Diplomarbeit

Verwaltung großer Datenmengen für die effiziente Anwendung des Apriori-Algorithmus zur Wissensentdeckung in Datenbanken

Frank Wiechers

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

Datum
November 1997

Betreuer:

Prof. Dr. Katharina Morik
Dipl. Inf. Peter Brockhausen

Zusammenfassung

Die vorliegende Arbeit befaßt sich mit der Reimplementierung des Apriori-Algorithmus zur Entdeckung von Assoziationsregeln vor dem Hintergrund der effizienten Verwaltung sowohl (sehr) großer Daten- als auch Hypothesenmengen. Deshalb wird nach einer kurzen Einleitung zunächst in Kapitel 2 die Problematik der Entdeckung von Assoziationsregeln näher beleuchtet, ehe in Kapitel 3 auf Datenstrukturen eingegangen wird, die eine effiziente Verarbeitung großer Datenmengen ermöglichen. Der Kern dieser Arbeit liegt in Kapitel 5, in dem die Reimplementierung des Apriori-Algorithmus beschrieben wird. Anschließend folgen Experimente in Kapitel 5.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Überblick

Der weitverbreitete Gebrauch von Strichkodes für die meisten kommerziellen Produkte, die zunehmende Computerisierung im Handelsgeschehen (z.B. durch den Einsatz von Kreditkarten) und die Fortschritte in der Datenhaltung (z.B. durch CD-ROM) haben uns in den letzten Jahren mit einer Unmenge an Daten konfrontiert, die wir in unserer heutigen Situation kaum vollständig verarbeiten und interpretieren können.

In Wirtschaft, Wissenschaft, öffentlichen Verwaltungen und anderen Anwendungsgebieten hat das explosionsartige Wachstum der Datenmengen und der sie verwaltenden Datenbanken bei den Anwendern geradezu das Bedürfnis nach einer neuen Generation von Techniken und Tools erzeugt, die den Anwender geschickt und automatisch bei der Datenanalyse unterstützen sollen. Gerade diese Techniken und Tools sind Gegenstand des relativ neuen Anwendungsgebietes der *Wissensentdeckung in Datenbanken (KDD[1])* [Fayyad et al., 1996].

Allgemein gesprochen kann die Wissensentdeckung in Datenbanken als nichttriviale Extraktion impliziter, vorher unbekannter und potentiell nützlicher Informationen angesehen werden [Frawley et al., 1992]. Die Wissensentdeckung in Datenbanken beschreibt einen mehrstufigen Prozeß der komplexen und langwierigen Interaktion eines Menschen mit einer (sehr großen) Datenmenge, der zu der Entdeckung von Wissen führt, indem die in den Daten durch Anwendung von *Data Mining Methoden* entdeckten Muster (Modelle) interpretiert werden [Brachman und Anand, 1996]. Das Ergebnis dieses Prozesses kann eine Spezifikation für eine bestimmte KDD-Anwendung sein, die beispielsweise eingesetzt werden kann beim Marketing zur Klassifikation von Kunden und zur Vorhersage von Kundenverhalten, beim Informationsmanagement und bei der Entscheidungsfindung (*Decision Support Problem*) vieler Unternehmen, bei der Prozeßsteuerung und Fehlersuche in der Fertigung und bei WWW-basierten Online-Informationsanbietern zur effizienteren Bewertung des Benutzerverhaltens zum Zweck der Verbesserung der zur Verfügung gestellten Dienstleistungen.

[1] engl. **K**nowledge **D**iscovery in **D**atabases

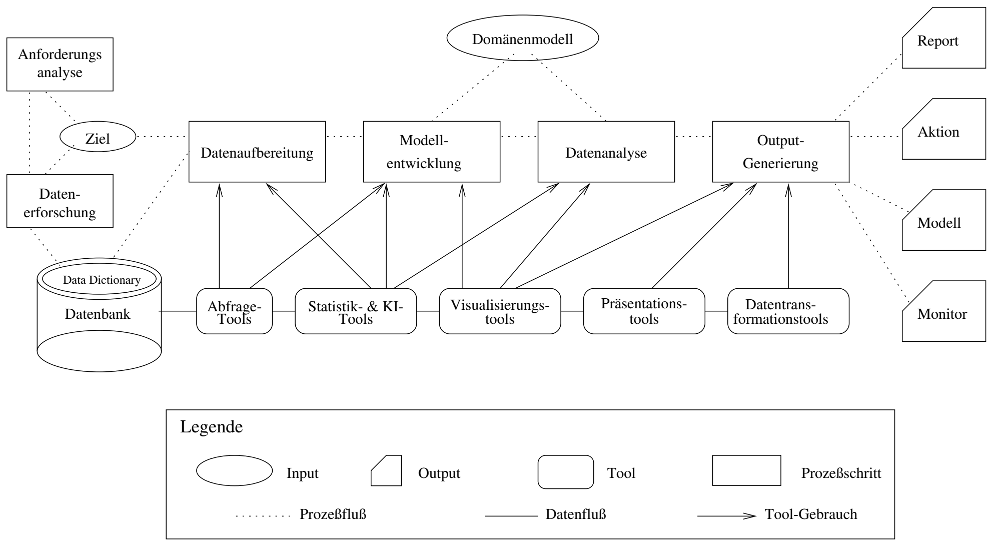


Abbildung 1.1: Der KDD-Prozeß.

Nach [Brachman und Anand, 1996] läßt sich der Prozeß der Wissensentdeckung in Datenbanken (KDD-Prozeß) in die folgenden Prozeßschritte einteilen (siehe Abb. 1.1):

- Anforderungsanalyse (*Task Discovery*): Die Anforderungen an den KDD-Prozeß und der daraus resultierenden KDD-Anwendung müssen nach eingehender Beschäftigung mit dem Anwender und dessen Organisation festgehalten werden. Dieser Schritt ist zwar sehr zeitaufwendig und schwierig, weil die Anforderungen und Ziele des Anwenders am Anfang oftmals relativ unklar sind, aber absolut notwendig.

- Datenerforschung (*Data Discovery*): Der Analytiker muß sich intensiv mit den zu analysierenden Daten, die der Anwender zu diesem Zweck zur Verfügung stellt, auseinandersetzen, um die Struktur, den Umfang und die Qualität der Daten kennenzulernen und zu bewerten, damit festgestellt werden kann, ob die Ziele des Anwenders mit den zur Verfügung gestellten Daten überhaupt erfüllt werden können.

- Datenaufbereitung (*Data Cleaning*): Da die zu analysierenden Daten in der Regel nicht vor dem Hintergrund, später einmal eine Wissensentdeckung darauf durchzuführen, gesammelt wurden und teilweise fehlerhaft oder unvollständig sein können, ist zunächst eine Aufbereitung der Daten erforderlich, um sie in die für die Analyse geeignete Repräsentation zu überführen.

- Modellentwicklung (*Model Development*): Aufgrund der Tatsache, daß in vielen KDD-Anwendungen (z.B. Analyse von Marketingdaten) zum einen die Informationen in der zu analysierenden Datenmenge zu mannigfaltig sind und zum anderen die Kardinalität der Datenmenge so groß ist, daß sie nicht mehr effizient verwaltet werden kann, versucht man zum Zwecke der Komplexitätsreduktion eine Teilmenge aus den Daten zu selektieren, auf die die Analyse fokussiert wird, und gegebenenfalls Parameter für die Analyse einzuschränken, da nicht alle Variablen in einer Analyse von Nutzen sind (*Sampling*). Der Analytiker setzt sich mit den gegebenen Daten auseinander und fokussiert aufgrund seiner Erfahrung und seines Hintergrundwissens den zu analysierenden Teilausschnitt der Daten. Dabei können Tools zur Visualiserung sehr hilfreich sein. Diese Interaktion mit den Daten führt zur Formulierung von Hypothesen und zur entsprechenden Wahl einer geeigneten Hypothesensprache.

Zu den wichtigsten Schritten innerhalb der Modellentwicklung gehören die folgenden Punkte:

  - Datensegmentierung (*Data Segmentation*)
  - Modellauswahl (*Model Selection*): Es gibt eine Vielzahl von Modellen zur Analyse von (sehr) großen Datenmengen.
  - Parameterauswahl (*Parameter Selection*): Innerhalb des gewählten Modells müssen die zu betrachtenden Parameter ausgewählt werden.

- Datenanalyse (*Data Analysis*): Der Analytiker hat eine Hypothese bezüglich der zu analysierenden Daten aufgestellt. Daraufhin wird ein Datenanalysetool benutzt, um ein Modell aus den Daten aufzubauen, damit anhand dessen die Hypothese entweder bestätigt oder verworfen werden kann. Die folgenden Teilschritte gehören zu den wichtigsten innerhalb der Datenanalyse:

  - Modellspezifikation (*Model Specifikation*): Ein spezifisches Modell wird hier angegeben.
  - Modellanpassung (*Model Fitting*): Spezifische Parameter eines Modells werden nötigenfalls basierend auf den Daten bestimmt. Während in einigen Fällen das Modell unabhängig von den Daten konstruiert wird, wird es in anderen Fällen an Trainingsdaten angepaßt.
  - Evaluierung (*Evaluation*): Das Modell wird anhand der Daten evaluiert.
  - Modellverfeinerung (*Model Refinement*): Das in der Modellentwicklung angegebene initiale Modell wird in Abhängigkeit des Ergebnisses der Evaluierung iterativ verfeinert.

Diese Teilschritte werden durch *Data Mining* Techniken, mit deren Hilfe Modelle an gegebene Daten angepaßt bzw. Muster aus den vorliegenden

Daten bestimmt werden, unterstützt. Die verschiedenen *Data Mining* Verfahren können anhand der Lernaufgabe, die sie bewältigen, charakterisiert werden:

- *Klassifikation*: Eine wichtige Anwendung von *Data Mining* ist die Durchführung von Klassifikationen in einer großen Menge von Daten, wobei ein Datensatz basierend auf den Werten bestimmter Attribute klassifiziert wird. Entscheidungsbaumbasierte Klassifikationsverfahren, wie C4.5 [Quinlan, 1993] aus dem Bereich des Maschinellen Lernens und CART [Breiman et al., 1984] aus dem Bereich der Statistik, benutzen beispielsweise eine Attribut-Werte-Repräsentation, während das aus dem Bereich des Maschinellen Lernens stammende System FOIL [Quinlan, 1990] einen relationalen Repräsentationsformalismus verwendet (Hypothesen werden in funktionsfreie Hornklauseln angeben). In C4.5 besteht die Möglichkeit Entscheidungsbäume in Klassifikationsregeln zu transformieren.
- *Regellernen*: Regeln können in einem Attribut-Werte-Repräsentationsformalismus oder in der Prädikatenlogik 1. Stufe repräsentiert werden. Für Assoziationsregeln, die beispielsweise Zusammenhänge zwischen den Artikeln in den Verkaufstransaktionen eines Supermarktes wiederspiegeln, wird ein Attribut-Werte-Repräsentationsformalismus verwendet [Agrawal et al., 1993]. Im Gegesatz dazu benutzen die Techniken aus der induktiven logischen Programmierung (ILP) die Prädikatenlogik 1. Stufe, um Regeln in einer aussagekräftigen und flexiblen Form zu repräsentieren. So können mit den beiden ILP-Verfahren, RDT und RDT/DB, das ILP-Regellernverfahren mit direkter Datenbankankopplung, relationale und damit komplexere Regeln entdeckt werden [Brockhausen und Morik, 1996], [Morik und Brockhausen, 1997].
- *Clusteranalyse*: Die Clusteranalyse beschreibt einen Prozeß zur Gruppierung physikalischer oder abstrakter Objekte in Klassen von ähnlichen Objekten, wobei die spärlich und dicht besetzten Stellen innerhalb der Datenmenge identifiziert werden. So werden die gesamten Verteilungen in der Datenmenge entdeckt. Innerhalb der Statistik konzentriert sich die Clusteranalyse hauptsächlich auf die distanzbasierte Clusteranalyse. Systeme, die auf der statistischen Clustering-Methode basieren, wie z.B. AutoClass [Cheeseman und Stutz, 1996], benutzen eine Bayes'sche Klassifikationsmethode.

  Im Gegensatz zur Klassifikation wird eine Menge von Daten mit Hilfe der Clusteranalyse nicht nach einer vorausbestimmten Menge von Klassen, sondern nach dem Prinzip „Maximiere die Intraclass-Ähnlichkeit und minimiere die Interclass-Ähnlichkeit“ gruppiert.

  Innerhalb des Maschinellen Lernens bezieht sich die Clusteranalyse oft auf das *unüberwachte Lernen*, da die Klassen, zu denen ein Objekt gehört, nicht vordefiniert sind, oder auf das *Conceptual Clustering*.

In [Ng und Han, 1994] wird der Clustering-Algorithmus CLARANS für sehr große Datenmengen vorgestellt, der auf einer randomisierten Suche basiert und aus den beiden in der Statistik gebräuchlichen Clustering-Algorithmen, PAM (Partitioning Around Medoids) und CLARA (Clustering Large Applications) entstanden ist.

– *Regression*: Bei der im Bereich der Statistik anzusiedelnden Regression geht es um das Lernen einer Funktion, die ein Datenelement auf einen reellen Wert abbildet, der in einer Variablen repräsentiert wird, um in Abhängigkeit des Variablenwertes eine Vorhersage zu treffen. Die dabei verwendete Abbildung wird dadurch erzeugt, indem lineare und nicht-lineare Kombinationen von grundlegenden Funktionen (Sigmoids, Splines, Polynome) an Kombinationen der Werte von Eingabevariablen angepaßt werden [Elder und Pregibon, 1996].

Weitere wichtige *Data Mining Techniken* sind z.B. die Entdeckung kausaler Zusammenhänge in Daten mit Hilfe von probabilistischen graphischen Modellen auf der Grundlage von Bayes'schen Netzwerken [Cooper und Herskovits, 1992], sowie Techniken zur Visualisierung der entdeckten Muster [Keim und Kriegel, 1996].

- Output-Generierung (*Output Generation*): Geeignete Ausgaben einer Analyse können eine textuelle Beschreibung eines Trends oder eine sorgfältig ausgearbeitete Graphik zur Verdeutlichung der im Modell geltenden Beziehungen sein. Weiterhin können Beschreibungen ausgegeben werden, die dem Benutzer in Abhängigkeit des in den Daten entdeckten Wissens durchzuführende Aktionen vorschreiben. Schließlich ist ein Monitor, der wieder in die Datenbank eingefügt wird und dazu dient, beim Eintreten einer bestimmten Bedingung, einen Alarm oder eine Aktion auszulösen, eine nützliche Art von Ausgabe.

Der Einsatz verschiedener *Data-Mining* Techniken für verschiedene Arten von Datenbanken, wie z.B. relationale, objektorientierte, deduktive, multimediale, Internet-basierte oder heterogene Datenbanken ermöglicht die Entdeckung verschiedener Wissensarten [Chen et al., 1996]. So können beispielsweise aus der relationalen Datenbank eines Einzelhandelsunternehmens Verkaufstransaktionen der Kunden analysiert werden, um Assoziationen unter den Artikeln zu erhalten. Da es sich im allgemeinen um eine Vielzahl von Artikeln handelt, gibt es exponentiell viele zu testende Hypothesen, wobei eine Hypothese für eine Menge von zusammen in den Transaktionen vorkommenden Artikeln steht. Mit Hilfe des Apriori-Algorithmus können Assoziationen gefunden und daraus Assoziationsregeln gelernt werden [Agrawal et al., 1996].

