Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jörn Slotta

Vergleich von Algorithmen zur Assoziationsanalyse basierend auf Webserver Logfiles

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jörn Slotta

Vergleich von Algorithmen zur Assoziationsanalyse basierend auf Webserver Logfiles

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. sc. pol. Gerken
Zweitgutachter: Prof. Dr.-Ing. Hübner

Eingereicht am: 23. April 2012

Jörn Slotta

Thema der Arbeit
Vergleich von Algorithmen zur Assoziationsanalyse basierend auf Webserver Logfiles

Stichworte
Assoziationsanalyse, Apriori, CSV, FP-Growth, FP-Tree, Logfile

Kurzzusammenfassung
Im Verlauf der Bachelorarbeit wird ein Programm zur Analyse von Webserver Logfiles erstellt. Diese werden im ersten Schritt zu CSV-Dateien umgewandelt und dann weiterverwendet. Als Analyseverfahren wird die Assoziationsanalyse genutzt. Innerhalb der Assoziationsanalyse werden mehrere Algorithmen angewendet, welche miteinander verglichen werden. Außerdem wird beschrieben, wie die Assoziationsanalyse funktioniert, was die Werte „Support“ und „Konfidenz“ bedeuten und wofür diese benutzt werden. Als Datenquelle dient ein Webserver Logfile des Departments Informatik der HAW-Hamburg. Desweiteren werden die Ergebnisse dargestellt und ausgewertet.

Jörn Slotta

Title of the paper
Comparing algorithms for the association analysis based on webserver logfiles

Keywords
association analysis, Apriori , CSV, FP-Growth, FP-Tree, Logfile

Abstract
In the process of this bachelor’s thesis, a program is developed to analyze webserver logfiles. In the first step, these will be transformed to CSV-data for further use. For the analytical method, an association analaysis is used. Within the association analysis, several algorithms are applied and compared with each other. Furthermore, this work describes how the association analysis works, what the measures "support" and "confidence" signify, and what they are applied for. The data source is a webserver logfile from the Department of Computer Science of the Hamburg University of Applied Sciences. Lastly, the results will be presented and evaluated.

Inhaltsverzeichnis

1. Einleitung

In Zeiten, in denen jeder Artikel mit einem EAN-Code versehen ist und die meisten Geschäfte über Scannerkassen verfügen, ist das Einkaufen und speziell das Bezahlen für den Kunden eine Erleichterung. Doch nicht nur der Kunde profitiert von diesem System. Die Scannerkassen sind in der Regel an größere Systeme mit Datenbanken angebunden, wodurch die Möglichkeit besteht, alle Einkäufe (Transaktionen) vollkommen automatisiert zu speichern. Diese gesammelten Daten können nun mithilfe der Assoziationsalanyse ausgewertet und ihre Zusammenhänge analysiert werden. So entstehen Regeln wie: „80% aller Kunden, die Windeln kaufen, kaufen auch Bier“. Dieser Vorgang wird auch Warenkorbanalyse, dem häufigsten Einsatzgebiet der Assoziationsanalyse, genannt. In Kapitel 2.1 wird näher auf die Assoziationsanalyse eingegangen.
Mithilfe der entstandenen Regeln hat das Geschäft unter anderem die Möglichkeiten, Laufwege zu optimieren und spezielle Werbung oder Aktionen zu schalten (Ester und Sander, 2000, S. 159f.). Das Prinzip ist auch auf die digitale Welt übertragbar, hier können Kunden personalisierte Werbung oder, bei der Betrachtung eines Artikels, das Kaufverhalten anderer Kunden angezeigt bekommen („Kunden, die dieses Produkt gekauft haben, kauften auch [...]“). Somit werden die Kunden indirekt angeregt, weitere Artikel zu erwerben, ohne dass diese „nutzlos“ erscheinen. Denn beim Kauf eines Druckers ist die Wahrscheinlichkeit, dass Druckerpapier vorgeschlagen wird, größer als die Wahrscheinlichkeit für die Empfehlung von Erdbeermarmelade.

1.1. Motivation

Die Assoziationsanalyse ist nicht nur für eine Warenkorbanalyse nützlich. Sie kann, wie im Fall dieser Arbeit, auch auf einen Weblog angewendet werden. Die verschiedenen Seitenaufrufe einer Sitzung sind in diesem Fall (wie schon bei den Einkäufen) eine Transaktion. So können Zusammenhänge zwischen Seitenaufrufen erstellt werden. Durch diese Zusammenhänge kann zum Beispiel die Notwendigkeit erkannt werden, spezielle Hyperlinks hinzuzufügen, damit der Anwender nicht nur über Umwege auf eine Seite gelangt (vergleichbar mit der Laufwegoptimierung im Geschäft).

1.2. Zielsetzung

Ziel der Arbeit ist es, die Assoziationsanalyse auf ein Webserver Logfile anzuwenden. Hierfür wird dieses zuerst als CSV-Datei (siehe Kapitel 1.5) gespeichert, um im nächsten Schritt auf die CSV-Datei sowohl den Apriori-Algorithmus wie auch den FP-Growth Algorithmus anzuwenden. Für das entwickelte Programm soll es irrelevant sein, ob in der CSV-Datei Seitenaufrufe oder Einkäufe stehen. Unabhängig vom Inhalt sind die Daten vergleichbar und können, solange die CSV-Datei wie vorgegeben formatiert ist, analysiert werden. Ein weiteres Ziel dieser Arbeit ist es, die beiden Algorithmen aus Sicht der Effizienz zu vergleichen und diesen Vergleich auszuwerten.

1.3. Abgrenzungen

In dieser Arbeit werden nur die beiden Algorithmen „Apriori“ und „FP-Growth“ implementiert und verglichen (Vorstellung der Algorithmen in Kapitel 2.2). Alle anderen existierenden Algorithmen für eine performante Assoziationsanalyse werden nicht berücksichtigt, da dies den Rahmen einer Bachelorarbeit überschreiten würde.

1.4. Aufbau der Arbeit

Die Arbeit wird mit einer kurzen Einführung in das CSV-Dateiformat, welches in dieser Arbeit für den Import der Datensätze benutzt wird, und der Beschreibung eines Weblogs, der als Datenbasis dient, begonnen. Darauffolgend wird in die Grundlagen der Assoziationsanalyse eingeführt. In dem Zuge wird die Analyse kurz formal beschrieben sowie die beiden Werte „Support“ und „Konfidenz“ dargestellt. Desweiteren wird ein allgemeiner Ablauf dieser geliefert und die beiden in dieser Arbeit verwendeten Assoziationsalgorithmen erläutert. Im anschließenden Hauptteil werden das Design der Applikation und die einzelnen Implementierungsschritte beschrieben. Außerdem werden die beiden Algorithmen auf die Transkaktionen des Wegblogs angewendet und verglichen. Im nachfolgenden Schlussteil wird das Ergebnis des Vergleiches ausgewertet und ein Fazit in Hinblick auf die Zielsetzung der Arbeit gezogen.

1.5. CSV

CSV steht für „Comma-Separated Values“ und ist ein Dateiformat. Es wird oft dort angewandt, wo eine Tabelle in Textform benötigt wird. Für das Format CSV gibt es keinen allgemeinen Standard. Dennoch sind verschiedene Spezifikationen für CSV vorhanden. Das hat zur Folge,

dass CSV-Dateien sehr unterschiedlich interpretiert werden können. Das Trennzeichen, dass die Spalten voneinander separiert, kann zum Beispiel unterschiedlich definiert sein. Besagtes Trennzeichen war in der Grundidee von CSV, wie der Name vermuten lässt, das Komma. Mittlerweile gibt es allerdings auch diverse Implementierungen mit anderen Trennzeichen, wie zum Beispiel Semikolon, Leerzeichen, Tabulator oder Doppelpunkt.
Soll nun eine Tabelle oder Datenbank als CSV-Datei exportiert und später (womöglich auch von einer anderen Anwendung) wieder importiert werden, muss die Formatierung genau festgelegt sein. Zur Verdeutlichung dient als Spalteninhalt ein Datumsfeld „02/03/04“, dieses kann ohne Festlegung der Formatierung wie folgt interpretiert werden:

- 02.03.2004 (u.a. die in Deutschland übliche Schreibweise [1])
- February 03, 2004 (u.a. die in den USA übliche Schreibweise)
- 2002-03-04 (u.a. die in China übliche Schreibweise [2])

Auf das zusätzliche Problem, dass nicht ersichtlich ist, um welches Jahrhundert es sich handelt, (2000, 1900, 1800, ...) wird nicht weiter eingegangen.
In der Regel ist der interne Aufbau einer solchen CSV-Datei immer ähnlich. Vergleicht man ihn mit dem Aufbau einer Tabelle, so existiert je Tabellenzeile auch eine Zeile in der CSV-Datei. In dieser Zeile stehen die Einträge aus dem jeweiligen Feld der Tabelle. Die Trennung der einzelnen Felder wird, wie oben bereits erwähnt, mit dem definierten Trennzeichen bewirkt. Jede Zeile wird mit einem Zeilenumbruch beendet (siehe Abbildung 1.1).

Abbildung 1.1.: Abbildung einer Tabelle auf eine beispielhafte CSV-Datei

Optional stehen in der ersten Zeile einer CSV-Datei (vergleichbar mit einer Tabelle) die Spaltennamen. Außerdem sollte jeder Datensatz (jede Zeile) die gleiche Anzahl an Spalten beinhalten, dies wird jedoch nicht immer eingehalten (Shafranovich, 2005).

[1] nach DIN 1355-1
[2] nach ISO 8601

1.6. Weblog

Unter Weblog wird in dieser Arbeit die Protokollierung von Aktionen eines Webservers bezeichnet, somit ist nicht der sogenannte „Blog“ gemeint, der ebenfalls unter dem Namen Web-Log bekannt ist.
In einer solchen Log-Datei eines Webservers können unterschiedliche Informationen protokolliert werden. Grundsätzlich wird für jede vom Webserver erhaltene Anfrage eine Zeile in das Weblog geschrieben. Der Aufbau einer solchen Zeile ist nicht immer einheitlich vorgegeben, allerdings besteht der Aufbau in der Regel aus Standardelementen. Der Log kann unter anderem folgende Daten beinhalten:

- IP-Adresse des Benutzers
- Dazugehöriger Zeitstempel
- Anfrage des Benutzers
- Dateigröße in Bytes
- Benutzter Browser
- Benutztes Betriebssystem
- ...

(Heindl, 2003, S.101f.)
So auch auf dem Webserver des Departments Informatik der HAW-Hamburg, auf dem ebenfalls jede Anfrage im Weblog gespeichert wird. Unter anderem sind dies Anfragen auf ein auf der Seite vorhandenes Element, wobei hier Bilder (sowohl animiert als auch nicht animiert), Stylesheets oder auch Javascripte als Element bezeichnet werden. Durch die Speicherung jeder Anfrage entsteht die Problematik, dass das Log mehrere Datensätze je Seitenaufruf beinhaltet. Die Daten des Weblogs sind in mehrere Spalten eingeteilt, wobei zu beachten ist, dass in dieser Arbeit nur die Spalten der IP-Adresse, der Zeitstempel und die Adresse der angeforderten Seite relevant sind. Alle anderen Spalten werden für die Assoziationsanalyse nicht benötigt und im Laufe der Implementierung entfernt. Da es sich um personenbezogene Daten handelt, werden IP-Adressen unter Berücksichtigung des Zeitstempels zusammengefasst, um den Datenschutz zu wahren. Beim Zusammenfassen werden IP-Adresse und Zeitstempel durch eine eindeutige Transaktionsnummer ersetzt. Hierbei gilt:
Seitenaufrufe von derselben IP werden zu einer Transaktion zusammengefasst, wenn seit dem

letzten Aufruf weniger als dreißig Minuten vergangen sein sollten. Sind mehr als dreißig Minuten seit der letzten Anfrage vergangen, wird eine neue Transaktion erstellt. Duplikate werden ignoriert, dadurch werden Mehrfacheinträge zu einem Eintrag zusammengefasst. Durch das Entfernen der nicht relevanten Spalten beinhalten die Datensätze der einzelnen Elemente eines Seitenaufrufs die gleichen Werte und werden durch die Ignoranz von Duplikaten nicht weiter beachtet. Somit ist die oben genannte Problematik, dass jedes Element einer Seite einen Eintrag im Weblog hat, nicht mehr gegeben. Dieses Vorgehen wird in Kapitel 3.3.2 weiter ausgeführt.

2. Grundlagen

Um ein grundsätzliches Verständnis für die Assoziationsanalyse zu bekommen, werden in dem kommenden Kapitel alle Grundlagen dieser erläutert. Zu diesen Grundlagen gehören eine allgemeine Erklärung der Assoziationsanalyse, ihr Sinn und ihre Funktion sowie der wichtigen Werte *Support* und *Konfidenz*. Außerdem wird der allgemeine Ablauf der Assoziationsanalyse beschrieben, die in dieser Arbeit genutzten Algorithmen benannt und ihre Arbeitsweise dargestellt. Anhand von Beispielen wird versucht, die Materie so verständlich wie möglich zu erläutern, damit die Umsetzung im Hauptteil nachvollziehbar ist.

2.1. Assoziationsanalyse

In der Assoziationsanalyse werden Daten analysiert und deren Abhängigkeiten aufgezeigt. Als Ergebnis der Assoziationsanalyse, die auf einen Datenbestand angewendet wird, erhält man Assoziationsregeln. Diese Assoziationsregeln beschreiben die signifikanten Abhängigkeiten der zu untersuchenden Merkmale (Säuberlich, 2000, S. 43). Sie lassen sich als Wenn-Dann-Aussagen formulieren ($A \Rightarrow B$ = Wenn A, dann B) und „eignen sich dadurch zur Durchführung von Untersuchungen im Marketing, im Controlling und auch außerhalb betriebswirtschaftlicher Einsatzfelder“. Der Bedingungsteil wird auch als Regelrumpf, Prämisse oder Antezedens bezeichnet, während der Folgerungsteil auch Regelkopf, Konklusion oder Sukzedens genannt wird (Gabriel, Gluchowski und Pastwa, 2009, S. 161).
Wichtig für die Assoziationsanalyse sind Maßzahlen. Sie werden benötigt, um Aussagen über die produzierten Assoziationsregeln und deren Inhalte zu machen. *Support* und *Konfidenz* sind die Maßzahlen, die sich gegenüber anderer durchgesetzt haben (Farkisch, 2011, S. 109) und werden deshalb folgend in einzelnen Kapiteln behandelt.
Ein Einsatzgebiet der Assoziationsanalyse ist die Warenkorbanalyse. Für diese haben Assoziationsverfahren einen besonderen Stellenwert (Petersohn, 2005, S. 102). Die meisten Geschäfte verfügen über elektronische Scannerkassen, die einen Artikel anhand des Barcodes, der europäischen Artikelnummer (EAN), eindeutig identifizieren und damit alle notwendigen Informationen aus einer Datenbank abrufen. Beim Abscannen der EAN kann der Artikel gemeinsam mit den restlichen Artikeln des Einkaufes automatisiert als Transaktion in einer

Datenbank gespeichert werden. Die Menge aller Transaktionen bildet die Datenbasis, auf der die Assoziationsanalyse angewendet werden kann. Nach der Analyse der Datenbasis ist bekannt, welche Artikel statistisch häufig gemeinsam gekauft werden. Ein Beispiel: Kunden, die einen 3D-Fernseher gekauft haben (Regelrumpf), kauften zu 60% auch eine 3D-Brille (Regelkopf). Anhand solcher Informationen können die Läden bzw. die Warenanordnung umstrukturiert werden, um Laufwege zu optimieren oder attraktive (Lock-)Angebote zu schalten (Görz, Rollinger und Schneeberger, 2003, S. 568).
Diese für den Handel ausgelegte Analyse lässt sich auch auf andere Anwendungsgebiete, wie zum Beispiel das E-Commerce, übertragen. Dort ist es heute schon üblich, dass bei einem Kauf bzw. bei der Ansicht eines Artikels zusätzlich weitere „passende“ Artikel angezeigt werden. Durch die eindeutige Identifikation eines Kunden können spezielle Angebote erstellt bzw. personalisierte Werbung eingesetzt werden.
Außerdem kann nicht nur die Warenkorbanalyse durchgeführt werden, sondern auch das Surf- bzw. Suchverhalten der Kunden analysiert werden (Petersohn, 2005, S. 102), ähnlich wie die in dieser Arbeit behandelte Auswertung von Webserver Logfiles.

Beispiele zur Verdeutlichung der Maßzahlen mithilfe der Warenkorbanalyse folgen nach der jeweiligen Erklärung der Maßzahl.

2.1.1. Formale Beschreibung

Es sind diverse Mengen für die Assoziationsanalyse notwendig. Ausgangspunkt hierfür ist die Menge $I = \{I_1, \ldots, I_m\}$ (die Menge aller Items), im Beispiel einer Warenkorbanalyse alle Artikel. Desweiteren steht T für eine Transaktion. Diese Transaktion besteht ebenfalls aus einer Menge von Items, ist also Teilmenge von I ($T_x \subseteq I$). Sie stellt somit zum Beispiel den individuellen Einkauf eines Kunden dar. Die Menge aller Transaktionen T bildet die Datenbasis $D = \{T_1, \ldots, T_n\}$. Hat man eine Assoziationsregel in der Form $X \rightarrow Y$, müssen die Menge X im Regelrumpf und die Menge Y im Regelkopf zueinander disjunkt und echte Teilmengen der Itemmenge I sein. Diese Regel wird von einer Transaktion T genau dann erfüllt, wenn $(X \cup Y) \subseteq T$, das heißt, wenn alle durch die Regel abgebildeten Items auch in der Transaktion vorhanden sind (Bankhofer und Vogel, 2008, S. 261 und Petersohn, 2005, S. 103).

2.1.2. Support

> „Support is an important measure because a rule that has very low support may occur simply by chance. A low support rule is also likely to be uninteresting from a business perspective because it may not be profitable to promote items that customers seldom buy together [...].“(Tan, Steinbach und Kumar, 2006, S. 330)

Der *Support* ist die relative Häufigkeit eines Items, bzw. einer Itemmenge in der Datenbasis (Bankhofer und Vogel, 2008, S. 261). Das heißt, dass die Anzahl der Datensätze, die diese Regel unterstützen, ins Verhältnis zur Gesamtanzahl aller Datensätze gesetzt wird. Ein höherer Anteilswert lässt auf eine größere Relevanz schließen (Gabriel, Gluchowski und Pastwa, 2009, S. 162).

$$sup^{1}(X) = \frac{|\{T \in D | X \subseteq T\}|}{|D|}$$

Für den *Support* einer Assoziationsregel $X \rightarrow Y$ sieht die formale Beschreibung wie folgt aus:

$$sup(X \rightarrow Y) = \frac{|\{T \in D | (X \cup Y) \subseteq T\}|}{|D|}$$

(Petersohn, 2005, S. 103)
Wie oben bereits angesprochen, sind Regeln mit einem hohen *Support* wesentlich relevanter als jene, die einen geringen Wert aufweisen und somit nur von wenigen Transaktionen erfüllt werden. Aus diesem Grund ist es möglich für den *Support* einen Minimalwert festzulegen und nur die Regeln zu betrachten, die diese Anforderung erfüllen (Bollinger, 1996, S. 258). Diese Untergrenze wird unter anderem in den Algorithmen genutzt (siehe Kapitel 2.1.4).

Beispiel

Gegeben ist ein Händler, der alle Einkäufe in einer Datenbank speichert. Die Anzahl der Einkäufe beträgt 100.000. 10.000 Einkäufe beinhalten einen 3D-Fernseher, 7.500 Einkäufe beinhalten eine 3D-Brille und 5.000 Einkäufe beinhalten sowohl Fernseher als auch Brille. Der *Support* für die entsprechenden Werte berechnet sich wie folgt:

$$sup(3D - Fernseher) = \frac{10000}{100000} = 0,10 = 10\%$$

$$sup(3D - Brille) = \frac{7500}{100000} = 0,075 = 7,5\%$$

[1] Hier und im weiterem Verlauf der Arbeit meint sup() den Support im Gegensatz zur üblichen Bezeichnung des Supremums

$$sup\,(3D - Fernseher \rightarrow 3D - Brille) = \frac{5000}{100000} = 0,05 = 5\%$$

$$sup\,(3D - Brille \rightarrow 3D - Fernseher) = \frac{5000}{100000} = 0,05 = 5\%$$

In 10% aller Transaktionen wurden 3D-Fernseher gekauft, in 7,5% aller Transaktionen wurde eine 3D-Brille gekauft und in 5% aller Transaktionen wurden 3D-Fernseher und 3D-Brille gemeinsam gekauft. Außerdem wird bei der Berechnung des Supportwertes in den letzten beiden Fällen deutlich, dass die Supportberechnung von Regeln kommutativ ist.
Das Beispiel wird im Kapitel der *Konfidenz* erneut aufgegriffen und erweitert.

2.1.3. Konfidenz

> „Confidence, [...], measures the reliability of the inference made by a rule." (Tan, Steinbach und Kumar, 2006, S. 330)

Mit der Maßzahl *Konfidenz* „werden die Mächtigkeit und die Stärke [des Zusammenhangs] einer Regel ausgedrückt" (Farkisch, 2011, S. 109). Bei einer Regel $X \rightarrow Y$ muss, um auf den Wert der *Konfidenz* zu kommen, die Anzahl der Transaktionen, die $X \rightarrow Y$ erfüllen, ins Verhältnis zu den Transaktionen, die den Prämissenteil X der Regel erfüllen, gesetzt werden (Bollinger, 1996, S. 258). Somit ergibt sich formal:

$$conf\,(X \rightarrow Y) = \frac{|\{T \in D|\,(X \cup Y) \subseteq T\}|}{|\{T \in D|X \subseteq T\}|} = \frac{sup\,(X \rightarrow Y)}{sup\,(X)}$$

(Petersohn, 2005, S. 104)
Wie an der formalen Beschreibung zu erkennen ist, werden „der Support der Vereinigungsmenge von X und Y ins Verhältnis zum Support der Menge X gesetzt." Da *Support* und *Konfidenz* Verhältnisgrößen sind, gilt für den Wertebereich:

$$sup\,(X \rightarrow Y)\,, conf\,(X \rightarrow Y) \in [0; 1]$$

(Bankhofer und Vogel, 2008, S. 162).
Wie schon bei dem Wert für *Support* kann auch für die *Konfidenz* ein Minimalwert vom Benutzer festgelegt werden. Auch hier werden nur die Regeln betrachtet, deren Wert größer oder gleich dem Minimalwert ist (Bollinger, 1996, S. 258).

