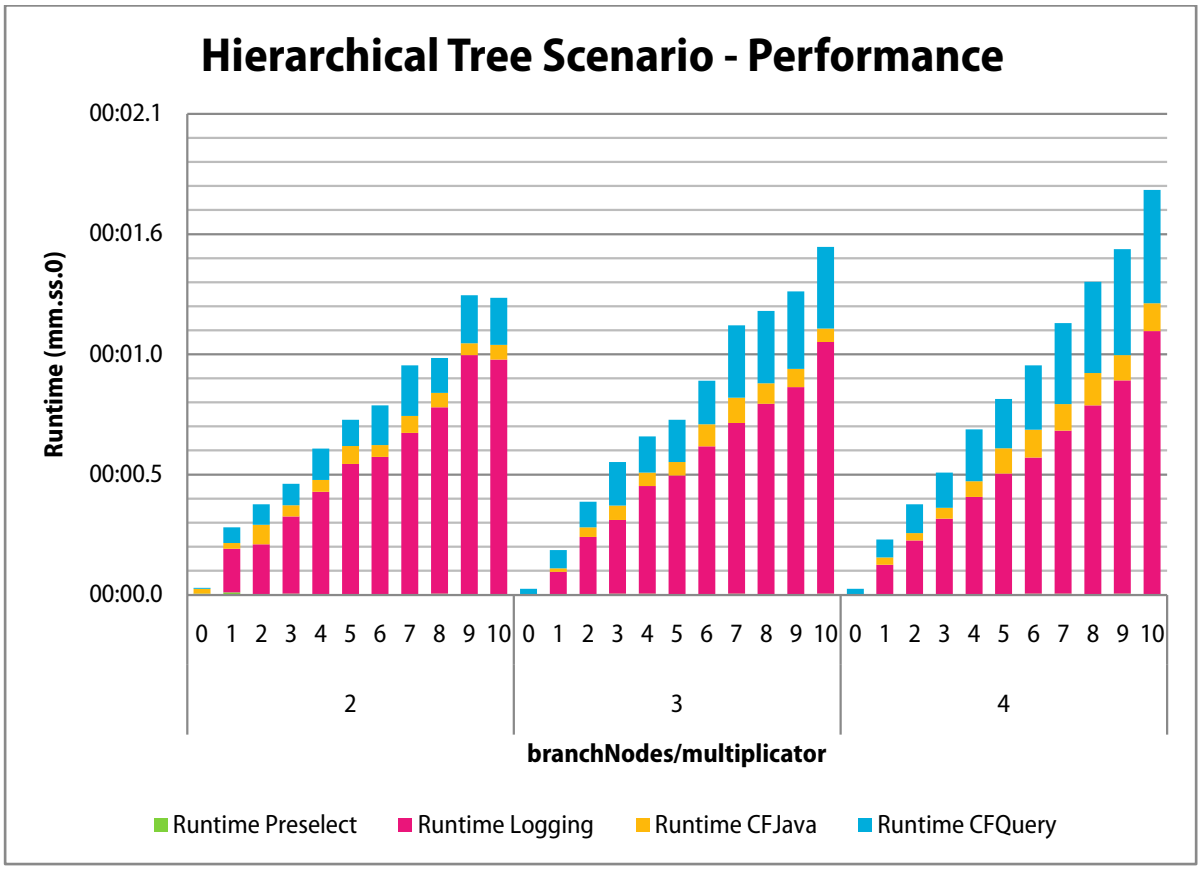


Abbildung 10: Ergebnisse der Evaluation des Hierarchical Tree Scenarios.

Die drei Blöcke repräsentieren Datensets der Grösse von etwa 2000, etwa 90'000 und etwa 1,2 Millionen Transaktionen (von links nach rechts). Minimal wurden 0 Ketten injiziert, maximal 10 von jeder Länge von 2 bis 10 Transaktionen, sprich maximal 90 Ketten insgesamt. Es wurden immer alle Ketten gefunden, es gab keine überschüssigen Treffer.

Was auffällt, ist, dass die Queries zum Expandieren der Nodes (*Runtime CFQuery*) bei einer grossen Datenmenge etwas länger zu dauern scheinen. Dies liegt wahrscheinlich daran, dass der Umgang mit einer bedeutend grösseren Datenmenge für das DBMS etwas Mehraufwand bedeutet.

[30] Die Abkürzung „CF" steht für *ChainFinder*, wobei jedoch spezifisch das *Pattern-Matching* gemeint ist, was jedoch vorliegend dasselbe ist, da der *ChainFinder* das *Pattern-Matching* durchführt.

Die Ausführungszeit der Berechnungen für das Pattern-Matching innerhalb von Java (*Runtime CFJava*) ist nicht abhängig von der Datensetgrösse. Die Candidate Selection (*Runtime Preselect*) ist zu vernachlässigen, da dieses immer nur eine einzige Root zurückgibt und hier keine Optimierungen möglich sind.

Die Zeit für das Speichern der gefundenen Ketten (*Runtime Logging*) ist nicht abhängig von der Datensetgrösse. Was jedoch auffällt, ist, dass die Speicherung (*Logging*) der Ketten mehr Zeit braucht als das Auffinden(*CFJava + CFQuery*) von diesen.

4.4.2 Das Simdata-Szenario

In diesem Szenario werden die Hintergrunddaten nicht generiert, sondern aus bereits in der Datenbank existierenden Tabellen in die Stage geladen. Sie enthalten Transaktionsdaten, welche ähnliche Charakteristiken aufweisen wie reale Daten, wobei also davon ausgegangen werden kann, dass bereits Ketten („Zufallstreffer") darin enthalten sind.

Stage („SimdataStage"): Neben den minimal notwendigen Feldern *Transaktions-ID*, *Absender*, *Begünstigter*, *Datum, Sachbearbeiter* und *Betrag* sind für den verwendeten Injektor (und den Checker) noch die Felder *Ketten-ID* und *Position in der Kette* wichtig. Der Injektor fügt noch die Felder *Start-Transaktion der Kette* und *End-Transaktion der Kette* ein, welche vom Checker eine schnelle und gründliche Überprüfung der Treffermenge ermöglichen. Mehr dazu später unter „SimdataChecker".

InjektorU („SimdataInjectorU")**:** Dies ist der Erste von zwei Injektoren, der für das vorliegende *Simdata-Szenario* verwendet werden kann. Anders als im *Hierarchical-Tree-Szenario* werden Ketten an zufällig ausgewählte Konten angefügt, was diese per Definition zu einem Root werden lässt. Der Buchstabe „U" steht für „uniform", was damit begründet ist, dass dieser Injektor Ketten von jeder Länge von 2 bis 10 Transaktionen gleich verteilt anhängt.

InjectorP („SimdataInjectorP"): Das Ziel dieses Injektors ist es, die Patterns so in das Datenset zu integrieren, wie es mehr der Realität entspricht. Der Buchstabe „P" steht dabei für „Probability". Er sucht die Roots zwar ebenfalls zufällig aus, jedoch erhält nicht jeder dieser Roots dann genau eine Kette. 50% der ausgewählten Konten erhalten eine Kette, 20% der Konten werden 3 Ketten zugeordnet, in weiteren 20% 5 Ketten und in den letzten 10% 20 Ketten. Die Länge der Ketten ist gleich verteilt, ebenfalls im Bereich von 2 bis 10 Transaktionen.

Dieses Verhalten macht ihn weit weniger deterministisch als der InjectorU. Im Durchschnitt werden also pro Root 4,1 Ketten generiert[31]. Die genaue Anzahl Ketten weicht jedoch stark von der erwarteten Anzahl ab, wenn nur wenige Wiederholungen gemacht werden oder der Multiplikator klein ist.