In dieser Arbeit wird eine Reimplementierung des Apriori-Algorithmus angegeben, mit der sowohl große Ausgangsdatenmengen als auch große Mengen von Hypothesen, die aufgrund ihrer Kardinalität nicht im Hauptspeicher untergebracht werden können, effizient verwaltet werden. Nachdem zunächst in Kapitel 2 die theoretischen Hintergründe des Apriori-Algorithmus angesprochen werden, werden in Kapitel 3 Datenstrukturen zur Verwaltung großer Datenmengen betrachtet. Danach wird in Kapitel 4 näher auf die Implementierung eingegangen. In Kapitel 5 werden Experimente angegeben. Abschließend wird eine Schlußbetrachtung aufgestellt.

Kapitel 2

Entdeckung von Assoziationsregeln

Das Problem der Entdeckung von Assoziationsregeln wird motiviert durch das Auffinden wichtiger Zusammenhänge zwischen den Artikeln einer gegebenen Datenmenge von Einkaufstransaktionen, wobei das Vorhandensein einiger Artikel in einer Transaktion das Vorhandensein anderer Artikel in derselben Transaktion impliziert. Eine in einer Datenmenge von Einkaufstransaktionen entdeckte Assoziationsregel lautet beispielsweise „In 67% der Fälle, in denen Cola und Saft zusammen eingekauft werden, wird auch Bier gekauft."

Bevor auf den Apriori-Algorithmus zur Entdeckung von Assoziationsregeln eingegangen wird, wird hierfür zunächst eine formale Problemdarstellung angegeben.

2.1 Formale Problemdarstellung

Im folgenden werden formale Aspekte angesprochen, die eine theoretische Grundlage für die Diskussion des Apriori-Algorithmus bereiten:

Definition 1 $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ *mit* $n \in \mathbb{N}$ *sei eine Menge von Literalen.*

Jedes Literal steht für ein Item bzw. für ein bestimmtes Objekt. Ein Item ist z.B. ein Artikel, der zusammen mit anderen in einem Supermarkt eingekauft wird. Desweiteren sei eine Datenmenge von Transaktionen gegeben. Eine Beispieldatenmenge von Transaktionen ist in Abbildung 2.1 dargestellt.

Definition 2 $\mathcal{D}$ *sei eine Menge von Transaktionen* t_i *mit* $0 \leq i \leq |\mathcal{D}| - 1$*, wobei* $|\mathcal{D}|$ *die Anzahl der Transaktionen in* $\mathcal{D}$ *angibt.*

Definition 3 *Eine Transaktion* t_i *mit* $0 \leq i \leq |\mathcal{D}| - 1$ *sei eine Menge von Items, d.h.* $t_i \subseteq \mathcal{I}$*.*

Transaktion	Items
t_0	Saft, Cola, Bier
t_1	Saft, Cola, Wein
t_2	Saft, Wasser
t_3	Cola, Bier
t_4	Saft, Cola, Bier, Wein
t_5	Wasser
t_6	Schokolade, Cola, Chips
t_7	Schokolade, Schinken, Brot
t_8	Brot, Bier

Abbildung 2.1: Eine Datenmenge von Transaktionen.

Definition 4 *Ein k-Itemset sei eine Menge von k Items aus $\mathcal{I}$.*

In Abbildung 2.1 stellt die Menge {Cola, Bier} z.B. ein 2-Itemset dar. Eine Transaktion t_i mit $0 \leq i \leq |\mathcal{D}|-1$ enthält den k-Itemset X, wenn die Beziehung $X \subseteq t_i$ gilt. In Abbildung 2.1 enthalten die Transaktionen t_0, t_3 und t_4 den 2-Itemset {Cola, Bier}.

Definition 5 *Der Support eines k-Itemsets X sei die Anzahl der Transaktionen, die X enthalten, d.h.* $support(X) = |\{t \in \mathcal{D} \mid X \subseteq t\}|$.

Definition 6 *Der minimale Support s_{min} sei ein vom Benutzer spezifizierter Wert, der angibt, wie oft jeder k-Itemset mindestens in der Datenmenge $\mathcal{D}$ vorkommen muß, damit er als Muster von k Items für den Benutzer „interessant" ist.*

Der minimale Supportwert s_{min} liegt entweder als absolute Zahl, die zwischen 1 und $|\mathcal{D}|$ liegen muß, oder als Prozentangabe vor.

Definition 7 *Ein Large k-Itemset sei ein k-Itemset X, für den die Beziehung $support(X) \geq s_{min}$ gilt.*

Der 2-Itemset {Cola, Bier} kommt in Abbildung 2.1 in den Transaktionen dreimal vor, weist somit einen Supportwert von 3 auf und ist damit beispielsweise bei einem minimalen Supportwert s_{min} von 2 ein Large 2-Itemset. Nach [Agrawal et al., 1993] gilt der folgende Satz:

Satz 1 *Jede Teilmenge X' eines Large Itemsets X ist ebenfalls ein Large Itemset, da für jedes $X' \subset X$ gilt:* $\quad support(X') \geq support(X)$.

Beweisidee: Jede Teilmenge X' eines Large Itemsets X kommt mindestens genauso häufig wie ihre Obermenge X vor. In der Regel ist der Supportwert von X' aber größer als der von X, weil nicht jede Transaktion, die X' enthält, auch die zusätzlichen Items aus der Menge $X \setminus X'$ enthält. ■

Definition 8 *Ein Candidate k-Itemset sei ein k-Itemset $ci_k = \{i_1, \ldots, i_k\}$, für den gilt:*

$$\forall j \in \{1, \ldots, k\} : \ \{i_1, \ldots, i_k\} \setminus i_j \ \textit{ist ein Large (k-1)-Itemset}$$

Ein Candidate k-Itemset setzt sich somit aus k Large $(k-1)$-Itemsets zusammen.

Definition 9 *Eine Assoziationsregel sei eine Regel der Form $X \to Y$, die aus einem Large Itemset X in der Prämisse und einem Large Itemset Y in der Konklusion besteht. X und Y müssen disjunkt sein, bilden aber zusammen einen Large k-Itemset $X \cup Y$ mit $k \geq 2$.*

Der Support einer Assoziationsregel $X \to Y$ ist somit gleich dem Support des Large Itemsets $X \cup Y$. Die Assoziationsregel $\{Saft, Cola\} \to \{Bier\}$ weist somit einen Supportwert von $support(\{Saft, Cola, Bier\}) = 2$ auf.

Definition 10 *Die Konfidenz einer Assoziationsregel $X \to Y$ wird definiert durch*

$$confidence(X \to Y) = \frac{support(X \cup Y)}{support(X)}$$

Die Konfidenz gibt die Stärke einer Assoziationsregel $X \to Y$ an, die besagt, wie groß der Anteil der Transaktionen in $\mathcal{D}$ ist, die die Prämisse X und gleichzeitig zusammen mit X die Konklusion Y enthalten, während sich der Support auf die „statistische Signifikanz" der Regel bezieht [Agrawal et al., 1993].

In Abbildung 2.1 ergibt sich die Konfidenz der Assoziationsregel $\{Saft, Cola\} \to \{Bier\}$ mit

$$\begin{aligned} confidence(\{Saft, Cola\} \to \{Bier\}) &= \frac{support(\{Saft, Cola\} \cup \{Bier\})}{support(\{Saft, Cola\})} \\ &= \frac{|\{t_0, t_4\}|}{|\{t_0, t_1, t_4\}|} \ = \ \frac{2}{3} \ \approx \ 67\% \end{aligned}$$

Analog zur Angabe eines minimalen Supportwertes kann auch eine untere Schranke c_{min} für den Konfidenzwert einer Assoziationsregel $X \to Y$ vom Benutzer angegeben werden.

Definition 11 *Eine Assoziationsregel $X \to Y$ mit $support(X \to Y) \geq s_{min}$ sei für den Benutzer relevant, wenn der Konfidenzwert der Regel mindestens c_{min} beträgt, ansonsten sei sie irrelevant.*

Somit ergeben sich bei einem minimalen Supportwert $s_{min} = 2$ und einer minimalen Konfidenz $c_{min} = 75\%$ aus der Beispieldatenmenge in Abbildung 2.1 die folgenden für den Benutzer relevanten Assoziationsregeln:

Assoziationsregel	Konfidenz
{ Saft, Bier } → { Cola }	100%
{ Cola, Wein } → { Saft }	100%
{ Wein } → { Cola, Saft }	100%
{ Saft } → { Cola }	75%
{ Bier } → { Cola }	75%
{ Wein } → { Cola }	100%
{ Wein } → { Saft }	100%

Die Lernaufgabe besteht nun darin, alle Assoziationsregeln $X \rightarrow Y$ mit $support(X \rightarrow Y) \geq s_{min}$ und $confidence(X \rightarrow Y) \geq c_{min}$ in der Datenmenge $\mathcal{D}$ zu entdecken.[1]

2.2 Der Apriori–Algorithmus zum Entdecken von Assoziationsregeln

Mit Hilfe des Apriori-Algorithmus [Agrawal et al., 1996] werden zunächst alle häufig vorkommenden Muster in Form von Large Itemsets in der Datenmenge $\mathcal{D}$ ermittelt, damit die anschließende Generierung der Assoziationsregeln nur auf diejenigen Muster fokussiert wird, die mindestens so oft in der Datenmenge vorkommen wie es der Benutzer durch Angabe des minimalen Supportwertes s_{min} fordert. Daher wird jedes Large k-Itemset li_k ($k \geq 2$) in jeweils zwei disjunkte nicht-leere Teilmengen X und Y aufgeteilt ($li_k = X \cup Y$), um Assoziationsregeln der Form $X \rightarrow Y$ zu generieren. Wenn die Beziehung

$$confidence(X \rightarrow Y) = \frac{support(X \cup Y)}{support(X)} \geq c_{min}$$

gilt, dann wird die Regel $X \rightarrow Y$ mit der Konfidenz $confidence(X \rightarrow Y)$ und dem Support $support(X \cup Y)$ ausgegeben. Die Konfidenz gibt somit das Verhältnis des Supportwertes von $X \cup Y$ zum Supportwert von X an.

2.2.1 Entdeckung der Large Itemsets

Der Apriori-Algorithmus führt mehrere Durchläufe bzw. Iterationen über der Datenmenge $\mathcal{D}$ aus, um sukzessive alle Large Itemsets zu berechnen. In der ersten Iteration werden einfach alle häufig vorkommenden Items bzw. Large 1-Itemsets bestimmt. Danach konstruiert der Algorithmus in jeder Iteration aus der in der vorherigen Iteration entdeckten Menge der Large Itemsets eine Hypothesenmenge aller sich daraus ergebenden Candidate Itemsets, ermittelt die Supportwerte für jeden Candidate Itemset in den Transaktionen und selektiert auf der Basis

[1] $\mathcal{D}$ kann beispielsweise als Datenfile oder Datenbanktabelle vorliegen.

des minimalen Supportwertes die tatsächlichen Large Itemsets aus der Menge der Kandidaten heraus. Die auf diese Weise gewonnenen Large Itemsets dienen wiederum für die nächste Iteration als Basis zur Generierung einer neuen Hypothesenmenge. Dieser Prozeß wird solange wiederholt bis keine neuen Large Itemsets mehr gefunden werden können. Einen Überblick über den Apriori-Algorithmus gewährt Abbildung 2.2. Die Schritte 4 – 7 dienen dabei der Bestimmung der Large k-Itemsets für $k \geq 2$.

Der Apriori-Algorithmus

1. Preprocessing der Datenmenge $\mathcal{D}$ (siehe Kapitel 4 Implementierung)

2. $L_1 :=$ Menge der Large 1-Itemsets

3. Setze den Iterationszähler k auf 2

4. Bilde aus der Menge L_{k-1} der Large $(k-1)$-Itemsets die Hypothesen- bzw. Kandidatenmenge der Candidate k-Itemsets

   $$C_k := \{ci_k = \{i_1, i_2, \ldots, i_n\} \mid \forall j \in \{1, \ldots, n\} : \{i_1, i_2, \ldots, i_n\} \setminus i_j \in L_{k-1}\},$$

   indem jeweils k Large $(k-1)$-Itemsets zu einem Candidate k-Itemset vereinigt werden

5. Bestimme die Supportwerte für die Candidate k-Itemsets $ci_k \in C_k$ mit Hilfe eines Scan-Vorgangs über die Transaktionen in der Datenmenge $\mathcal{D}$

6. Filtere aus C_k alle Large k-Itemsets heraus und trage sie in $L_k = \{\, ci_k \in C_k \mid support(ci_k) \ \geq \ s_{min} \,\}$ ein

7. Wenn $L_k \neq \{\emptyset\}$, erhöhe k um 1 und gehe nach 4.

8. Sonst beginne mit der Regelgenerierung.

Abbildung 2.2: Der Apriori-Algorithmus im Überblick.

Betrachtet man die Beispieldatenmenge aus Abbildung 2.1, so erhält man nach dem ersten Scannen der Transaktionen innerhalb des Preprocessings der Datenmenge $\mathcal{D}$ zunächst die Menge C_1 aller Candidate 1-Itemsets mit ihren Supportwerten:

Itemmenge	Support
{ Cola }	5
{ Saft }	4
{ Bier }	4
{ Wein }	2
{ Wasser }	2
{ Schokolade }	2
{ Brot }	2
{ Chips }	1
{ Schinken }	1

Bei einem angenommenen minimalen Supportwert s_{min} von 2 wird die Menge L_1 der Large 1-Itemsets wie folgt aus denjenigen Kandidaten aufgebaut, die mindestens einen Supportwert von 2 aufweisen (vgl. Schritt 2 des Apriori-Algorithmus in Abbildung 2.2):

$$L_1 = \{ \{ \text{Cola} \}, \{ \text{Saft} \}, \{ \text{Bier} \}, \{ \text{Wein} \}, \{ \text{Wasser} \}, \{ \text{Schokolade} \}, \{ \text{Brot} \} \}$$

Jetzt werden mit Hilfe der Schritte 3 und 4 des Apriori-Algorithmus (siehe Abb. 2.2) aus L_1 alle 2-er Kombinationen gebildet, um Candidate 2-Itemsets zu erhalten. Diese werden in die Menge C_2 eingetragen, die damit aus

$$\binom{|L_1|}{2} = \binom{7}{2} = 21$$

2-Itemsets besteht.