Beispiel

Zur Verdeutlichung wird das Beispiel vom *Support* wieder aufgegriffen und erweitert. Gegebene Informationen sind:
Anzahl aller Transaktionen: 100.000
Anzahl der Transaktionen, die 3D-Fernseher enthalten: 10.000
Anzahl der Transaktionen, die 3D-Brillen enthalten: 7.500
Anzahl der Transaktionen, die sowohl 3D-Fernseher als auch 3D-Brillen enthalten: 5.000
Support(3D-Fernseher) = 0,1
Support(3D-Brille) = 0,075
Support(3D-Fernseher $\rightarrow$ 3D-Brille) = 0,05
Support(3D-Brille $\rightarrow$ 3D-Fernseher) = 0,05
Mit diesen Werten kann die *Konfidenz* für die Regeln 3D-Brille $\rightarrow$ 3D-Fernseher und 3D-Fernseher $\rightarrow$ 3D-Brille wie folgt berechnet werden:

$$conf\,(3D - Brille \rightarrow 3D - Fernseher) = \frac{0,05}{0,075} = 0,\bar{6} \cong 66,7\%$$

$$conf\,(3D - Fernseher \rightarrow 3D - Brille) = \frac{0,05}{0,1} = 0,5 = 50\%$$

In 66,7% aller Transaktionen, in denen eine 3D-Brille gekauft wurde, wurde auch ein 3D-Fernseher gekauft. In 50% aller Transaktionen, in denen ein 3D-Fernseher gekauft wurde, wurde auch eine 3D-Brille gekauft. Dieses Beispiel wurde in Abbildung 2.1 als Venn-Diagramm grafisch aufbereitet.

2.1.4. Allgemeiner Ablauf

Ein *brute-force* Denkansatz, in dem für jede mögliche Regel der Support- und Konfidenzwert berechnet wird, ist in der Praxis undenkbar. Diese Herangehensweise ist enorm teuer, da die Anzahl der Regeln R, die aus einer Itemmenge gebildet werden, exponentiell zu der Anzahl der Items d ansteigt, wie die folgende Formel zeigt:

$$R = 3^d - 2^{d+1} + 1$$

Daher soll die Anzahl der erstellten Regeln reduziert werden. Dies geschieht in den zwei Phasen, die in der Assoziationsanalyse durchlaufen werden (Tan, Steinbach und Kumar, 2006, S. 331).

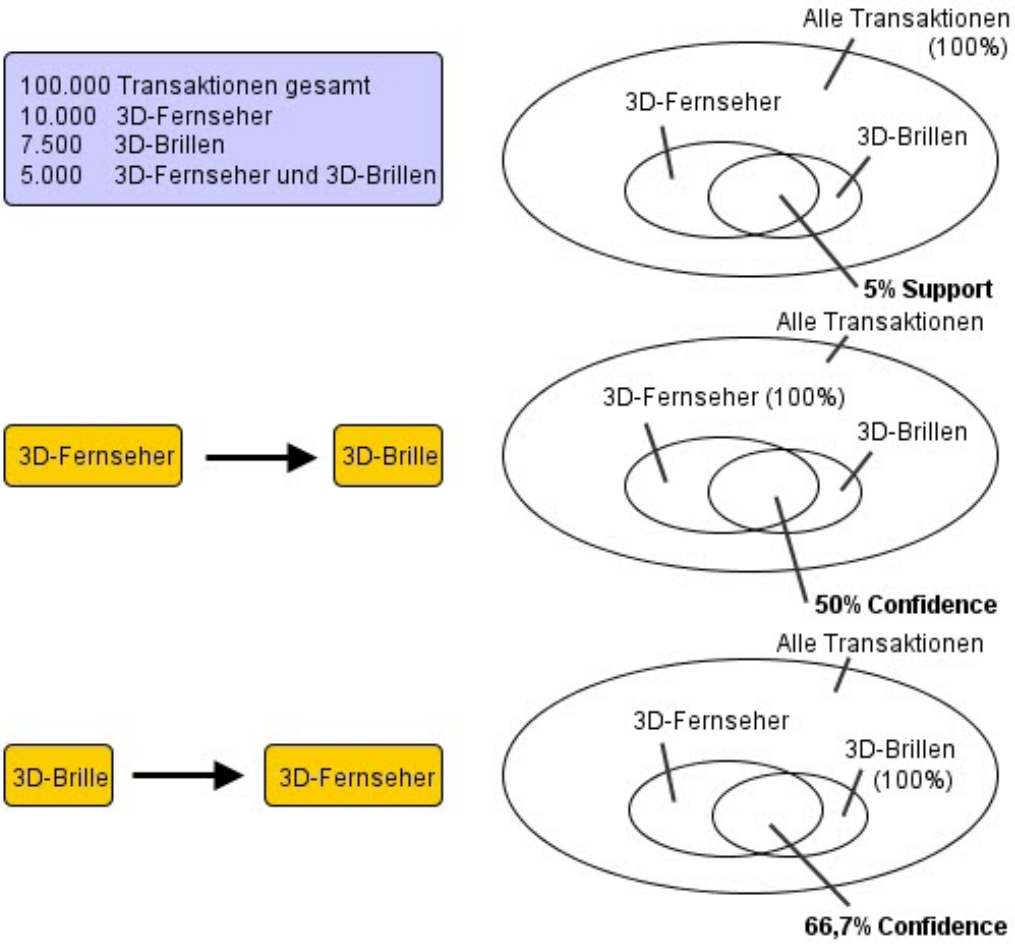


Abbildung 2.1.: Visualisierung von *Support* und *Konfidenz* anhand eines Beispiels (Quelle: in Anlehnung an Bankhofer und Vogel, 2008, S. 263)

Erste Phase: In der ersten Phase werden alle *sets of items* (Itemsets) einer Datenbasis gesucht, deren Supportwert größer oder gleich dem angegebenen Mindestsupport ist. Diese Itemsets werden im Weiteren als „large itemsets“, „frequent itemsets“ oder auch „häufige Itemmengen“ bezeichnet (Petersohn, 2005, S. 104).
Für die häufige Itemmenge H gilt $H \subseteq I$[2]. Außerdem gilt für H formal:

$$sup\,(H) = \frac{|\{T \in D : H \subseteq T\}|}{|D|}\,[3] \geq sup_{min}$$

Da die Kombinierungsmöglichkeiten der Items aus der Ausgangsitemmenge I in der Regel sehr groß sind (siehe Abbildung 2.2 für nur fünf Items), ist die Aufgabe der Assoziationsalgorithmen, ein möglichst effizientes Finden aller Itemmengen, jedoch ohne Untersuchung aller Itemmengen auf ihren Supportwert (Bankhofer und Vogel, 2008, S. 263f.). Häufig werden innerhalb der Algorithmen zuerst sogenannte Kandidaten (*candidate itemsets*) ermittelt, die die Mengen darstellen, deren *Support* noch nicht festgestellt wurde (Petersohn, 2005, S. 104).

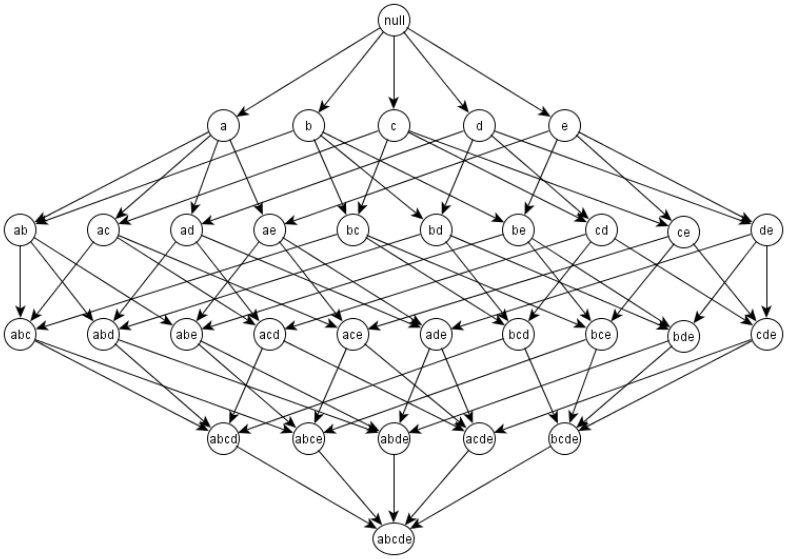


Abbildung 2.2.: Alle Kombinierungsmöglichkeiten bei fünf Elementen (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 332)

[2] Bankhofer und Vogel nutzen das Symbol der echten Teilmenge ($\subset$) sowohl als Symbol für die echte Teilmenge ($\subset$) wie auch als Symbol für die Teilmenge ($\subseteq$). Im weiteren Verlauf der Arbeit wird das Symbol der echten Teilmenge durch das Symbol der Teilmenge ersetzt, wenn die Teilmenge und nicht die echte Teilmenge gemeint ist. Um die Lesbarkeit der Formeln nicht zu verschlechtern, wird diese Änderung durch eine Fußnote gekennzeichnet. Wird tatsächlich die echte Teilmenge in einer Formel von Bankhofer und Vogel genutzt, wird diese ungeändert übernommen.

[3] Änderung des Symbols der echten Teilmenge durch das Symbol der Teilmenge

Zweite Phase: Sind alle häufigen Itemmengen gefunden, werden in der zweiten Phase unter Verwendung dieser die Assoziationsregeln erstellt. Dies geschieht, indem alle Regeln der Form $H' \rightarrow H - H'$ für alle $H' \subset H$ generiert werden, deren *Konfidenz* größer oder gleich des Minimalwertes ($conf_{min}$) ist, formal dargestellt:

$$conf\,(H' \rightarrow H - H') = \frac{|\{T \in D : H \subseteq T\}|}{|\{T \in D : H' \subseteq T\}|}^{4} \geq conf_{min}$$

(Bankhofer und Vogel, 2008, S. 264)

Zur Veranschaulichung dient folgendes Beispiel:
Angenommen $ABCD$ und AB sind *large itemsets*, dann kann über die Berechnung der *Konfidenz*

$$conf\,(AB \rightarrow CD) = \frac{|\{T \in D : ABCD \subseteq T\}|}{|\{T \in D : AB \subseteq T\}|} \geq conf_{min}$$

bestimmt werden, ob die Regel $AB \rightarrow CD$ gilt (Petersohn, 2005, S. 104).

2.2. Algorithmen

Es gibt viele Algorithmen, die der Erstellung von Assoziationsregeln dienen. In dieser Arbeit werden jedoch nur zwei von ihnen genutzt. Der Erste ist der sogenannte Apriori-Algorithmus (siehe Kapitel 2.2.1), der andere ist der FP-Growth Algorithmus, der in Kapitel 2.2.2 vorgestellt wird. Die Wahl fiel auf diese beiden Algorithmen, da der Apriori als klassisches Verfahren gilt (Bollinger, 1996, S. 972) und der FP-Growth eine ganz andere Arbeitsweise verfolgt und ohne die Generierung von Kandidaten auskommt. Beide Algorithmen werden in den nachfolgenden Kapiteln erklärt und beschrieben.

2.2.1. Apriori

Der Apriori-Algorithmus wurde vom IBM-Almaden-Forschungszentrum entwickelt und ist das klassische Verfahren zum Generieren der Assoziationsregeln. Es existieren einige Verfahren, die auf dem Apriori-Algorithmus basieren. Zu nennen wären hier zum Beispiel *AprioriTid* oder *AprioriHybrid* (Farkisch, 2011, S. 111). Auf diese Entwicklungen wird in dieser Arbeit jedoch nicht weiter eingegangen.
Nachfolgend sind einige Informationen gegeben (Ester und Sander, 2000, S. 160), um die Funktionsweise des Algorithmus' verständlicher zu erläutern:

[4] Änderung des Symbols der echten Teilmenge durch das Symbol der Teilmenge

- Items innerhalb der Itemsets sind lexikographisch sortiert. Wenn ein Itemset X beispielsweise aus den Items $\{x_1, x_2, x_3, \ldots, x_k\}$ besteht, gilt für die Items innerhalb des sets, dass $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_k$.
- Die Länge der Itemsets wird gemessen durch die Anzahl der Elemente. Ein Itemset mit k-Elementen wird auch k-Itemset genannt.

Das Herzstück des Algorithmus' ist die Erzeugung der *large itemsets* (siehe Kapitel 2.1.4) (Farkisch, 2011, S. 112). Hierfür werden *candidate itemsets* erzeugt, sozusagen potenzielle *large itemsets*, deren Supportwert noch nicht ermittelt wurde. Die Generierung der *candidate itemsets* erfolgt stufenweise (iterativ). Je k-ter Iteration des Apriori-Algorithmus werden nur die *large itemsets* und *candidate itemsets* der Länge k betrachtet (Petersohn, 2005, S. 108). Die Generierung der Kandidaten beruht nach Agrawal und Srikant auf der Aussage: *„The basic intuition is that any subset of a large itemset must be large."*. Zur Verdeutlichung dieser Aussage dient die Abbildung 2.3.

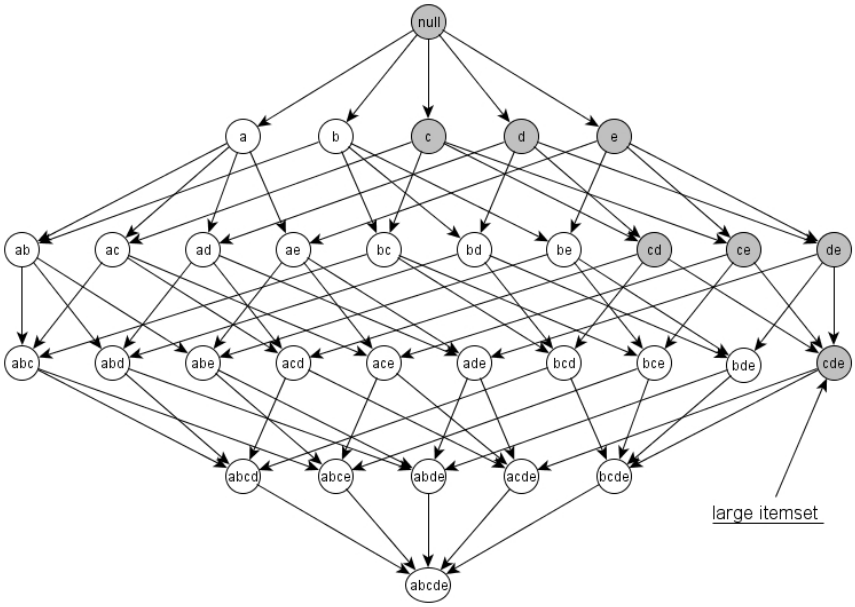


Abbildung 2.3.: Eine Darstellung des Apriori-Prinzips: Ist $\{cde\}$ ein *large itemset*, müssen alle *subsets* ebenso *large* sein (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 334)

Außerdem ergeben sich aus der Aussage zwei Schritte:

join step: Der erste Schritt ist der sogenannte *join-step* und bewirkt laut Agrawal und Srikant: *„[...] the candidate itemsets having k items can be generated by joining large itemsets having*

k-1 items [...]“. Es werden sozusagen vorläufige *candidate sets* erstellt, die alle möglichen (k+1)-elementigen Kombinationen beinhalten (Hettich, Hippner und Wilde, 2000, S. 973). Vergleichbar ist dieser Schritt mit dem „Join“-Befehl einer Datenbank. Es wird jedes k-elementige *large itemset* p mit dem letzten Item eines k-elementigem *large itemset* q erweitert, wenn die ersten (k-1) Elemente identisch sind (Abbildung 2.4) (Ester und Sander, 2000, S. 163).

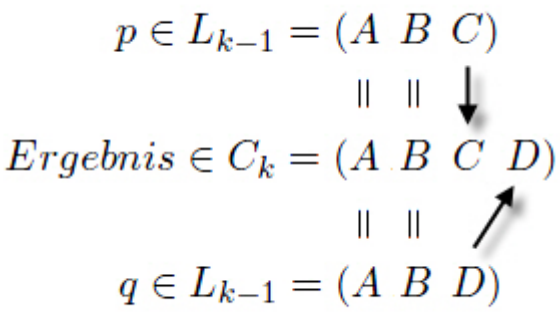


Abbildung 2.4.: Generierung eines 4-elementigen Kandidaten aus 3-elementigen *large itemsets* (Quelle: In Anlehnung an Ester und Sander, 2000, S. 163)

prune step: Der zweite Schritt wird *prune-step* oder auch „Pruning“ genannt und bewirkt laut Agrawal und Srikant: *„[...] deleting those that contain any subset that is not large.“*. Das heißt, dass alle (durch den *join-step*) erstellten k-elementigen Kandidaten entfernt werden, die eine (k-1)-elementige Teilmenge besitzen, die nicht in der Menge der (k-1)-elementigen *large itemsets* vorkommen (Ester und Sander, 2000, S. 163f.). Pruning wird auch häufig in Verbindung mit Entscheidungsbäumen genannt und bedeutet so viel wie abschneiden oder beschneiden. Durch diesen Schritt sollen keine (überflüssigen) Unterbäume entwickelt werden, die später wieder verworfen werden (Witten und Frank, 2001, S. 174). Verdeutlicht wird der Vorteil des Prunings in Abbildung 2.5.
Durch diese beiden Schritte werden sehr viel kleinere *candidate sets* erstellt (Agrawal und Srikant, 1994a, S. 3).

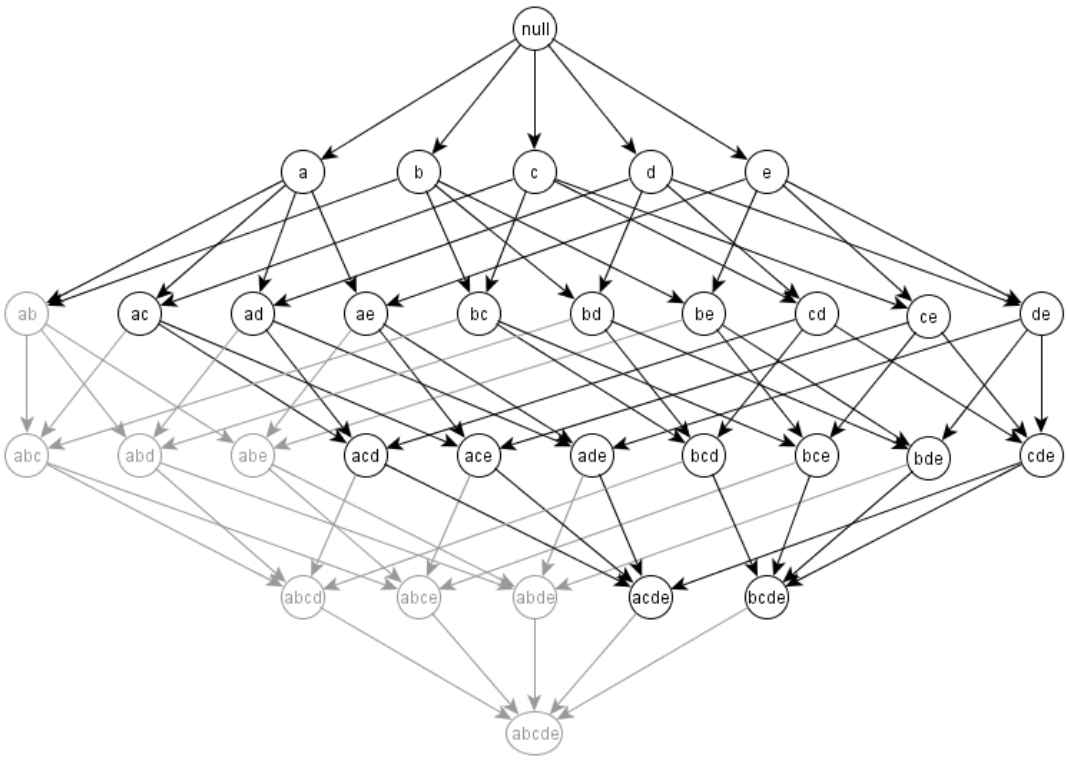


Abbildung 2.5.: Resultat des *prune-steps*: Ist $\{ab\}$ kein *large itemset*, kann kein *superset* von $\{ab\}$ large sein und der Baum wird radikal beschnitten (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 335)

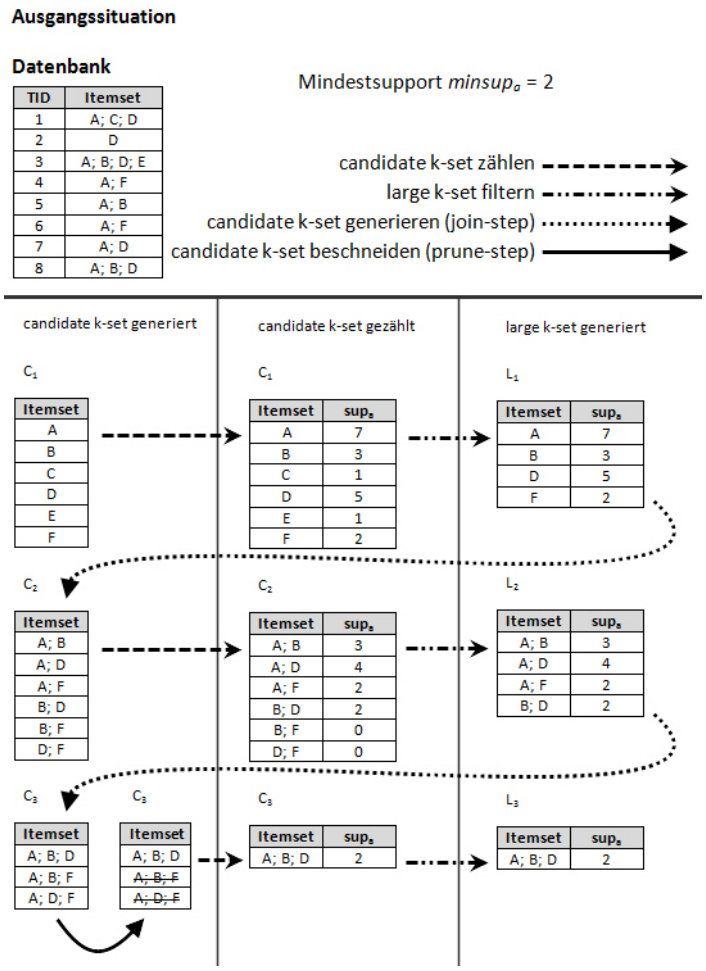
Arbeitsweise des Algorithmus

Der eigentliche Algorithmus für die Ermittlung der *large itemsets* durchläuft drei Phasen.

1. Generierung des k-ten *candidate sets* C_k mittels *join-step* und *prune-step*.
2. Supportermittlung aller k *canditate itemsets* $c \in C_k$.
3. Ermittlung aller k *large itemsets*.

(Petersohn, 2005, S. 108)
Die Ermittlung der *large itemsets* ist beendet, wenn sich keine neuen *candidate itemsets* erzeugen lassen.