Checker („SimdataChecker"): Zuerst wird die Zahl der gefundenen Konten und Transaktionen mit derjenigen verglichen, welche bei der Injektion integriert worden sind. Um den Test zu bestehen, muss die jeweilige Anzahl gefundener Einträge mindestens so gross sein, wie diejenige der Injizierten. Zusätzlich wird noch überprüft, ob jede Transaktion und jedes Konto (sprich deren ID) in der Treffermenge vorkommen. Zuletzt wird noch getestet, ob die Ketten, welche vom Injektor erzeugt wurden, exakte Entsprechungen in der Ergebnismenge haben. Jede Transaktion von zwei zu prüfenden Ketten muss den gleichen Eintrag haben betreffend der *ID*, der *Start-Transaktion* und derjenigen der *End-Transaktion*. Gleichzeitig müssen alle Elemente der Kette in ihrer Position innerhalb der Kette übereinstimmen.

[31] 50% * 1 + 20% * 3 + 20% * 5 + 10% * 20 = 410% = 4,1.

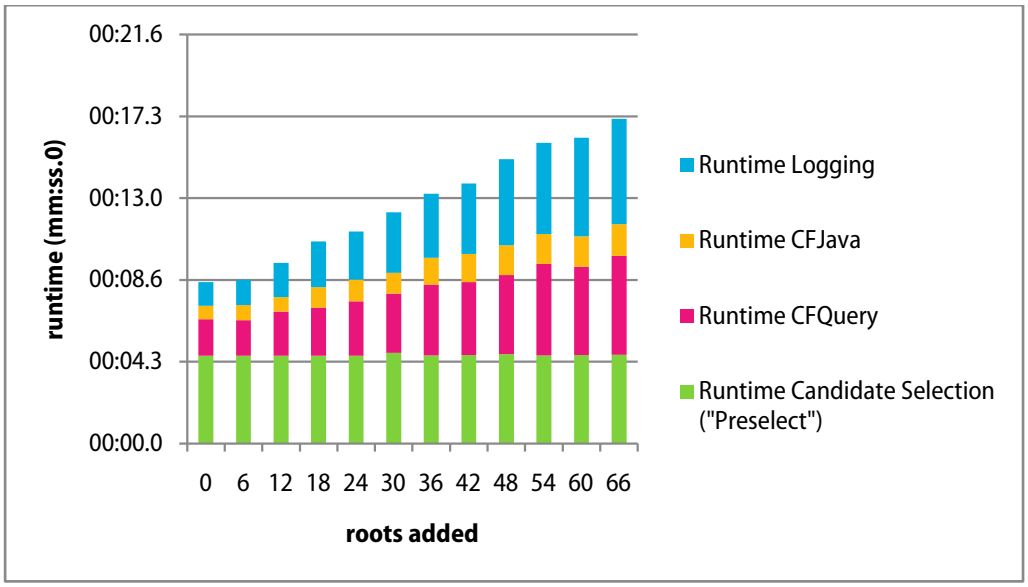


Abbildung 11: Evaluation eines Simdata-Laufs.

„roots added" entspricht der Anzahl Konten, die durch den Injektor hinzugefügte Ketten erhalten haben.

Evaluation: Die obere Grafik wurde auf dem kleinsten Datenset mit 0.36 Millionen Transaktionen gemacht. Die Laufzeit des *ChainFinders* ist im Falle von 0 hinzugefügten Ketten nicht nahe null, da bereits in den Hintergrunddaten zufälligerweise Patterns enthalten sind, die der *ChainFinder* auffindet.

Interessant ist übrigens, dass sich die Laufzeit der *Candidate Selection* weitgehend unabhängig zur Anzahl hinzugefügter Roots verhält, was Rückschüsse über deren Skalierbarkeit ermöglicht, die Gegenstand des nachstehenden Kapitels sind.

4.5 ChainFinder-Evaluation

4.5.1 Java-Chainfinder vs. SQL-ChainFinder

Während der Entwicklung wurde der *ChainFinder* einmal rudimentär in SQL nachprogrammiert. Dies deshalb, weil dessen Java-Implementation per Voreinstellung jedes Konto als verdächtig und somit als Teil des RootSets einstufte, womit eine Überprüfung nach Ketten für jedes Konto sequenziell durchgeführt wurde. Dies generierte viele einzelne kleine Queries und somit eine erhöhte Gesamtlaufzeit. Aufgrund dieser sehr hohen Ausführungszeit schien es naheliegend zu überprüfen, ob sich in reinem SQL nicht eine ähnliche aber schnellere Lösung implementieren liess (vgl. nachstehende Abbildung 12 „Baseline-Analyse").

Der so Entstandene rudimentäre SQL-ChainFinder bestand aus einer Serie von SQL-Queries, welche Transaktionsketten einer bestimmten Länge finden konnten. Diese Lösung konnte nicht so gut konfiguriert werden wie der *Java-ChainFinder* und war auch nicht in der Lage, beliebig lange Ketten zu finden. Des weiteren wurden teilweise falsche Ergebnisse geliefert (wenn nach Ketten mit einer Länge grösser 2 gesucht wurde). Die Verlängerung der zu suchenden Kette um ein Konto verlängerte den getätigten Join über eine zusätzliche Tabelle. Somit entstand also eine Query, die mit einem Join über 10 Tabellen arbeitete, falls Ketten der Länge 10 gesucht wurden.

Am Ende stand fest, dass es auch in grossen Datensets via SQL relativ einfach und schnell möglich war, Ketten zu finden, welche eine Länge von 2 Transaktionen (also 3 Konten) haben. Dieser Umstand führte zu einer Optimierung der *Candidate Selection* im *Java-ChainFinder*. Im Vergleich zur ursprünglichen Version des *Java-ChainFinders* konnte durch die optimierte *Candidate Selection* eine Performancesteigerung vom Faktor 10 und mehr erreicht werden.

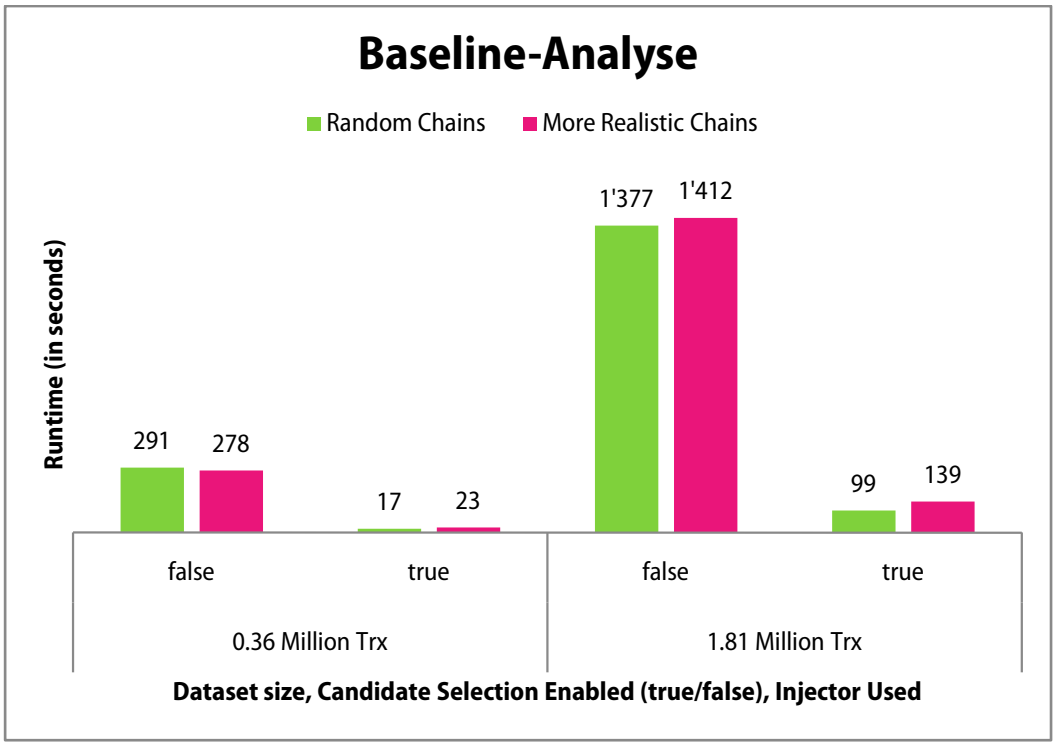


Abbildung 12: Die Baseline-Analyse.