Als nächstes werden in Schritt 5 des Algorithmus (siehe Abb. 2.2) die Supportwerte der Kandidaten aus C_2 durch ein zweites Scannen der Transaktionen in der Beispieldatenmenge bestimmt:

Itemmenge	Support	Itemmenge	Support
{ Cola, Saft }	3	{ Bier, Wein }	1
{ Cola, Bier }	3	{ Bier, Wasser }	0
{ Cola, Wein }	2	{ Bier, Schokolade }	0
{ Cola, Wasser }	0	{ Bier, Brot }	1
{ Cola, Schokolade }	1	{ Wein, Wasser }	0
{ Cola, Brot }	0	{ Wein, Schokolade }	0
{ Saft, Bier }	2	{ Wein, Brot }	0
{ Saft, Wein }	2	{ Wasser, Schokolade }	0
{ Saft, Wasser }	1	{ Wasser, Brot }	0
{ Saft, Schokolade }	0	{ Schokolade, Brot }	1
{ Saft, Brot }	0		

Alle 2-Itemsets, die mindestens zweimal vorkommen, werden in die Menge der Large 2-Itemsets L_2 aufgenommen (vgl. Schritt 6 in Abb. 2.2):

$L_2 = \{ \{ \text{Cola, Saft} \}, \{ \text{Cola, Bier} \}, \{ \text{Cola, Wein} \}, \{ \text{Saft, Bier} \}, \{ \text{Saft, Wein} \} \}$

Damit ist die zweite Iteration des Algorithmus abgeschlossen und die Ausführung des Algorithmus wird in Schritt 4 fortgesetzt. Die Kandidatenmenge C_3 der Candidate 3-Itemsets wird wie folgt gebildet:

- Aus der Menge L_2 werden jeweils zwei Large 2-Itemsets $li_2^1 = \{i_1^1, i_2^1\}$ und $li_2^2 = \{i_1^2, i_2^2\}$ zu einem 3-Itemset ci_3 vereinigt, so daß

  $$|ci_3| = |li_2^1 \cup li_2^2| = 3$$

  gilt. Folglich gibt es in beiden Mengen ein übereinstimmendes Item. Es gelte o.B.d.A. $i_1^1 = i_1^2$. Der 3-Itemset ci hat damit folgenden Aufbau:

  $$ci_3 = \{ i_1^1, i_2^1, i_2^2 \}$$

  Somit können die beiden Large 2-Itemsets { Cola, Saft } und { Cola, Bier }, die das Item *Cola* gemeinsam haben, zu einem 3-Itemset { Cola, Saft, Bier } vereinigt werden.

- Aufgrund von Satz 1 achtet man darauf, daß für jedes ci_3 und für jedes 2-Itemset s, das in ci_3 enthalten ist, folgende Bedingung gilt:

  $$\forall\, s \subset ci_3 : s \in L_2$$

  Mit li_2^1 und li_2^2 sind bereits zwei von den drei 2-Itemsets, die in dem 3-Itemset ci_3, enthalten sind, Large 2-Itemsets. Der dritte noch zu untersuchende 2-Itemset ist $\{i_2^1, i_2^2\}$, der die beiden unterschiedlichen Items aus li_2^1 und li_2^2 enthält. Wenn dieser ebenfalls ein Large 2-Itemset ist, dann kann ci_3 als Candidate 3-Itemset in die Hypothesenmenge C_3 aufgenommen werden, ansonsten wird ci_3 verworfen.

  Der 3-Itemset { Cola, Saft, Bier } umfaßt mit { Cola, Saft } und { Cola, Bier } bereits zwei Large 2-Itemsets. Es muß also nur noch ermittelt werden, ob der übrige darin enthaltene 2-Itemset { Saft, Bier } ebenfalls ein Large 2-Itemset ist. Da die Beziehung

  $$support(\{ Saft,\ Bier \}) = |\{t_0,\ t_4\}| = 2\ \geq s_{min}$$

  gilt, ist der 2-Itemset { Saft, Bier } ein Large 2-Itemset. Damit wird { Cola, Saft, Bier } als Candidate 3-Itemset in die Hypothesenmenge C_3 aufgenommen.

  Im Gegensatz dazu ist im 3-Itemset { Cola, Bier, Wein }, der sich aus der Vereinigung der beiden Large 2-Itemsets { Cola, Bier } und { Cola, Wein } ergibt, mit { Bier, Wein } eine 2-elementige Teilmenge enthalten, die keinen Large 2-Itemset darstellt. Deshalb kann hier mit Hilfe des Algorithmus a *priori* geschlossen werden, daß der 3-Itemset { Cola, Bier, Wein } kein Large 3-Itemset sein kann, weil nicht all seine Teilmengen Large 2-Itemsets sind. So wird der 3-Itemset { Cola, Bier, Wein } nicht in die Kandidatenmenge C_3 aufgenommen.

Insgesamt hat C_3 folgende Gestalt:

$$C_3 = \{ \{ \text{Cola, Saft, Bier} \}, \{ \text{Cola, Saft, Wein} \} \}$$

Nach dem dritten Scannen der Transaktionen der Beispieldatenmenge erhält man die tatsächlichen Supportwerte für jeden Kandidaten aus C_3 und die darauf basierende Menge L_3 der häufigen 3-elementigen Itemmengen:

Itemmenge	Support
{ Cola, Saft, Bier }	2
{ Cola, Saft, Wein }	2

$$L_3 = \{ \{ \text{Cola, Saft, Bier} \}, \{ \text{Cola, Saft, Wein} \} \}$$

Um die Menge C_4 zu generieren, müssen jeweils zwei Large 3-Itemsets, die zwei Items gemeinsam haben, vereinigt werden und dabei einen 4-Itemset erzeugen, der aus vier Large 3-Itemsets besteht. Aus den beiden Large 3-Itemsets in L_3 läßt sich zwar ein 4-Itemset mit { Cola, Saft, Bier, Wein } bilden, jedoch ist er kein Candidate 4-Itemset, da mit { Saft, Bier, Wein } bereits eine 3-elementige Teilmenge kein Large 3-Itemset ist. Somit sind die Mengen C_4 sowie L_4 leer. Die Entdeckung der Large Itemsets ist damit an dieser Stelle beendet.

Haben die zu entdeckenden Muster in Form von Large Itemsets eine maximale Kardinalität von N, so muß der Apriori-Algorithmus insgesamt N-mal die Datenmenge zu ihrer Entdeckung scannen. Bei einer Variante des Apriori-Algorithmus, dem AprioriTID-Algorithmus [Agrawal und Srikant, 1994], wird die eigentliche Datenmenge nach der ersten Iteration nicht mehr zum Zählen der Supportwerte der Kandidaten benutzt. Stattdessen wird dazu eine spezielle Menge verwendet, in der jede Transaktion durch eine Menge aller darin vorkommenden Kandidaten ersetzt wird. Enthält eine Transaktion keinen Candidate Itemset, so gibt es für diese Transaktion keinen Eintrag in der speziellen Menge. Deshalb kann gerade in späteren Iterationen die Anzahl der Einträge in dieser speziellen Menge geringer sein als die Anzahl der Transaktionen in der Datenmenge. Außerdem kann jeder einzelne Eintrag kürzer sein als die korrespondierende Transaktion in der Datenmenge, weil in späteren Iterationen wahrscheinlich nur sehr wenig Kandidaten in den Transaktionen enthalten sind. Jedoch wird gerade in früheren Iterationen und für sehr große Datenmengen sehr viel Speicher benötigt, da jeder Eintrag in der speziellen Menge durch die Aufnahme aller in der Transaktion vorkommenden Kandidaten sehr viel größer als die korrespondierende Transaktion sein kann.

Die beiden Algorithmen, Apriori und AprioriTID, generieren zuerst Kandidaten aus den Large Itemsets der vorherigen Iteration und scannen danach erst die Transaktionen in der Datenmenge, um die tatsächlichen Supportwerte der Kandidaten zu ermitteln. Es gibt zwei weitere Algorithmen zur Entdeckung von Assoziationsegeln, AIS [Agrawal et al., 1993] und SETM [Houtsma und Swami, 1993], bei denen dieses nicht der Fall ist, denn beide Algorithmen generieren während des Scannens der Transaktionen neue Kandidaten. Nach dem Lesen einer Transaktion wird jedes darin enthaltene Large k-Itemset li_k aus der vorherigen Iteration durch weitere Items aus der Transaktion erweitert, um neue Candidate $(k+1)$-Itemsets zu generieren und in eine neue Kandidatenmenge C_{k+1} einzutragen, ohne jede k-elementige Teilmenge eines Candidate $(k+1)$-Itemsets aus C_{k+1} daraufhin zu testen, ob sie ein Large k-Itemset ist. Deshalb liegt hier der Nachteil darin, daß im Gegensatz zum Apriori-Algorithmus mehr Candidate $(k+1)$-Itemsets unnötigerweise generiert und gezählt werden, die nicht zu der Menge der Large $(k+1)$-Itemsets gehören können, weil sie k-elementige Teilmengen enthalten, die nicht in der Menge der Large k-Itemsets enthalten sind (vgl. Satz 1, Seite 8).

2.2.2 Die Regelgenerierung

Um Assoziationsregeln zu generieren, wird jeder Large k-Itemset li_k mit $k \geq 2$ in einen Large m-Itemset a_m mit $1 < m < k$ als Prämisse und einen Large $(k-m)$-Itemset $(li_k \setminus a_m)$ als Konklusion der Assoziationsregel zerlegt. Dann werden alle Assoziationsregeln $a_m \rightarrow (li_k \setminus a_m)$ ausgegeben, für die gilt:

$$confidence(a_m \rightarrow (li_k \setminus a_m)) = \frac{support(li_k)}{support(a_m)} \geq c_{min}$$

2.2.2.1 Ein einfacher Algorithmus

Nach [Agrawal et al., 1996] werden aus jedem Large k-Itemset li_k mit $k \geq 2$ $(k-1)$-elementige Teilmengen $a_{k-1} \subset li_k$ gebildet und der Reihe nach getestet, ob alle Assoziationsregeln der Form $a_{k-1} \rightarrow (li_k \setminus a_{k-1})$ eine Konfidenz von mindestens c_{min} aufweisen können. Sobald ein Test positiv ausfällt, wird die Regel ausgegeben und die Prämisse a_{k-1} festgehalten, um daraus alle $(k-2)$-elementigen Teilmengen $a_{k-2} \subset a_{k-1}$ zu erzeugen, um Regeln der Form $a_{k-2} \rightarrow (li_k \setminus a_{k-2})$ zu generieren, usw. Dieses Vorgehen beschreibt einen Tiefendurchlauf, bei dem neue zu testende Regeln erzeugt werden, indem jeweils aus einer Regel, die eine Konfidenz von mindestens c_{min} aufweist, solange ein Element aus der Prämisse entfernt und in die Konklusion eingefügt wird, bis entweder der berechnete Konfidenzwert der neuen Regel kleiner als c_{min} ist oder die Prämisse nur noch aus einem einzigen Element besteht (siehe Abbildung 2.3).

Die Menge {A B C D E} sei z.B. ein Large 5-Itemset und nur die beiden Regeln {A C D E} $\rightarrow$ {B} und {A B C E} $\rightarrow$ {D} können eine Konfidenz von mindestens c_{min} aufweisen.

Algorithmus 1 zur Regelgenerierung

1. $m = k$;
   $a_m = li_k$;

2. $A = \{ a_{m-1} \,|\, a_{m-1} \subset a_m \;\wedge\; a_{m-1}\ ist\ ein\ Large\ (m-1)-Itemset\}$

3. Berechne für alle $a_{m-1} \in A$: $confidence(a_{m-1} \rightarrow (li_k \setminus a_{m-1}))$

4. Wenn $confidence(a_{m-1} \rightarrow (li_k \setminus a_{m-1})) \geq c_{min}$, dann gebe die Assoziationsregel $a_{m-1} \rightarrow (li_k \setminus a_{m-1})$ aus mit $confidence(a_{m-1} \rightarrow (li_k \setminus a_{m-1}))$ und $support(li_k)$;
   Wenn $|a_{m-1}| > 1$, dann setze a_m auf a_{m-1} und gehe nach 2

Abbildung 2.3: Der einfache Algorithmus zur Regelgenerierung im Überblick.

Im Laufe des Algorithmus werden aus den beiden Prämissen, {A C D E} und {A B C E}, neue auf Mindestkonfidenz zu testende Regeln erzeugt, indem je ein Element aus der Prämisse gelöscht und gleichzeitig in die Konklusion eingefügt wird. Insgesamt werden die folgenden Regeln mit einer 2-elementigen Konklusion auf Mindestkonfidenz getestet:

Neue Regeln aus {A C D E}	Neue Regeln aus {A B C E}
1. {A C D} → {B E}	5. {A B C} → {D E}
2. {A D E} → {B C}	6. {A B E} → {D C}
3. {C D E} → {B A}	7. {B C E} → {D A}
4. {A C E} → {B D}	8. {A C E} → {D B}

Die Schwachpunkte des einfachen Algorithmus zeigen sich in zwei Beobachtungen. Die erste Beobachtung ist, daß der Konfidenzwert von Regel 1 nicht c_{min} erreicht, weil {E} ⊂ {B E} und damit die Prämisse {A C D} eine Teilmenge der Prämisse {A B C D} der Regel {A B C D} → {E} ist, die aber laut Voraussetzung keinen Konfidenzwert von c_{min} vorweisen kann, denn es gilt:

$$\begin{array}{rrcl} & support(\{A\,C\,D\}) & \geq & support(\{A\,B\,C\,D\}) \\ \Longleftrightarrow & \frac{support(\{A\,B\,C\,D\,E\})}{support(\{A\,B\,C\,D\})} & \geq & \frac{support(\{A\,B\,C\,D\,E\})}{support(\{A\,C\,D\})} \\ \Longleftrightarrow & conf(\{A\,B\,C\,D\} \rightarrow \{E\}) & \geq & conf(\{A\,C\,D\} \rightarrow \{B\,E\}) \end{array}$$

Gleiches gilt auch für die Regeln 2, 3, 5, 6 und 7. Im allgemeinen gilt das folgende Lemma:

Lemma 1 *Für jeden Large Itemset li und jeden nicht-leeren Itemset a mit* $a \subset li$ *gilt: Wenn die Konfidenz der Assoziationsregel* $a \rightarrow (li \setminus a)$ *kleiner als* c_{min} *ist,* *dann ist auch die Konfidenz aller Assoziationsregeln der Form* $a' \rightarrow (li \setminus a')$ *mit* $a' \subset a$ *nicht größer als* c_{min}.

Beweis: Dieses folgt daraus, daß jede Teilmenge a' von a mindestens genauso häufig wie a vorkommt, d.h. $support(a') \geq support(a)$, und damit gilt:

$$\begin{aligned} \frac{support(li)}{support(a')} &\leq \frac{support(li)}{support(a)} \\ \Longleftrightarrow \quad confidence(a' \rightarrow (li \setminus a')) &\leq confidence(a \rightarrow (li \setminus a)) \end{aligned}$$

■

Die zweite Beobachtung ist, daß im obigen Beispiel gleiche Regeln mehrfach getestet werden, wie z.B. die Regeln 4 und 8.