Beispiel Im folgenden Beispiel wird die Funktionsweise des Apriori-Algorithmus verdeutlicht (siehe Abbildung 2.6). Als Ausgangssituation dient eine Datenbank mit acht Transaktionen. Der Mindestwert für *Support* beträgt 2 [5]. Iteration 1 ($k = 1$):



TID	Itemset
1	A; C; D
2	D
3	A; B; D; E
4	A; F
5	A; B
6	A; F
7	A; D
8	A; B; D

C_1

Itemset
A
B
C
D
E
F

C_1

Itemset	sup_a
A	7
B	3
C	1
D	5
E	1
F	2

L_1

Itemset	sup_a
A	7
B	3
D	5
F	2

C_2

Itemset
A; B
A; D
A; F
B; D
B; F
D; F

C_2

Itemset	sup_a
A; B	3
A; D	4
A; F	2
B; D	2
B; F	0
D; F	0

L_2

Itemset	sup_a
A; B	3
A; D	4
A; F	2
B; D	2

C_3

Itemset
A; B; D
A; B; F
A; D; F

C_3

Itemset
A; B; D
~~A; B; F~~
~~A; D; F~~

C_3

Itemset	sup_a
A; B; D	2

L_3

Itemset	sup_a
A; B; D	2

Abbildung 2.6.: Verdeutlichung von Apriori an einem Beispiel (Quelle: in Anlehnung an Petersohn, 2005, S. 110)

- Erste Phase - Es werden alle Kandidaten in das *candidate set* zusammengeführt.
- Zweite Phase - Es wird der *Support* aller k-elementigen Mengen berechnet.
- Dritte Phase - Alle Elemente, deren *Support* größer als der Mindestsupport ist, werden zum *large k-set* zusammengefasst.

[5] In den Beispielen handelt um die absoluten Werte und nicht um die relative Häufigkeit, da ein Umrechnen die Beispiele unnötig verkomplizieren würde

Iteration 2 ($k = 2$):

- Erste Phase - *join-step* - Aus den large $k-1$-set werden alle Kandidaten in das *candidate k-set* zusammengeführt.
- Erste Phase - *prune-step* - Alle Mengen, deren Teilmengen nicht *large* sind, werden entfernt (keine vorhanden).
- Zweite Phase - Es wird der *Support* aller k-elementigen Mengen berechnet.
- Dritte Phase - Alle Elemente, deren *Support* größer als der Mindestsupport ist, werden zum *large k-set* zusammengefasst.

Iteration 3 ($k = 3$):

- Erste Phase - *join-step* - Aus den large $k-1$-set werden alle Kandidaten in das *candidate k-set* zusammengeführt.
- Erste Phase - *prune-step* - Alle Mengen, deren Teilmengen nicht *large* sind, werden entfernt ({A; B; F} wird entfernt, weil {B; F} nicht *large* ist und {A; D; F} wird entfernt, weil {D; F} nicht *large* ist).
- Zweite Phase - Es wird der *Support* aller k-elementigen Mengen berechnet.
- Dritte Phase - Alle Elemente, deren *Support* größer als der Mindestsupport ist, werden zum *large k-set* zusammengefasst.

Da nur noch ein Item in der häufigen Itemmenge vorhanden ist, wird der Algorithmus an dieser Stelle beendet, weil keine weiteren Kandidaten generiert werden können.

Nachteil der Kandidatengenerierung

Auch wenn der Apriori-Algorithmus in den meisten Fällen, durch das Reduzieren der Kandidatenmengen, effizient ist und ein gutes Ergebnis liefert, gibt es Situationen, in denen das nicht der Fall ist, zum Beispiel bei einem niedrigen Mindestwert für *Support*, einer großen Anzahl von Mustern oder langen Mustern. Zur Verdeutlichung dienen folgende Beispiele:

- Existieren 10^4 1-elementige *frequent itemsets*, erzeugt der Apriori mehr als 10^7 2-elementige Kandidaten, welche wiederum getestet werden müssen.
- Für ein 100-elementiges *frequent itemset* $\{a_1, \ldots, a_{100}\}$ müssen insgesamt $2^{100} - 2 \approx 10^{30}$ Kandidaten generiert werden.
- Es ist langwierig, die Datenbasis immer wieder zu durchsuchen, um den Supportwert für die Kandidaten zu bestimmen.

(Han u. a., 2004, S. 54)
Ein alternativer Ansatz wird im Kapitel 2.2.2 beschrieben.

Regelerzeugung

Für die Erzeugung von Regeln werden für jedes *large itemset* l alle nicht leeren echten Teilmengen a von l erstellt ($a \subset l$). Für jede echte Teilmenge a wird eine Regel der Form $a \Rightarrow (l - a)$ erstellt, wenn der Wert der *Konfidenz* dieser Regel größer oder gleich dem Mindeswert für die *Konfidenz* ist.
Diese Herangehensweise kann durch folgende Betrachtung verbessert werden:
Angenommen, aus einer Teilmenge a von l kann keine Regel erstellt werden, so kann auch aus jedem *subset* von a keine Regel erstellt werden und muss daher nicht betrachtet werden.

Beispiel: Angenommen, die Regel $ABC \Rightarrow D$ erfüllt nicht die Mindestkonfidenz, dann tut dies auch die Regel $AB \Rightarrow CD$ nicht, da laut Agrawal und Srikant: „*[...] the support of any subset $\tilde{a}$ of a must be as great as the support of a. Therefore, the confidence of the rule $\tilde{a} \Rightarrow (l - \tilde{a})$ cannot be more than the confidence of $a \Rightarrow (l - a)$.*“.

Aus dieser Erkenntnis kann gefolgert werden, dass, wenn die Regel $(l - c) \Rightarrow c$ gilt, auch die Regel $(l - \tilde{c}) \Rightarrow \tilde{c}$ gelten muss, wenn $\tilde{c}$ eine nicht leere Teilmenge von c ist.

Beispiel: Gilt die gegebene Regel $AB \Rightarrow CD$, so müssen auch die Regeln $ABC \Rightarrow D$ und $ABD \Rightarrow C$ gelten.
(Agrawal und Srikant, 1994b, S. 13)

Diese Aussage hat Ähnlichkeit mit der Eigenschaft, auf der die Kandidatengenerierung beruht: „*The basic intuition is that any subset of a large itemset must be large.*“(Agrawal und Srikant, 1994a, S. 489). Aus diesem Grund wird bei der Implementierung (Beschreibung in Kapitel 3.6) die gleiche Methode wie bei der Kandidatengenerierung genutzt.

2.2.2. FP-Growth

Der Algorithmus ***Frequent Pattern-Growth*** unterscheidet sich von Apriori grundlegend dadurch, dass keine Kandidaten (*candidate sets*) erzeugt werden und er somit nicht nach dem „generate-and-test“ Prinzip arbeitet. Er arbeitet mit einer kompakten Datenstruktur (Tan, Steinbach und Kumar, 2006, S. 363), dem sogenannten ***Frequent Pattern-Tree***, der aus den Transaktionen der Datenbank aufgebaut wird. Aus diesem können dann die *frequent itemsets* extrahiert werden (Petersohn, 2005, S. 120).

Darstellung eines FP-Tree

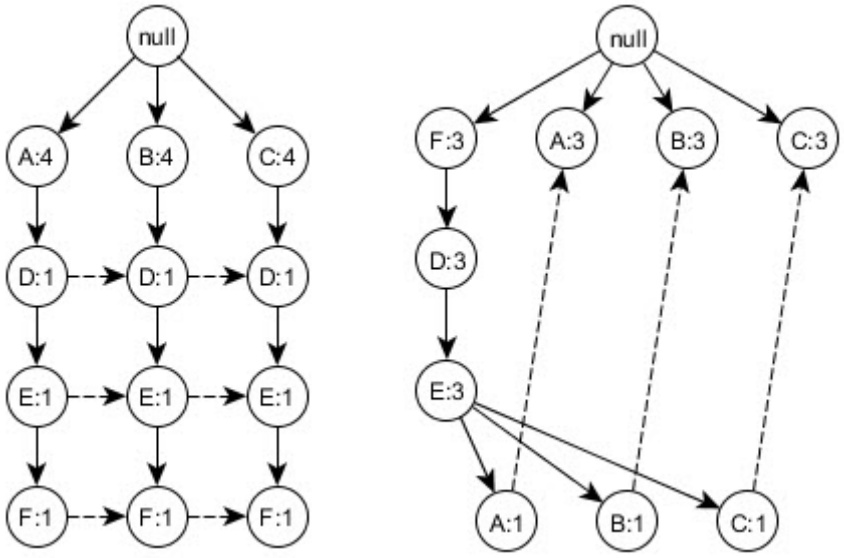
Ein FP-Tree repräsentiert die Datenbasis in einer komprimierten Form. Es wird jede Transaktion der Datenbasis betrachtet und diese als Pfad in dem Baum eingefügt. Da jeder Knoten des Baumes nur ein 1-elementiges *large item* darstellt, und nicht ein ganzes Itemset, können sich Pfade auch überschneiden oder gemeinsame Knoten haben. Je mehr Pfade sich überlappen, desto größer ist die Komprimierung der Datenbasis.
Der Startknoten, auch *root* oder Wurzel genannt, wird mit dem „$null$"-Symbol dargestellt (Tan, Steinbach und Kumar, 2006, S. 363). Alle Knoten, die die 1-elementigen *frequent items* repräsentieren, beinhalten folgende Informationen:

- Zeiger (oder auch *Pointer* genannt) auf Elternknoten
- Zeiger auf Kindknoten
- Name
- Zähler des Vorkommens im aktuellen Pfad
- Zeiger auf den nächsten gleichnamigen Knoten.

Zudem exisiert noch eine Tabelle, die sogenannte *header table*, in der je Item ein Zeiger auf den Knoten mit dem ersten Vorkommen dieses Items gehalten wird. Die Items in der Tabelle sind nach dem absteigenden Supportwert sortiert.
Unter Angabe des Minimalwertes für *Support* und der Datenbasis läuft der sequentielle Aufbau des FP-Baumes wie folgt ab:

1. Die Datenbasis wird durchlaufen und das Vorkommen aller Items gezählt.
2. Die 1-elementigen *frequent items* werden nach dem Supportwert sortiert.
3. Die Datenbasis wird ein zweites Mal durchlaufen und der Baum erstellt. Die Transaktionen werden nacheinander betrachtet und alle Items, die infrequent sind, aus der Menge genommmen.
4. Die reduzierte Menge wird nach dem Supportwert sortiert und jedes Element nacheinander (nach absteigendem Supportwert) in den Baum eingefügt.

(Petersohn, 2005, S. 120f.) Später wird der Aufbau des Baumes noch anhand eines Beispiels verdeutlicht.
Die Größe des FP-Baumes ist in der Regel kleiner, als die Größe der unkomprimierten Datenbasis, da in Transaktionen auch die gleichen Items vorkommen können. Im besten Fall haben



(a) Sortierter FP-Tree

(b) Unsortierter FP-Tree

Abbildung 2.7.: Verschiedene Anordnungen von FP-Trees (Quelle: In Anlehnung an Han u. a., 2004, S. 60)

alle Transaktionen die gleichen Mengen, dann hat der Baum nur einen Ast mit Knoten. Der schlechteste Fall ist, wenn jede Transaktion unterschiedliche eindeutige Mengen beinhaltet. In diesem Fall hat der Baum die gleiche Größe wie die unkomprimierte Datenbasis, benötigt allerdings mehr Speicher, da noch die Zeiger verwaltet werden.
Auch wird durch die absteigende Sortierung des Supportwertes nicht immer gewährleistet, dass der kleinste Baum erzeugt wird. Werden folgende Transaktionen ($\{A, D, E, F\}$, $\{B, D, E, F\}$, $\{C, D, E, F\}$, $\{A\}$, $\{A\}$, $\{A\}$, $\{B\}$, $\{B\}$, $\{B\}$, $\{C\}$, $\{C\}$, $\{C\}$) mit einem Mindestsupportwert von 3 betrachtet, resultiert daraus die Menger der einelementigen *large itemsets* (L_1) der Form ($\{A : 4\}$, $\{B : 4\}$, $\{C : 4\}$, $\{D : 3\}$, $\{E : 3\}$, $\{F : 3\}$). Durch die sortierte Ordnung $A \longmapsto B \longmapsto C \longmapsto D \longmapsto E \longmapsto F$, entsteht der Baum, der in Abbildung 2.7a zu sehen ist. Der Baum zählt zwölf Knoten (ohne *Root*).
Besteht für die gleichen Transaktionen eine zufällig Ordnung $F \longmapsto D \longmapsto E \longmapsto A \longmapsto B \longmapsto C$, entsteht der Baum, der in Abbildung 2.7b zu sehen ist. Dieser Baum zählt nur neun Knoten (ohne *Root*) und umfasst daher weniger als der sortierte Baum (Han u. a., 2004, S. 60).

Beispiel für den Aufbau des Baumes

Um den Aufbau des Baumes zu verdeutlichen, wird dieser folgend in einem Beispiel Schritt für Schritt erklärt. Grundlage ist der Datenbestand, der bereits in dem Apriori-Beispiel genutzt wurde. Die Datenbasis ist in Abbildung 2.8 zu sehen, der Mindestwert für *Support* beträgt wieder 2.

Datenbank

TID	Itemset
1	A; C; D
2	D
3	A; B; D; E
4	A; F
5	A; B
6	A; F
7	A; D
8	A; B; D

L_1

Itemset	sup_s
A	7
D	5
B	3
F	2

Abbildung 2.8.: Beispiel für den Aufbau eines FP-Tree; Datenbasis und L_1 (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 364)

1. Die Datenbasis wird einmal durchsucht und alle 1-elementigen *large itemsets* gespeichert. Danach werden diese nach dem Supportwert absteigend sortiert (L_1). Nun wird erneut durch die Datenbasis iteriert (siehe Abbildung 2.8).

2. Die erste Transaktion/der erste Datensatz wird betrachtet (A, C, D). Das Item C wird entfernt, da $C \notin L_1$. Es werden die Knoten A und D erstellt und unter Betrachtung der Reihenfolge ein Pfad erstellt ($null \longmapsto A \longmapsto D$). Der Zähler jeden Knotens hat den Wert 1. Der FP-Tree nach dem ersten Datensatz ist in Abbildung 2.9 zu sehen.

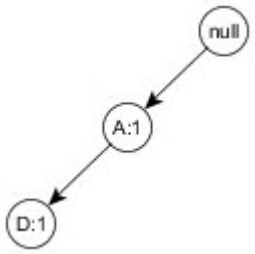


Abbildung 2.9.: Beispiel für den Aufbau eines FP-Tree; Erster Datensatz gelesen (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 364)

3. Der zweite Datensatz wird gelesen (D). Dieser enthält nur ein 1-elementiges *large itemset*. Da dieses Itemset noch kein Kindknoten der *Root* ist (es keinen gemeinsamen Präfix mit einem anderen Pfad hat), wird ein neuer Knoten erstellt und ein Pfad generiert ($null \longmapsto D$). Auch hier hat der Zähler den Wert 1. Der FP-Tree nach dem zweiten Datensatz ist in Abbildung 2.10 zu sehen.

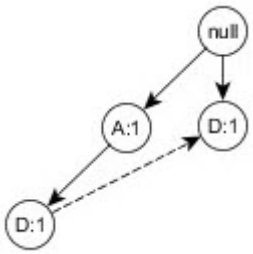


Abbildung 2.10.: Beispiel für den Aufbau eines FP-Tree; Zweiter Datensatz gelesen (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 364)

4. Der dritte Datensatz wird gelesen (A, B, D, E). Das Item E wird entfernt, da $E \notin L_1$. Das Resultat (A, B, D) wird in eine sortierte Reihenfolge gebracht (nach L_1). Da (A, D, B) und die schon vorhandene Transaktion (A, D) identische Präfixe haben, nämlich A und D, überlappen sich die Pfade $null \longmapsto A \longmapsto D$ und $null \longmapsto A \longmapsto D \longmapsto B$. Aus diesem Grund werden die Zähler für A und D um 1 erhöht, während der Zähler des neu erstellten Knotens B auf 1 steht. Der FP-Tree nach dem dritten Datensatz ist in Abbildung 2.11 zu sehen.

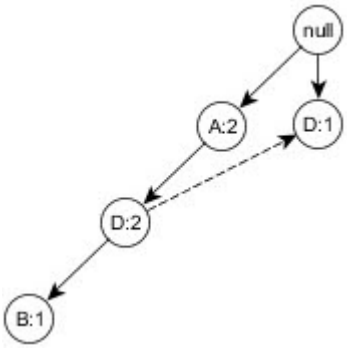


Abbildung 2.11.: Beispiel für den Aufbau eines FP-Tree; Dritter Datensatz gelesen (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 364)

5. Dieser Prozess geschieht ebenfalls mit allen anderen Transaktionen. Wurde er für alle Transaktionen durchgeführt, ist der Baum fertig erstellt. Der fertige Baum mit der dazugehörigen *header table* für das Beispiel ist in der Abbildung 2.12 zu sehen.

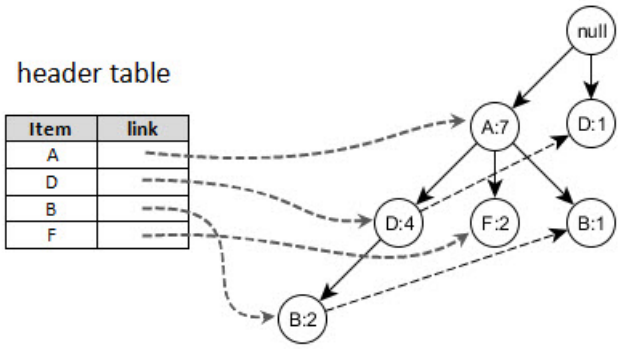


Abbildung 2.12.: Beispiel für den Aufbau eines FP-Tree; Letzter Datensatz gelesen (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 364)

Die gestrichelten Pfeile sind die Pointer, die zwischen den Knoten auf die gleichen Items verweisen. Diese dienen dazu, die verschiedenen Pfade, die mit dem gleichen Item enden, schnell zu finden (Tan, Steinbach und Kumar, 2006, S. 364ff.).

Funktionsweise des FP-Growth Algorithmus

Der Algorithmus „FP-Growth“ hat nun die Aufgabe, diesen FP-Tree zu deuten und aus ihm die *large itemsets* zu ermitteln. Die Deutung ist abhängig von dem Aufbau des FP-Trees. Der einfachste Aufbau ist, wenn er nur einen Pfad (von der *root*) enthält, solch ein Baum wird auch „single prefix-path tree“ (siehe (a) in Abbildung 2.13) genannt. Die Pfade können zwei verschiedene Formen haben. Die erste Form ist der sogenannte „single path portion“ (siehe (b) in Abbildung 2.13), in diesem Fall hat der Pfad oder der Teilpfad immer nur maximal einen Kindknoten. Die zweite Form ist der „multipath portion“ (siehe (c) in Abbildung 2.13), in dem der Pfad mehrere Verzweigungen beinhaltet. Der *multipath* (die Erklärung der Deutung eines *multipath* folgt später) bekommt eine *pseudo-root* zugewiesen. Diese *pseudo-root* ersetzt den letzten Knoten des *single path* (der Erste, der mehr als ein Kind hat; darauf wird später noch eingegangen).
An einem Beispiel, basierend auf Abbildung 2.13, wird folgend die Deutung eines Baumes grob erklärt.
Um den kompletten Baum zu deuten, können erst die beiden Teilbäume gedeutet werden. Wenn dies geschehen ist, kann aus den „Teilergebnissen“ das Gesamtergebnis für den Baum

erstellt werden.
Als erstes wird der Teilbereich des „single path“ betrachtet. Eine Eigenschaft des *single path* ist, dass jeder Knoten mit einem anderen vereinigt werden kann. Der Zähler dieser Vereinigung hat den kleinsten Wert des Knotens. Somit ist die Menge aller *large itemsets* aus $P = \{(a : 10), (b : 8), (c : 7), (ab : 8), (ac : 7), (bc : 7), (abc : 7)\}$.
Aus der Deutung des *multipath* ergibt sich das *large itemset* $Q = \{(d : 4), (e : 3), (f : 3), (df : 3)\}$. Da Q ein „Unterpfad“ von P ist, muss, um alle *frequent itemsets* des Baumes zu erhalten, noch das Kreuzprodukt aus P und Q ermittelt werden. Der Zähler des Produktes wird wie gewohnt auf den des kleinsten Vorkommens gesetzt. Es entsteht somit die Menge $P\text{x}Q = \{(ad : 4), (bd : 4), (cd : 4), (abd : 4), (acd : 4), (bcd : 4), (abcd : 4), (ae : 3), (be : 3), (ce : 3), (abe : 3), (ace : 3), (bce : 3), (abce : 3), (af : 3), (bf : 3), (cf : 3), (abf : 3), (acf : 3), (bcf : 3), (abcf : 3), (adf : 3), (bdf : 3), (cdf : 3), (abdf : 3), (acdf : 3), (bcdf : 3), (abcdf : 3)\}$. Die *frequent itemsets* eines solchen *single prefix-path* FP-Trees bestehen also aus dem *frequent set* von P aus dem von Q und schließlich aus dem Kreuzprodukt dieser beiden *frequent itemsets* (P x Q) (Han u. a., 2004, S. 65f.).