Die Baseline-Analyse basiert auf dem sog. *Simdata-Szenario*. Sprich es wurden simulierten Realdaten mit einem Umfang von 0.36 Millionen und 1.81 Millionen Transaktionen verwendet. Der Anteil der neu injizierten Transaktionen wurde auf 0.1% festgesetzt, unabhängig vom vewendeten Injektor, was ungefähr ein praxisnaher Wert sein dürfte. Die Details zu den verwendeten Injektoren sind im Kapitel 4.4.2 erläutert, wobei „Random Chains" die Verwendung des *InjectorU* impliziert und „More Realistic Chains" diejenige des *InjectorP*.

Kurz vor dem Abgabetermin entstand aus einem Experiment eine komplett funktionsfähige Version eines reinen *SQL-ChainFinders*, also einer Implementation des ChainFinder-Algorithmus in einer DB2-Stored Procedure. Die Skalierbarkeit dieser Version ist erheblich verbessert, da keine Joins über mehr als zwei Tabellen stattfinden, was in der ersten SQL-Implementation noch anders war.

Die nachstehende Grafik zeigt die Performance der neusten SQL-ChainFinder-Version im Vergleich mit dem Java-ChainFinder mit optimierter *Candidate-Selection*. Die SQL-Implementation ist dabei etwa 10 Mal schneller als der Java-ChainFinder mit der optimierten Candidate Selection. In einem Vergleich mit dem Java-ChainFinder ohne optimierte Candidate Selection ist er etwa 100 Mal schneller auf simulierten Realdaten.

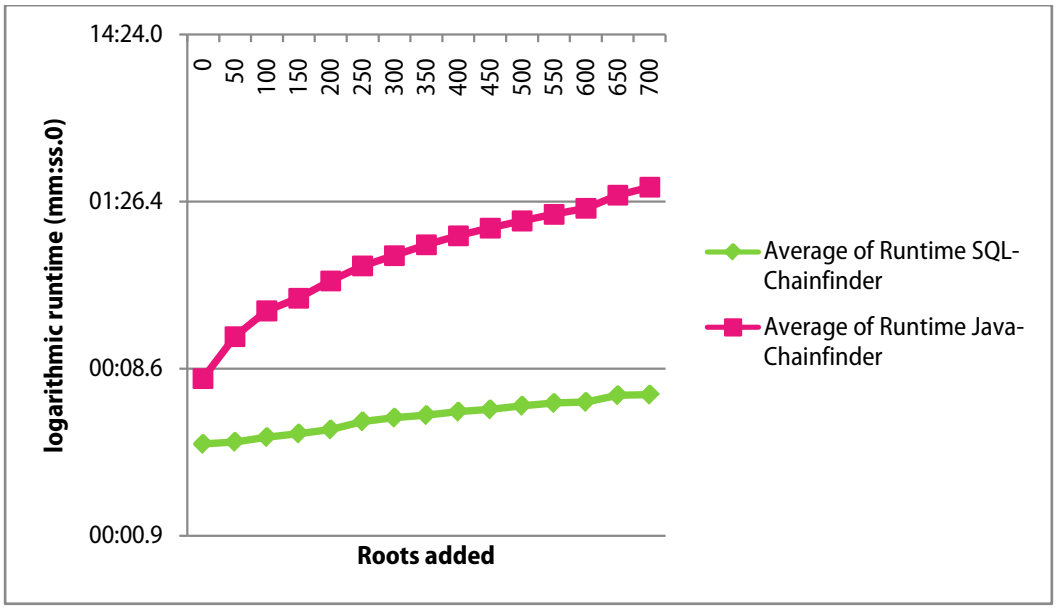


Abbildung 13: Performance-Vergleich von SQL-ChainFinder und Java-Chainfinder.

Datenset: Simulierte Realdaten (0.37 Millionen Transaktionen), wobei der *InjectorP* verwendet wurde. Der Anteil der neu injizierten Ketten macht bei *„Roots added" = 62* ziemlich genau 0.1% der Gesamttransaktionszahl aus.

In einigen Läufen ergab deren Analyse durch den *Checker* (*„SimdataChecker"*), dass eine einzige Kette nicht gefunden. Interessanterweise wurde sie fast immer sowohl vom Java-Chainfinder, als auch vom SQL-Chainfinder nicht gefunden, was eher auf einen Fehler im Checker oder Injektor hinzudeuten scheint, als beim Pattern-Matching. Dieser Fehler lässt sich vernachlässigen, wenn es um die Messung der Performance geht. Allerdings deutet er darauf hin, dass sowohl die neue SQL-Implementation des ChainFinders, als auch die Java-Version sich ziemlich gut entsprechen.

4.5.2 Änderungen am Source Code des Java-ChainFinders

Die folgenden Absätze betreffen die getätigten Änderungen am Source Code des Java-ChainFinders, sowohl in Bezug auf seine Evaluation und Evolution sowie seiner Einbindung in die Workbench. Dieses Thema wurde schon in der Aufgabenstellung angesprochen, vgl. Kapitel 3.6

Proxy/Konfigurations-Klasse: In seiner ursprünglichen Version hatte der *ChainFinder* eine statische Konfiguration. Wenn er jedoch automatisch unter verschiedenen Einstellungen getestet werden soll, musste mit diesem Ansatz gebrochen werden. Die Konfiguration konnte nach den Änderungen dynamischer angepasst werden, wobei die Methoden dazu über den Proxy des *ChainFinder* aufgerufen werden konnten.

Konsistenz-Checks: Für einen einzigen Konsistenz-Check mussten aus Performance-Gründen zwei weitere Werte in die Tabelle mit den verdächtigen Transaktionen[32] geloggt werden. Jede einzelne Transaktion hatte ja die Zuteilung zu einer Kette. Nun musste bei jedem Element einer solchen Kette noch die Information über die *ID der Starttransaktion* gespeichert werden, wie auch diese der *End-Transaktion innerhalb der Kette*. Sind Indizes auf den richtigen Feldern gesetzt, ermöglicht das einen schnellen Abgleich mit den injizierten Ketten. Die geprüften Konsistenzbedingungen sind folgende:

- Ist die Starttransaktion und die Endtransaktion in der Kette dieselbe.

[32] Vgl. Kapitel 2.2.

- Sind alle einzelnen Transaktionen innerhalb der Kette an der korrekten Position (dieser Check ist möglich, weil sowohl der *Injektor*, als auch der *ChainFinder* diese Positionsangabe speichern).

Dies ist der einzige wirklich wichtige Check. Allerdings macht er es nötig, dass die zusätzlichen Felder sowohl bei der Injektion, als auch bei der Resultatsmenge vorhanden sind. Wäre dies nicht so, würde allein die Überprüfung der Resultate ein Problem von relativ hoher Komplexität schaffen.

Die optimierte Candidate Selection: Die Implementation der optimierten *Candidate Selection*, sprich des SQL-Preselects, machte kaum Änderungen am Java-Code nötig. Es musste nur das SQL-Statement angepasst werden, welches das Rootset abfragte. Gleichzeitig musste in der *WHERE-Klausel* noch die Fuzzy-Ratio angegeben werden, sprich die maximal zulässige Abweichung beim Betrag und die maximal zulässige Abweichung betreffend des Datums.