2.2.2.2 Ein schnellerer Algorithmus

Der schnellere Algorithmus [Agrawal et al., 1996] (siehe Abb. 2.4) betrachtet die Konklusionen der Regeln, um die Schwachpunkte, die in den beiden Beobachtungen zum einfachen Algorithmus sichtbar sind, auszugleichen. Aus diesem Grund wird zunächst die Regel $a \rightarrow (li \setminus a)$ in $(li \setminus concl) \rightarrow concl$ umgewandelt, indem $(li \setminus a)$ durch $concl$ und a durch $(li \setminus concl)$ ersetzt werden. Jetzt kann folgendes Lemma angegeben werden:

Lemma 2 *Jede Assoziationsregel* $(li \setminus concl^*) \rightarrow concl^*$, *wobei* $concl^*$ *eine nichtleere Teilmenge von* $concl$ *sei, weist genau dann eine Konfidenz von mindestens* c_{min} *auf, wenn die Assoziationsregel* $(li \setminus concl) \rightarrow concl$ *eine Konfidenz von mindestens* c_{min} *besitzt.*

Beweis: Es gilt:

$$\begin{aligned} & & concl^* &\subset concl \\ \Longleftrightarrow & & support(li \setminus concl^*) &\leq support(li \setminus concl) \\ \Longleftrightarrow & & \frac{support(li)}{support(li \setminus concl^*)} &\geq \frac{support(li)}{support(li \setminus concl)} \\ \Longleftrightarrow & & confidence((li \setminus concl^*) \rightarrow concl^*) &\geq confidence((li \setminus concl) \rightarrow concl) \end{aligned}$$

■

Die Aussage des obigen Lemmas ist ähnlich zu der Eigenschaft, die besagt, daß jede Teilmenge eines Large Itemsets ebenfalls ein Large Itemset sein muß (vgl. Satz 1, Seite 8). Daher bietet es sich auch hier in Analogie zur Entdeckung der Large Itemsets an, jeden Large k-Itemset, der die Konsequenz einer für den Benutzer relevanten Regel bildet, aufzubewahren und daraus Large $(k+1)$-Itemsets zu generieren, die in neue zu testende Regeln als Konsequenzen eingesetzt werden (vgl. Schritt 3 in Algorithmus 2 zur Regelgenerierung in Abb. 2.4).

Algorithmus 2 zur Regelgenerierung

1. $H_1 = \{\, concl_1 \,|\, concl_1 \;\; ist \;\; eine \;\; 1 - elementige \;\; Teilmenge \;\; von \;\; l_k \,\}$
   $m = 1$

2. Berechne für alle $concl_m \in H_m$: $\quad confidence((li_k \setminus concl_m) \rightarrow concl_m)$
   Wenn $confidence((li_k \setminus concl_m) \rightarrow concl_m) \geq c_{min}$, dann gebe die Regel $(li_k \setminus concl_m) \rightarrow concl_m$ mit
   der Konfidenz $confidence((li_k \setminus concl_m) \rightarrow concl_m)$ und
   dem Support $support(li_k)$ aus.
   $C_m = C_m \cup concl_m$

3. Nehme die m-elementigen Konklusionen der für den Benutzer relevanten Regeln und generiere daraus $(m+1)$-elementige Konklusionen für neue zu testende Regeln:

   $$H_{m+1} := \{\{i_1, i_2, \ldots, i_{m+1}\} \;|\; \forall j \in \{1, \ldots, (m+1)\} : \{i_1, i_2, \ldots, i_{m+1}\} \setminus i_j \in C_m\}$$

4. Wenn $H_{m+1} \neq \emptyset$, dann erhöhe m um 1 und gehe nach 2.
   Sonst Ende.

Abbildung 2.4: Der schnellere Algorithmus zur Regelgenerierung im Überblick.

Im obigen Beispiel werden somit die beiden Konsequenzen {B} und {D} der beiden für den Benutzer relevanten Regeln, {A C D E} $\rightarrow$ {B} und {A B C E} $\rightarrow$ {D}, in der Menge C_1 aufbewahrt (siehe Schritt 2 in Abb. 2.4), um daraus den Large 2-Itemset {B D} zu generieren, der in die Menge H_2 der 2-elementigen Konklusionen für neue zu testende Regeln gespeichert wird (siehe Schritt 3 in Abb. 2.4). Dann wird aus dem Large Itemset {A B C D E} eine neue auf Mindestkonfidenz zu testende Regel {A C E} $\rightarrow$ {B D} generiert, indem der Large 2-Itemset {B D} aus H_2 als Konklusion und die übrigen Items aus {A B C D E} als Prämisse in die Regel eingesetzt werden (siehe Schritt 2 in Abb. 2.4). Diese Regel ist die einzige mit einer 2-elementigen Konklusion, die bei dem hier vorgestellten schnelleren Algorithmus zur Regelgenerierung aus dem Large Itemset {A B C D E} erzeugt und getestet wird. Folglich wird im Vergleich zum einfachen Algorithmus nur die 4. Regel {A C E} $\rightarrow$ {B D} getestet.

Kapitel 3

Datenstrukturen zur Verwaltung großer Datenmengen

Dieses Kapitel dient der Vorbereitung auf das nächste Kapitel, das sich mit der Implementierung des Apriori-Algorithmus unter dem Aspekt der Verwaltung großer Datenmengen befaßt. Deshalb werden in diesem Kapitel Datenstrukturen behandelt, die einen effizienten Umgang mit großen Datenmengen ermöglichen.

3.1 Problembeschreibung

Die Anwendung des Apriori-Algorithmus verlangt nach einer effizienten Verwaltung einer sich von Iteration zu Iteration in der Größe dynamisch verändernden Menge von Itemsets. Deshalb müssen bei der Implementierung des Apriori-Algorithmus Datenstrukturen für die Verwaltung einer Menge von Itemsets eingesetzt werden, die das Einfügen von Itemsets möglichst effektiv unterstützen und zum Zwecke der Kandidaten- oder Regelgenerierung einen effizienten Zugriff auf die Itemsets mit den dazugehörigen Supportwerten bereitstellen.

Würde man bei n Items, die in der zu untersuchenden Datenmenge vorkommen, in Iteration k eine direkt mit den Candidate k-Itemsets adressierbare Tabelle reservieren, in der es einen Platz für jeden überhaupt theoretisch möglichen Candidate k-Itemset geben würde, so würde die direkt adressierbare Tabelle aus

$$\binom{n}{k}$$

Einträgen bestehen, die je einen Candidate k-Itemset mit den dazugehörigen Supportwert enthalten würden. Ein Zugriff auf ein Candidate k-Itemset und dem zugehörigen Supportwert würde daher direkt über den Candidate k-Itemset in einer garantierten Zeit von $O(1)$ realisierbar sein.

Das Problem ist, daß beispielsweise allein für alle Candidate 2-Itemsets, die aus 10000 in der Datenmenge vorkommenden Items gebildet werden können, eine direkt adressierbare Tabelle mit

$$\binom{10000}{2} \approx 50$$

Millionen Einträgen benötigt werden würde. Eine Tabelle, die eine derart große Kardinalität aufweist, kann praktisch nicht im verfügbaren Hauptspeicher des Rechners allokiert werden. Wenn aber nur 1000 der insgesamt 10000 in der Datenmenge vorkommenden Items häufig vorkommen, so wären nur

$$\binom{1000}{2} = 499500$$

der knapp 50 Millionen Einträge der direkt adressierbaren Tabelle, die für alle überhaupt möglichen Candidate 2-Itemsets mit den dazugehörigen Supportwerten reserviert werden würden, tatsächlich belegt. Das bedeutet, daß hier sehr viel Speicherplatz verschwendet werden würde, weil die Menge der tatsächlich abgespeicherten Itemsets sehr viel kleiner als die Anzahl der Einträge in der Tabelle sein würde. Deshalb bietet sich hier die Anwendung von Hashing [Knuth, 1973], [Cormen et al., 1990] als effektive Alternative zur direkten Adressierung einer Tabelle an, denn Hashing benutzt typischerweise eine Tabelle, deren Größe proportional zur Anzahl der tatsächlich zu speichernden Itemsets ist. Anstatt direkt über den Itemset auf die entsprechende Position in der Tabelle zuzugreifen, wird beim Hashing die Tabellenposition, auf die zugegriffen werden soll, zunächst aus den Items des Itemsets berechnet. Im Durchschnitt benötigen Einfüge- und Suchoperationen beim Hashing eine Zeit von $O(1)$. In den meisten Fällen wird Hashing zur Verwaltung von Daten eingesetzt, die komplett im verfügbaren Hauptspeicher bearbeitet werden können.

Darüberhinaus muß geeignet auf Situationen eingegangen werden, in denen die Menge der aktuell zu verwaltenden Itemsets nicht komplett in dem zur Verfügung stehenden Hauptspeicher gehalten werden kann. Aus diesem Grund kommen Datenstrukturen, wie externes Hashing [Enbody und Du, 1988] oder B^+-Bäume [Comer, 1979], in Betracht, die eine effiziente externe Verwaltung von Daten ermöglichen.

Nachfolgend wird näher auf die soeben angesprochenen Datenstrukturen eingegangen.

3.2 Internes Hashing

Die Verwaltung einer Menge von Itemsets geschieht beim internen Hashing mit Hilfe einer Tabelle (Hashtabelle) der Kardinalität m, die mit den von $0, \ldots, m-1$ indizierten Positionen m Verweise auf Paare von Itemset und Supportwert enthält. Es wird hier vorausgesetzt, daß sowohl die Hashtabelle als auch die gesamte Menge der zu betrachtenden Itemsets, auf die innerhalb der Hashtabelle verwiesen wird, vollständig im verfügbaren Hauptspeicher untergebracht werden können.

Zum Einfügen eines neuen Itemsets oder zum Zugriff auf die Supportwerte von Itemsets muß die Position innerhalb der Hashtabelle gefunden werden, die einen freien bzw. den betreffenden Verweis auf das Paar von Itemset und Supportwert enthält. Zu diesem Zweck wird eine Hashfunktion definiert, die einen Itemset als Eingabe erwartet, um daraus unmittelbar die Position innerhalb der Hashtabelle zu berechnen, an der sich der Verweis befindet.

Da in der Regel die Beziehung

$$\binom{n}{k} \gg m$$

gilt, d.h. die Menge aller aus n Items potentiell erzeugbaren k-Itemsets wesentlich größer als die Anzahl m der zur Verfügung stehenden Positionen in der Hashtabelle ist, läßt es sich nicht vermeiden, daß durch die Hashfunktion zwei Itemsets auf dieselbe Position abgebildet werden und somit eine *Kollision* ausgelöst wird, auf die mit einer speziellen Strategie zur *Kollisionsbehandlung* reagiert werden muß. In diesem Zusammenhang spielt der sogenannte *Auslastungsfaktor*

$$\alpha = \frac{Anzahl\ aktuell\ belegter\ Hashtabellenpositionen}{Anzahl\ aller\ Hashtabellenpositionen}$$

eine wichtige Rolle. Je höher der Auslastungsfaktor α desto wahrscheinlicher sind Kollisionen. Kollisionen treten aber auch schon bei kleinem α mit relativ hoher Wahrscheinlichkeit auf. So besagt z.B. das sogenannte „Geburtstagsparadoxon“ (vgl. [Feller, 1968]), daß bei 23 oder mehr Personen, die sich gemeinsam auf einer Geburtstagsparty befinden, zwei von ihnen mit einer Wahrscheinlichkeit von über 50 Prozent am selben Tag Geburtstag haben. Bei 47 Personen liegt diese Wahrscheinlichkeit schon über 95 Prozent. Mit anderen Worten ausgedrückt, bedeutet dies, daß in einer Hashtabelle mit 365 Einträgen, die mit 47 aktuell gespeicherten Schlüsseln zu 12.9 Prozent gefüllt ist, die Wahrscheinlichkeit größer als 95 Prozent ist, daß zumindest zwei Schlüssel eine Kollision verursacht haben. Folglich sind Kollisionen auch in spärlich besetzten Hashtabellen kaum zu vermeiden.

Beim Einsatz von internem Hashing kommt es folglich darauf an, daß durch die Wahl einer geeigneten *Hashfunktion* jede Position in der Hashtabelle angesprochen wird und die einzufügenden Daten zur Minimierung der Anzahl der Kollisionen möglichst gleichmäßig über die Hashtabelle verteilt werden. Daneben müssen Kollisionen durch eine gesonderte *Kollisionsbehandlung* effizient aufgelöst werden.

3.2.1 Die Hashfunktion

Allgemein wird durch die Hashfunktion eine Abbildung $h : K \rightarrow \{0, \ldots, m-1\}$ definiert, die jedem Schlüssel k aus der Menge aller möglichen Schlüssel K einen Index (Hashadresse) $h(k)$ mit $0 \leq h(k) \leq m-1$ zum Zugriff auf eine Hashtabelle mit m Positionen zuordnet.

Dabei kommt es besonders darauf an, daß die Hashfunktion

- surjektiv ist, also alle Positionen mit $0 \leq h(k) \leq m-1$ in der Hashtabelle erfaßt,
- die Schlüssel möglichst gleichmäßig und willkürlich über die Positionen in der Hashtabelle verteilt, wobei jede Position $h(k)$ für einen willkürlich gewählten Schlüssel k mit gleicher Wahrscheinlichkeit ausgewählt wird, um die Anzahl der Kollisionen und den damit verbundenen Aufwand zur Kollisionsbehandlung zu minimieren und
- einfach und schnell zu berechnen ist, denn komplizierte Berechnungen kosten gerade bei einer Vielzahl von Hashfunktionsaufrufen enorm viel Rechenzeit.

Ein naheliegendes und einfaches Verfahren zur Erzeugung einer Hashadresse $h(k)$ zu einem gegebenen Schlüssel $k \in \mathbb{N}_0$ ist, den Rest von k bei ganzzahliger Division durch die Hashtabellengröße m zu nehmen (Divisions-Rest-Verfahren):

$$h(k) = k \ mod \ m \quad \text{mit } 0 \leq h(k) \leq m-1$$

Hier ist eine geeignete Wahl von m von Vorteil. Beispielsweise liefert $h(k)$ bei der Wahl von $m = 2^i$ nur die i niederwertigen Bits der Dualdarstellung von k, ohne die restlichen Bits bei der Hashadressenberechnung zu betrachten.

Es sei z.B. eine Hashtabelle mit einer Größe von $m = 2^8 = 256$ gegeben, um darin Variablennamen, die aus bis zu drei Buchstaben bestehen, zu verwalten. Jeder Buchstabe wird als ASCII-Zeichen gespeichert. Daher kann jeder Variablenname als 24-Bit Zahl, die sich aus drei ASCII-Zeichen zusammensetzt, repräsentiert werden.

Das Problem ist, daß durch die Hashfunktion $h(k) = k \ mod \ 256$ für jeden aus drei ASCII-Zeichen bestehenden Variablennamen $k = C_3C_2C_1$ nur das erste Zeichen C_1 für die Hashadressenberechnung herangezogen wird, weil der Drei-Zeichen-Schlüssel $C_3C_2C_1$ als 24-Bit Zahl mit dem numerischen Wert $C_3 \cdot 256^2 + C_2 \cdot 256^1 + C_3 \cdot 256^0$ betrachtet wird und durch die modulo 256 Berechnung auf den Wert C_1 reduziert wird.

Da durch die Hashfunktion $h(k) = k \ mod \ 256$ nur das letzte Zeichen ausgewählt wird, werden die Schlüssel $X1$, $X2$, $X3$, $Y1$, $Y2$ und $Y3$, wie folgt abgebildet:

$$\begin{aligned} h(\text{X1}) = h(\text{Y1}) = \text{'1'} \\ h(\text{X2}) = h(\text{Y2}) = \text{'2'} \\ h(\text{X3}) = h(\text{Y3}) = \text{'3'} \end{aligned}$$

Die tatsächlichen Hashtabellenpositionen ergeben sich aus den ASCII-Codes für '1', '2' und '3' und sind demnach 50, 51 bzw. 52. So werden die sechs Schlüssel

$X1$, $X2$, $X3$, $Y1$, $Y2$ und $Y3$ in ein gemeinsames Cluster von drei benachbarten Hashadressen abgebildet. Überdies werden fast gleiche Schlüssel auf benachbarte Hashadressen abgebildet, so daß man sagen kann, daß $h(k)$ Cluster innerhalb der Menge der Schlüssel bewahrt. Im Gegensatz dazu „bricht" eine geeignete Hashfunktion $h(k)$ Cluster innerhalb der Menge der Schlüssel auf, ohne sie zu bewahren oder die Menge der Schlüssel noch stärker zu clustern.