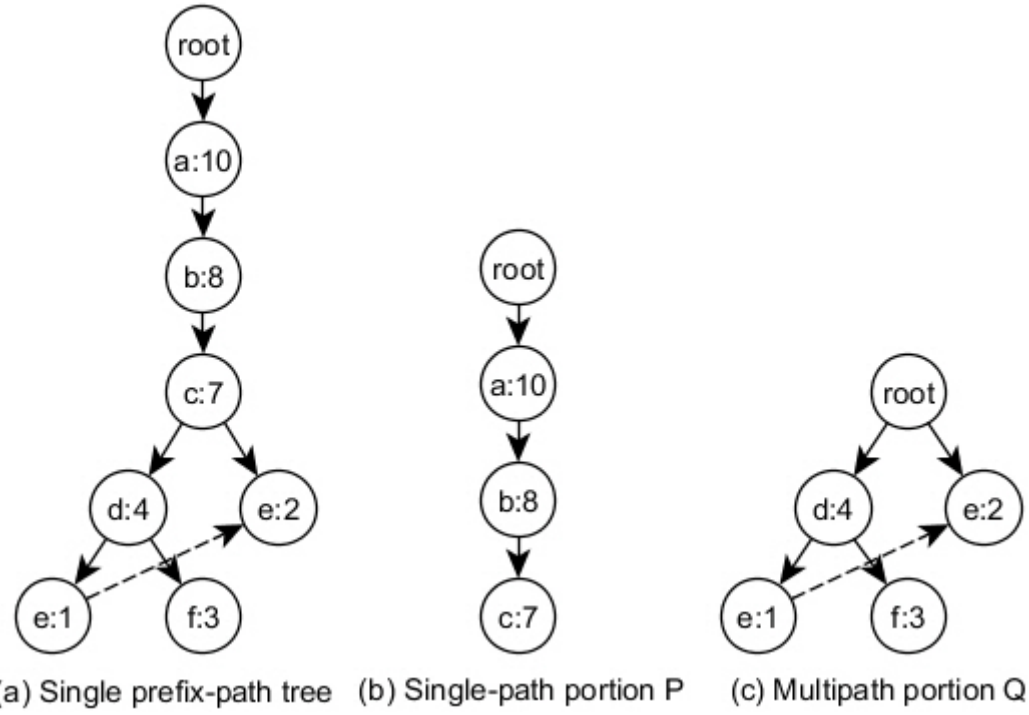


Abbildung 2.13.: Deutung eines *single prefix-path trees* (Quelle: Han u. a., 2004, S. 65)

Um die frequenten Muster aus einem *multipath* zu bestimmen, werden folgende Eingenschaften eines FP-Trees genutzt:

1. **Node-link property:** „*For any frequent item a_i, all the possible patterns containing only frequent items and a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header.*“

2. **Prefix path property:** „*To calculate the frequent patterns with suffix a_i, only the prefix subpathes of nodes labeled a_i in the FP-tree need to be accumulated, and the frequency count of every node in the prefix path should carry the same count as that in the corresponding node a_i in the path.*“

(Han u. a., 2004, S. 61ff.)

Der FP-Growth-Methode werden ein Baum und ein *frequent itemset* α (beim ersten Aufruf ist $\alpha = \emptyset$) übergeben. Besteht der Baum aus nur einem Pfad, ist die Vereiningung des *frequent itemsets* α mit den Knoten des Pfades die gesuchte Menge der *large itemsets*. Besteht der Baum aus mehreren Ästen (*multipath*), wird die Methode rekursiv mit einem Teilbaum aufgerufen.
Zu Beginn wird jedes Item a_i der *header table* betrachtet (beginnend beim letzten Item der Tabelle) und die *frequent pattern* $\beta = a_i \cup \alpha$ erzeugt. Durch die oben genannte Eigenschaft (*node-link property*) werden über die *Pointer* alle Knoten des entsprechenden *frequent item* gefunden. Durch sie werden die Pfade dieser Knoten gesammelt und die sogenannte „conditional pattern base“ gebildet. Diese „conditional pattern base“ beinhaltet die Pfade, die mit dem angegebenen *frequent pattern* enden, jedoch ohne die Knoten der *frequent pattern*. Der Zähler aller Knoten in einem betrachteten Pfad wird, dank der „prefix path“-Eigenschaft, auf den Wert des Vorkommens des einzelnen Elements gesetzt. Durch die Änderung des Wertes wird erneut der Supportwert für jedes Item überprüft und die Items, welche die Minimalgrenze nicht erreichen, entfernt. Aus dem „reduzierten“ Pfad wird ein neuer Baum, der sogenannte „conditional FP-tree“ erzeugt, welcher dem erneuten Aufruf der Methode FP-Growth(), zusammen mit dem *itemset* β, übergeben wird (Petersohn, 2005, S. 122).
Um diesen Vorgang zu verdeutlichen, werden die Schritte an einem Beispiel durchgeführt. Die Grundlage für das folgende Beispiel bildet der Baum (i) in Abbildung 2.14. Der Mininmalwert für den *Support* beträgt 2.
Da der Baum immer von unten nach oben (*bottom-up*) durchlaufen wird, sind jeweils nur die Pfade interessant, die mit dem betrachteten Item enden. In Abbildung 2.14 werden alle Pfade für die entsprechenden Items gezeigt. Wie bereits oben erwähnt, dienen die *Pointer* dazu, alle

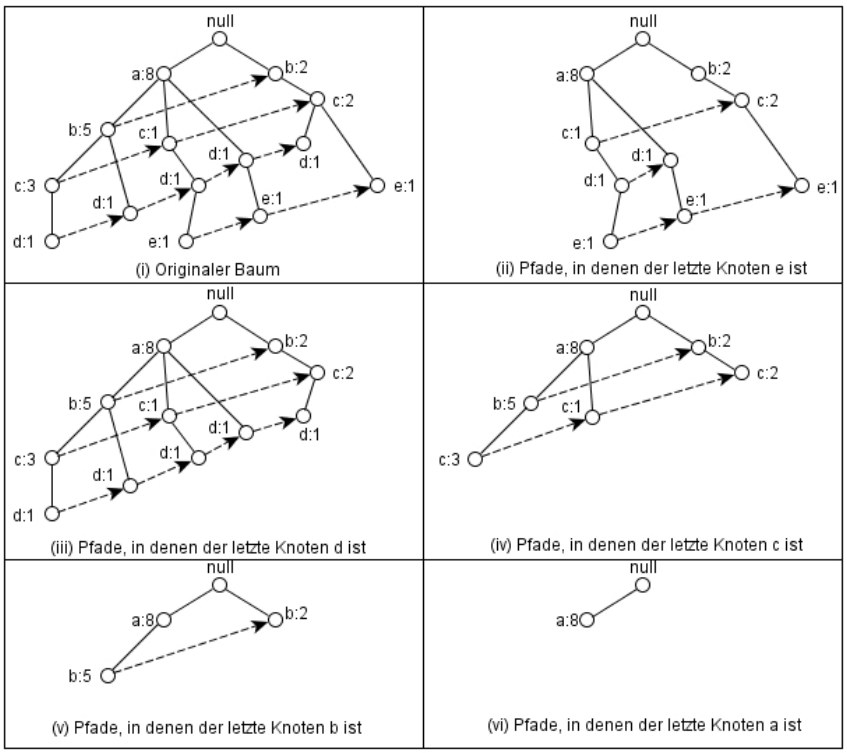


Abbildung 2.14.: Beispielbaum mit den einzelnen Pfaden (Quelle: In Anlehnung an Tan, Steinbach und Kumar, 2006, S. 367)

Pfade des Items zu finden. Dabei interessieren nur die Elternknoten, da die Kinder aufgrund der *bottom-up* Anordnung bereits abgearbeitet wurden. Mithilfe der Pfade, die mit dem Item enden, kann nun die sogenannte „conditional pattern base“ erstellt werden.
Dabei interessieren nur die Elternknoten, da die Kinder aufgrund der *bottom-up* Anordnung bereits abgearbeitet wurden. Mithilfe der Pfade, die mit dem Item enden, kann nun die sogenannte „conditional pattern base“ erstellt werden.
Wird beispielsweise das Item „e“ betrachtet, bilden die dazugehörigen Pfade (siehe Abbildung 2.14 (ii)) die *conditional pattern base* von e ({{a, c, d: 1},{a, d: 1},{b, c: 1}}). Mithilfe der *conditional pattern base* wird im nächsten Schritt ein neuer FPBaum der sogenannte „conditional FP-Tree“ erstellt. Bei der Erstellung des Baumes wird geprüft, ob alle Items der *conditional pattern base* den Mindestsupport erfüllen. a:2, c:2 und d:2 erfüllen diese Vorraussetzung, b:1 tut dies nicht und wird deswegen entfernt. Nun wird die FP-Growth-Methode mit dem *conditional FP-Tree* und e aufgerufen. Daraus resultiert, dass im nächsten Schritt nicht mehr die Pfade betrachtet werden, die mit e enden, sondern nur die Subpfade betrachtet werden, die mit ae, ce und de

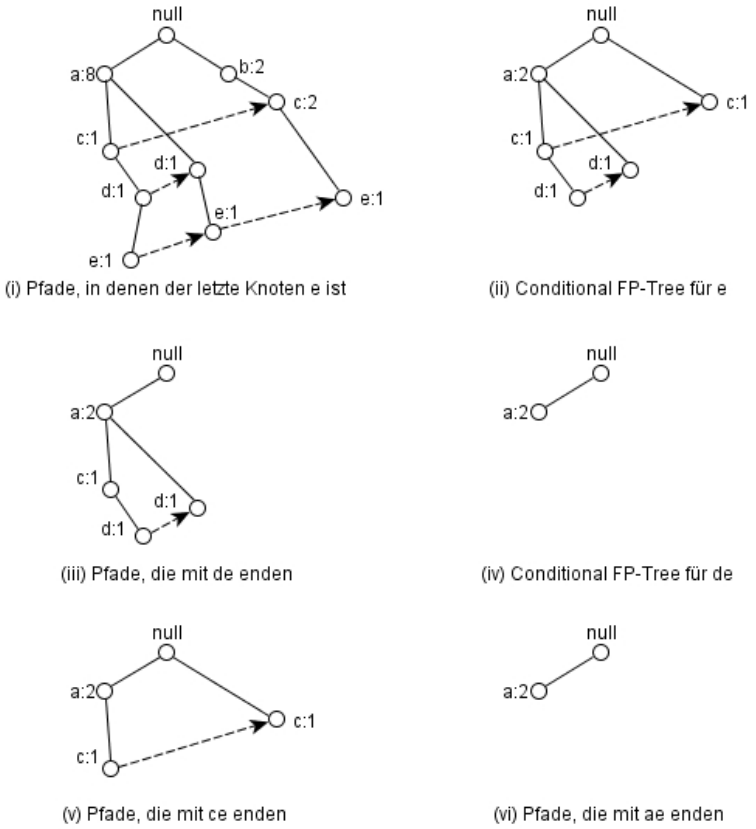


Abbildung 2.15.: Beispiel zum Finden von *frequent itemsets*, die mit e enden (Quelle: in Anlehnung an Tan, Steinbach und Kumar, 2006, S. 368)

abschließen (siehe Abbildung 2.15).
Auch bei den Subpfaden wird von unten nach oben gearbeitet. Somit wird zuerst das *itemset* de in die Menge der *frequent itemsets* hinzugefügt und dessen *conditional pattern base* ({{a, c: 1},{a: 1}}) erstellt (zu sehen in Abbildung 2.15 (iii)). Im Laufe der Erstellung des *conditional FP-Tree* wird der Knoten c entfernt, da er den Mindestsupport nicht erfüllt (siehe Abbildung 2.15 (iv)). Nach der Erstellung wird er zusammen mit de der FP-Growth-Methode übergeben. Daraus resultiert, dass ade zu den *frequent sets* hinzugefügt wird. Da aus ade allerdings kein *conditional FP-Tree* erzeugt werden kann, wird nun das *itemset* ce betrachtet. Das *itemset* ce wird zu der Menge der *frequent itemsets* hinzugefügt. Da auch hier kein *conditional FP-Tree* erzeugt werden kann, wird die Menge ae betrachtet. Mit der Menge ae wird identisch verfahren wie bereits mit der Menge ce.
Somit wurden alle *frequent itemsets* generiert, die die Menge e enthalten ({{e}, {d, e}, {c, e}, {a, e},

$\{a, d, e\}\}$). Für alle weiteren Items kann die Generierung der *frequent itemsets* in der gleichen Art durchgeführt werden. Das Endergebnis wird immer mit dem Zwischenergebnis vereiningt. Wurde dieser Schritt für alle Items der *header table* durchgeführt, ist der komplette Baum gedeutet und das *frequent itemset* ist vollständig (Tan, Steinbach und Kumar, 2006, S. 367ff.).

Regelerzeugung

Da in der Literatur zu dem FP-Growth Algorithmus keine Methodiken zur Regelerzeugung vorgestellt werden, wird die Theorie der Regelerzeugung von Apriori angewandt, welche bereits in Kapitel 2.2.1 vorgestellt wurde. Aufgrund dieser Thematik wird im späteren Vergleich der Algorithmen nur die Suche nach den *frequent itemsets* einbezogen und nicht die Regelerzeugung selbst.

3. Implementierung & Design

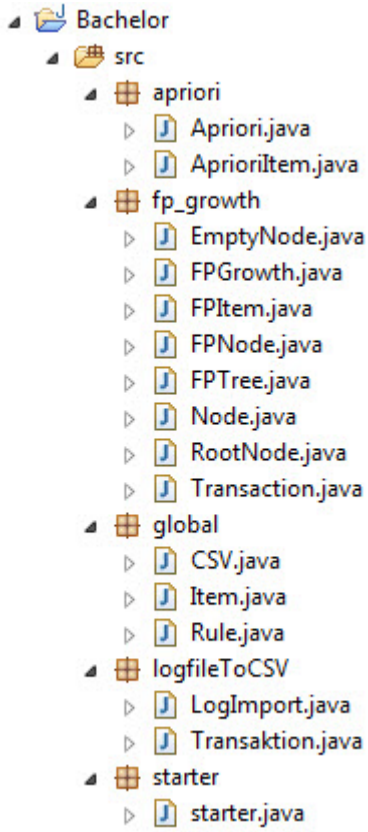


Abbildung 3.1.: Auszug des *Package Explorers*

Folgend wird die Implementierung und das Design für eine erfolgreiche Assoziationsanalyse vorgestellt. Abbildung 3.1 zeigt den *Package Explorer* des Projektes, der alle erstellten Klassen und *Packages* anzeigt.
Begonnen wird mit einem fachlichen Datenmodell in Form eines einfach dargestellten Klassendiagramms, welches die Struktur der Applikation zeigt. Nachdem der grobe Aufbau erklärt wurde, wird die Bearbeitung des Logfiles in seiner „Rohform“ vorgestellt, danach wird auf die verschiedenen Klassen für die Algorithmen und dann auf die Implementierung der Algorith-

men selbst eingegangen. Am Schluss wird noch die Regelgenerierung und deren Formatierung angesprochen.

3.1. Fachliches Datenmodell

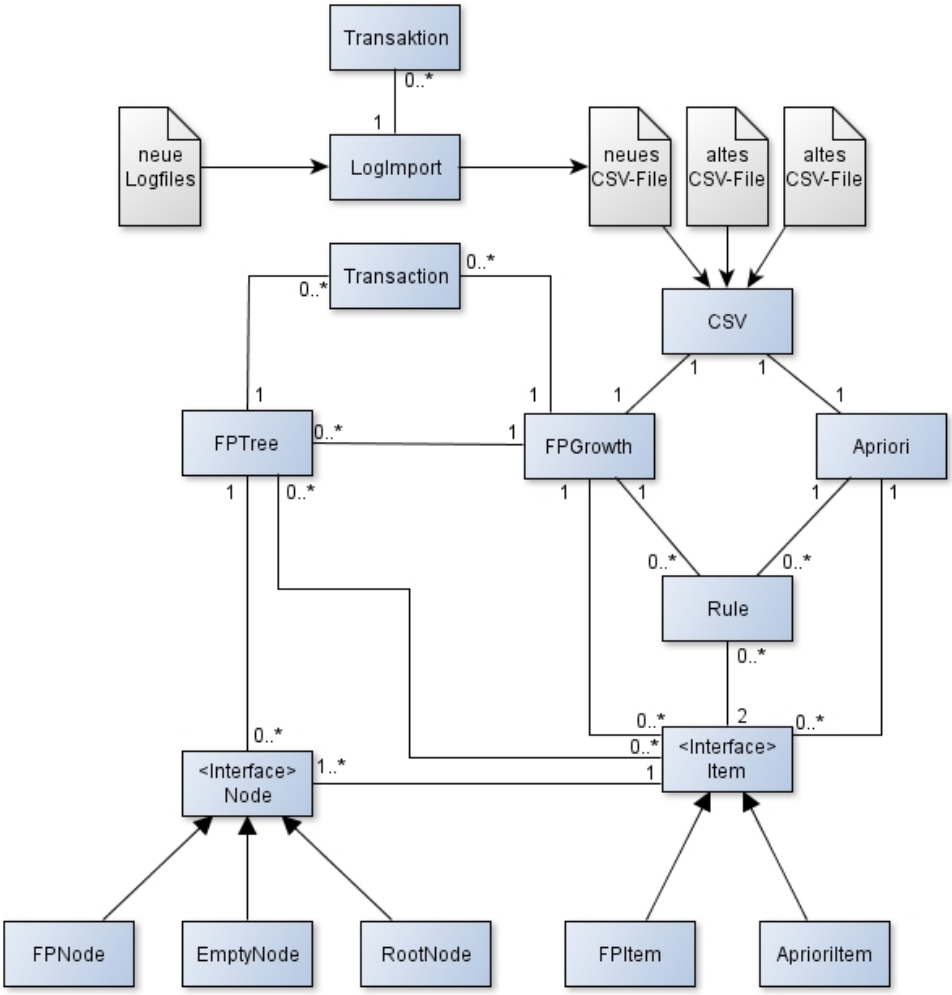


Abbildung 3.2.: Vereinfachtes komplettes Klassendiagramm

Um eine grobe Übersicht über die Struktur der Anwendung zu geben, ist in Abbildung 3.2 ein vereinfachtes Klassendiagramm zu sehen, welches die Abhängigkeiten der einzelnen Klassen darstellt. Die Klasse „LogImport“ im oberen Teil der Abbildung 3.2 zeigt, dass sie mithilfe der neuen Logfiles (welche noch nicht eingelesen wurden) und der Klasse „Transaktion“ eine

neue CSV-Datei erstellt. Diese neue CSV-Datei wird zusammen mit den bereits vorhandenen CSV-Dateien von der Klasse „CSV“ eingelesen. Ein Objekt der Klasse „CSV“ wird von dem Apriori-Algorithmus (realisiert in der Klasse „Apriori“) als Datenbasis genutzt. Die von dem Apriori-Algorithmus genutzten Itemsets werden durch die Klasse „AprioriItem“ repräsentiert, welche das Interface „Item“ implementiert. Außerdem generiert der Algorithmus eine Menge von Regeln, welche durch die Klasse „Rule“ repräsentiert werden.
Auch der Algorithmus FP-Growth, der in der Klasse „FPGrowth“ implementiert ist, nutzt ein CSV-Objekt als Datenbasis. Da in diesem Algorithmus die einzelnen Datensätze sortiert und reduziert werden, dient die Klasse „Transaction“ dazu, diese darzustellen. Außerdem stellt die Klasse „FPTree“ einen *Frequent Pattern Tree* dar, welcher aus einzelnen Knoten besteht. Jede Knotenart implementiert das Interface „Node“. Vorhandene Knotenarten sind „FPNode“ (ein normaler Knoten), „EmptyNode“ (ein leerer Knoten) und „RootNode“ (ein Knoten, der die Wurzel eines Baumes repräsentiert). Ein Item des Algorithmus' FP-Growth wird durch die Klasse „FPItem“, welche ebenfalls das Interface „Item“ implementiert, definiert.

Folgend wird ab Kapitel 3.3 die Implementierung der einzelnen Klassen näher erläutert. Zu jeder Klasse ist dort auch das dazugehörige Klassendiagramm zu finden.

3.2. Ablauf der Ausführung

In Abbildung 3.3 ist ein Sequenzdiagramm zu sehen, welches den groben Ablauf der Ausführung des Programms darstelllt. An dieser Stelle soll betont werden, dass es nur einen groben Überblick verschaffen soll und nicht vollständig ist.
In dem Diagramm ist die Klasse „starter“ zu sehen, welche Aufgaben an die anderen Klassen deligiert. Über `transform(logfilePath, csvName)` wird die Klasse „LogImport“ angewiesen, alle Logfiles in dem übergebenen Verzeichnis auszulesen, zusammenzufassen, zu anonymisieren und als CSV-Datei mit dem Namen $csvName$ zu speichern.
Nun wird ein CSV-Objekt erstellt, dessen Konstruktor der Verzeichnispfad zu den CSV-Dateien übergeben wird. Alle Dateien in diesem Pfad werden eingelesen und innerhalb des Objektes als Datenbasis gespeichert.
Nun werden die Algorithmusobjekte erstellt. Hierfür dienen die Klassen „Apriori“ bzw. „FPGrowth“. Über den Konstruktor werden ihnen die Mindestwerte für *Support* und *Konfidenz*, sowie das CSV-Objekt übergeben. Aus dem CSV-Objekt holen sich die Klassen über die Methode `getData()` die Datenbasis.
Wird `useAlgorithm()` auf dem FPGrowth-Objekt angewendet, wird die Datenbasis für die

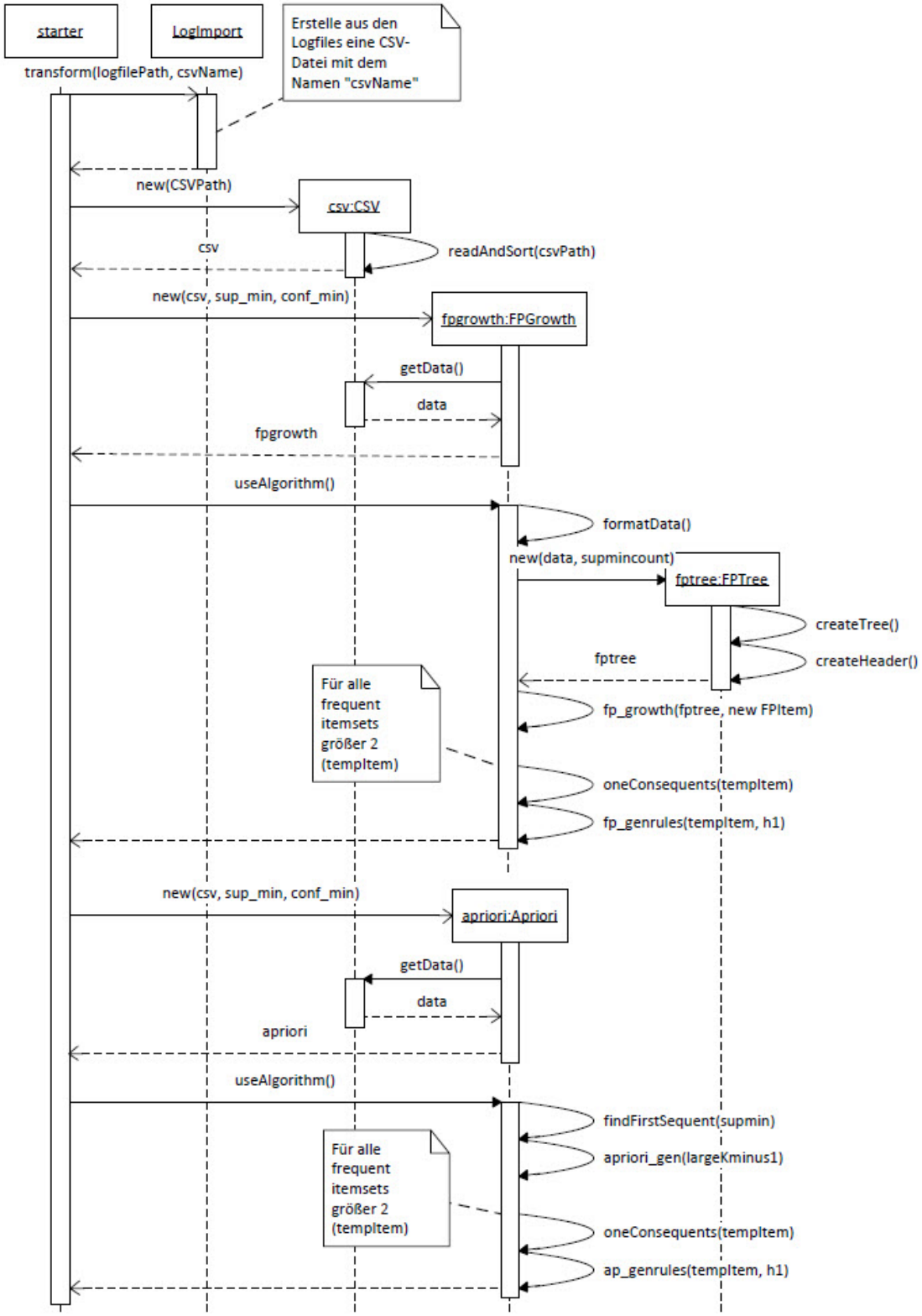


Abbildung 3.3.: Sequenzdiagramm des groben Ablaufs

interne Weiterverarbeitung umgeformt. Daraufhin wird ein neues FPTree-Objekt erstellt und wenn dessen *header table* nicht leer ist, die methode `fp_growth(fptree, item)` aufgerufen, wobei das übergebene Item keine Elemente beinhaltet. Da das Sequenzdiagramm, wie bereits erwähnt, sehr kompakt ist, wird an dieser Stelle auf Kapitel 3.8 verwiesen.
Für die Regelerzeugung wird jedes *frequent itemset*, das mehr als ein Element beinhaltet, der Methode `oneConsequents(itemset)` übergeben. Danach dient der Rückgabewert ($h1$) zusammen mit dem *itemset* der Methode `fp_genrules(itemset, h1)` als Parameter. Wurde dies für jedes *frequent itemset* durchgeführt, beinhaltet eine Klassenvariable ($ruleSet$), die Menge aller Regeln, die den Mindestwert für *Konfidenz* erfüllen.
Wird `useAlgorithm()` auf dem Apriori-Objekt angewendet, werden mithilfe der Methode `findFirstSequent(supmin)` alle einelementigen *frequent itemsets* ermittelt. Danach werden aus diesen $(k-1)$-elementigen *frequent itemsets* mithilfe der Methode `apriori_gen(k-1frequentItems)` alle k-elementigen Kandidaten generiert. Das Vorkommen der Kandidaten in der Datenbasis wird gezählt und wenn ein Kandidat den Mindestsupport erfüllt, der Menge der *frequent itemsets* hinzugefügt. Auch hier wird für den genaueren Ablauf auf Kapitel 3.6 verwiesen.
Die Regelgenerierung erfolgt auf dieselbe Art wie die des FPGrowth-Objekts.