Nötige Änderungen für die Performance Evaluation: Es wurden einige Stoppuhren über einige Code-Abschnitte verteilt. Jede einzelne Methode derjenigen Klasse, welche die Resultatenmenge in die Datenbank schreibt, wurde gemessen, sprich die nötige Ausführungszeit in Millisekunden erfasst und dem Evaluator übergeben. Zusätzlich wurden Anzahl und die benötigte Zeit der abgesetzten Queries erfasst, welche nichts mit dem Logging der Resultatsmenge zu tun hatten. Sie wurden in ihrer Gesamtheit gemessen, wobei jedoch die Ausführungszeit des Preselects in einem dedizierten Feld erfasst wurde, damit eine genauere Evaluation in Bezug auf die erreichte Effizienzsteigerung möglich war.

Implikation dieser Änderungen bezüglich der Entwicklung eines Frameworks: Für die Umsetzung der Konsistenz-Überprüfungen, wie auch für die Umsetzung der Performance-Evaluation, waren Änderungen an der Implementation des ChainFinders nötig. Erstere haben wahrscheinlich kaum einen Einfluss auf die Performance selbst. Die zusätzlich geloggten Informationen verändern zwar die generelle Ausführungszeit eines Durchlaufs. Jedoch ist die Zeit, welche für das Logging benötigt wird, separat erfasst und kann so von der totalen Laufzeit abgezogen werden. Die Erfassung der Performance ist durch direkte Feldzugriffe und nicht etwa durch *Reflection* oder ein *Listener-Pattern* implementiert. Sie sollten so nur sehr wenig Rechenzeit in Anspruch nehmen, wegen der tiefen Komplexität im Vergleich zu den beiden anderen erwähnten möglichen Methoden. Ihre Informationen werden erst nach Abschluss eines Durchlaufs vom Evaluator in die Evaluationstabelle geschrieben.

4.6 Architektur des Evaluations-Frameworks

Betrachtet man den Code der Workbench, dürften viele Dinge selbsterklärend sein, beziehungsweise sich mithilfe der Kommentare im Code einfach herausfinden lassen.

Es wurde ein MVC-Pattern verwendet. Der Controller ist als Singleton implementiert und die sehr schlanke Klasse „Model“ bietet sich als Superklasse für die meisten künftigen Klassen an. Ihre Hauptfunktionalität liegt darin, einen noch einfacheren Zugriff auf den Controller und Statusausgaben im Frontend zu ermöglichen.

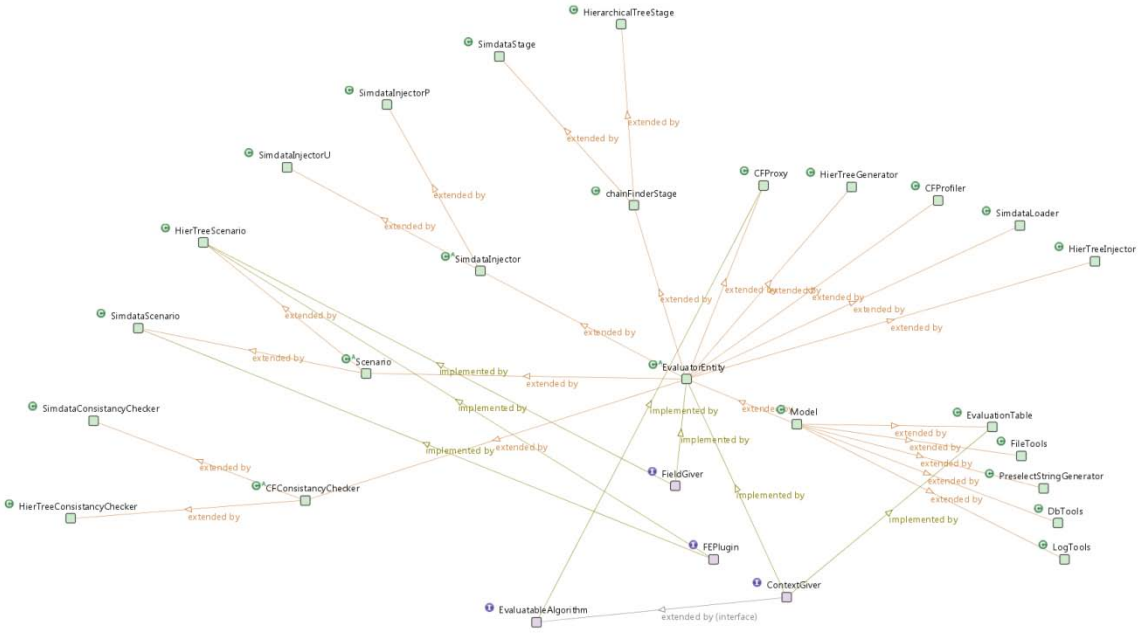


Abbildung 14: Überblick über die Klassenhierarchie der Workbench (Auszug)

Durch die Verwendung der Template-Engine entstand der Begriff des *Kontexts*. Im Kontext sind die Template-Variabeln mit ihren entsprechenden Werten gespeichert. Das entsprechende Interface nennt sich *ContextGiver*. Ein weiteres Konzept ist dasjenige eines *FieldGivers*. Die meisten Klassen wie Szenarien, Generatoren / Loader, Proxies, und Injektoren, die direkt mit einem Evaluationslauf zu tun haben, haben sowohl einen Kontext als auch das Verhalten eines *FieldGivers*. Haben sie letzteres bedeutet dies nichts anderes, als dass vorgesehen ist, dass sie Werte in die Evaluationstabelle schreiben. Es gibt noch eine Klasse *„EvaluatorEntity"*, welche den Umgang mit diesen beiden Verhaltensweisen (arbeiten mit Templates und die Vorhaltung von Werten für die Evaluationstabelle) vereinfachen. Sehr viele Klasse erben von *EvaluatorEntity*. Selbst wenn in der zukünftigen Verwendung der Workbench weitere Konzepte in derjenigen Kategorie entstehen würden, wo sich die Injektoren, Generatoren, etc. befinden, könnten diese wahrscheinlich davon profitieren, wenn sie in sie in ihrer Implementation von *EvaluatorEntity* erben.

Wenn dies nicht der Fall wäre, lohnt es sich vielleicht, eine Implementation des Interfaces *ContextGiver* oder *FieldGiver* in Betracht zu ziehen.

Die vorliegende Architektur ist relativ schlank, weil sie keine tiefen Klassenhierarchien benutzt, und flexibel, weil sie generell vermehrt Composition statt Inheritance („Vererbung") verwendet. Die Unterstützung von anderen Datenbankstandards ist durch die Verwendung von JDBC und dem Template-System gegeben, welches mit allen SQL-Dialekten umgehen könnte. Sollte jedoch ein komplett neuer Algorithmus, welcher einen anderen Datenbankstandard als DB2 nutzt, implementiert werden, könnte dies einen Mehraufwand bedeuten (da das Management der Datenbankverbindungen angepasst werden müsste).

Das Frontend bietet bewusst nur die wichtigsten Funktionalitäten. Im Entwicklungsprozess gab es aber durchaus auch Builds, welche eine aufwändigere GUI hatten. Was dabei auffiel, ist, dass sich ein schlechter Trade-Off entwickelte. Durch viele Indirektionen und generell eine zu hohe Komplexität entstanden relativ hohe Kosten durch eine erschwerte Einarbeitung, aber nur beschränkter Nutzen durch die interaktivere GUI. Denn je mehr Funktionsumfang das Frontend hatte, desto eher wurde es zu einem zweiten Eclipse, zu einer Neuerfindung des Rades. Deswegen wurden

die nicht essenziellen Features zurückgenommen und es wurde auch im Workflow auf „Composition“ gesetzt, sprich in diesem Kontext die Zusammenarbeit von mehreren Standardanwendungen.

4.7 Workflow

In diesem Kapitel werden zwei spezifische Aspekte des praktischen Umgangs mit dem ChainFinder und dessen Umgebung genauer erläutert. Das volle Verständnis dieser Ausführungen hat der Leser erst dann, wenn er mit den Grundfunktionen des Evaluations-Framework bekannt ist. Dieses befindet sich auf der mit der vorliegenden Arbeit mitgelieferten CD-ROM, die den kommentierten Programmcode enthält.