Laut [Knuth, 1973] wird eine bessere Verteilung der durch $h(k)$ produzierten Hashadressen erreicht, wenn m als Primzahl gewählt wird. Allerdings müssen bei der Wahl einer Primzahl in Verbindung mit der Divisions-Rest-Methode bestimmte Vorkehrungen getroffen werden. Insbesondere soll für m keine Primzahl der Form $m = r^k \pm a$ gewählt werden, wobei r die Basis der Schlüsselmenge K ist und k und a kleine Integerzahlen sind.

Als Beispiel werden wieder die Drei-Zeichen-Schlüssel $C_3C_2C_1$ betrachtet, wobei die Basis des Zeichensatzes wieder aus 8 Bits besteht und demzufolge r auf 256 gesetzt wird. Setzt man die Größe der Hashtabelle auf $m = 65537$ (Fermatsche Primzahl mit dem Wert $2^{16} + 1$) und berechnet man

$$h(C_3C_2C_1) = (C_3 \cdot 256^2 + C_2 \cdot 256^1 + C_1 \cdot 256^0)\ mod\ 65537,$$

so erhält man als Resultat

$$h(C_3C_2C_1) = (C_2C_1 - C_3)_{256}\,.$$

Das heißt, daß $h(C_3C_2C_1)$ als Zahl zur Basis 256 einfach eine Differenz von Produkten von Zeichen darstellt. Allgemein tendiert der Wert von $h(C_3C_2C_1)$ dazu, einfach eine Summe oder Differenz von Produkten von Zeichen C_i zu berechnen, wenn m in Form von $m = r^k \pm a$ angegeben wird. Solch eine Hashfunktion wird weder Cluster von Schlüssel „aufbrechen", noch wird sie Schlüssel gleichmäßig und willkürlich auf die Hashtabellenpositionen verteilen.

Weitere Methoden zur Wahl einer Hashfunktion sind z.B. die *multiplikative Methode*, die *Faltung*, die *Mid-Square-Methode* und die *Truncation*.

Bei der multiplikativen Methode werden Hashfunktionen der Form

$$h(k) = \lfloor m\ \ (kA\ \ mod\ \ 1)\rfloor$$

erzeugt, wobei sich „$kA\ \ mod\ \ 1$" auf die Nachkommastellen von kA bezieht. Ein Schlüssel k wird hier zunächst mit einer Konstanten A, $0 < A < 1$ multipliziert. Dann werden die Nachkommastellen mit m multipliziert und der ganzahlige Teil dieses Ergebnisses als Hashadresse genommen. Eine modulo m Berechnung ist hier nicht notwendig.

Beim Folding werden die Schlüssel in mehrere Abschnitte zerlegt. Die einzelnen Abschnitte werden dann z.B. aufaddiert (multipliziert, subtrahiert, etc.), um die Hashadresse zu erhalten. Ein 9-stelliger Schlüssel $k = 013402122$ kann z.B. in die drei Abschnitte 013, 402 und 122 zerlegt werden, die daraufhin zur

Hashadresse 537 aufaddiert werden. Ist die Größe der Hashtabelle mit $m = 1000$ beispielsweise eine 4-stellige Zahl, so kann die modulo m Berechnung für $h(k)$ eingespart werden, wenn sichergestellt wird, daß nur 3-stellige Hashadressen berechnet werden.

Die Mid-Square-Methode entnimmt einen Ausschnitt aus der Mitte eines Schlüssel, interpretiert ihn als Zahl und quadriert ihn, um die Hashadresse zu berechnen. Aus dem 9-stelligen Schlüssel $k = 013402122$ kann man z.B. die mittleren Ziffern 402 herausnehmen und quadrieren, um $h(k) = 402^2 = 161604$ zu erhalten. Wenn die Hashadresse 161604 die Hashtabellengröße m übersteigt, so können z.B. die mittleren vier Ziffern des Ergebnisses als Hashadresse $h(k) = 6160$ interpretiert werden, oder die Hashadresse ergibt sich zu $h(k) = 161604 \ mod \ m$.

Bei der Truncation werden einfach Teile des Schlüssels gelöscht und die übrigen Ziffern (oder Bits oder Zeichen) zur Berechnung der Hashadresse benutzt. Wenn z.B. der Schlüssel $k = 013402122$ gegeben ist, können bis auf die letzten drei alle Ziffern ignoriert werden. So erhält man $h(k) = 122$. Eine modulo m Berechnung für $h(k)$ kann dann eingespart werden, wenn sichergestellt werden kann, daß $h(k)$ für jedes k kleiner als m ist. Während die Truncation kaum Berechnungszeit erfordert, neigt sie dazu, die Schlüssel nicht willkürlich und gleichmäßig über die Positionen der Hashtabelle zu verteilen. Deshalb wird sie oft in Verbindung mit den anderen oben genannten Methoden eingesetzt und selten alleine verwendet.

In [Lum et al., 1971] wird das Verhalten verschiedener Typen von Hashfunktionen, wie das Divisions-Rest-Verfahren, die multiplikative Methode, die Mid-Square-Methode, die Faltung und die algebraische Verschlüsselung untersucht und miteinander verglichen. Das Ergebnis dieser Untersuchung ist, daß das Divisions-Rest-Verfahren insgesamt die beste Performanz vorweisen kann. Deshalb werden im Hinblick auf die Implementierung des Apriori-Algorithmus Hashfunktionen, die auf dem Divisions-Rest-Verfahren basieren, betrachtet.

Um Hashfunktionen für k-Itemsets $\{i_1, \ldots, i_k\}$ zu definieren, wird zunächst vorausgesetzt, daß jedes Item i_j mit $1 \leq j \leq k$ als natürliche Zahl interpretiert werden kann. Dann können beispielsweise folgende Hashfunktionen, die auf dem Divisions-Rest-Verfahren basieren, für Hashtabellen der Größe m aufgestellt werden:

$$\begin{aligned}
h_1(\{i_1, \ldots, i_k\}) &= (z_1 \cdot i_1 + \cdots + z_k \cdot i_k) \ mod \ m \quad , \\
& \text{ wobei } z_1, \ldots, z_k \text{ uniform verteilte 32-Bit-Zufallszahlen sind} \\
h_2(\{i_1, \ldots, i_k\}) &= (p_1 \cdot i_1 + \cdots + p_k \cdot i_k) \ mod \ m \quad , \\
& \text{ wobei } p_1, \ldots, p_k \text{ Primzahlen sind} \\
h_3(\{i_1, \ldots, i_k\}) &= (z_{i_1} \, xor \, z_{i_1+i_2} \, xor \cdots xor \, z_{i_1+\cdots+i_k}) \ mod \ m \quad , \\
& \text{ wobei die Items } i_1, \ldots, i_k \text{ den Zugriff auf ein Array von} \\
& \text{ uniform verteilten 32-Bit-Zufallszahlen steuern} \\
& \text{ (siehe Abb. 3.1)}
\end{aligned}$$

Position	0	1	2	3	4	5	6	7	8	$\cdots$
Zufallszahl	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	$\cdots$
für $h(\{1,3,4\})$		z_1			z_4				z_8	

$$h(\{1,3,4\}) = (z_1 \ xor \ z_4 \ xor \ z_8) \ mod \ m$$

Abbildung 3.1: Beispiel zur Hashfunktion h_3

Die soeben aufgezählten Hashfunktionen werden direkt innerhalb des Apriori-Algorithmus getestet, bei dem die zu untersuchende Datenmenge aus 100 verschiedenen Items und 1000 Transaktionen besteht. Durchschnittlich besteht jede Transaktion aus 14 Items. Um die Large Itemsets zu bestimmen, werden hier 4 Iterationen benötigt. Als Hashtabellengröße wird in jeder Iteration eine an der Größe der Hypothesenmenge C_k der Candidate k-Itemsets angepaßte Primzahl gewählt. Daher ergibt sich in jeder Iteration ein unterschiedlicher Auslastungsfaktor α der Hashtabelle. Es werden für jede Hashfunktion zehn Programmdurchläufe des Apriori-Algorithmus mit denselben Daten und jeweils verschiedenen Zufallszahlen für die Hashfunktionen h_1 und h_3 und verschiedenen Primzahlen für die Hashfunktion h_2 durchgeführt.

Der Test dient der Bestimmung der Effizienz der Hashfunktionen h_1, h_2 und h_3, indem die Anzahl der Kollisionen (in Prozent), die bei dem Einfügen der Candidate Itemsets in die Hashtabellen in den Iterationen 2 bis 4 auftreten, jeweils über zehn Programmdurchläufe gemittelt werden:

	Iteration 2 ($\alpha = 0.40$)	Iteration 3 ($\alpha = 0.41$)	Iteration 4 ($\alpha = 0.26$)
h_1	8%	9%	8%
h_2	4%	28%	12%
h_3	9%	9%	6%

Die Hashfunktionen h_1 und h_3 weisen im Gegensatz zur Funktion h_2 für jede Iteration eine relativ konstante Anzahl an Kollisionen auf. Bei h_2 kommt es zwar im Vergleich zu den anderen beiden Funktionen in Iteration 2 bei der Verarbeitung der Candidate 2-Itemsets zu verhältnismäßig wenig Kollisionen, doch in Iteration 3 steigt die Anzahl der Kollisionen relativ stark an, so daß hier ein größerer Aufwand zur Kollisionsbehandlung entsteht, der sich negativ auf die Laufzeit auswirkt. Werden noch größere Mengen von Candidate Itemsets verarbeitet, so würde sich ein Anstieg in der Anzahl der Kollisionen noch negativer auf die Laufzeit auswirken.

Die Funktion h_3 verhält sich in den Iterationen 2 und 3 ähnlich wie Funktion h_1. Sie schneidet jedoch in Iteration 4 besser als h_1 ab, so daß sie dort mit weniger Aufwand für Kollisionsbehandlungen auskommt als h_1. Beide Funktionen, h_1 und h_3, benutzen Zufallszahlen. Allerdings unterscheiden sie sich darin, wieviele Zufallszahlen sie benutzen und wie sie auf die Zufallszahlen zur Berechnung der

Hashadresse zugreifen. Bei der Hashfunktion h_1 wird ein Array von Zufallszahlen benutzt, das in der Größe proportional zur längsten Transaktion in der Datenmenge ist. Die Hashadresse eines Candidate k-Itemsets $ci_k = \{ i_1, i_2, \ldots, i_k \}$ wird berechnet, indem jeweils die Produkte aus dem i.ten Item in ci_k mit der i.ten im Array gespeicherten Zufallszahl für $1 \leq i \leq k$ gebildet und aufaddiert werden und auf das Ergebnis der Addition anschließend modulo m gerechnet wird. Im Gegensatz zur Hashfunktion h_1 benötigt die Hashfunktion h_3 ein viel größeres Array von Zufallszahlen, dessen Größe sowohl von der maximalen Transaktionslänge als auch von dem größten Wert für ein Item abhängt, denn bei der Berechnung der Hashadresse für $ci_k = \{ i_1, i_2, \ldots, i_k \}$ fungieren die Items aus ci_k als Index auf das Zufallszahlenarray. In Abbildung 3.1 errechnet sich z.B. die Hashadresse für den Candidate 3-Itemset $\{ 1, 3, 4 \}$ dadurch, indem drei Zufallszahlen durch *xor* verknüpft werden und auf das Ergebnis dieser Operation anschließend modulo m gerechnet wird. Die drei Zufallszahlen werden mit Hilfe der drei Items 1, 3 und 4 aus dem Zufallszahlenarray abgerufen. Zunächst wird mit Item 1 die Zufallszahl unter der Position 1 im Array abgerufen. Dann werden nacheinander die beiden Items 3 und 4 als Offset auf die jeweils aktuelle Arrayposition aufgeschlagen, um die Zufallszahlen unter den Positionen 4 ($= 1 + 3$) und 8 ($= 1 + 3 + 4$) zu erhalten. Somit benötigt man bei einer Anzahl von 100 Items, die durch die natürlichen Zahlen von 0 bis 99 repräsentiert werden können, und bei einer maximalen Transaktionslänge von 10 Items ein Zufallszahlenarray mit $99 + 98 + 97 + \ldots + 91 + 90 = 945$ Einträgen. Dieses führt z.B. dazu, daß bei vielen Items und langen Transaktionen im Gegensatz zu h_1 sehr viel mehr Speicher für das Array mit den Zufallszahlen für h_3 bereitgestellt werden muß.

3.2.2 Strategien zur Kollisionsbehandlung

Obgleich eine geeignete Hashfunktion die Anzahl der Kollisionen minimiert, indem sie für eine gleichmäßige Verteilung aller Schlüssel (hier: Verweise auf Itemsets) auf die gesamten Positionen innerhalb der Hashtabelle sorgt, müssen dennoch Maßnahmen zur Behandlung von auftretenden Kollisionen eingeleitet werden. Dazu gibt es zwei grundlegende Strategien [Cormen et al., 1990]:

1. *Separate Chaining*: Die Verweise aller Itemsets, die auf dieselbe Position in der Hashtabelle abgebildet werden, werden außerhalb der Hashtabelle in einer linearen Liste (Überlaufliste) miteinander verkettet. In den einzelnen Positionen der Hashtabelle werden lediglich die Listenköpfe gespeichert.

2. *Offenes Hashing*: Bei einer Kollision werden solange weitere Positionen innerhalb der Hashtabelle untersucht, bis entweder eine freie Position (offene Stelle) oder der gesuchte Verweis gefunden wird. Deshalb muß für jeden Itemset eine beliebige Reihenfolge bzw. Permutation aller Hashadressen (Sondierungsfolge) gebildet werden, durch die festgelegt wird, in welcher

Reihenfolge die Hashadressen beim Eintreten einer Kollision inspiziert werden. In [Gonnet und Baeza-Yates, 1990] und [Ottmann und Widmayer, 1990] werden verschiedene offene Hashverfahren untersucht, wobei es jedesmal darauf ankommt, eine geeignete Sondierungsfolge zu finden, um bei der Suche nach der richtigen Position möglichst wenig bereits besetzte Positionen inspizieren zu müssen. Einige der in [Gonnet und Baeza-Yates, 1990] beschriebenen Verfahren realisieren dieses durch Inkaufnahme eines erhöhten Aufwandes beim Einfügen der Elemente zum Aufbau der Hashtabelle, wobei beispielsweise lange Sondierungsfolgen verkürzt, indem kurze Sondierungsfolgen verlängert werden.

Eines der günstigsten Verfahren ist das *Double Hashing*, weil dessen Performanz sehr nahe an die des „idealen" Schemas des *uniformen Hashings* heranreicht, in dem jede mögliche Sondierungsfolge mit gleicher Wahrscheinlichkeit für einen Such- oder Einfügevorgang gewählt wird (vgl. [Cormen et al., 1990]).