3.3. Anonymisierung der Logfiles

Für die Implementierungstätigkeit des Datenimports sind verschiedene Schritte notwendig. Begonnen wird mit der Zusammenfassung der Logfiles, in der alle wichtigen Informationen aus dem Log extrahiert werden. Daraufhin werden diese zusammengefassten Informationen als CSV-Datei abgespeichert, damit sie von den Algorithmen importiert und genutzt werden können. Diese Schritte werden nachfolgend näher erläutert.

3.3.1. Vorverarbeitung der Logfiles

Als Basisdaten des Imports dienen die Webserver Logfiles, welche von dem Department Technik und Informatik der HAW-Hamburg bereit gestellt werden. Da die Rohdaten noch Spalten beinhalten, die für das Anwendungsgebiet der Assoziationsanalyse unwichtig sind, war die Idee sie „vorzufiltern“, das heißt, alle Spalten zu entfernen, die für die weitere Entwicklung nicht erforderlich sind. Anfangs war die Grundidee, dass dies bei jedem Logfile von Hand geschieht. Da allerdings das bearbeitende Logfile (von 2.November 2011 - 12.März 2012) ca. 2,6 Millionen Einträge beinhaltet und rund 670 Megabyte groß ist, ließ es sich zum einen nicht mehr von jeder Bearbeitungssoftware öffnen und zum anderen ist die manuelle Bearbeitung

jedes neuen Logfiles in der Praxis nicht praktikabel.
Aus diesem Grund wurde auf die Vorfilterung verzichtet und im weiteren Ablauf die benötigten Informationen (IP-Adresse, Zeitstempel und Seitenaufruf) direkt aus den Zeilen geholt.

3.3.2. Schritt 1 - Zusammenfassung der Logfiles

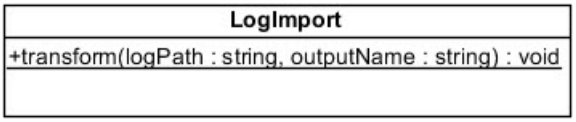


Abbildung 3.4.: Klassendiagramm „LogImport“

In Abbildung 3.4 ist das Klassendiagramm der Klasse „LogImport“ gegeben. Diese dient dazu, die Logfiles zusammenzufassen, die Daten zu anonymisieren und nur die benötigten Informationen in einer CSV-Datei zu speichern. Hierfür wird der Methode `transform(logPath, outputName)` ein Verzeichnispfad übergeben, aus dem alle Logfiles nacheinander ausgelesen werden. Dabei muss beachtet werden, dass die Namensgebung der einzelnen Logs wohl durchdacht sein sollte. Empfohlen wird eine Speicherung in der Form YYYY-MM-DD[1], da das Auslesen des Verzeichnisses intern wie folgt implementiert ist:

```
File dir = new File(path);
File[] fileArray = dir.listFiles();
```

Angenommen, es werden monatlich die Logfiles manuell und nicht automatisiert gespeichert, dann kann es vorkommen, dass diese nicht am ersten eines Monats, sondern erst am zweiten oder dritten eines Monats gespeichert werden. Anhand eines Beispiels wird die Ausführungsreihenfolge im YYYY-MM-DD Format mit der im DD-MM-YYYY Format verglichen. Die Logs für Januar, Februar, März und April werden in beiden Fällen in vier Logfiles gespeichert:
Fall 1 - Werden die Logfiles wie nachstehend benannt: 01-01-2012.log (1), 03-02-2012.log (2), 02-03-2012.log (3) und 01-04-2012.log (4); lautet die Bearbeitungsreihenfolge für diese 1 - 4 - 3 - 2.
Fall 2 - Werden die Logfiles wie nachstehend benannt: 2012-01-01.log (1), 2012-02-03.log (2), 2012-03-02.log (3) und 2012-04-01.log (4); lautet die Bearbeitungsreihenfolge 1 - 2 - 3 - 4.

Die Implementierung erfordert eine chronologische Anordnung der Logs, wobei das älteste Log als erstes eingelesen und bearbeitet werden muss. Weicht die Reihenfolge der eingelesenen

[1] YYYY bezeichnet das vierstellige Jahr, MM den zweistelligen Monat und DD den zweistelligen Tag.

Logs ab, kann es zu Problemen mit den Zeitstempeln kommen.
Die Idee hinter dem Prinzip, ein ganzes Verzeichnis anstelle einer einzelnen Datei einzulesen, ist folgende:
Angenommen, es werden jeden Monat die Logfiles gespeichert, so ist es möglich, diese in einem Ordner zu speichern, ohne sie umständlich zusammenzufügen. Bei dem Zusammenfügen entsteht nicht nur zusätzlicher Aufwand, sondern auch das Problem, dass je nach Bearbeitungsprogramm eventuell eine Zeilenbegrenzung erreicht wird.
Wird das in dieser Arbeit verwendete Logfile (2.631.315 Zeilen) beispielsweise von Microsoft Excel 2007 geöffnet, erscheint eine Fehlermeldung, da nur 1.048.576 Zeilen unterstützt werden.

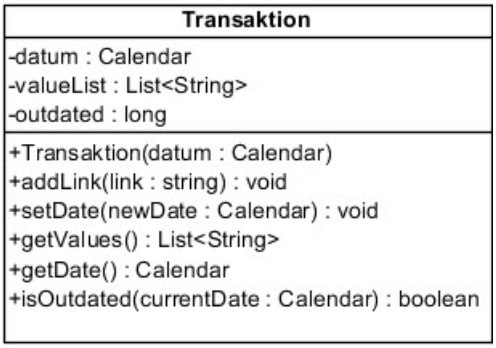


Abbildung 3.5.: Klassendiagramm „Transaktion“

Da es sich bei den IP-Adressen um sensible Daten handelt und man diese nicht direkt in der Assoziationsanalyse benötigt, werden sie zusammen mit dem Zeitstempel zu Transaktionen vereinigt. Aus diesem Grund wurde die Klasse „Transaktion“ implementiert (siehe Abbildung 3.5). Ein Objekt dieser Klasse repräsentiert eine Transaktionen aus dem Logfile und beinhaltet die Informationen:

- **datum:** Zeitstempel der Transaktion, bestehend aus dem Datum sowie aus der Uhrzeit der Anfrage.
- **valueList:** Liste der aufgerufenen Seiten.
- **outdated:** Ein Zeitintervall (in Minuten), das angibt, wie lange ein Seitenaufruf der gleichen IP derselben Transaktion zugeordnet wird.

Ablauf:

Zu Beginn wird zeilenweise durch das Logfile iteriert und aus jeder Zeile die IP-Adresse (*String*), der Zeitstempel (*Calendar-Objekt*) und der Seitenaufruf (*String*) geholt. Mithilfe der IP-Adresse wird nun in der *ipMap* (*HashMap*) geschaut, ob zu dieser IP eine aktive Transaktion existiert. Ist dies der Fall, wird der Zeitstempel betrachtet.
Ist die letzte Anfrage länger als dreißig Minuten her, wird eine neue Transaktion (Objekt der Klasse „Transaktion“) erstellt und als *value* zusammen mit der IP-Adresse als *key* in die *ipMap* geschrieben und somit der vorhandene Eintrag überschrieben. Sie wird sozusagen als neue Transaktion und somit unabhängig von der vorhandenen Transaktion betrachtet. Außerdem wird die Transaktion in das *resultSet* (*HashSet*) geschrieben. Dasselbe passiert, wenn eine IP-Adresse noch nicht in der *ipMap* vorhanden ist. Das *resultSet* beinhaltet am Ende der Ausführung alle vorhandenen Transaktionen. Eine beispielhafte Umformung ist in Abbildung 3.6 zu sehen.
Sind seit der letzten Anfrage weniger als dreißig Minuten vergangen, wird der Zeitstempel der Transaktion aktualisiert und der dazugehörige Seitenaufruf dem Objekt hinzugefügt.
Es werden keine mehrfachen Seitenaufrufe gespeichert (keine Duplikate erlaubt), da es für die Assoziationsanalyse nicht von Bedeutung ist. Um die *ipMap* möglichst klein zu halten, wird diese nach jeweils 5.000 eingelesenen Zeilen bereinigt. Die „Reinigung“ beinhaltet, unter Betrachtung des Zeitstempels der aktuellen Zeile, eine Iteration durch die *Map*. Jeder Zeitstempel eines Eintrags wird mit dem aktuellen Zeitstempel verglichen und bei einer Differenz von mehr als dreißig Minuten aus der *Map* entfernt. Mit dem Wert 5.000 wurde in dieser Implementierung die beste Bearbeitungsgeschwindigkeit gemessen.
Ist die Zusammenfassung der Logfiles für das komplette Verzeichnis erfolgt, existiert in Form des *resultSets* eine solide Basis, um bei Schritt 2 fortzufahren.

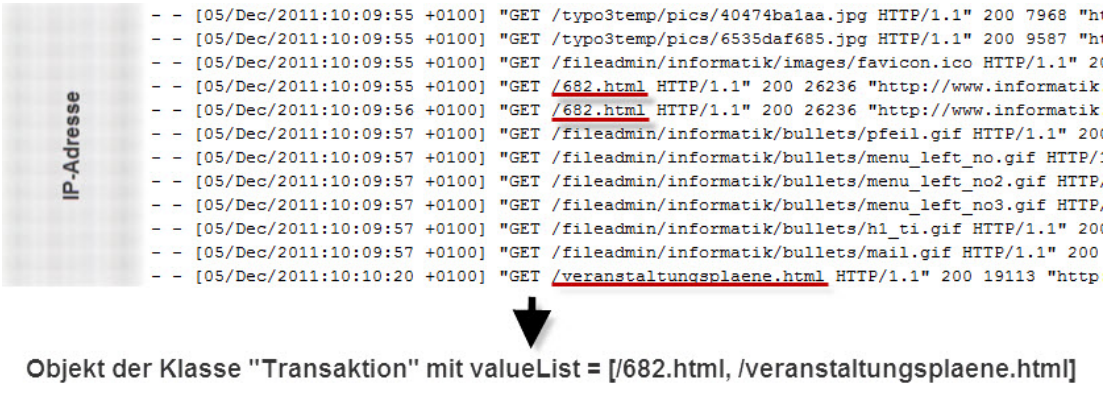


Abbildung 3.6.: Beispielhafte Umformung eines Logfiles zu einem Objekt „Transaktion“

3.3.3. Schritt 2 - Sicherung der Daten als CSV

Bevor die Sicherung der Daten beginnen kann, muss klar festgelegt werden, wie die CSV-Datei formatiert ist. Ohne eine klare Festlegung der Formatierung kann das spätere fehlerlose Einlesen der CSV-Dateien nicht sichergestellt werden. In dem Fall dieser Arbeit wird als Trennzeichen das Semikolon gewählt. Die Liste der in der Transaktion vorhandenen Elemente wird mit einer eckigen öffnenden Klammer eingeleitet und mit einer eckigen schließenden Klammer abgeschlossen. Innderhalb dieser Liste werden die einzelnen Elemente mithilfe von Kommata separiert. Zur Verdeutlichung folgt der Aufbau für die erste Transaktion, die die Elemente A, B und C enthält.

$$1; [A, B, C]$$

Um die zusammengefassten Logfiles (in Kapitel 3.3.2) nun in einer CSV-Datei zu speichern, wird durch das *resultSet* iteriert und jede Transaktion betrachtet (siehe Listing 3.1). Da nur die Transaktionen von Bedeutung sind, die mehr als eine Seite beinhalten, wird jede auf ihre Seitenaufrufe hin untersucht und nur die mit mindestens zwei Aufrufen in die CSV-Datei geschrieben. Auch wird, da durch diese Reduktion Datensätze wegfallen, die Transaktionsnummer neu generiert, damit diese fortlaufend ist und keine Lücken aufweist.

```
1 Iterator<Transaktion> it = resultSet.iterator();
2 while (it.hasNext()){
3   currentTransaktion = it.next();
4   if (currentTransaktion.getValues().size() > 1){
5     out.write(counter+" ; "+currentTransaktion.getValues()+"\r\n
          ");
6     counter++;
7   }
8 }
```

Listing 3.1: Sicherung der Transaktionen

So werden diese Daten im gewünschten Import-Format für die CSV-Dateien gespeichert (Spalte 1 -> (neue) Transaktionsnummer; Spalte 2 -> Seitenaufrufe). Ein Ausschnitt der erstellten CSV-Datei ist in Abbildung 3.7 zu sehen.

```
1;[/134.html, /1962.html, /1968.html, /1978.html, /1980.html, /1988.html, /1964.h
2;[/stisys.html, /studierende.html]
3;[/index.php?id=1669, /studierende.html]
4;[/meisel.html, /wp_robot_vision.html]
5;[/54.html, /index.php?id=1669]
6;[/forschung_kooperation.html, /studierende.html]
7;[/studierende.html, /veranstaltungsplaene.html, /1781.html, /691.html, /681.htm
8;[/studierende.html, /hv.html]
9;[/studierende.html, /649.html]
10;[/kontakt.html, /professoren.html, /heitmann.html, /10.html]
11;[/professoren.html, /meisel.html, /wp_robot_vision.html]
12;[/westhoff.html, /6.html]
13;[/professoren.html, /index.php?id=33, /baran.html]
```

Abbildung 3.7.: Ausschnitt aus der CSV-Datei nach der Anonymisierung

3.4. Implementierung des Datenimports

Für den Datenimport ist eine Klasse namens „CSV“ zuständig (siehe Abbildung 3.8). Bei der Erstellung eines Objektes dieser Klasse wird, über den Konstruktor, der Pfad zu den CSV-Dateien übergeben. Alle Dateien, die sich in diesem Ordner befinden, werden eingelesen und verarbeitet. Hierbei wird die in Kapitel 3.3.3 angesprochene Formatierung vorausgesetzt. Haben die Dateien intern eine andere Formatierung, kann die Interpretation dieser fehlerhaft sein. Wird die Formatierung allerdings eingehalten, ist es irrelevant, welche Daten eingelesen werden. Somit ist das Programm auch für andere Anwendungszwecke der Assoziationsanalyse geeignet. Beispielsweise könnte eine Transaktion anstelle der Seitenaufrufe auch die Artikel eines Einkaufs beinhalten.

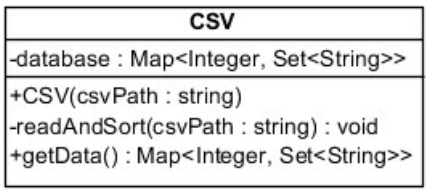


Abbildung 3.8.: Klassendiagramm „CSV“

Dadurch, dass auch andere CSV-Dateien eingelesen werden können, werden bei Objekterstellung erneut alle Transaktionen entfernt, deren zweite Spalte weniger als oder genau einen Eintrag enthält. Diese Daten werden intern in einer *HashMap* gespeichert, die als *key* einen *Integer* und als *value* ein Set mit *Strings* beinhaltet. Über einen *Getter* kann diese *Map* aus dem Objekt geholt werden.

3.5. Darstellung der Regeln und Itemsets

Im folgenden Abschnitt werden das Interface „Item“ und alle Klassen für die Darstellung der Regeln und Itemsets hinreichend erklärt. Angefangen wird mit dem Interface „Item“, welches von den Klassen „AprioriItem“ und „FPItem“ implementiert wird, danach folgen die beiden erwähnten Klassen und die Klasse „Rule“. Wie schon bei den zuvor beschrieben Klassen können die Abhängigkeiten in Abbildung 3.2 eingesehen werden.

3.5.1. Item

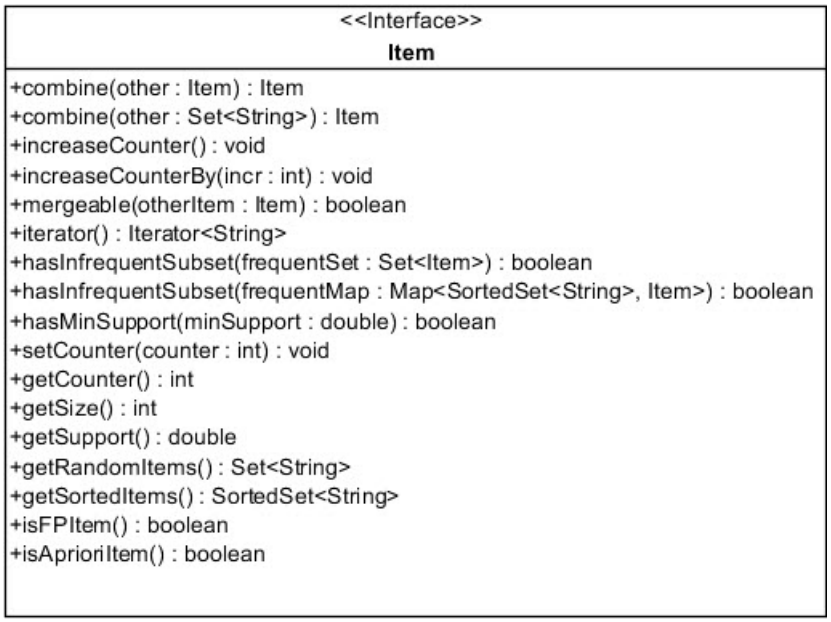


Abbildung 3.9.: Klassendiagramm des Interface „Item“

„Item“ ist ein Interface zur Repräsentation der in den Algorithmen vorhanden Itemsets. Das Klassendiagramm des Interfaces „Item“ ist in Abbildung 3.9 zu sehen. Die Beschreibung der einzelnen Methoden erfolgt hier, damit dies nicht doppelt in den Beschreibungen der implementierenden Klassen geschehen muss. Die Methoden und deren Beschreibung sind wie folgt:

- **combine(Item other):** Erstellt ein neues Item, welches die Vereinigung der Elemente aus diesem und dem Item *other* beinhaltet.
- **combine(Set<String> other):** Erstellt ein neues Item, welches die Vereinigung der Elemente aus diesem Item und denen der Menge *other* beinhaltet.

- **increaseCounter():** Erhöht den Zähler des Vorkommens um Eins und führt eine erneute Berechnung des Supportwertes durch.
- **increaseCounterBy(int incr):** Erhöht den Zähler des Vorkommens um *incr* und führt eine erneute Berechnung des Supportwertes durch.
- **mergeable(Item otherItem):** Prüft, ob beide k-elementigen Items wirklich k-Elemente beinhalten. Wenn dies gegeben ist, werden schrittweise die Elemente beider Items verglichen. Sind alle Elemente bis auf das k-te Element identisch, können die beiden Items miteinander vereinigt werden und die Methode gibt den Wert *true* zurück. Sind sie nicht identischt, wird der Wert *false* zurückgegeben.
- **iterator():** Gibt einen Iterator über die Elemente des Itemsets zurück.
- **hasInfrequentSubset(Map<Set<String>,Item> frequentMap):** Diese Methode bekommt im k-ten Schritt eine Liste aller $k-1$-elementigen *frequent sets* übergeben. Mithilfe dieser Liste kann geprüft werden, ob alle Subsets des aktuellen Items Elemente dieser Liste sind. Je nachdem, ob dies der Fall ist oder nicht, wird *true* bzw. *false* zurück gegeben. Diese Methode realisiert den in Kapitel 2.2.1 angesprochenen „prune-step“, der nur im Apriori-Algorithmus zur Anwendung kommt.
- **hasInfrequentSubset(Set<Item> frequentSet):** Die Methode hat die gleiche Funktion wie die oben angegebene, ist allerdings reduziert für die Verwendung der Regelgenerierung im FP-Growth Algorithmus.
- **hasMinSupport(double minSupport):** Prüft, ob der aktuelle Supportwert größer oder gleich dem übergebenen Mindestsupport ist. Als Rückgabe dient hier wieder ein Boolean *true* oder *false*.
- **isAprioriItem():** Gibt zurück, ob es sich bei dem Objekt um ein AprioriItem handelt (*true*) oder nicht (*false*).
- **isFPItem():** Gibt zurück, ob es sich bei dem Objekt um ein FPItem handelt (*true*) oder nicht (*false*).