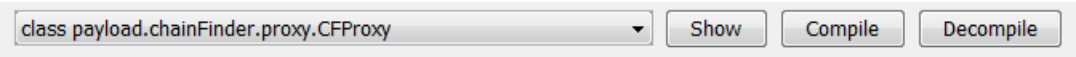
4.7.1 Implementation neuer Algorithmen und Szenarien

Um den schnellen Einstieg in die Arbeit mit der Workbench zu ermöglichen, ist bereits ein zweiter Algorithmus mit dem Namen „MiniMouse“ und ein minimales Evaluationssystem für ihn in die Workbench integriert. Ein künftiger Entwickler, welcher das Projekt übernehmen würde, wüsste also bereits, wo er ansetzen könnte.

Soll ein neues Szenario implementiert werden, lässt sich dieses als Plugin in die GUI einklinken. Dieser Mechanismus funktioniert automatisch. Leider wurden dem nächsten Verwender keine Hilfsmittel in die Hand gegeben, die eine einfache Erstellung des GUI Codes ermöglichen. Dieses Problem ist nicht existent, wenn der GUI-Builder Matisse oder ein vergleichbares Produkt und das Wissen zum Umgang damit zur Verfügung steht.

4.7.2 Erstellen und Debuggen von Templates

In Abschnitt „Tools“ im Evaluations-Framework hat es einen Browser, der die SQL-Templates im entsprechenden Verzeichnis auflistet. Beim Klick auf ein solches wird dieses in einem Editor geöffnet, welcher über ein speziell angepasstes Syntax-Higlighting verfügt. Diejenigen Abschnitte, welche dynamisch verändert werden können (*„Template-Variabeln“*) sind rot eingefärbt und beginnen mit einem Dollar-Zeichen ($). So ist die typische Bezeichnung für die Transaktionstabelle *$trxTable*. Im unteren Abschnitt befindet sich eine Auswahlliste („Dropdown“), welche die verschiedenen Klassen auflistet, welche effektiv mit Templates arbeiten. Jede dieser Klassen hat bestimmte Informationen in ihrem *Kontext*.



Dieser kann mit dem entsprechenden Button („show“) angezeigt werden. Die anderen beiden Schaltflächen ermöglichen die Kompilation, respektive die Dekompilation des momentanen Codes im Editor. Bei der Kompilation werden alle momentan vorhandenen Template-Variablen mit denjenigen Werten des Kontextes der gewählten Klasse in der Auswahlliste ersetzt. Umgekehrt werden bei der Dekompilation alle Zeichenketten im Editor-Fenster durch ihre entsprechenden Variablen im Kontext der gewählten Klasse ersetzt.

Hinweise:

- **Kein Save-Button:** Es wurde absichtlich keine Funktion zur Speicherung der Änderungen an einem Template in der Workbench implementiert. Eine solche Funktion wäre eine grosse Gefahr für die Integrität eines Evaluators. Die Überschreibung eines Templates mit seiner kompilierten Form würde gezwungenermassen zu einem Fehler bei der Ausführung des nächsten Laufes führen, sollten überhaupt Template-Variablen vorher existiert haben. Die Dekompilation könnte unter Umständen zu falschen Ergebnissen führen, wenn Werte innerhalb des Kontextes mehrfach benutzt werden. Zugleich könnten auch Stellen des

Textes im Editor fälschlicherweise mit Template-Variabeln überschrieben werden, wenn diese Werte nicht dieselbe Semantik haben im SQL.

- **Template-Debugging:** Wird eine Template-Variable in einem SQL-Template verwendet, welche aber nicht im Kontext der aufrufenden Klasse definiert ist, wirft die Datenbank normalerweise eine Exception. [33] Aber nicht immer generiert eine solche Unachtsamkeit semantisch oder syntaktisch ungültiges SQL. Passiert dies nicht, kann ein Durchlauf scheinbar erfolgreich sein. Der Editor kann vewendet werden, um nicht gesetzte Template-Variabeln zu erkennen. Dazu muss nur das entsprechende Template geöffnet werden und der richtige Klassenname im entsprechenden DropDown selektiert werden. Wenn nach einem Klick auf „compile" keine roten Bereiche mehr im Editor zu sehen sind, bedeutet dies, dass der Kontext alle für dieses Template nötigen Template-Variabeln beinhaltet. Ein Klick auf „decompile" zeigt an, wie viele Stellen der statischen Werte innerhalb des Editorfensters möglicherweise[34] durch ihre Entsprechungen im Kontext der jeweiligen Klasse ersetzt werden könnten.

4.7.3 Aufbereitung der Ergebnisse in Excel

Es bietet sich an, eine Datenbank-IDE zu verwenden, welche die Evaluations-Tabellen nach Microsoft-Excel exportieren kann (wie beispielsweise „TOAD for DB2" von Quest Software). Im Query-Fenster im Abschnitt „SQL-Explorer" des Frontends wird nach einem abgeschlossenen Lauf jeweils eine SQL-Query angezeigt, welche alle Felder in der entsprechenden Evaluationstabelle selektiert. Die Laufzeiten werden in Millisekunden gespeichert, die automatisch generierte Abfrage generiert jedoch noch derivative Spalten, welche dieselben Werte im Excel-internen Zeitformat repräsentieren.

Nach dem Import in Excel bietet sich die Verwendung von Pivot-Tabellen und –Diagrammen an. Diese aggregieren die Daten nach derselben Methode, wie es *„GROUP BY"*-Queries tun würden, jedoch bedeutend komfortabler und mit einfacherer Umwandlung in Diagramme.

Die einzige Aggregations-Funktion, welche nicht oder nur umständlich mit diesem Workflow (sprich in Excel unter der Verwendung der Pivot-Tabellen) zu nutzen wäre, und gleichzeitig naheliegend und interessant wäre, ist MEDIAN(). Doch bei der Verwendung von diesem ergeben sich zusätzliche Herausforderungen, welche an der folgenden Tabelle illustriert sein sollen:

ID	Runtime A („RA")	Runtime B („RB")	Runtime TOTAL (= RA + RB)
Lauf 1	5	3	8
Lauf 2	7	3	10
Lauf 3	5	4	9
A: Median über alle Felder	5	3	9
B1: Median über ein Feld - DB (Lösung 1)	5 (Median)	3	8
B2: Median über ein Feld - DB (Lösung 2)	5 (Median)	2	10
C: Median über ein Feld, Durchschnitt über alle anderen	5 (Median)	2.66	9
D: Median über DB, Durchschnitt bei den anderen Feldern über diejenigen ID's, welche denselben Eintrag im Median-Feld haben.	5 (Median)	3.5	8.5

[33] Doch bevor diese geworfen wird, gelangt die gesendete Query noch in ihrer kompilieren Form auf die Error-Konsole. Ist darin noch eine nicht ersetzte Template Variable enthalten (erkennbar durch ein führendes Dollar-Zeichen), deutet das auf die Ursache des Fehlers hin.

[34] Bevor sie wirklich ersetzt werden, sollte sichergegangen werden, dass die Semantik des SQL durch die Ersetzung nicht verändert wird. Vor allem wenn mehrere Template-Variabeln denselben Wert enthalten, ist diese Prüfung wichtig.

Die grünen Felder sind nicht Teil des ursprünglichen Datensets. Der aufgezeigte Fall ist recht hypothetisch und sollte nur zeigen, welche Arten es unter anderem gäbe, den Median zur Aggregierung unserer Evaluationstabellen zu verwenden. Es soll nicht bewertet werden, welche Methode robuster ist oder die Semantik realer Messergebnisse besser erhält. Allerdings kann gesagt werden, das A und D wahrscheinlich schwierig zu implementieren sind in der momentanen Architektur, respektive dem momentanen Workflow (DB2 + Java + Excel). Sind künftige Analysen unter Zuhilfenahme derartiger Aggregationen gewünscht, müsste vielleicht Excel durch ein dediziertes Statistikprogramm ersetzt werden.