Der Kern dieses Verfahrens ist die Anwendung einer Hashfunktion der Form

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \; mod \; m \quad ,$$

wobei h_1 und h_2 Hilfsfunktionen sind. Die Variable i steht für die i.te zu untersuchende Position innerhalb der Sondierungsfolge. Zu Beginn wird bei einem Zugriff auf die Hashtabelle die Position $h_1(k)$ geprüft. Nachfolgende zu inspizierende Positionen ergeben sich jeweils aus der vorherigen durch Aufaddieren eines Offsets $h_2(k)$ und anschließender modulo m Berechnung. Damit sichergestellt ist, daß notfalls die gesamte Hashtabelle der Größe m untersucht werden kann, muß $h_2(k)$ relativ prim zu m sein, denn sonst haben $h_2(k)$ und m einen größten gemeinsamen Teiler $d > 1$, so daß bei der Suche nach dem Element k nur $\frac{1}{d}$ Prozent der Hashtabelle geprüft werden. Folglich sind in jeder Sondierungsfolge sämtliche Positionen der Hashtabelle enthalten, wenn

- m eine Zweierpotenz ist und die Bildmenge von $h_2(k)$ nur aus ungeraden Zahlen besteht oder
- m eine Primzahl ist und die Bildmenge von $h_2(k)$ aus ganzen Zahlen besteht, die kleiner als m sind, z.B.

$$\begin{aligned} h_1(k) &= k \; mod \; m \quad , \\ h_2(k) &= 1 + (k \; mod \; m') \quad , \end{aligned}$$

  wobei m' ein bißchen kleiner als m ist (z.B. $m-1$ oder $m-2$).

Im Gegensatz zur 1. Strategie wird bei der 2. Strategie zum einen kein zusätzlicher Speicher zur Verwaltung von Listen benötigt und zum anderen besteht nicht die Gefahr, daß viele Positionen in der Hashtabelle unbesetzt bleiben, weil alle Verweise auf Itemsets in wenigen aber dafür langen Überlauflisten gehalten werden.

3.2.3 Erweiterung

Ein Nachteil von Hashverfahren ist, daß im allgemeinen die Elemente, die in der Hashtabelle gespeichert sind, nicht effizient in Sortierreihenfolge ausgegeben werden können. Es besteht aber die Möglichkeit, die abzuspeichernden Elemente in ihrer Einfügereihenfolge mit Hilfe von zusätzlichen Zeigern zu verketten.

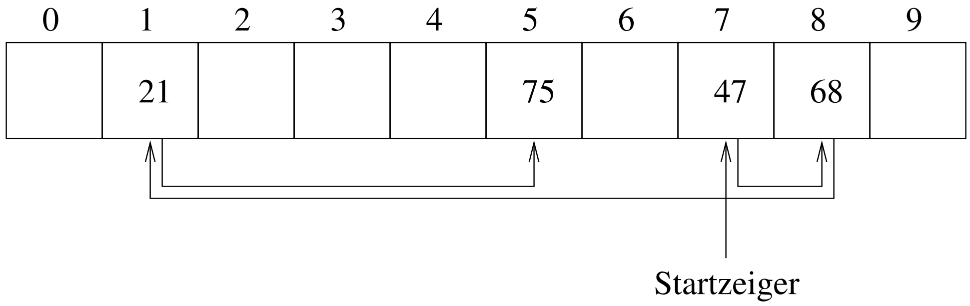


Abbildung 3.2: Verkettung der Elemente in einer Hashtabelle über die Einfügereihenfolge

Abbildung 3.2 zeigt eine Hashtabelle mit 10 Positionen, in der die Elemente 47, 68, 21 und 75 in dieser Reihenfolge eingefügt werden.

Eine sortierte Ausgabe der in einer Hashtabelle befindlichen Elemente ist dann effizient möglich, wenn die Elemente in Sortierreihenfolge in die Hashtabelle eingefügt worden sind. In diesem Fall muß die durch die zusätzlichen Zeiger aufgebaute Liste nur sequentiell ab dem Element, das zuerst eingefügt worden ist, durchlaufen werden.

3.3 Externes Hashing

Bei dem oben angesprochenen internen Hashing ist das Einfügen von Daten über den vorgesehenen Hauptspeicherplatz hinaus unmöglich. Werden die zu bearbeitenden Datenbestände größer als das sie als Ganzes im Hauptspeicher gehalten werden können, müssen sie auf externen Speichermedien verwaltet werden. Aus diesem Grund sind dynamische Hashverfahren interessant, die in Anlehnung an das interne Hashing einen schnellen durchschnittlichen Zugriff in $O(1)$ auf die in Datenblöcken organisierte Datenmenge ermöglichen [Enbody und Du, 1988], in die die zu verarbeitenden Itemsets mit ihren Supportwerten abgespeichert werden.

Ein Datenblock definiert die kleinste zu übertragende Einheit zwischen Hauptspeicher und externem Speichermedium und bildet die Grundlage zu Performanzanalysen externer Verfahren. Die meisten Rechner haben eine feste Blockgröße für den Transfer von Daten zwischen Hauptspeicher und externen Speichermedien, typischerweise 1024 Bytes oder ein Vielfaches davon. Damit können mehrere Datensätze in einem Block untergebracht werden, so daß eine Blockkapazität b definiert wird, die angibt, wieviele Datensätze neben Verwaltungsinformationen in einem Block gespeichert werden können.

Anstelle einer Hashtabelle wird eine aus Datenblöcken bestehende Datei (Hashdatei) verwendet, in der jeder einzelne Block durch seine relative Adresse, beginnend bei 0, direkt angesprochen werden kann. Wenn eine Hashdatei von m Blöcken mit den Blockadressen 0 bis $m-1$ aufgrund von Einfügeoperationen (hier: Einfügen von Itemset mit Supportwert) voll wird, so wird die Hashdatei zur Aufnahme weiterer Daten erweitert, indem ein neuer leerer Block mit der Blockadresse m an das Ende der Hashdatei angehängt wird. Sobald die Kapazität der Hashdatei erneut ausgeschöpft ist, wird jeweils ein neuer leerer Block an das Ende der Hashdatei angefügt.

Beim externen Hashing ordnet eine Hashfunktion $h(k)$ einem Schlüssel k die relative Adresse $h(k)$ mit $0 \leq h(k) \leq m-1$ des Blocks zu, in dem der Datensatz mit Schlüssel k zu speichern bzw. zu suchen ist.

Das Problem ist, daß man nicht einfach ein und dieselbe Hashfunktion bei sich änderndem m verwenden kann, weil sonst nicht garantiert werden kann, daß gespeicherte Datensätze auch wiedergefunden werden. Außerdem muß eine globale Reorganisation, also das Umspeichern sämtlicher gespeicherter Daten gemäß einem veränderten m aus Effizienzgründen vermieden werden. Dieses Problem kann gelöst werden, indem nur Teilbereiche des externen Speicherbereichs – meist einzelne Datenblöcke – reorganisiert werden und indem festgehalten wird, für welche Teilbereiche eine Reorganisation erfolgte und welche neue Hashfunktion dabei verwendet wurde.

Dynamische Hashverfahren zur Verwaltung extern repräsentierter Datenbestände sind beispielsweise das sogenannte *erweiterbare Hashing* und das *externe lineare Hashing*.

3.3.1 Erweiterbares Hashing

Das Prinzip des erweiterbaren Hashings ist, daß der Hashfunktionswert benutzt wird, um ein zusätzliches Verzeichnis zu indizieren, welches die Blockadressen enthält, unter denen der Datenbestand extern abgespeichert ist [Fagin et al., 1979]. Sofern angeforderte Blöcke sich nicht schon aufgrund von vorhergehenden Zugriffen im Hauptspeicher befinden, werden sie dorthin transferiert. Falls der zur Verfügung stehende hauptspeicherinterne Pufferbereich für Blöcke komplett belegt ist, muß bei einem Transfer ein anderer Block ausgelagert werden.

Das Verzeichnis besteht aus einer zusammengefaßten Repräsentation eines

Tries [Enbody und Du, 1988] der Tiefe n, der 2^n Blockadressen beeinhaltet. Zur Adressierung der diskbasierten Hashdatei werden n Bits des berechneten Hashfunktionswertes[1] benutzt, um auf die jeweiligen Blockadressen innerhalb des Verzeichnisses zuzugreifen.

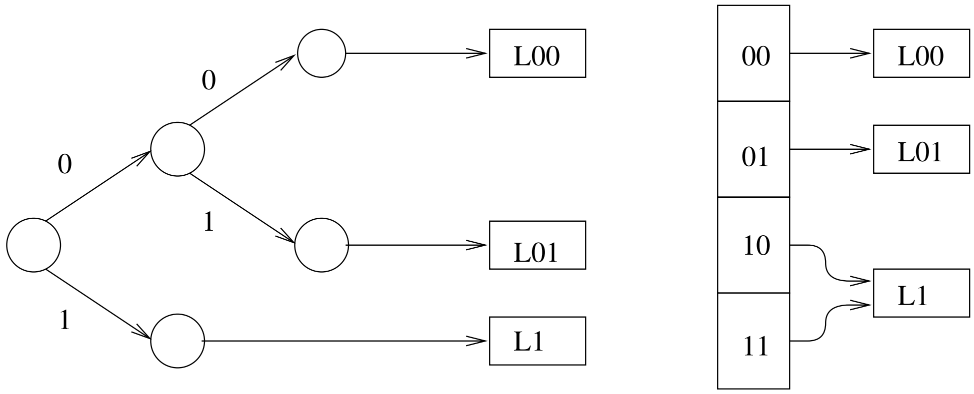


Abbildung 3.3: Ein *Trie* und ein Verzeichnis zur Repräsentation des *Tries*

Abbildung 3.3 illustriert die Beziehung zwischen einem *Trie* und seiner Repräsentation als Verzeichnis, das sich wie folgt formiert:

Am Anfang besteht die Hashdatei aus einem einzigen Block und das Verzeichnis besitzt einen Eintrag zur Adressierung dieses Blocks. Die Tiefe des *Tries* ist 0, denn 0 Bits jedes Hashfunktionswertes werden untersucht, um zu bestimmen, in welchem Block ein Datensatz abgelegt wird, d.h., daß alle neu einzufügenden Daten zunächst in Block L0 abgespeichert werden. Wenn dieser Block im Laufe der Zeit durch das Einfügen eines neuen Datensatzes „überläuft" (*Overflow*), dann wird der Block in die beiden Blöcke L0 und L1 aufgeteilt. Das Verzeichnis wird verdoppelt, um die Adresse des neuen Blocks L1 unterzubringen. Infolgedessen erhöht sich die Tiefe des Verzeichnisses um 1.

Nach dem Splitten genügt es, ein einziges Bit des Hashfunktionswertes zu untersuchen, um festzustellen, ob neue einzufügende Datensätze in Block L0 oder L1 abgelegt werden sollen. Sobald die Kapazität einer dieser beiden Blöcke ausgeschöpft ist (z.B. L0), findet wie zuvor eine Aufsplittung statt. Der Block L0 wird in L00 umbenannt und ein neuer Block L01 reserviert. Jetzt werden die in Block L00 gespeicherten Datensätze zwischen Block L00 und L01 gesplittet, indem die beiden untersten Bits der Hashwerte der einzelnen Schlüssel zu den abgespeicherten Datensätzen untersucht werden. Wenn die beiden untersten Bits des Hashwertes $h(k)$ den Zustand 00 besitzen, wird der Datensatz mit dem Schlüssel k in Block L00 abgelegt, und wenn sie den Zustand 01 besitzen, wird der Datensatz mit dem Schlüssel k in Block L01 gespeichert. Wieder wird das Verzeichnis verdoppelt, damit die Adresse des dritten Blocks L01 einen Platz darin findet.

[1] Der Hashfunktionswert kann beispielsweise aus 32 Bit bestehen.

Diese Verdoppelung verursacht ein mehrfaches Abspeichern der Adresse des nicht gesplitteten Blocks L1. Nun hat das Verzeichnis vier Einträge, eine Tiefe von 2 und indiziert die Blöcke L00, L01 und L1 (siehe Abb. 3.3).

Sollte der Block L1 gesplittet werden, so besitzt das Verzeichnis bereits einen Platz für die Adresse des neuen Blocks, da der Block L1 redundanterweise zweimal im Verzeichnis adressiert wird. Jeder Block hat eine Tiefe n_b die angibt, wieviele Bits des Hashfunktionswertes benötigt werden, um auf die Blockadresse zuzugreifen. Bei einer Verzeichnistiefe von n, d.h. n Bits des Hashfunktionswertes müssen untersucht werden, damit alle Einträge des Verzeichnisses adressiert werden können, erscheint die Adresse eines Blocks mit der Tiefe n_b genau 2^{n-n_b}-mal im Verzeichnis. In Abbildung 3.3 haben die Blöcke L00 und L01 eine Tiefe von 2 und werden genau $2^{2-2} = 2^0 = 1$-mal adressiert, wohingegen Block L1 eine Tiefe von 1 aufweist und genau $2^{2-1} = 2^1 = 2$-mal adressiert wird.

Nach der Aufsplittung eines Blocks erhöht sich seine Tiefe n_b um 1. Das Verzeichnis muß immer dann verdoppelt werden, wenn die Tiefe eines Blocks die Tiefe des Verzeichnisses übersteigt.

3.3.2 Externes lineares Hashing

Das externe lineare Hashing ist ein verzeichnisloses Verfahren, bei dem die Hashfunktion h aus zwei Hashfunktionen h_l und h_{l+1} besteht, durch die die gesamte dynamisch wachsende Hashdatei adressiert werden kann [Larson, 1988]. Ist eine Hashdatei von einer anfänglichen Größe von m_0 Blöcken durch das Einfügen von Datensätzen auf eine aktuelle Größe von m Blöcken angewachsen, wobei $m_0 \cdot 2^l \leq m < m_0 \cdot 2^{l+1}$ für eine natürliche Zahl l gilt, die die Anzahl der kompletten Hashdateiverdoppelungen angibt, so adressiert h_l den Adreßbereich $0, \ldots, m_0 \cdot 2^l - 1$ und h_{l+1} den Adreßbereich $0, \ldots, m_0 \cdot 2^{l+1} - 1$. Die Hashdatei wird infolge von Einfügeoperationen vergrößert, indem ein neuer Block mit der Adresse m an das Dateiende angefügt wird. Danach wird ein Block mit der Adresse i ausgewählt, dessen zugehörige Datensätze durch Anwendung von h_{l+1} auf die beiden Blöcke i und m verteilt werden. Die Adresse i des zu splittenden Blocks ergibt sich zu $i = m - m_0 \cdot 2^l$. Damit durchläuft i der Reihe nach die Adressen 0 bis $m_0 \cdot 2^l - 1$ (siehe Abb. 3.4). Am Anfang sind i und l gleich 0. Nach jeder abgeschlossenen Verdoppelung der ursprünglichen Hashdatei wird i wieder auf 0 gesetzt.

Um Datensätze abzuspeichern und auszulesen, wird zum einen die Hashfunktion h_l zur Adressierung der Blöcke mit den Adressen i bis $m_0 \cdot 2^l - 1$ verwendet, d.h. für alle noch nicht gesplitteten Blöcke. Zum anderen wird die Funktion h_{l+1} zur Adressierung der Blöcke 0 bis $i - 1$ und $m_0 \cdot 2^l$ bis m benutzt, d.h. für bereits gesplittete Blöcke und für Blöcke, die seit der letzten Verdoppelung der Hashdatei neu hinzugefügt worden sind.