Zudem beinhaltet das Interface noch einen *Setter* für die Anzahl des Vorkommens eines Items und einen *Getter* für (1) die Menge mit den Elementen (in sortierter oder unsortierter Reihenfolge), (2) deren Anzahl, (3) den Supportwert und (4) die Anzahl des Vorkommens des Items.

Die beiden Klassen „AprioriItem“ und „FPItem“ implementieren dieses Interface und werden folgend erklärt.

AprioriItem

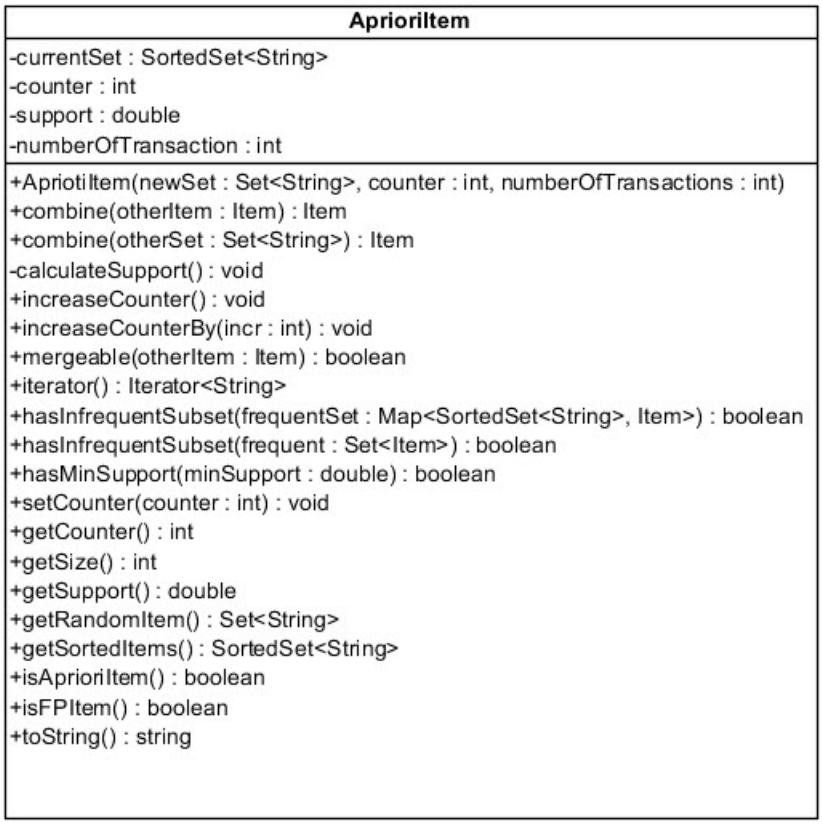


Abbildung 3.10.: Klassediagramm „AprioriItem“

Da in dem Apriori-Algorithmus *large itemsets* und Kandidaten (*candidate itemsets*) generiert werden, werden diese durch eine eigene Klasse namens „AprioriItem“ repräsentiert (siehe Klassendiagramm in Abbildung 3.10). Bei der Erzeugung eines Objektes der Klasse „AprioriItem“ werden die Menge der Elemente (*Set<String>*), die Anzahl des Vorkommens (*counter*) und die Gesamtanzahl aller Transaktionen (*numberOfTransaction*) dem Konstruktor übergeben. Mithilfe dieser Daten kann intern der aktuelle Supportwert berechnet und gespeichert werden. Aufgrund der Anforderung, dass der Apriori-Algorithmus „[...] items within a transaction or itemset [...] sorted in lexicographic order“ erwartet (Han und Kamber, 2001, S. 231), werden die Mengen intern als *TreeSet* implementiert und erfüllen damit diese Bedingung.

```
1 INSERT INTO C_k
2 SELECT p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}
3 FROM L_{k-1} as p, L_{k-1} as q
4 WHERE p.item_1 = q.item_1 and
5       ...
6       p.item_{k-2} = q.item_{k-2} and
7       p.item_{k-1} < q.item_{k-1};
```

Listing 3.2: Pseudocode zum Ablauf des *join-steps* (Quelle: Petersohn, 2005, S. 109)

Das Listing 3.2 zeigt den Pseudocode für den *join-step*. In diesem werden zwei Items miteinander verglichen, indem das $i$te Element eines Items mit dem $i$ten Element des anderen Items verglichen wird. Dies ist nur möglich, da die einzelnen Elemente lexikographisch geordnet sind. Ist das $k-1$te Element des einen Items kleiner als das $k-1$te Element des anderen Items, können diese Items zu einem k-elementigen Kandidaten zusammengefügt werden. Der Vergleich p.item$_{k-1}$ < q.item$_{k-1}$ dient nur zur Sicherstellung, dass keine Duplikate generiert werden (Han und Kamber, 2001, S. 231). In der Applikation wird dies dadurch sichergestellt, dass jedes Item nur genau einmal mit einem anderen Item verglichen wird. Wurde beispielsweise Item A schon mit Item B verglichen, wird nicht noch einmal Item B mit Item A verglichen. Aus diesem Grund wird in der Implementation die Prüfung auf $\neq$ anstelle von $<$ durchgeführt. Die Prüfung, ob die Vorbedingungen des *join-steps* erfüllt werden, wird durch die Methode `mergeable(Item otherItem)` durchgeführt. Liefert sie den Boolean „*true*“ zurück, wird durch `combine(Item otherItem)` ein neues Itemobjekt erstellt und zurückgegeben (siehe oben).
Auch der *prune-step* ist in der Klasse AprioriItem implementiert worden.

```
1 forall itemset c ∈ C_k do
2   forall (k-1)-subsets s of c do
3     if (s ∉ L_{k-1}) then
4       delete c from C_k;
```

Listing 3.3: Pseudocode zum Ablauf des *prune-steps* (Quelle: Agrawal und Srikant, 1994a, S. 490)

In dem Listing 3.3 wird der Pseudocode für den *prune-step* gezeigt, der in der Methode `hasInfrequentSubset(Map<SortedSet<String>,Item> frequentSet)` umgesetzt wurde. Als Parameter (*frequentSet*) wird ihr eine Map aller $k-1$ elementigen *large sets* übergeben.

Es werden alle Subsets eines Kandidaten erstellt und geprüft, ob diese in der Map vorhanden sind. Je nach Ergebnis wird der entsprechende Boolean zurück gegeben.

FPItem

Abbildung 3.11.: Klassendiagramm „FPItem“

Um innerhalb des FP-Growth Algorithmus' ein einzelnes Element oder ein Itemset in der Implementation zu repräsentieren, existiert eine Klasse „FPItem“. Ein FPItem hält intern ein *HashSet* mit *Strings*, einen Zähler für das Vorkommen des Items, die Anzahl aller Transaktionen und den Supportwert. In diese Klasse sind die Methoden `mergeable(Item otherItem)` und `hasInfrequentSubset(Set<Item> frequentSet)` ausschließlich für den *join- und prune-step* der Regelgenerierung vorhanden und werden für das Finden aller *large itemsets* nicht benötigt. Der Aufbau dieser beiden Methoden ist bereits im Kapitel der Klasse „AprioriItem“ in den Listings 3.2 und 3.3 beschrieben.

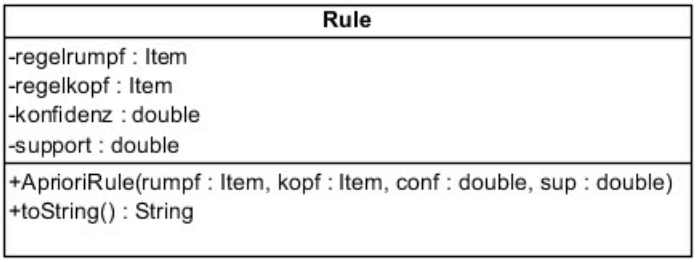


Abbildung 3.12.: Klassendiagramm „Rule“

3.5.2. Rule

Eine Regel, die aus den *large itemsets* gebildet werden kann, wird durch ein Objekt der Klasse „Rule“ repräsentiert. Wie bereits in den Grundlagen erwähnt, besteht eine Regel aus einem Regelrumpf und einem Regelkopf, wobei eine Regel „Wenn Regelrumpf, dann Regelkopf“ gelesen wird.
Wird ein Objekt der Klasse „Rule“ erstellt, werden der Regelkopf, der Regelrumpf, der Supportwert und der Konfidenzwert der Regel dem Konstruktor übergeben (siehe Klassendiagramm in Abbildung 3.12). Der Supportwert der Regel ist der Supportwert des „Superitem“. Mit dem Begriff „Superitem“ ist das Item gemeint, aus dem die Regel hervorgeht. Wird beispielsweise die Regel $A \Rightarrow B$ betrachtet, ist ihr Superitem das *frequent itemset* AB.
Die Regelerstellung basiert auf der Tatsache, dass bei einer beispielhaften Regel $\{A\} \rightarrow \{B\}$ die *Konfidenz* durch die Formel

$$conf\left(\{A\} \rightarrow \{B\}\right) = \frac{sup(\{A\})}{sup(\{A, B\})}$$

berechnet wird. Da ein Objekt der Klasse „Rule“ allerdings keine komplexe Logik beinhaltet, sondern ihr alle Werte über den Konstruktor übergeben werden, wird die eigentliche Regelerstellung im Kapitel 3.6 näher erläutert.

3.6. Implementierung des Apriori Algorithmus

Für das Anwenden des in Kapitel 2.2.1 erläuterten Apriori-Algorithmus’ dient die Klasse „Apriori“ (siehe Klassendiagramm in Abbildung 3.13). Bei Objekterstellung wird sowohl ein CSV-Objekt wie auch der Minimalwert für *Support* und *Konfidenz* übergeben. Die Implementierung des Algorithmus’ basiert auf dem Pseudocode aus dem Listing 3.4.

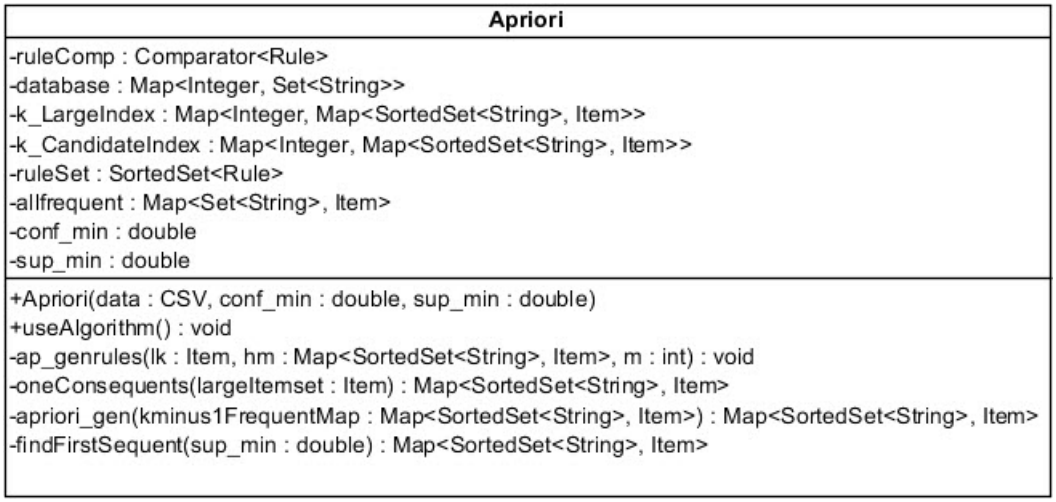


Abbildung 3.13.: Klassendiagramm „Apriori“

```
1  L_1 = findFirstSequent(sup\_min);
2  for (k = 2; L_{k-1} ≠ ∅; k++){
3    C_k = apriori_gen(L_{k-1});
4    forall transactions t ∈ D {
5      C_t = subset(C_k, t);
6      forall candidates c ∈ C_t do
7        c.count++;
8    }
9    L_k = {c ∈ C_k | c.count ≥ minsup}
10 }
11 Answer = ⋃_k L_k;
```

Listing 3.4: Pseudocode zum Ablauf des Apriori-Algorithmus’ (Quelle: In Anlehnung an Agrawal und Srikant, 1994a, S. 489)

Intern sind zwei *HashMaps* für die Speicherung der Kandidaten (*k_CandidateIndex*) und der *large itemsets* (*k_LargeIndex*) zuständig. Innerhalb dieser Maps werden Einträge gespeichert, die als *key* einen Index und als *value* wieder eine *Map<SortedSet<String>,Item>* mit den *itemsets* als Menge von *Strings* und dem dazugehörigem Item-Objekt enthält. Ein Beispiel für die Map *k_CandidateIndex* ist in Abbildung 3.14 zu sehen.
Am Anfang des Algorithmus’ muss einmal die Menge aller einelementigen *large itemsets* erstellt werden. Dies geschieht durch die private Methode `findFirstSequent(double sup_min)`. Bei der Ausführung wird durch alle Items (*String*) iteriert und diese in ein Set eingefügt. Ist

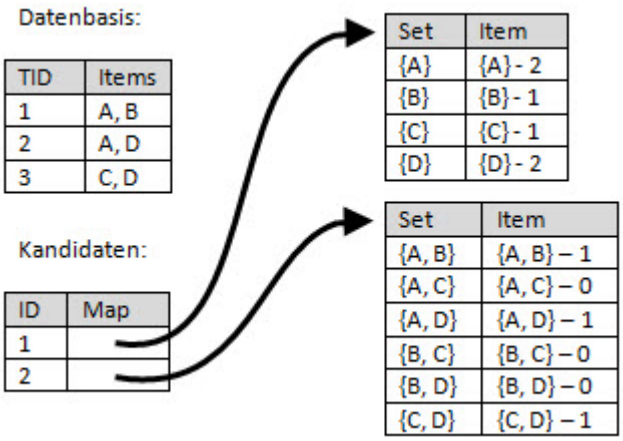


Abbildung 3.14.: Beispiel für *k_CandidateIndex*

das Set in der *candidateMap* noch nicht vorhanden, wird aus diesem Set ein Item erstellt und das Set als *key* und das Item als *value* in die *candidateMap* gespeichert. Ist es bereits vorhanden, wird das entsprechende Item aus der Map geholt und sein Vorkommen durch die Methode Item.`increaseCounter()` erhöht. Sind alle einelementigen Kandidaten erstellt, wird durch die Map iteriert und jedes Item auf dessen Supportwert hin untersucht. Ist der Wert größer oder gleich dem Mindestsupport, wird der Eintrag Set → Item (key → value) in die *resultMap* geschrieben. Diese wird am Ende der Methode zurückgegeben. Die Methode `findFirstSequent(sup_min)` wird nur einmalig am Anfang des Algorithmus' aufgerufen.
Die Methode `apriori_gen(Map<SortedSet<String>,Item> kminus1frequentMap)` generiert aus k-1-elementigen *large itemsets* die k-elementigen Kandidaten. Innerhalb dieser Methode werden sowohl der *join-step* (siehe Listing 3.2) wie auch der *prune-step* durchgeführt (siehe Listing 3.3). Da der *prune-step* und der *join-step* innerhalb der Klasse „AprioriItem" geschehen, wurde die Implementierung dieser Schritte in Kapitel 3.5.1 beschrieben.
Wurden die k-elementigen Kandidaten generiert, wird sowohl durch alle Transaktionen wie auch durch die k-elementigen Kandidaten iteriert und das Vorkommen der Kandidaten gezählt. Danach wird der Supportwert jedes Kandidaten mit dem Mindestsupport verglichen und alle Kandidaten, die diesen Mindestwert oder einen größeren Supportwert haben, in der Map der k-elementigen *large itemsets* gespeichert.
Dieser Vorgang wird so lange wiederholt, bis L_k leer ist und somit keine Elemente mehr enthält. Zu diesem Zeitpunkt ist der Apriori-Algorithmus beendet und die Klassenvariable *k_LargeIndex* beinhaltet alle *large itemsets*, aus denen unter Betrachtung der *Konfidenz* die Regeln erstellt werden können.

Für die Regelerstellung folgt ein Listing, welches den Ablauf mithilfe von einem Pseudocode darstellt:

```
1  forall large k-itemsets l_k, k ≥ 2 do begin
2    H_1 = oneConsequents(l_k);
3    call ap_genrules(l_k, H_1)
4  end
5
6  procedure oneConsequents(l_k: large k-itemset)
7    L = {(k-1)-itemsets l_{k-1} | l_{k-1} ⊂ l_k};
8    forall l_{k-1} ∈ L do begin
9      conf = support(l_k)/support(l_{k-1});
10     if (conf ≥ minconf) then
11       output the rule l_{k-1} ⇒ (l_k − l_{k-1});
12   end
13
14 procedure ap_genrules(l_k: large k-itemset, H_m: set of m-item
       consequents)
15   if (k > m+1) then begin
16     H_{m+1} = apriori_gen(H_m);
17     forall h_{m+1} ∈ H_{m+1} do begin
18       conf = support(l_k)/support(l_k − h_{m+1});
19       if (conf ≥ minconf) then
20         output the rule (l_k − h_{m+1}) ⇒ h_{m+1};
21       else
22         delete h_{m+1} from H_{m+1}
23     end
24     call ap_genrules(l_k, H_{m+1});
25   end
```

Listing 3.5: Pseudocode zur Regelerzeugung (Quelle: In Anlehnung an Agrawal und Srikant, 1994b, S. 13f.)

Wie in dem Listing 3.5 zu sehen ist, sind für die Regelerstellung zwei neue Methoden und die schon bekannte Methode `apriori_gen()` zuständig. Die Methode `oneConsequents(Item largeitemset)` dient dazu, alle einelementigen Folgerungen (auch *consequents*) zu finden. Als Folgerung wird hier der hintere Teil einer Regel bezeichnet, der bereits als Regelkopf bekannt ist. Die Regeln mit den einelementigen Folgerungen, deren Konfidenzwert den Wert der Mindestkonfidenz erfüllt, werden in dem *ruleSet* gespeichert und eine Menge dieser „erfolgreichen“

Folgerungen H_1 wird zurückgegeben.
Diese Menge von einelementigen Items H_1 wird zusammen mit dem *large itemset* l_k der Methode `ap_genrules()` übergeben. Beim Aufruf wird auf die Abbruchbedingung geprüft, denn wenn $k \leq m + 1$, sind bereits alle möglichen Regeln erstellt worden. Wird die Abbruchbedingung nicht erfüllt, wird die Methode `apriori_gen()` aufgerufen. Diese dient dazu, erst durch den *join-step* (siehe Listing 3.2) alle möglichen (m+1)-elementigen Regelköpfe zu finden, um diese mithilfe des *prune-steps* (siehe Listing 3.3) zu reduzieren und die reduzierte Menge als mögliche Regelköpfe in H_{m+1} zu speichern.
Danach wird jedes Element von H_{m+1} betrachtet und geprüft, ob der Wert der *Konfidenz* größer als der Mindestwert für *Konfidenz* ist. Wenn dies der Fall ist, wird eine Regel erstellt und in das *ruleSet* hinzugefügt. Ist dies nicht der Fall, wird das Element aus der Menge H_{m+1} entfernt. Danach wird `ap_genrules()` rekursiv mit dem *large itemset* l_k und der Menge H_{m+1} aufgerufen.
So werden für jedes *large itemset* alle Regeln erstellt nach der Theorie, die in Kapitel 2.2.1 beschrieben wurde.

3.7. Erzeugung des FP-Tree

Im folgenden Abschnitt wird die Grundlage des FP-Growth Algorithmus' erklärt, der Aufbau eines *Frequent Pattern Tree*. Hierfür werden die dazugehörigen Klassen (wenn noch nicht geschehen) beschrieben, deren Zusammenhänge in Abbildung 3.2 eingesehen werden können. Dieser Abschnitt bildet die Grundlage für die im nachfolgenden Kapitel angesprochene „Implementierung des FP-Growth Algorithmus'".

3.7.1. Transaction

Die Klasse „Transaction" repäsentiert in der Implementation des FP-Growth Algorithmus' eine Transaktion. Eine Menge dieser Transactions bildet die Datenbasis für den FP-Tree und wird gebildet, indem eine Umformung im FP-Growth Algorithmus durchgeführt wird. Ein Objekt der Klasse „Transaction" beinhaltet eine *LinkedList* mit Items und einen Zähler, der angibt, wie oft diese Transaktion ausgeführt werden soll. Da eine Transaktion nach den Elementen der *header table* sortiert werden soll (siehe im nächsten Abschnitt „FPTree"), existiert innerhalb der Klasse ein *Comparator<Item>*, in dem die Sortierung vorgegeben wird. Außerdem besitzt die Klasse „Transaction" folgende Methoden:

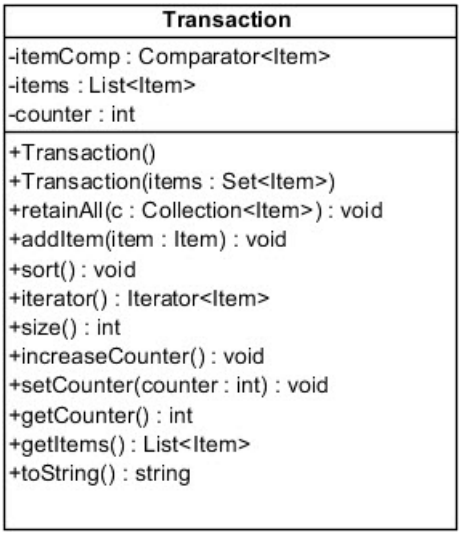


Abbildung 3.15.: Klassendiagramm „Transaction“

- **retainAll(Collection<Item> c):** Entfernt alle Items aus der internen Liste, die nicht die Elemente (Items) der Collection[2] *c* enthält.
- **addItem(Item item):** Fügt ein Item der Liste hinzu.
- **sort():** Sortiert die Liste nach dem Comparator.
- **iterator():** Gibt einen Iterator über die Elemente der Liste zurück.
- **size():** Gibt die Anzahl der Items in der Liste zurück.
- **increaseCounter():** Erhöht den Zähler um 1.

Zusätzlich zu diesen Methoden gibt es noch einen *Setter*, sowie einen *Getter* für den Zähler und einen *Getter* für die Liste mit Items.
Durch `retainAll(Collection<Item> c)` wird ein wichtiger Schritt zur Erstellung eines *Frequent Pattern Trees* realisiert. Als Parameter wird die Menge aller Items der *header table* übergeben und somit alle nicht frequenten Items aus der Transaktion entfernt.