Der Fall B wäre noch am leichtesten in SQL zu implementieren. Leider bietet DB2 auch keine Median-Funktion, diese kann aber dank der OLAP-Funktion „row_number()" relativ einfach nachgebaut werden.

```
SELECT * FROM
  (SELECT row_number() over (partition by MULTIPLICATOR order by RT_TOT) as RN,
  t1.*
  from EVAL.UNIFORM_2_1_0_10_1_3 t1)
WHERE RN = 2
```

Hier ist das Median-Feld *RT_TOT* und es gibt in den Daten jeweils 3 Zeilen für jeden getesteten *MULTIPLICATOR*. Wäre diese Anzahl Zeilen gerade, wäre die Implementierung schwieriger. Aber in der Regel lässt sich dieses Problem umschiffen, indem einfach eine ungerade Anzahl von „Runs" im Szenario gewählt wird:

Auf der anderen Seite könnte man das auch in Java lösen, wobei allerdings dazu zu sagen ist, dass die Workbench vom Design her eigentlich nicht darauf ausgelegt ist, die Aggregationen selbst zu machen.

5 Diskussion

Praxisbezug

In der Einleitung wurde aufgezeigt, wie die Relevanz von automatisierten Suchverfahren nach betrügerischen Transaktionen in der Realität sein könnte. Ebenfalls wurden unternehmenspolitische Aspekte angeschnitten und es wurde gefragt, welche Implikationen solche Betrugsfahndungsmethoden für einzelne Wirtschaftsteilnehmer oder die Gesellschaft hat.

Der ChainFinder ist ein Algorithmus, der relativ einfache Muster findet. Die Qualität der Ergebnisse lässt sich relativ leicht evaluieren, weswegen sich sein Beispiel gut eignete, um Konzepte zur Evaluation von Pattern-Matcher zu entwickeln. Im Verlauf der Arbeit wurden Szenarien entworfen, um die Performance und Korrektheit der Resultate des ChainFinders zu testen. Eines davon ist praxisnah gehalten (siehe Kapitel 4.4.2), ein anderes ist vollkommen synthetisch (siehe Kapitel 4.4.1). Die Datenstruktur und die Muster, nach denen gesucht wurden, haben Praxisbezug. Auch die Plattformen (DB2 und Java) sind so in der realen Welt zu finden. Allerdings wurden nur „relativ" kleine Datensets gestestet mit einigen Millionen Transaktionen. Es gab auch keine Einschränkungen, welche Funktionen von DB2 benutzt werden durften und welche nicht. In der Praxis könnte das durchaus anders sein.

Umfang der Tests

Der ChainFinder wurde nicht in allen seinen möglichen Einstellungen getestet. Ebenso wurden in der Problembeschreibung viele weitere Ebenen aufgezeigt, auf denen ein Algorithmus zur Betrugserkennung oder ganz allgemein ein beliebiger Algorithmus evaluiert werden könnte.

Wahrscheinlich bringt es für den Autor eines Algorithmus mehr, wenn er ein einfach zu erlernendes und flexibles Evaluationssystem zur Verfügung hat, als wenn er die Evaluation an eine dritte Partei weitergibt. Der nötige Wissenstransfer stellt einen unverhältnismässigen Aufwand dar. Zudem müssen Informationen über die internen Abläufe im Betrieb preisgegeben werden, was den geschäftlichen Geheimhaltungsinteressen zuwiderläuft.

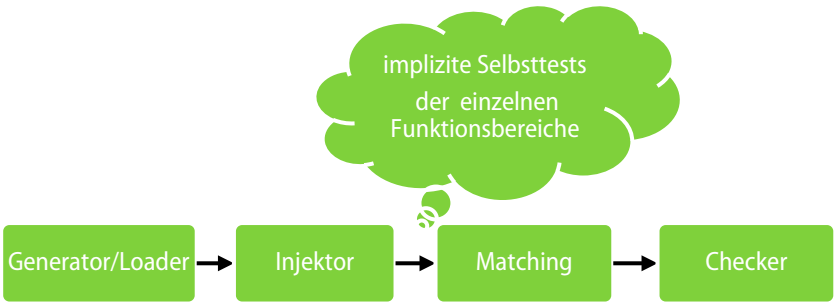
Die durchgeführten Tests reichten aber aus, um Konzepte eines Evaluators oder eines Frameworks für die Evaluation und die Evolution von beliebigen Algorithmen zur Betrugserkennung herauszuarbeiten und sie nachher in Code zu giessen. Zu diesem Zeitpunkt wäre es für den Autor relativ einfach, die ausgeschlossenen Evaluationen doch noch durchzuführen. Beispielsweise könnte der Frage nachgegangen werden, wie hoch denn der Performance-Verlust durch die Messung der Performance ist.

Für eine zukünftige Verwendung müsste die Frage beantwortet werden, ob die Funktionalität zum Selbsttest Bestandteil des Frameworks sein soll, oder eine andere Lösung angebracht wäre. Wenn ja, würde dies heissen, dass eine nicht essenzielle Funktionalität die Komplexität erhöht und somit Anpassung des Frameworks auf einen neuen Algorithmus erschwert würde. Wäre das Framework anhand zweier oder mehrerer Algorithmen entwickelt worden, wäre das Ergebnis (das Framework) wahrscheinlich für weitere ähnliche Algorithmen besser geeignet.

Das Framework veranschaulicht die entwickelten Konzepte gut, wenn man die Klassenhierarchie betrachtet, da beinahe jedes der in dieser Arbeit beschriebenen Konzepte durch eine Klasse repräsentiert wird (Vgl. die erarbeiteten Konzepte von Kapitel 4.3 und 4.6). An vielen Stellen wurden Automatismen eingeführt, welche die Wartbarkeit der Codes erhöhen sollten. Ausserdem setzt die ganze Framework sehr stark auf Composition und nicht (mehr) auf Ver-

erbung. Dadurch wird eine relativ gute Isolation der Funktionsbereiche ermöglicht und um eine Änderung oder Erweiterung durchführen zu können, müssen nicht mehr allzu viele Bereiche des Codes verstanden werden.

Während der Entwicklung des Evaluationssystems wurde die Erkenntnis gemacht, dass eine gegenseitige Abhängigkeit zwischen Teilen des Evaluationssystems (Generator, Injector, Checker) und des ChainFinders besteht. Das ganze Evaluationssystem testet sich sozusagen stetig selbst. Funktioniert ein Teil dieser Kette nicht, wie er sollte, besteht eine gute Chance, dass man am Ende des Laufs Informationen zur Verfügung hat, die bei der Fehlersuche helfen.



Ausser dem genannten impliziten Test fanden noch manuelle Tests während der Entwicklung der Injektoren und Generatoren statt. Mir dem geschaffenen Framework wäre es sehr leicht, die genannten Funktionsbereiche zu evaluieren und evolvieren, da dabei die genau gleichen Methoden zum Einsatz kämen, die über weite Teile dieser Arbeit erarbeitet konzeptionell entwickelt, verwendet und implementiert wurden.

Java-ChainFinder vs. SQL-ChainFinder

Im Verlauf dieser Arbeit wurde das Konzept der Candidate Selection zur Optimierung des Graph-Pattern-Matchings aufgezeigt und eine entsprechende Methode für den Java-ChainFinder implementiert. Sie schränkte die für ihn zu verarbeitende Datenmenge ein und hatte die Form einer einzigen SQL-Query. Genauer gesagt handelte es sich um einen Join der Transaktionsdatentabelle mit sich selbst (mit Nebenbedingungen bezüglich des Transaktionsdatums und des Betrags). Es wurde jeder Eintrag mit jedem Eintrag maximal einmal abgeglichen und dabei alle Konten gefunden, die mindestens einer Zweierkette angehören. Wird nun der ChainFinder schneller durch die optimierte Candidate Selection, stellt sich die Frage, inwieweit es vorteilhaft ist, seine Funktion in SQL zu implementieren. Um diese Frage zu beantworten, müssen allerdings noch weitere Aspekte in Betrag gezogen werden.