Das Kriterium für das Erweitern der Hashdatei um einen Block ist beim externen linearen Hashing üblicherweise der Belegungsfaktor

$$\frac{n}{b \cdot m}$$

der Hashdatei, wobei n die Anzahl der aktuell gespeicherten Datensätze bezeichnet. Überschreitet der Belegungsfaktor in Folge einer Einfügeoperation einen festgesetzten Schwellenwert, so wird die Hashdatei jeweils um einen Block erweitert. Da aber beim externen linearen Hashing der nächste zu splittende Block unabhängig vom einzufügenden Datensatz festliegt, kann keine Rücksicht darauf genommen werden, ob der Datensatz noch in dem ihm zugeordneten Block einen Platz findet oder nicht. Deshalb werden spezielle *Überlaufblöcke* benötigt, wenn ein neu einzufügender Datensatz gemäß der Hashfunktion h auf einen Block (Primärblock) abgebildet wird, der bereits mit b darin enthaltenen Datensätzen voll ist und nicht direkt gesplittet wird.

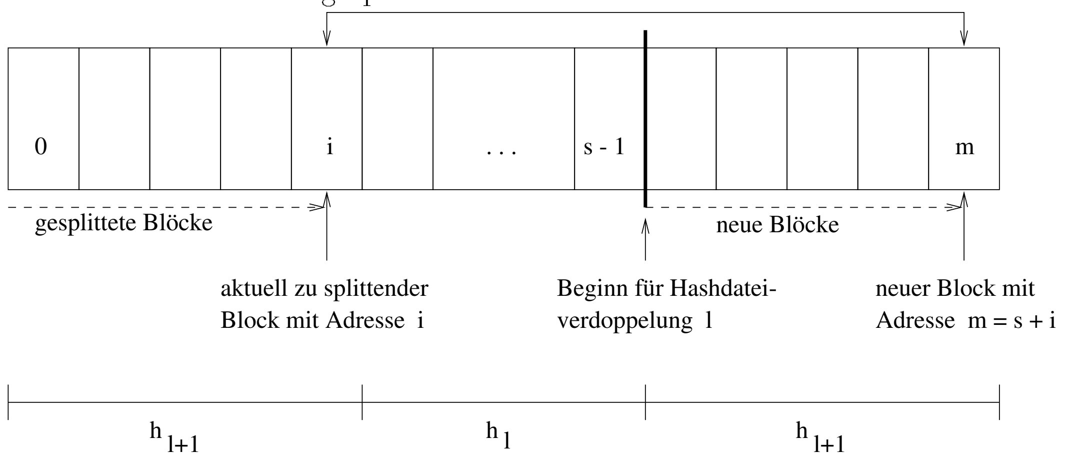


Abbildung 3.4: Externes lineares Hashing. Der Wert von s beträgt $m_0 \cdot 2^l$ und gibt die Anzahl der Blöcke an, die sich in einer Hashdatei befinden, die anfangs aus m_0 Blöcken bestand und sich l-mal in der Größe verdoppelt hat.

3.3.2.1 Überlaufblöcke

Für Primärblöcke, die zwar schon mit b gespeicherten Datensätzen ihre Kapazität voll ausgeschöpft haben und dennoch neue Datensätze aufnehmen müssen, werden zu diesem Zweck Überlaufblöcke reserviert. Überlaufblöcke können separat in einer eigens dafür vorgesehenen Datei abgelegt werden und theoretisch selbst wieder mit Hilfe von externem linearen Hashing verwaltet werden [Ramamohanarao und Sacks-Davis, 1984]. Allerdings hat diese Vorgehensweise den Nachteil, daß mehrere unabhängige Dateien verwaltet werden müssen, die eigentlich logisch zusammengehören.

Das „Buddy-In-Waiting"-Verfahren [Seltzer und Yigit, 1991] stellt einen Mechanismus zur Verfügung, der sowohl die Abbildung aller Primär- und damit verbundenen Überlaufblöcke auf eine einzige Hashdatei unterstützt als auch die einfache lineare Reihenfolge der zu splittenden Primärblöcke beibehält. Überlaufblöcke werden deshalb intern für jede Expansion (Verdoppelung der Anzahl der Primärblöcke) der Hashdatei reserviert. Diese Überlaufblöcke werden dann von jedem Primärblock benutzt, dem mehr als b Datensätze zugeordnet werden. Abbildung 3.5 zeigt die Anordnung von Primär- und Überlaufblöcken innerhalb derselben Hashdatei.

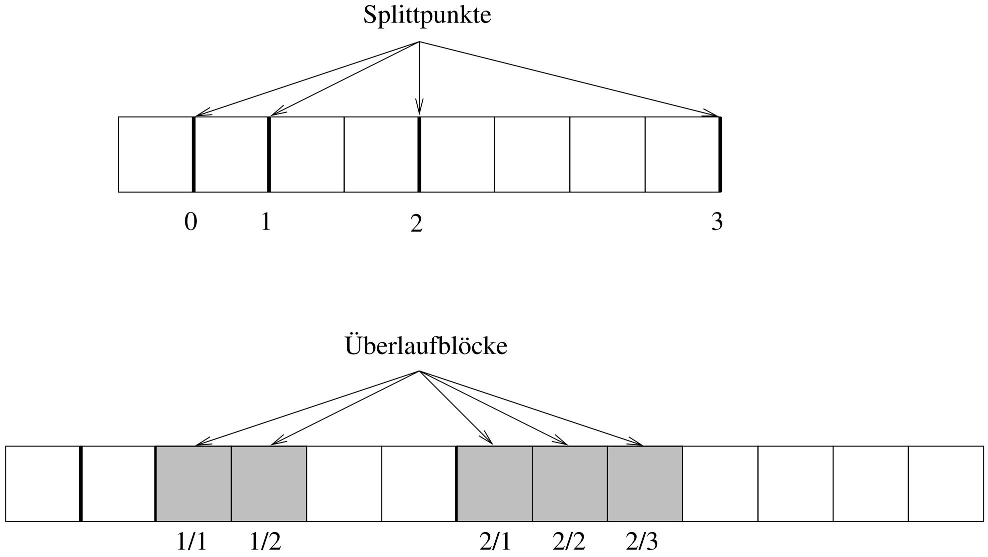


Abbildung 3.5: Mit dem Erreichen eines Splittpunktes hat sich die Anzahl der Blöcke in der Hashdatei verdoppelt. In dieser Abbildung werden zwei Überlaufblöcke am ersten und drei Überlaufblöcke am zweiten Splittpunkt reserviert.

3.3.2.2 Pufferverwaltung

Während die Blöcke kleinerer Hashdateien komplett im verfügbaren Hauptspeicher gepuffert werden können, ist es für größere Hashdateien praktisch unmöglich. Deshalb bietet sich bei Hashdateien, deren Blöcke in ihrer Gesamtheit nicht komplett im Hauptspeicher gepuffert werden können, eine LRU-Ersetzungsstrategie[2] der Blöcke an [Seltzer und Yigit, 1991].

Die gepufferten Blöcke werden in LRU-Reihenfolge verkettet, damit eine schnelle Ersetzung von Blöcken ermöglicht werden kann. Ein effizienter Zugriff auf Primärblockpuffer wird über ein im Hauptspeicher befindliches logisches Feld von Blockzeigern sichergestellt, während auf die Pufferbereiche für Überlaufblöcke effizient zugegriffen werden kann, indem die jeweiligen Pufferbereiche direkt mit

[2] LRU = Least Recently Used. Diejenigen Blöcke im Puffer, auf die innerhalb der letzten Zugriffe am wenigsten zugegriffen worden ist, werden zuerst durch neu einzulagernde Blöcke ersetzt.

dem Puffer, der den Vorgängerblock (entweder ein Primär- oder ein anderer Überlaufblock) enthält, verbunden werden. Das bedeutet, daß sich ein Überlaufblock

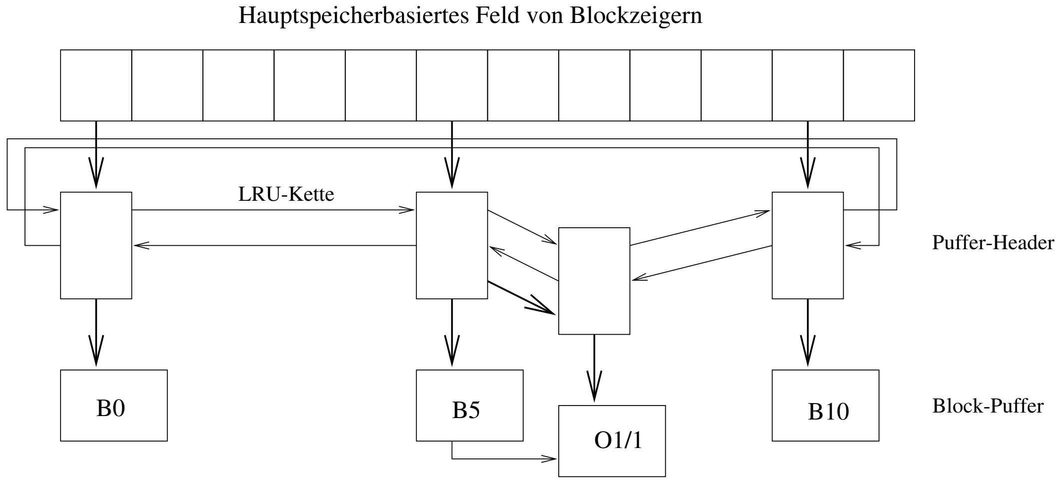


Abbildung 3.6: Drei direkt über das hauptspeicherbasierte Feld von Blockzeigern adressierbare Primärblöcke sind in die Block-Puffer B0, B5 und B10 eingelagert. Der im Block-Puffer O1/1 gespeicherte Überlaufblock ist sowohl physikalisch mit dem Puffer-Header als auch logisch mit dem Block-Puffer des zugehörigen Primärblocks B5 verbunden.

nur dann im Puffer befindet, wenn der zugehörige Primärblock bereits im Puffer eingelagert worden ist. Bei der Auslagerung eines Primärblocks werden ebenfalls alle damit verbundenen Überlaufblöcke ausgelagert. Abbildung 3.6 veranschaulicht die Datenstrukturen für das Puffermanagement.

3.3.2.3 Bemerkung zur Speicherplatzausnutzung

Man kann zeigen, daß bei Gleichverteilung der Datensätze die erwartete Speicherplatzausnutzung in einem dynamischen Hashverfahren, das mit rekursiver Halbierung (Verteilung von Datensätzen aus einem Block auf zwei Blöcke) arbeitet, ohne Berücksichtigung von Überlaufblöcken, bei $ln\, 2$, also etwa 69%, liegt [Fagin et al., 1979], [Larson, 1978]. Versucht man jedoch einen konstant hohen Belegungsfaktor zu erreichen, so ergibt sich zwischen zwei Dateiverdoppelungen eine gewisse Diskontinuität bei der erwarteten Anzahl der zu einem Primärblock gehörigen Überlaufblöcke, denn gegen Ende der Dateiverdoppelung wird die Anzahl der Überlaufblöcke, die zu einem bereits gesplitteten Primärblock gehören, wesentlich geringer sein als die Anzahl derer, die zu einem noch nicht gesplitteten Block gehören. Dieser Effekt läßt sich mit Hilfe *partieller Expansionen* abschwächen [Larson, 1980]. Dazu werden etwa in einer ersten partiellen Expansion die Inhalte von jeweils zwei auf drei Datenblöcke, von denen einer die Hashdatei vergrößert, und in einer zweiten partiellen Expansion die Inhalte entsprechend von drei auf vier Datenblöcke verteilt. Nichtsdestotrotz wird häufig die Speicherplatzausnutzung der Überlaufblöcke deutlich hinter derjenigen der Primärblöcke zurückbleiben, insbesondere dann, wenn die Kapazität der Überlaufblöcke groß ist.

3.4 B^+-Bäume

Der B^+-Baum als Variante des B-Baumes ist eine weitere Alternative zur Organisation großer extern zu verwaltender Datenmengen [Comer, 1979]. Im Gegensatz zum B-Baum sind beim B^+-Baum die Verweise auf Datensätze bzw. die Datensätze selbst zusammen mit den kompletten Suchschlüsseln nur in den Blättern, die untereinander verkettet sind, zu finden. Deshalb können im Gegensatz zum B-Baum alle n in einem B^+-Baum gespeicherten Datensätze in einer Zeit von $O(n)$ sequentiell in Sortierordnung ausgegeben werden. Die inneren Knoten im B^+-Baum dienen ausschließlich dem Routing nach einem bestimmten Blatt, in dem ein Datensatz eingefügt oder gesucht werden soll, und müssen im Gegensatz zum B-Baum keine kompletten Suchschlüssel führen (siehe Präfix-B^+-Bäume, Seite 38).

Die Knoten eines Baumes enthalten eine größere Menge an Datenelementen, in denen die Suche erfolgt und die im allgemeinen am Stück in einem Block mit zusätzlichen Verwaltungsinformationen durch eine I/O-Operation zwischen externem Speichermedium und hauptspeicherinternem Puffer transferiert werden. Von diesen Datenelementen geht deshalb eine entsprechend hohe Zahl von Kanten zu Nachfolgerknoten aus. In diesem Zusammenhang spricht man von der *Ordnung* d eines B^+-Baumes, die besagt, daß jeder Knoten höchstens $2 \cdot d$ Einträge und $2 \cdot d + 1$ Zeiger auf Nachfolgerknoten enthalten darf[3] (siehe Seite 37, Punkt 3). Tatsächlich variiert die Anzahl der Datenelemente von Knoten zu Knoten, doch muß jeder Knoten mindestens d Datenelemente und entsprechend $d + 1$ Zeiger aufweisen. Folglich ist jeder Knoten zumindest zur Hälfte gefüllt.

Eine Suche in einem solchen Baum startet von der Wurzel und wird in Analogie zu Binärbäumen über Kanten in Nachfolgerknoten fortgesetzt, die mittels einer I/O-Operation als Block in den Hauptspeicher geladen werden, wenn sie nicht im Hauptspeicher gepuffert sind. Aufgrund der Tatsache, daß der B^+-Baum balanciert ist, wird die Suchkomplexität und der damit verbundene I/O-Aufwand minimiert. Der Vorteil des B^+-Baumes ist, daß nach jeder Einfügeoperation ein balancierter Zustand aufrechterhalten wird, d.h., daß jedes Blatt dieselbe Tiefe besitzt. So durchläuft der längste Pfad in einem Baum mit n Datenelementen $log_d n$ Knoten, wobei d die Ordnung des Baumes spezifiziert. Die Höhe des B^+-Baumes läßt sich um so geringer halten, je mehr Datenelemente in einem Knoten bzw. Block untergebracht werden können. Daher besteht z.B. eine einfache Maßnahme darin, in einem Block nicht den gesamten zu suchenden Datensatz,

[3] d wird auch als minimaler Verzweigungsgrad des B^+-Baumes bezeichnet.

sondern nur den Suchschlüssel für den Satz zusammen mit einem Verweis auf ihn unterzubringen. Wenn n die Gesamtzahl der Suchschlüssel und k die Zahl der Nachfolger pro Knoten ist, so ergibt sich der Aufwand für Blockzugriffe grob zu $O(log_k n)$. In Abbildung 3.7 wird ein B^+-Baum mit einer Ordnung und einer Tiefe von drei dargestellt.