3.7.2. Node

Das Interface „Node“ wird von den Klassen „FPNode“, „EmptyNode“ und „RootNode“ implementiert. Es folgen die Methoden und ihre Funktion, die das Interface kennt:

[2] „*A collection represents a group of objects*“(*Java API 1.4.2 - Collection*)

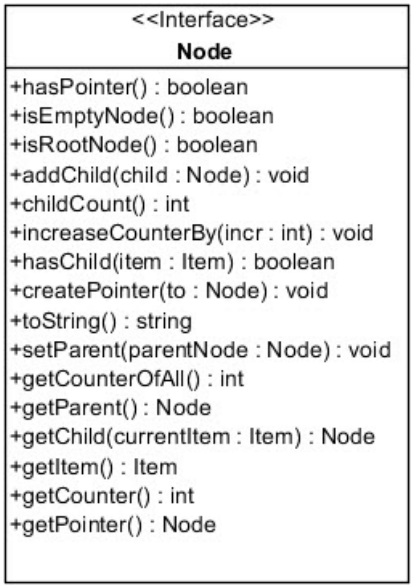


Abbildung 3.16.: Klassendiagramm des Interface „Node“

- **hasPointer():** Liefert den Boolean *true*, wenn der Knoten auf einen anderen Knoten zeigt und dieser nicht leer ist, ansonsten *false*.
- **isEmptyNode():** Liefert *true*, wenn die Node eine EmptyNode ist, ansonsten *false*.
- **isRootNode():** Liefert *true*, wenn die Node eine RootNode ist, ansonsten *false*.
- **addChild(Node child):** Fügt die Node $child$ in die Menge der Kindknoten ein.
- **childCount():** Gibt die Anzahl der Kindknoten zurück.
- **increaseCounterBy(int incr):** Erhöht den Zähler des Knotens um $incr$.
- **hasChild(Item item):** Prüft, ob ein Kindknoten vorhanden ist, der auf dem Item $item$ basiert.
- **createPointer(Node to):** Erstellt den Zeiger auf die Node to.
- **setParent(Node parentNode):** Setzt den Elternknoten auf $parentNode$.

Außerdem kennt es *Getter* für den Elternknoten, einen Kindknoten, das Item, den Zähler und den Knoten, auf den der aktuelle Knoten zeigt.

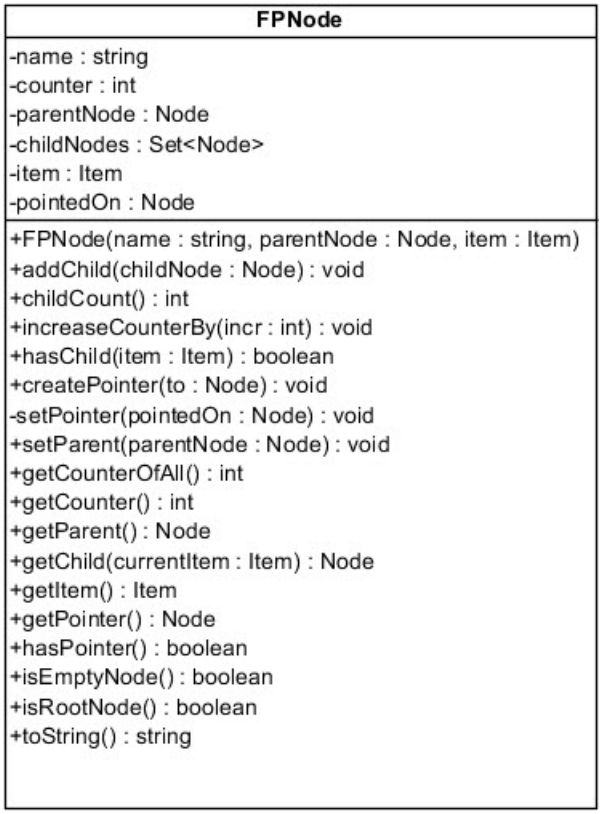


Abbildung 3.17.: Klassendiagramm „FPNode“

FPNode

Ein Objekt der Klasse „FPNode“ repräsentiert einen Knoten des *Frequent Pattern Trees* und implementiert das Interface „Node“. Ein solcher Knoten beinhaltet Informationen über seinen Elternknoten, eine Menge von Kindknoten, einen Namen, einen Zähler für das Vorkommen des Knotens, das Item, von dem der Knoten abstammt und einen „Zeigerknoten“ (siehe Klassendiagramm in Abbildung 3.17). Der Knoten wird in Kapitel 2.2.2 verdeutlicht und stellt die Grundlage für einen FPTree (Kapitel 3.7.3) dar. Wird `createPointer(Node to)` aufgerufen, erfolgt die Abfrage `hasPointer()`. Zeigt die Node auf eine EmptyNode, gibt `hasPointer()` den Wert *false* zurück und mithilfe von `setPointer(to)` wird der Zeiger auf *to* gesetzt. Zeigt die Node allerdings schon auf eine FPNode (Rückgabe *true*), wird die Methode `createPointer(to)` rekursiv auf den gezeigten Knoten aufgerufen.

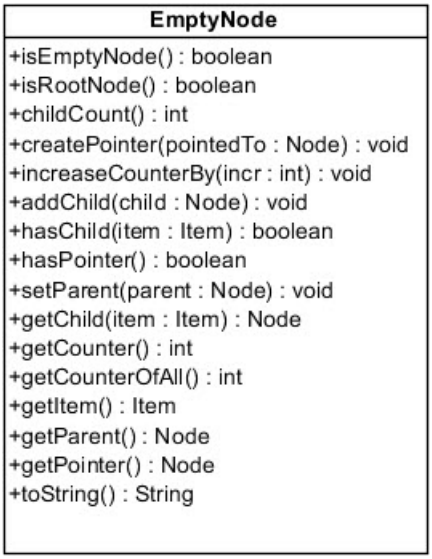


Abbildung 3.18.: Klassendiagramm „EmptyNode“

EmptyNode

Ein Objekt der Klasse „EmptyNode“ wird als Platzhalter für einen Knoten ohne Inhalt genutzt. Sie implementiert das Interface Node und führt keine zusätzlichen Variablen, die Werte speichern.

RootNode

Ein Objekt der Klasse „RootNode“ wird einmalig von einem Objekt der Klasse „FPTree“ erstellt. Es repräsentiert den sogenannten Wurzelknoten und hält nur eine Menge von Kindern. Die Methode `isRootNode()` liefert bei einem RootNode Objekt immer *true* und nur die Methoden, welche die Kindknoten betreffen, liefern einen Wert zurück.

3.7.3. FPTree

Um bei dem FP-Growth Algorithmus die Datenbasis in einer komprimierten Form darzustellen, existiert die Klasse „FPTree“ (siehe Klassendiagramm in Abbildung 3.20). Dieser Baum besteht aus verschiedenen Knoten (auch Nodes genannt) und hält intern eine Menge von Transaktionen (Objekte der Klasse „Transaction“), eine Menge aller Knoten, die *header table*, die RootNode und den Mindestwert für den *Support*. Um einen FP-Tree zu erstellen, werden zwei Schritte durchgeführt. Im ersten Schritt wird die *header table* erstellt. Hierfür wird durch alle Transaktionen iteriert und das Vorkommen der Elemente gezählt. Diese Elemente (in-

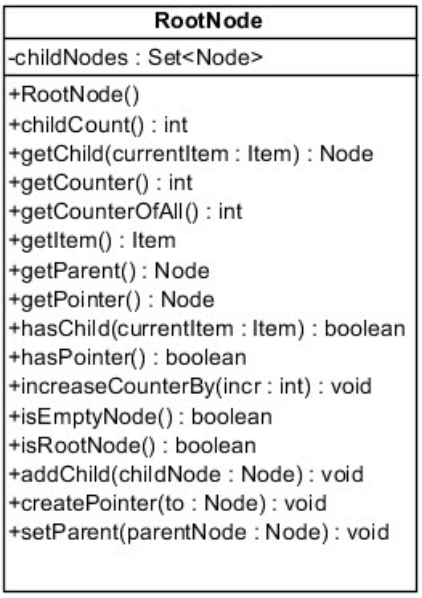


Abbildung 3.19.: Klassendiagramm „RootNode“

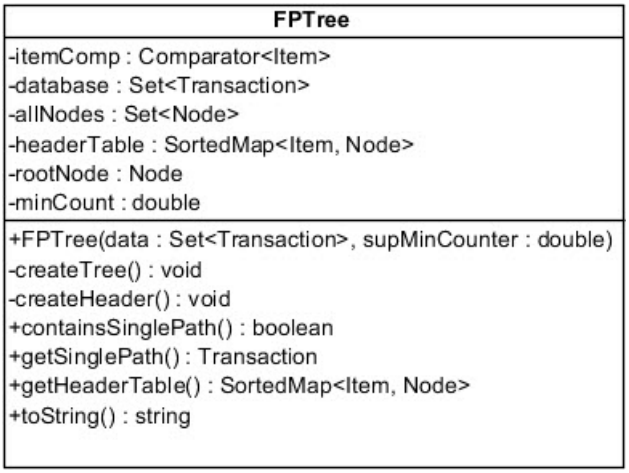


Abbildung 3.20.: Klassendiagramm „FPTree“

tern Objekte der Klasse Item) werden auf ihren Supportwert hin untersucht und, wenn sie *frequent* sind, in der *header table* (intern eine *TreeMap<Item, Node>*) als *key* gespeichert. Als *values* bekommen die einzelnen Elemente der *header table* eine EmptyNode zugewiesen. Ein Comparator dient der Sortierung der *header table* nach Items.
Im zweiten Schritt wird der eigentliche Baum erstellt. Jede Transaktion wird einzeln betrachtet. Durch die in Kapitel 3.7.1 genannte Methode `retainAll(Collection<Item> c)` werden alle Elemente entfernt, die nicht in der *header table* vorhanden sind. Da der für die *header table* genutzte Comparator und der Comparator der „Transaction“ der gleiche ist, ist somit die Anordnung der Elemente aus der Transaktion dieselbe wie die der Elemente in der *header table*.
Für jedes Item einer Transaktion (*currentItem*) wird geprüft, ob der aktuell betrachtete Knoten (*currentNode*) einen Kindknoten für das Item besitzt. Wenn ja, wird `currentNode = currentNode.getChild(currentItem)` gesetzt und der Zähler des Kindknotens um den Zähler der Transaktion erhöht. Existiert noch kein Kindknoten, wird dieser erstellt, sein Vorkommen auf das der Transaktion gesetzt und als die neue *currentNode* definiert. Außerdem wird geprüft, ob der Wert in der *header table* für das *currentItem* eine EmptyNode ist. Ist dies der Fall, ist der neue Wert die *currentNode*, wenn nicht, wird auf der vorhandenen FPNode die Methode (siehe Kapitel 3.7.2) `createPointer(currentNode)` aufgerufen.
Die Klasse „FPTree“ beinhaltet zusätzlich noch folgende Methoden:

- **containsSinglePath():** Prüft, ob alle Nodes in dem Baum (inklusive der Wurzel „root-Node“) nur maximal einen Kindknoten haben. Wenn ja, wird der Boolean *true* zurückgegeben, ansonsten der Boolean *false*.
- **getSinglePath():** Gibt ein Objekt der Klasse Transaction zurück, welches den Pfad repräsentiert. Es wird die Node des letzten Elements aus der *header table* geholt und mithilfe der Methoden `getParent()` und `getItem()` ein Pfad zusammengestellt.

Außerdem hat die Klasse „FPTree“ einen *Getter* für die *header table*.

3.8. Implementierung des FP-Growth Algorithmus'

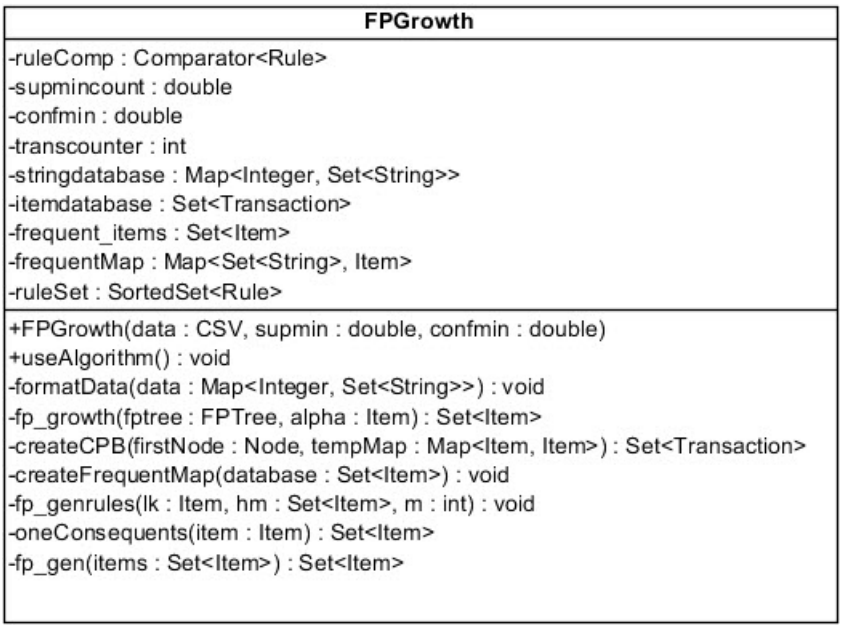


Abbildung 3.21.: Klassendiagramm „FPGrowth"

Um den in Kapitel 2.2.2 vorgestellten FP-Growth Algorithmus anzuwenden, dient die Klasse „FPGrowth". Bei der Erstellung eines Objektes dieser Klasse, wird ein CSV-Objekt sowie der Mindestwert für den *Support* und die *Konfidenz* (prozentual) übergeben. Das übergebene CSV-Objekt, dass die Datenbasis repräsentiert, wird für die interne Darstellung in eine Menge von Transaktionen umgeformt. Da in jedem FPItem das jeweilige Vorkommen gespeichert wird, kann mithilfe der Gesamtzahl aller Transaktionen und dem prozentualen Mindestwert für den *Support* das benötigte Mindestvorkommen für ein *frequent itemset* berechnet werden.
Mithilfe der Menge von Transaktionen und dem Mindestvorkommen wird ein FPTree erstellt und dieser zusammen mit einem neu erstellten, leeren FPItem, der Methode `fp_growth(FPTree fptree, Item alpha)` übergeben. Diese Methode ist die Basis des FP-Growth Algorithmus' und dient dazu, den erstellten FP-Baum zu deuten. Der Pseudocode der Methode ist in dem Listing 3.6 zu sehen.

```
1  procedure fp_growth(Tree, α)
2   if Tree contains a single path P then
3    for each combination (denoted as β) of the nodes in the path P
4     generate pattern β∪α with support = minimum support of nodes
          in β;
5   else
6    for each a_i in the header of Tree
7     generate pattern β = a_i∪α with support = a_i.support;
8     construct β's conditional pattern base and then β's
          conditional FP-Tree Tree_β;
9     if Tree_β ≠ ∅ then
10     call fp_growth(Tree_β, β);
```

Listing 3.6: Pseudocode zur Deutung eines FP-Baumes des FP-Growth Algorithmus' (Quelle: In Anlehnung an Han und Kamber, 2001, S. 242)

Die Abfrage, ob der Baum nur einen *single path* beinhaltet, ist in der Klasse „FPTree" implementiert (`containsSinglePath()`). Liefert diese Methode den Boolean *true* zurück, werden die Items aus dem *single path* betrachtet und jede Teilmenge mit α vereinigt.
Existiert kein *single path*, sondern ein *multipath*, wird von hinten durch die *header table* des Baumes iteriert. Ein neues FPItem β wird erstellt, indem das aktuell betrachtete Item a_i mit α vereinigt wird. Im nächsten Schritt wird die *conditional pattern base* von β erstellt. Dies geschieht durch die private Methode `createCPB()`. Ihr wird die Node von a_i aus der *header table* übergeben. Mithilfe dieser Node können alle Pfade/Transaktionen erstellt werden, deren letztes Element das Item a_i ist. Für die Pfad-/Transaktionserstellung wird die Methode `getParent()` genutzt und der Zähler der Transaktion wird auf den der Node gesetzt. Über die Methode `getPointer()` kann die Methode `createCPB()` rekursiv aufgerufen werden und so weitere Transaktionen, die das Element a_i beinhalten, gefunden werden.
Mit dieser *conditional pattern base* kann ein neuer Baum erstellt werden, der sogenannte Betabaum ($Tree_\beta$). Wenn die *header table* des $Tree_\beta$ nicht leer ist, wird die Methode `fp_growth()` rekursiv mit dem $Tree_\beta$ und β aufgerufen. Ist die *header table* des Betabaumes leer, ist der Algorithmus beendet.
Wie in den Grundlagen schon erwähnt, existiert für den FP-Growth Algorithmus keine eigene Methode für die Regelerstellung. Aus diesem Grund wird nach dem gleichen Prinzip wie schon bei dem Apriori-Algorithmus vorgegangen. Basierend auf dem Listing 3.5 werden die Methoden `fp_gen()` basierend auf `apriori_gen()`, `fp_genrules()` basierend auf `ap_genrules()` und `oneConsequents()` implementiert.

3.9. Formatierung der Ausgabe

Für die Formatierung der Ausgabe existiert in der Klasse „Rule“ die Methode `toString()`, welche dafür sorgt, dass die Regel in der Form:
„Regelrumpf → Regelkopf | Support: <Supportwert> Konfidenz: <Konfidenzwert>“
ausgegeben wird. Die Anordnung, in welcher Reihenfolge die Regeln ausgegeben werden, wird durch den Comparator `ruleComp`, der in den Klassen „Apriori“ und „FPGrowth“ implementiert ist, festgelegt. Dieser Comparator wird dem *ruleSet* übergeben und sortiert die Regeln primär nach dem Supportwert (höchster zuerst). Wenn mehrere Regeln den gleichen Supportwert haben, werden diese nach dem Konfidenzwert sortiert. Aus diesem Grund werden die „wichtigsten“ Regeln zuerst angezeigt.
Abbildung 3.22 zeigt die Regeln und deren Ausgabe, wenn Beispielsweise ein Algorithmus mit einem Mindestsupport von 1% und einer Mindestkonfidenz von 80% ausgeführt wird.

```
[/veranstaltungsplaene.html] -> [/studierende.html]   Support: 0.15471052845859457  Konfidenz:0.8730280287946087
[/kontakt.html] -> [/professoren.html]   Support: 0.09749477512688978  Konfidenz:0.8003565062388592
[/54.html] -> [/index.php?id=1669]   Support: 0.08042233260049399  Konfidenz:0.9996626180836707
[/msdnaa-an.html] -> [/msdnaa.html]   Support: 0.05235729989414543  Konfidenz:0.9107648725212465
[/professoren.html, /studierende.html] -> [/kontakt.html]   Support: 0.05081019460955948  Konfidenz:0.9378757515030061
[/kontakt.html, /studierende.html] -> [/professoren.html]   Support: 0.05081019460955948  Konfidenz:0.8013698630136986
[/422.html] -> [/msdnaa.html]   Support: 0.03523057297179925 4  Konfidenz:0.9378612716763006
[/heitmann.html] -> [/professoren.html]   Support: 0.026192221046060 31  Konfidenz:0.8269065981148244
[/wahlfaecher0.html] -> [/studierende.html]   Support: 0.025866514670358005  Konfidenz:0.8411297440423654
[/heitmann.html, /kontakt.html] -> [/professoren.html]   Support: 0.022962299487012457  Konfidenz:0.9871645274212368
[/heitmann.html, /professoren.html] -> [/kontakt.html]   Support: 0.022962299487012457  Konfidenz:0.8766839378238342
[/studiengaenge.html] -> [/interessierte.html]   Support: 0.020573786065195 56  Konfidenz:0.9143546441495778
[/msdnaa-faq.html] -> [/msdnaa.html]   Support: 0.018402410227180196  Konfidenz:0.8268292682926829
[/msdnaa-an.html, /422.html] -> [/msdnaa.html]   Support: 0.017913850663626 74  Konfidenz:0.9649122807017544
[/veranstaltungsplaene.html, /vorlesungszeiten.html] -> [/studierende.html]   Support: 0.016746736150693484  Konfidenz:0.8371777476255088
[/suchen.html] -> [/index.php?id=33]   Support: 0.015443910647884265  Konfidenz:0.9878472222222222
[/1781.html] -> [/studierende.html]   Support: 0.012783975246315447  Konfidenz:0.8234265734265734
[/msdnaa-an.html, /msdnaa-faq.html] -> [/msdnaa.html]   Support: 0.011888282713134109  Konfidenz:0.877755110220441
[/meisel.html, /kontakt.html] -> [/professoren.html]   Support: 0.011779713921233342  Konfidenz:0.96875
[/meisel.html, /professoren.html] -> [/kontakt.html]   Support: 0.011779713921233342  Konfidenz:0.8628230616302187
[/kontakt.html, /veranstaltungsplaene.html] -> [/studierende.html]   Support: 0.011535434139456614  Konfidenz:0.8568548387096774
[/heitmann.html, /studierende.html] -> [/professoren.html]   Support: 0.011236869961729502  Konfidenz:0.8941684665226782
[/heitmann.html, /studierende.html] -> [/kontakt.html]   Support: 0.010938305784002388  Konfidenz:0.8704103671706264
[/studiengaenge.html, /studierende.html] -> [/interessierte.html]   Support: 0.010884021388052004  Konfidenz:0.9685990338164251
[/heitmann.html, /kontakt.html, /studierende.html] -> [/professoren.html]   Support: 0.010856879190076812  Konfidenz:0.9925558312655087
[/heitmann.html, /studierende.html, /professoren.html] -> [/kontakt.html]   Support: 0.010856879190076812  Konfidenz:0.9661835748879227
[/heitmann.html, /studierende.html] -> [/kontakt.html, /professoren.html]   Support: 0.010856879190076812  Konfidenz:0.8639308855291575
[/ti-labor.html] -> [/labore.html]   Support: 0.010802594794126428  Konfidenz:0.8105906313645621
[/stisys.html, /veranstaltungsplaene.html] -> [/studierende.html]   Support: 0.010748310398176045  Konfidenz:0.8703296703296703
[/veranstaltungsplaene.html, /professoren.html] -> [/kontakt.html]   Support: 0.010531172814374509  Konfidenz:0.8919540229885058
[/cms/index.php?id=118] -> [/index.php?id=118]   Support: 0.010341177428548164  Konfidenz:0.9338235294117647
[/index.php?id=118] -> [/cms/index.php?id=118]   Support: 0.010341177428548164  Konfidenz:0.8300653594771242
[/95.html] -> [/interessierte.html]   Support: 0.010178324240697012  Konfidenz:0.8278145695364238
```

Abbildung 3.22.: Ausgabe der Regelerstellung

4. Effizienzvergleich der Algorithmen

In diesem Kapitel werden die beiden Algorithmen hinsichtlich ihrer Effizienz miteinander verglichen. Es werden die Grundlagen, auf denen der Vergleich basiert, dargestellt. Mit Grundlagen sind sowohl die Hard- und Software, als auch die verarbeiteten Daten und der eingestellte Supportwert gemeint. In diesem Vergleich wird allerdings nur die Generierung der *frequent itemsets* beachtet, da die Regelgenerierung in beiden Algorithmen nahezu identisch abläuft. Die Ergebnisse der Algorithmen müssen nicht miteinander verglichen werden, da diese identisch sind und die beiden Algorithmen immer die gleichen *frequent itemsets* generieren.