Die durch den Verfasser der Arbeit entwickelten Injektoren arbeiteten nach dem Prinzip, zufällige Konten als Roots auszuwählen und ihnen dann eine Transaktion anzuhängen, welche den Root durch ihre Charakteristiken zu einem Element einer Zweierkette macht. Dieser Prozess lässt sich in ähnlicher Form iterativ fortführen, um die Ketten wachsen zu lassen. Diese Methode skalierte gut genug für Datensets mit Millionen von Transaktionen.

Nun stellt sich die Frage, ob die durch die Candidate Selection ermittelten Zweierketten nicht nach demselben Verfahren auf potenziell längere Ketten erweitert werden könnten. Dazu stellt man sich den ChainFinder als einen umgekehrten Injektor vor. Die bei der Candidate Selection gefundenen Ketten werden als Resultate betrachtet, die es zu „verlängern“ gilt. Der Vorteil an einem solchen Ansatz ist, dass alle Ketten gefunden werden und nur die jeweils längsten Ketten ausgehend vom Rootset in der Treffermenge erscheinen. Es werden mögliche Verlängerungen für die bereits gefundenen Ketten gesucht.

Um die gefundenen Zweierketten danach zu überprüfen, ob sie Teil einer Dreierkette sind, ist weniger Aufwand nötig als für die Candidate Selection. Eine Zweierkette besteht aus drei Konten. Nur für das Letzte müsste überprüft werden, ob es eine Transaktion mit entsprechenden Charakteristiken gibt, welche die Zweierkette zu einer gültigen Dreierkette machen würde. Der nötige Join müsste nun aber weniger Reihen miteinander vergleichen (nämlich eben nur die letzten Konten in der Zweierkette mit der Transaktionsdatentabelle). Der Grund dafür liegt darin, dass davon auszugehen ist, dass weniger Zweierketten existieren als Transaktionen. Nun gibt es auch zwingend weniger Dreierketten als Zweierketten, was impliziert, dass jede weitere Iteration dieses Verfahrens zwingend weniger teuer ist als die Vorhergehende.

Macht nun also die Candidate Selection den Java-ChainFinder bedeutend schneller, stellt sich die Frage, wie gross ihr Beitrag zur Lösung des Problems ist. Die Tatsache, dass sie den ChainFinder in der Baseline-Analyse mindestens 10x schneller gemacht hat, wirft zudem die Frage auf, ob der Java-ChainFinder nicht möglicherweise durch den skizzierten SQL-ChainFinder ganz ersetzt werden könnte.

Durch die erfolgreiche Implementierung des SQL-ChainFinders und die hier erläuterte Analyse wurde klar, dass die Candidate Selection eigentlich nicht bloss eine simple Vorfiltrierung der Datenbasis ist, sondern eigentlich schon das Grundproblem zum grössten Teil selber löst. Daraus lässt sich schliessen, dass entweder die Candidate Selection in einem gewissen Umfeld funktioniert womit dann der Java-ChainFinder durch den skizzierten SQL-ChainFinder ersetzt werden kann (weil dieser viel schneller ist). Oder aber sie lässt sich nicht in angemessener Zeit durchführen, womit ist sie selbst obsolet würde und damit auch gar keine Optimierung für den Java-ChainFinder darstellt. In den getesteten Szenarien ist die SQL-Implementation des ChainFinders überlegen. Im Rahmen der getesteten Szenarien dieser Arbeit müsste also gesagt werden, dass der Java-ChainFinder obsolet ist, weil die Candidate Selection den Java-ChainFinder nie langsamer machte.

Allerdings wurde dabei nicht zwingend an künftige Versionen des ChainFinders gedacht. Diese könnten Funktionalitäten bieten, welche gar nicht oder nur sehr schwierig in SQL zu implementieren sind. Der Autor nimmt an, dass sich der gesamte Funktionsumfang des gegebenen ChainFinders in allen Szenarien auch in der SQL-Implementierung zur Verfügung stehen würde und sogar in allen oben skizzierten Testanordnungen der Java-Implementation überlegen war. Das Ziel dieser Arbeit war es, den gegebenen ChainFinders zu evaluieren und zu evolvieren. Am Ende dieser Arbeit hat er sich jedoch dahingehend evolviert, dass er gar nicht mehr existiert, sondern nur noch seine Abbildung in SQL.

In der Praxis jedoch ist vielleicht die Anpassungsfähigkeit oder der schonende Umgang mit Datenbankressourcen wichtiger. Allerdings sind die Injektoren und Generatoren komplexer als der SQL-ChainFinder und könnten dann nicht in dieser Form verwendet werden, wenn dieser das nicht kann.

6 Future Work

Registrator Track

Während der ganzen Tests wurde die bisher unerwähnt gebliebene Funktion *„Registrator-Track"* des Java-ChainFinders nicht verwendet. Sie schien nicht zu funktionieren, der Grund für diese Tatsache wurde nicht ermittelt.

Wenn diese aktiv und funktionstüchtig ist, müssten alle Transaktionen einer gefundenen Kette vom gleichen Sachbearbeiter erfasst werden. Durch diese zusätzliche Einschränkung dürfte die Candidate Selection und auch der SQL-ChainFinder deutlich schneller sein, da bei der Abarbeitung des Joins potenziell weniger Reihen miteinander verglichen werden müssen. Es kann auch gesagt werden, dass der Suchraum durch diese strenge Anforderung an das Ergebnis viel kleiner sein dürfte. Wenn sich der Einsatz der SQL-Version des ChainFinders ohne aktivierten *Registrator-Track* in der Praxis wegen einer zu langen Laufzeit der Queries verbietet, so könnte man durch eine Aktivierung dieser Funktion diesem Problem evtl. Abhilfe verschaffen.

Subgraph Count

In einem Graph-Transaction-Setting macht die Erhebung der Anzahl der autonomen Subgraphen die Performance-Metrik „Subgraph Count vs. Datensetgrösse" möglich. Sie könnte besser verwertbare Ergebnisse liefern als „Laufzeit vs. Datensetgrösse". Beim Verfassen dieser Arbeit wurde eine Methode entwickelt, um schnell herauszufinden, wie viele Subgraphen sich mindestens im Datenset befinden.

Die Existenz eines Kontos ist genau dann gegeben, wenn es in einer Transaktion entweder in der Rolle des Absenders oder Empfängers auftritt. Das bedeutet wiederum, dass wenn ein Konto existiert und keine eingehende Transaktion hat, muss es zwingend mindestens eine abgehende Transaktion haben. Für jedes existierende Konto ohne eingehende Transaktion ist die Existenz eines autonomen Subgraphen im Datenset sicher. Dies ist dadurch zu begründen, dass jeder nur erdenkliche Subgraph, welcher dieses Konto enthält, nur andere Konten umfassen kann, welche mindestens eine eingehende Transaktion haben. Allerdings können auch Subgraphen existieren, die kein Konto ohne eingehende Transaktion enthalten, und somit mit der eben ausgeführten Methode nicht erkannt werden können.

Vielleicht könnte dieser Ansatz in dieser oder in einer weiter entwickelten Form bei der Evaluation von Graph-Pattern-Matcher nützlich sein.

Robustere Performance-Messungen innerhalb eines gegebenen Szenarios

Vielleicht gibt es eine bessere Lösung, als in jedem Szenario die Parameter stufenweise zu steigern und mit diesen Einstellungen einen oder mehrere Läufe durchzuführen. Nehmen wir ein sehr einfaches Szenario an: Es gibt Hintergrunddaten, welche sich nie ändern. Zuerst werden drei Läufe durchgeführt ohne eine zusätzliche Kette einzufügen, dann werden wieder drei Läufe auf den Hintergrunddaten mit einer hinzugefügten Kette durchgeführt, etc.

Nehmen wir an, das Szenario sieht ein Ende der Tests nach dem dritten Lauf mit 10 hinzugefügten Ketten vor. Insgesamt würden also 33 Läufe in 11 Konfigurationen durchgeführt.

Während eines bestimmten Laufs kann beispielsweise auf einer an der Evaluation beteiligten Maschinen erhöhte Rechenlast aus unbekannter Quelle anfallen. Dies äussert sich in den Performance-Kennzahlen zum entsprechenden

Lauf (er braucht länger, als er eigentlich sollte). Wenn nur ein Lauf einer bestimmten Konfiguration betroffen ist, würde dieser wenig Einfluss auf die Ergebnisse nach Aggregation mit der Median-Funktion haben.

Nehmen wir nun aber an, dass ein an der Evaluation beteiligter Rechner über eine längere Zeit (sagen wir über einen Viertel der Laufzeit des Szenarios) einer mittelstarken zusätzlichen Rechenlast aus unbekannter Quelle ausgesetzt ist. Da nun die drei Läufe mit einer gewissen Zahl von hinzugefügten Ketten jeweils hintereinander ablaufen, sind alle langsamer als sie sein sollten. Die Verwendung des Medians könnte dies nicht korrigieren. Es wäre jedoch möglich, diesen mittels Visualisierung und anschliessender optischer Inspektion der Performance-Daten aller Läufe des Szenarios zu erkennen. Wären nun aber die 33 Läufe des Szenarios in zufälliger anstatt ansteigender Abfolge durchgeführt worden, liesse sich der Fehler durch die Verwendung einer Aggregation mithilfe einer Median-Funktion korrigieren.

Bei der weiteren Verwertung des Frameworks könnte es vielleicht dahingehend erweitert werden, dass es die erläuterte Randomisierung der Läufe unterstützt.

Die Frage nach einer Beschreibungssprache

Es ist in den vorigen Kapiteln aufgefallen, dass die Notwendigkeit von geteilten Informationen über die Schnittstellen von I_A zu I_E und der Datenbank eine grosse Herausforderung an das Design eines Evaluators oder eines Frameworks zur Evaluation von multiplen Algorithmen und Datenbeständen stellt. Vielleicht sind weitere Forschungen sinnvoll, die auf eine Beschreibungssprache herauslaufen. So wären dann beispielsweise die zu generierenden Patterns in ihrer Struktur, die Informationen über *Pcontext* und die Schnittstellen in einem XML Dialekt spezifiziert. Dann müsste nur noch zugesichert werden, dass ein Algorithmus in Java den Interfaces in XML genügen müsste, wobei die Konfigurationseinstellungen, die Szenarien und die Plausibilitätsprüfungen dann ebenfalls dort (in XML) definiert wären. Ein solches Framework müsste dann die Definitionen nur noch in Code umsetzten – die I_E würde also weitestgehend die Charakteristik von interpretiertem Code haben. Die dadurch entstehenden Performance-Nachteile wären im Kontext der Konsistenz-Evaluation nur von begrenzter Bedeutung.

Geht es um die Performance-Evaluation, müsste die I_A wohl mit Vorteil ihre Daten selbst sammeln. Es könnte jedoch auch ein Interface mit Aussagen über Performance-Indikatoren, semantische Zusammenhänge, Statistiken über die Treffermenge respektive des Datensets und den Einstellungen in P_{query} und $P_{restrict}$ im XML definiert werden. Der Vorteil einer solchen „all things for all man"-Lösung wäre die die komplette Abstraktion von der Strategie. Die benötigten Informationen bezüglich Semantik und Kontext wären komplett in die XML-Datei isoliert (ebenfalls die Szenarien und Metainformationen über die Generatoren, Injektoren).

Der Evaluator lernt über sich selbst?

Wenn eine oder mehrere Konsistenzprüfungen fehlschlagen, dann könnte das Evaluationssystem versuchen, daraus zu lernen. Es müsste jedoch zu diesem Zweck den Wertebereich der Parameter haben und könnte dann durch *Machine Learning* selbst Optima für diese finden. Vielleicht wäre es noch nötig, die Resultate einmal manuell nach guten und schlechten Matches im Kontext der Praxis zu beurteilen. Wären solche Auswertungen vorhanden, würde dies dem Administrator des ChainFinders helfen, die Einstellungen in $P_{restrict}$ besser zu setzen.

Ein weiterer Punkt ist, dass ein Evaluator selbst seinen Einfluss auf die erhobenen Messwerte einzuschätzen hätte. Ein solcher Selbsttest setzt wahrscheinlich voraus, dass der Evaluator selbst seine Bestandteile gezielt an und abschalten

kann. Er müsste ebenfalls wissen in, wie weit er seine Fähigkeit verliert, gewisse Kennzahlen zu Performance und Korrektheit zu erheben, wenn er sich weniger in den ganzen Prozess der Inspektion einmischt.

7 Referenzen

Bolton, R., & David. (2002). Statistical fraud detection: A review. *Statistical Science*, *17*. Retrieved January 20, 2009, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.9299.

Gallagher, B. (2006). Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching (pp. 45-53).

Holder, L. B., Cook, D. J., & Djoko, S. (1994). Substructure discovery in the subdue system. *In Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, 169--180. doi: 10.1.1.44.6602.

Jiefeng Cheng, Yu, J., Bolin Ding, Yu, P., & Haixun Wang. (2008). Fast Graph Pattern Matching. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on* (pp. 913-922). doi: 10.1109/ICDE.2008.4497500.

Luo, B., & Hancock, E. (2001). Structural graph matching using the EM algorithm and singular value decomposition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *23*(10), 1136, 1120.

Makhoul, J., Kubala, F., Schwartz, R., & Weischedel, R. (1999). Performance measures for information extraction. *In Proceedings of DARPA Broadcast News Workshop*, 249--252. doi: 10.1.1.27.4637.

Obler, J. K., Oning, U. S., & An, J. T. (1992). Graph isomorphism is low for PP. *Comput. Complexity*, *2*, 301--330. doi: 10.1.1.26.685.

Park, B. G., Lee, K. M., Lee, S. U., & Lee, J. H. (2003). Recognition of partially occluded objects using probabilistic ARG (attributed relational graph)-based matching. *Computer Vision and Image Understanding*, *90*(3), 217-241. doi: 10.1016/S1077-3142(03)00049-3.

Porter, D. (2003). Insider Fraud: Spotting The Wolf In Sheep's Clothing. *Computer Fraud & Security*, *2003*(4), 12-15. doi: 10.1016/S1361-3723(03)04011-9.

Tong, H., Faloutsos, C., Gallagher, B., & Eliassi-Rad, T. (2007). Fast best-effort pattern matching in large attributed graphs (pp. 746, 737). ACM. Retrieved January 26, 2009, from http://dx.doi.org/10.1145/1281192.1281271.

Tsai, W., & Fu, K. (1979). Error-Correcting Isomorphisms of Attributed Relational Graphs for Pattern Analysis. *Systems, Man and Cybernetics, IEEE Transactions on*, *9*(12), 757-768. doi: 10.1109/TSMC.1979.4310127.

Ullmann, J. R. (1976). An Algorithm for Subgraph Isomorphism. *J. ACM*, *23*(1), 31-42. doi: 10.1145/321921.321925.

Valiente, G., & Martinez, C. (1997). An Algorithm for Graph Pattern-Matching. In International Informatics Series (Vol. 8, pp. 180–197). Carleton University Press. Retrieved from http://www.lsi.upc.es/ valiente/abs-wsp-1997.pdf.

Washio, T., & Motoda, H. (2003). State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, *5*(1), 59-68. doi: 10.1145/959242.959249.

Erklärung

.

Der Verfasser versichert, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

.....	.....
Nänikon, 5. Mai 2009	Stefan Amstein