4.1. Hard- und Software

Der Vergleich der Algorithmen wird natürlich mit identischer Hard- und Software durchgeführt. Das Testsystem verfügt über folgende Ressourcen:

- **CPU:** Intel Core 2 Duo T7500 mit 2,20 GHz
- **RAM:** 3 GB
- **OS:** Windows 7 Professional 32Bit SP1
- **Java:** JavaSE-1.6

4.2. Datenbasis

Als Datenbasis dient ein Logfile des Departments Informatik der Hochschule für angewandte Wissenschaften Hamburg. Dieses Logfile beinhaltet alle Seitenaufrufe vom 02.11.2011 12:14 Uhr bis zum 12.03.2012 10:31 Uhr und umfasst 2.631.315 Zeilen. Nachdem das Logfile zusammengefasst und als CSV-Datei gespeichert wurde, beinhaltet dieses noch 36.843 Transaktionen. Diese Transaktionen dienen als Datenbasis für den Vergleich der beiden Algorithmen.
Als Parameter dienen folgende Werte für den Mindestsupport.

- 10%
  ⋮

- 1%
- 0,9%
  ⋮
- 0,3%

Es folgen die jeweiligen Supportwerte, gefolgt von dem Mindestvorkommen, welches ein *itemset* besitzen muss, um in die Menge der *frequent itemsets* aufgenommen zu werden, und die Anzahl der *frequent itemsets* für genau diesen Anwendungsfall mit dem entsprechendem Supportwert:

- Mindestwert *Support*: 10%; Mindestvorkommen: 3685; Anzahl *frequent itemsets*: 6
- Mindestwert *Support*: 9%; Mindestvorkommen: 3316; Anzahl *frequent itemsets*: 8
- Mindestwert *Support*: 8%; Mindestvorkommen: 2948; Anzahl *frequent itemsets*: 11
- Mindestwert *Support*: 7%; Mindestvorkommen: 2580; Anzahl *frequent itemsets*: 11
- Mindestwert *Support*: 6%; Mindestvorkommen: 2211; Anzahl *frequent itemsets*: 14
- Mindestwert *Support*: 5%; Mindestvorkommen: 1843; Anzahl *frequent itemsets*: 20
- Mindestwert *Support*: 4%; Mindestvorkommen: 1474; Anzahl *frequent itemsets*: 23
- Mindestwert *Support*: 3%; Mindestvorkommen: 1106; Anzahl *frequent itemsets*: 34
- Mindestwert *Support*: 2%; Mindestvorkommen: 737; Anzahl *frequent itemsets*: 52
- Mindestwert *Support*: 1%; Mindestvorkommen: 369; Anzahl *frequent itemsets*: 145
- Mindestwert *Support*: 0.9%; Mindestvorkommen: 332; Anzahl *frequent itemsets*: 169
- Mindestwert *Support*: 0.8%; Mindestvorkommen: 295; Anzahl *frequent itemsets*: 199
- Mindestwert *Support*: 0.7%; Mindestvorkommen: 258; Anzahl *frequent itemsets*: 242
- Mindestwert *Support*: 0.6%; Mindestvorkommen: 222; Anzahl *frequent itemsets*: 278
- Mindestwert *Support*: 0.5%; Mindestvorkommen: 185; Anzahl *frequent itemsets*: 391
- Mindestwert *Support*: 0.4%; Mindestvorkommen: 148; Anzahl *frequent itemsets*: 703
- Mindestwert *Support*: 0.3%; Mindestvorkommen: 111; Anzahl *frequent itemsets*: 16864

4.3. Zeitmessung

Folgend wird dargestellt, wie sich die Dauer der Erstellung der *frequent itemsets* auf den gewählten Wert für den Mindestsupport auswirkt. Der Wert der Mindestkonfidenz wird für diesen Vergleich nicht mit herangezogen, da dieser ausschließlich für die Regelgenerierung benötigt wird.

Für jeden Supportwert wird der Algorithmus mehrmals durchgeführt. Die Ergebnisse der Zeitmessung werden addiert und daraus ein Mittelwert berechnet. Dies geschieht, da die gemessenen Werte leicht schwanken, vermutlich durch das unkontrollierbare Einsetzen des „Garbage Collectors“ oder der Runter- bzw. Rauftaktung der CPU (siehe „Varianz der Messwerte“, (Kreft und Langer, 2005)). Die Benutzung eines Mittelwertes und die leichten Schwankungen der Ergebnisse sind wenig problematisch, da im Vergleich die Werte sehr viel größer als die Schwankungen sind und gerade die mit einem geringen *Support* (siehe Kapitel 2.2.1) interessant sind.
Die Zeitmessung beginnt mit dem Aufruf von „useAlgorithm()“ und endet, wenn alle *frequent itemsets* erstellt wurden. Gemessen wird, indem mithilfe der Methode `System.nanoTime()` die Start- und Endzeit gespeichert und deren Differenz berechnet wird. Es bleibt die Ausführungszeit in Nanosekunden. Dieser Wert wird zur besseren Lesbarkeit auf Millisekunden gerundet und ausgegeben.

4.4. Auswertung der Ergebnisse

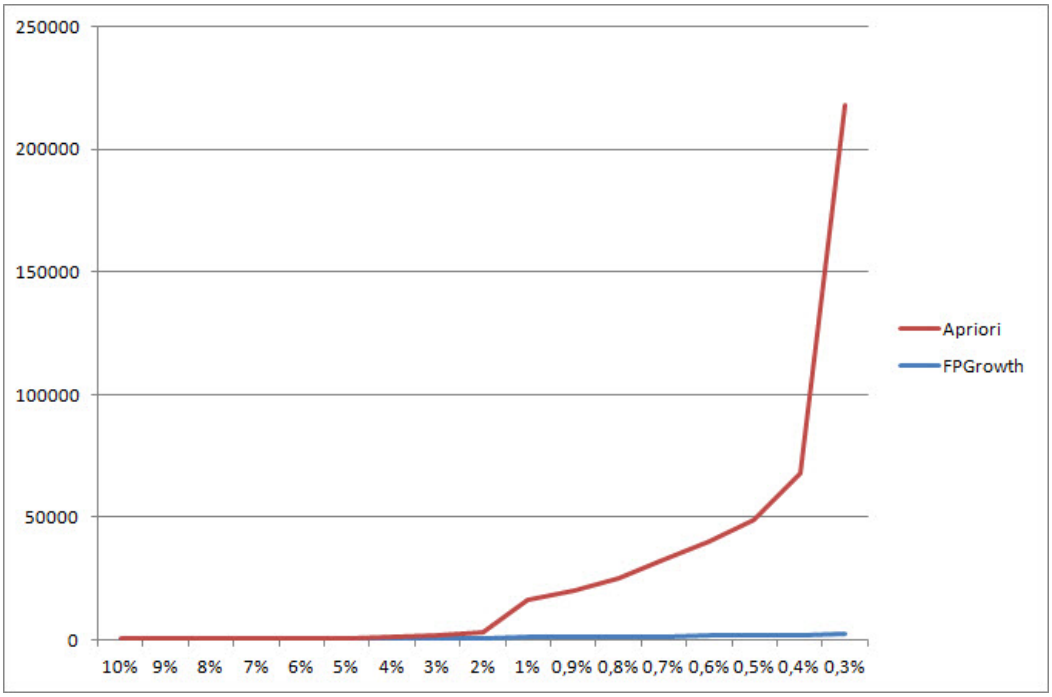


Abbildung 4.1.: Ergebnisdarstellung in einem Liniendiagramm

Um die Ergebnisse zu visualisieren, dienen zwei Diagramme. Das erste Diagramm (siehe Abbildung 4.1) zeigt die Daten grob skaliert. Die Maßzahlen der x-Achse sind die prozentualen Supportwerte, die y-Achse zeigt den durchschnittlichen Zeitverbrauch in Millisekunden. Dieses Diagramm dient weniger der exakten Angabe der Ergebnisse, sondern soll zeigen, wie enorm sich ab einem bestimmten Wert die Effizienz der beiden Algorithmen unterscheidet.

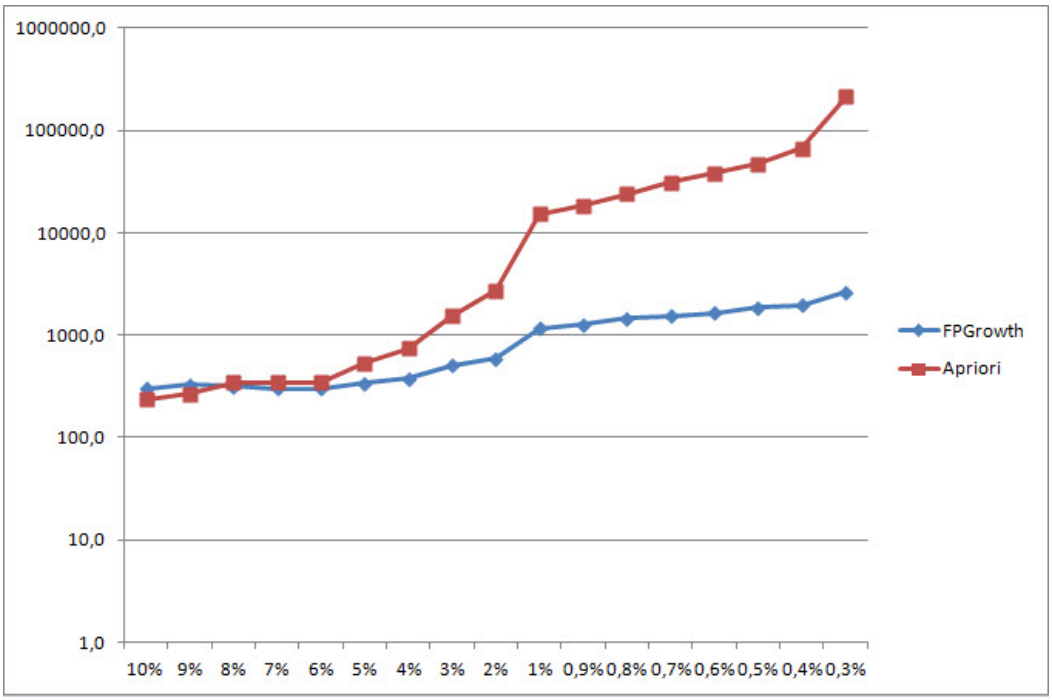


Abbildung 4.2.: Ergebnisdarstellung in einem logarithmisch zur Basis 10 skaliertem Liniendiagramm

Um die Ergebnisse genauer zu betrachten, dient das zweite Diagramm (Abbildung 4.2). Bei diesem Diagramm wurde die y-Achse logarithmisch zur Basis 10 skaliert. Durch die Skalierung werden die Extremen der Differenz visuell verringert, dafür können die Wertunterschiede in den hochprozentualen Bereichen besser verglichen und abgelesen werden.
In Abbildung 4.2 ist zu sehen, dass der Apriori-Algorithmus bei einer geringen Anzahl von *frequent itemsets* (also einem großem Mindestwert für *Support*) schneller als der FP-Growth Algorithmus ist. Dies verändert sich mit der Menge der resultierenden *frequent itemsets*. Ab einem Mindestsupport von 8% verschlechtert sich die Effizienz von Apriori so sehr, dass der FP-Growth Algorithmus ab diesem Zeitpunkt schneller ist.
Der Abbildung 4.1 ist zu entnehmen, dass zwischen 2% und 1% bei dem Apriori-Algorithmus ein enormer Geschwindigkeitseinbruch stattfindet. Werden die dazugehörigen Anzahlen der

erstellten *frequent itemsets* betrachtet, ist zu sehen, dass sich die Zahl von 53 (2%) auf 145 (1%) fast verdreifacht, sich die Berechnungszeit des Algorithmus' von ca. 2,7s auf ca. 15,2s mehr als verfünffacht hat. Bei dem FP-Growth Algorithmus hingegen hat sich die Bearbeitungszeit von ca. 0,6s auf ca. 1,2s nur verdoppelt.
Ein weiterer extremer Anstieg ist zwischen 0,4% und 0,3% zu sehen. Hier hat sich die Anzahl der erstellten *frequent itemsets* von 703 auf 16864 erhöht, fast vervierundzwanzigfacht. Für Apriori hat sich dadurch die Berechnungszeit von ca. 66,2s auf ca. 215s mehr als verdreifacht. Für den FP-Growth Algorithmus bedeutet diese Änderung eine Erhöhung der Berechnungszeit um ca. 0,6s.
Allgemein ist zu sagen, dass FP-Growth im Gegensatz zu Apriori keine extreme Steigung verzeichnet. Um die Geschwindigkeitsunterschiede noch einmal genau zu zeigen, wird der jeweilige Durchschnittswert des Apriori-Algorithmus' durch den des FP-Growth Algorithmus' geteilt. Das Ergebnis zeigt, um wieviel schneller der FP-Growth Algorithmus arbeitet. Die Werte können der Abbildung 4.3 entnommen werden. Werte kleiner 1 bedeuten, dass der Apriori-Algorithmus schneller ist.

Support	Quotient
10%	0,79
9%	0,81
8%	1,10
7%	1,16
6%	1,15
5%	1,58
4%	1,94
3%	3,02
2%	4,57
1%	13,03
0,9%	14,69
0,8%	16,16
0,7%	20,16
0,6%	23,41
0,5%	25,19
0,4%	33,43
0,3%	81,37

Abbildung 4.3.: Darstellung, wieviel fach schneller FP-Growth arbeitet

5. Fazit und Ausblick

Im Folgenden wird ein Fazit gezogen und im Zuge dessen werden kurz die vorgenommenen Arbeitsschritte sowie die gewonnenen Ergebnisse dargestellt. Desweiteren folgt ein Ausblick auf mögliche, weiterführende Erweiterungen.

5.1. Fazit

Die Assoziationsanalyse findet in verschiedenen Bereichen ihre Anwendung. In der vorliegenden Arbeit wurde sie auf Logfiles des Departments Informatik angewendet, mit dem Ziel, Assoziationen zwischen Seitenaufrufen herauszufinden. Für eine solche Analyse können verschiedene Algorithmen verwendet werden. Deswegen wurde in dieser Arbeit ein Vergleich in Hinblick auf die Effizienz der beiden Algorithmen Apriori und FP-Growth durchgeführt.
Begonnen wurde mit den Grundlagen der Assoziationsanalyse, in der die Begriffe *Support* und *Konfidenz* eingeführt und der allgemeine Ablauf der Analyse vorgestellt wurde. Desweiteren wurden die Arbeitsweisen und die Regelerzeugungen der beiden Algorithmen erläutert. Dabei muss darauf hingewiesen werden, dass der FP-Growth keine eigene Regelerzeugung besitzt und für ihn deshalb in der vorliegenden Arbeit die des Apriori-Algorithmus' übernommen wurde.
Darauffolgend wurden das fachliche Datenmodell sowie der grobe Ablauf der Ausführung dargestellt. Im weiteren Verlauf wurde das Programm implementiert und die einzelnen Schritte der Implementierung beschrieben. Dazu gehören die Anonymisierung der Logfiles, die Implementierung des Datenimports, die Darstellung der Items und Regeln sowie die Implementierungen der Algorithmen und deren Formatierung für die Regelausgabe.
Im Anschluss daran wurden die beiden Algorithmen hinsichtlich ihrer Effizienz miteinander verglichen. Dabei wurde sowohl die genutzte Hard- und Software, als auch die Datenbasis vorgestellt. Daraufhin wurden die Geschwindigkeiten der Erstellung der *frequent itemsets* verglichen und ausgewertet. Aufgrund der nahezu identischen Verfahren zur Regelerzeugung wurden diese in den Vergleich nicht einbezogen.
Insgesamt kann gesagt werden, dass bei dem Apriori-Algorithmus mit steigender Anzahl der *frequent itemsets* sich diese Erstellung stark verlangsamt. Der FP-Growth Algorithmus

hingegen verarbeitet auch große Datenmengen ohne starke Performanzeinbußen. Bei hohen Supportwerten und einer dementsprechend geringen Anzahl von *frequent itemsets* ist Apriori effizienter. Verringert sich der Supportwert, bestätigt sich die in Kapitel 2.2.1 angesprochene Problematik der Kandidatengenerierung, weswegen der FP-Growth Algorithmus schon ab einem Supportwert von 8% die *frequent itemsets* effizienter erstellt. Somit eignet sich FP-Growth für die Auswertung der Logfiles besser.

5.2. Ausblick

Die Ergebnisse der vorliegenden Arbeit können dazu verwendet werden, weitere Algorithmen zu implementieren und in den Vergleich einzubeziehen, um einen etwaigen effizienteren Algorithmus herauszufinden. In diesem Zuge könnte außerdem auf den Wert *Lift* eingegangen werden, welcher in dieser Arbeit nicht betrachtet worden ist, weil in der gängigen Literatur das Hauptaugenmerk auf die beiden verwendeten Werte *Support* und *Konfidenz* gelegt wird. Diesem Schema ist gefolgt worden.
Mithilfe der vorgestellten Algorithmen könnten auch andere Daten analysiert werden, solange sie der in Kapitel 3.3.3 angesprochenen Formatierung entsprechen. Somit können die erstellten Algorithmen zum Beispiel auch zur Warenkorbanalyse verwendet werden. Es stehen also noch weitere Anwendungsfelder offen.

A. CD

A.1. Quellcode

Die CD beinhaltet den gesamten entwickelten Quellcode. Dieser besteht sowohl aus dem Code für die Umformung der Logfiles zu einer CSV-Datei, als auch aus den, für die Anwendung der beiden Algorithmen, benötigten Klassen.

A.2. Datenbasis

Das verwendete Logfile und die daraus resultierende CSV-Datei befindet sich im Verzeichnis „Source“.

A.3. Vergleich

Die Daten des Vergleiches wurden in einer Excel-Tabelle zusammengetragen und daraus ein Diagramm erstellt, dieses befindet sich ebenfalls auf der CD.

A.4. Paper

Die Paper für die Algorithmen Apriori und FP-Growth befinden sich im Verzeichnis „Paper“.

A.5. Ordnerstruktur

Die Ordnerstruktur der CD wird folgend dargestellt und kurz beschrieben:

```
DVD/
 |_ BA_Slotta.pdf          Bachelorarbeit
 |_ /Abbildungen           Erstellte Grafiken im Format PDF,
                           JPG oder graphml
 |_ /Auswertung            Daten und Ergebnisse des Vergleiches
 |_ /Diagramme             Erstellte Diagramme
 |_ /Paper                 Paper zu den Algorithmen
 |_ /Source                Daten der entwickelten Applikationen
     |_ /Logfile           Logfile, das als Datenbasis diente
     |_ /CSV               Erstellte CSV-Datei
     |_ /src               Quellcode der Applikation
        |_ /apriori        Klassen für den Apriori-Algorithmus
        |_ /fp_growth      Klassen für den FP-Growth Algorithmus
        |_ /global         Klassen, die von beiden Algorithmen
                           genutzt werden
        |_ /logfileToCSV   Umwandlung vom Logfile zur CSV-Datei
        |_ /starter        Starter-Klasse
```

Literatur

Agrawal, Rakesh und Ramakrishnan Srikant (1994a). "Fast Algorithms for Mining Association Rules". In: *Proceedings 1994 Vldb Conference: Snatiago-Chile: VLDB '94*, S. 487–499. URL: http://rakesh.agrawal-family.com/papers/vldb94apriori.pdf.

Agrawal, Rakesh und Ramakrishnan Srikant (1994b). "IBM Research Report RJ9839". In: URL: http://www.almaden.ibm.com/cs/projects/iis/hdb/Publications/papers/vldb94_rj.pdf.

Bankhofer, Udo und Jürgen Vogel (2008). *Datenanalyse und Statistik: Eine Einführung für Ökonomen im Bachelor*. Gabler Verlag.

Bollinger, Toni (1996). "Assoziationsregeln - Analyse eines Data Mining Verfahrens". In: *Informatik Spektrum* 19, S. 257–261.

Ester, Martin und Jörg Sander (2000). *Knowledge Discovery in Databases - Techniken und Anwendungen*. Springer Verlag.

Farkisch, Kiumars (2011). *Data-Warehouse-Systeme Kompakt: Aufbau, Architektur, Grundfunktionen*. Springer Verlag.

Gabriel, Roland, Peter Gluchowski und Alexander Pastwa (2009). *Data Warehouse & Data Mining*. W3L GmbH.

Görz, Günther, Claus-Rainer Rollinger und Josef Schneeberger (2003). *Handbuch der künstlichen Intelligenz*. Oldenbourg.

Han, Jiawei und Micheline Kamber (2001). *Data Mining - Concepts and Techniques*. Academic Press.

Han, Jiawei u. a. (2004). "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach". In: *Data Mining and Knowledge Discovery* 8, S. 53–87.

Heindl, Eduard (2003). *Logfiles richtig nutzen*. Galileo Press GmbH.

Hettich, Stefanie, Hajo Hippner und Klaus D. Wilde (2000). "Assoziationsanalyse". In: *Das Wirtschaftsstudium (wisu)* 7, S. 970–978.

Kreft, Klaus und Angelika Langer (2005). *Java Performance: Micro-Benchmarking - Wie wirkt sich die HotSpot-Technologie aufs Micro-Benchmarking aus?* URL: http://www.angelikalanger.

com/Articles/EffectiveJava/22.JITCompilation/22.JITCompilation.html (besucht am 21. 04. 2012).

Oracle. *Java API 1.4.2 - Collection*. URL: http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Collection.html (besucht am 22. 03. 2012).

Petersohn, H. (2005). *Data Mining - Verfahren, Prozesse, Anwendungsarchitektur*. Oldenbourg Wissenschaftsverlag GmbH.

Shafranovich, Y. (2005). *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments: 4180. URL: http://tools.ietf.org/html/rfc4180 (besucht am 20. 02. 2012).

Säuberlich, Frank (2000). *KDD und Data Mining als Hilfsmittel zur Entscheidungsunterstützung*. Peter Lang Verlag.

Tan, Pang-Ning, Michael Steinbach und Vipin Kumar (2006). *Introduction to Data Mining*. Pearson Education Inc.

Witten, Ian H. und Eibe Frank (2001). *Data Mining - Praktische Werkzeuge und Techniken für das maschinelle Lernen*. Carl Hanser Verlag.

Abbildungsverzeichnis

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 23. April 2012 Jörn Slotta