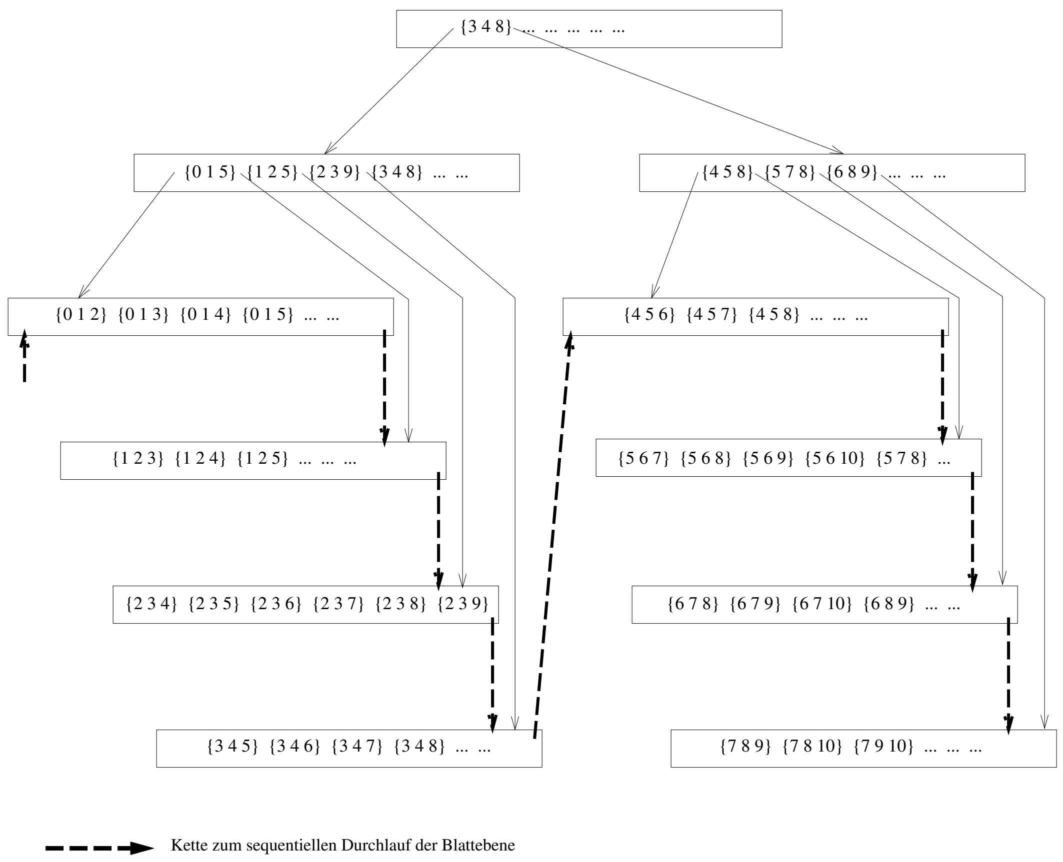


Abbildung 3.7: Ein B^+-Baum mit $d = 3$ und $h = 3$. Hier enthalten die Blätter die eigentlichen Verweise auf Itemsets bzw. die Itemsets selbst. Eine sequentielle Abarbeitung der untereinander verketteten Blätter realisiert die effiziente Verarbeitung der Itemsets in Sortierordnung in $O(n)$.

In [Lockemann et al., 1993] werden die folgenden Hauptmerkmale für B^+-Bäume angeführt:

- Der Kompromiß zwischen Zugriffs- und Änderungsaufwand zur Aufrechterhaltung der Balanciertheit des Baumes wird dadurch erzielt, daß die den Knoten des Baumes entsprechenden Blöcke nicht zwingend vollständig gefüllt werden. Vielmehr darf der Füllgrad zwischen 50 und 100% schwanken. Daher wirken sich Einfügeoperationen meist nicht ungünstig auf die Struktur des Baumes aus.

- Sequentielles Navigieren durch die in Sortierreihenfolge im Baum abgespeicherten Datensätze soll effizient unterstützt werden. Deshalb werden alle Schlüssel im Gegensatz zum B-Baum in den Blättern des Baumes geführt und die den Blättern entsprechenden Blöcke gemäß der über den Datensätzen definierten Sortierordnung verkettet. Daher ist ein sequentieller Durchlauf in $O(n)$ anstelle von $O(n \cdot log(n))$ möglich.

- Die Suchschlüssel in den Zwischenknoten sind insofern redundant, als sie solche in den Blättern wiederholen. Sie dienen daher ausschließlich dem Routing innerhalb des Baumes.

- Verweise auf Datensätze bzw. die Datensätze selbst werden ausschließlich in den Blättern geführt.

Damit läßt sich der B^+-Baum wie folgt definieren:

1. Jeder Pfad von der Wurzel zu einem Blatt hat dieselbe Länge h (Höhe des Baumes = Anzahl der gesehenen Knoten bzw. durchgeführten Blockzugriffe bei einer Suchoperation).

2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens $d+1$ und höchstens $2 \cdot d+1$ Nachfolger. Die Wurzel ist entweder selbst ein Blatt oder hat zwischen 2 und $2 \cdot d+1$ Nachfolger.

3. Jeder Knoten enthält eine Folge von Zeigern p_i und Suchschlüsseln s_j in der Anordnung $p_1, s_1, p_2, s_2, p_3, s_3, \ldots, p_n, s_n, p_{n+1}$ mit $d \leq n \leq 2 \cdot d$, wobei sich n auf die Anzahl der augenblicklich im Knoten geführten Suchschlüssel bezieht. Die Suchschlüssel s_j mit $1 \leq j \leq n$ seien in aufsteigender Reihenfolge sortiert. Die Zeiger p_j mit $1 \leq j \leq n$ haben in den Zwischenknoten und in den Blättern verschiedene Bedeutungen:

   (a) Für Zwischenknoten gilt: Die Zeiger verweisen auf die Nachfolgerknoten. Für alle Suchschlüssel x im Nachfolger p_1 gilt: $x \leq s_1$. Allgemein gilt für alle Schlüssel x im Nachfolger p_i für $1 \leq i \leq n$: $s_{i-1} < x \leq s_i$. p_{n+1} ist leer für alle Knoten auf derselben Blattebene außer dem letzten, für den gilt, daß alle Suchschlüssel x im Nachfolgerknoten zu p_{n+1} die Bedingung $s_n < x$ erfüllen.

   (b) Für Blätter gilt: p_i verweist auf den Datensatz (oder anstelle von p_i wird direkt auf den Datensatz zugegriffen) mit Suchschlüssel s_i für $1 \leq i \leq n$. Der Zeiger p_{2d+1} dient dem sequentiellen Navigieren, indem er auf das Blatt bzw. den Block verweist, der die in Sortierordnung direkt folgenden Datensätze enthält.

Die Suche nach einem Verweis bzw. Datensatz zu einem gegebenen Suchschlüssel s beginnt bei der Wurzel und folgt dort dem Zeiger p_i, für den entweder

1. $s \leq s_i$ für $i = 1$,
2. $s_{i-1} < s \leq s_i$ für $1 < i \leq n$ oder,
3. falls p_{n+1} existiert, $s_n < s$ gilt.

Diese Prozedur setzt sich solange rekursiv fort, bis ein Blatt erreicht wird, in dem abschließend nach einem übereinstimmenden Suchschlüssel gesucht wird. Wenn die Suche dort erfolglos abgebrochen wird, dann existiert kein Datensatz mit dem Suchschlüssel s.

3.4.1 Präfix-B^+-Bäume

Aufgrund der Tatsache, daß die inneren Knoten eines B^+-Baumes einzig und allein der Steuerung der Suche nach einem Blatt dienen, ist es nicht zwingend erforderlich, daß in den inneren Knoten möglicherweise relativ lange Suchschlüssel komplett abgespeichert werden, weil bei Suchvorgängen für das Routing innerhalb des Baumes oftmals die Betrachtung von Teilschlüsseln genügt. In [Bayer und Unterauer, 1977] werden daher Präfix-B^+-Bäume behandelt, die im Gegensatz zu einfachen B^+-Bäumen nur Teile der Suchschlüssel, nämlich Präfixe, in den inneren Knoten abspeichern. Abbildung 3.8 enthält ein einfaches Beispiel zur Illustration.

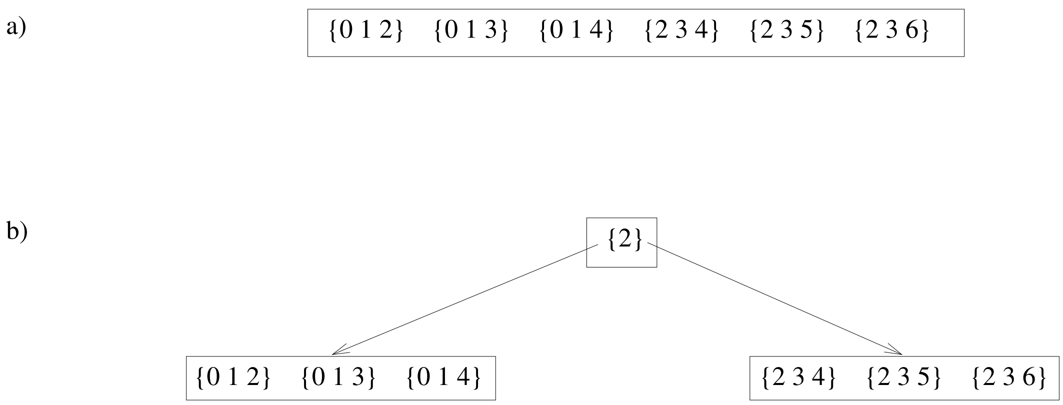


Abbildung 3.8: Um den Itemset { 2, 3, 4 } als Schlüssel mit dem zugehörigen Supportwert in das bereits volle Blatt in a) einzufügen, muß es wie in b) gesplittet werden. Der innere Knoten beinhaltet anstelle des kompletten Schlüssels lediglich ein Präfix davon.

Allgemein kann in dem Beispiel jeder Separator s, der zwei Knoten voneinander trennt, gewählt und in den inneren Knoten abgelegt werden, für den gilt:

$$\{0, 1, 4\} < s \leq \{3, 4, 5\}$$

Um Speicherplatz zu sparen, soll jedoch nach Möglichkeit einer der kürzesten Separatoren gewählt werden, denn dadurch kann erreicht werden, daß mehr Separatoren in den inneren Knoten bzw. in den dazugehörigen Blöcken gespeichert werden können. Dieses bewirkt gleichzeitig eine Erhöhung des Verzweigungsgrades der Knoten und eine damit verbundene Reduktion der Tiefe des Baumes. Folglich wird dadurch der Performanzflaschenhals bei Suchoperationen geweitet, weil weniger Knoten bzw. (externe) Blockzugriffe entlang der Suche von der Wurzel bis zu einem Blatt abgearbeitet werden.

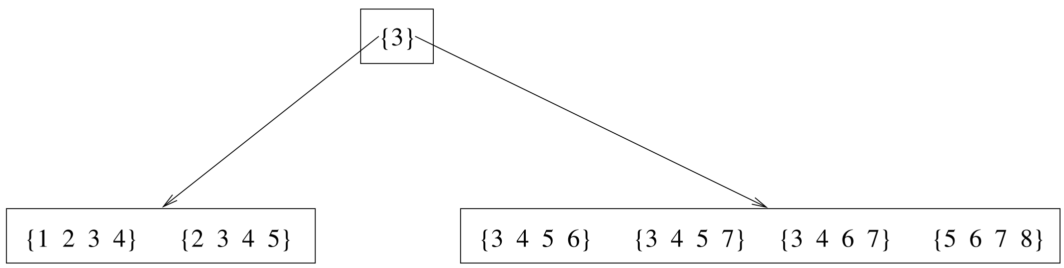
Weiterhin kann darauf verzichtet werden, daß ein Blatt oder ein innerer Knoten genau in der Mitte aufgeteilt wird. Stattdessen kann ein Intervall um den in der Mitte eines Knotens befindlichen Schlüssel gelegt werden, um darin nach einem geeigneten kürzesten Separator zu suchen. Diese Vorgehensweise ruft zwei entgegengesetzte Effekte hervor:

- Die Höhe des Präfix-B^+-Baumes neigt dazu abzunehmen.
- Die Speicherplatzausnutzung nimmt ab, weil mehr Knoten bzw. Blöcke auf der Blattebene und mehr aber dafür kürzere Einträge in den inneren Knoten gehalten werden müssen. Hier kann der Fall eintreten, daß Knoten bzw. Blöcke weniger als zur Hälfte gefüllt sind.

Es seien z.B. die folgenden Itemsets als Schlüssel in einem aufzuteilenden Blatt gespeichert:

$$\boxed{\{1, 2, 3, 4\}, \{2, 3, 4, 5\}, \{3, 4, 5, 6\}, \{3, 4, 5, 7\}, \{3, 4, 6, 7\}, \{5, 6, 7, 8\}}$$

Versucht man dieses Blatt in der Mitte zwischen $\{3, 4, 5, 6\}$ und $\{3, 4, 5, 7\}$ zu splitten, so erhält man $\{ 3, 4, 5, 7 \}$ als kürzesten Separator. Wird hingegen der Separator aus dem Intervall gewählt, das nicht wie oben die mittleren zwei sondern die mittleren vier Schlüssel enthält, so erhält man als kürzesten Separator 3, der ein Präfix des dritten Schlüssels $\{3, 4, 5, 6\}$ darstellt und nach oben in einen inneren Knoten wandert:



Kapitel 4

Implementierung

In diesem Kapitel wird auf die Implementierung des Apriori-Algorithmus unter dem Aspekt der Verarbeitung großer Datenmengen eingegangen. Zu diesem Zweck wird auf die in Kapitel 3 vorgestellten Datenstrukturen zur Verwaltung großer Datenmengen zurückgegriffen.

4.1 Ein Überblick

Der in Abbildung 4.1 dargestellte Apriori-Algorithmus kann in drei aufeinanderaufbauende Abschnitte gegliedert werden (vgl. Kapitel 2):

1. Vorverarbeitung und Bestimmung der häufig vorkommenden Items bzw. der Large 1-Itemsets (Zeile 01 - 08),

2. Bestimmung der Large k-Itemsets für $k \geq 2$ (Zeile 09 - 17) und

3. Generierung der Assoziationsregeln (Zeile 18)

Procedure <u>association_rule_mining</u> ($\mathcal{D}$, s_{min}, c_{min})
$\mathcal{D}$: Datenmenge von Transaktionen
s_{min}: minimaler Supportwert
c_{min}: minimaler Konfidenzwert

(01) $\mathcal{B}$:= $\emptyset$;
// Datei, in der die in Bitvektoren umgewandelten Transaktionen aus $\mathcal{D}$
// blockweise gespeichert werden
(02) assoctab := $\emptyset$;
// Assoziationstabelle für die zuordnende Abbildung: *Itemnummer* $\mapsto$ *Item*
// $assoctab(i)$ liefert das zur Itemnummer i gehörige Item
(03) preprocessing($\mathcal{D}$, $\mathcal{B}$, b_index, assoctab, s_{min});
(04) b_index := Integervektor mit den Integern $(0, \ldots, |\mathcal{B}| - 1)$;
// Blockindex zur Anzeige, ob ein Block b_i mit $i \in (0, \ldots, |\mathcal{B}| - 1)$
// betrachtet werden muß ((b_index(b_i) > 0), oder nicht ((b_index(b_i) $= 0$).
// Jedes Integer i enthält die Anzahl der noch zu betrachtenden Transaktionen in b_i.
(05) t_index := Bitvektor mit den Bits $(0, \ldots, |\mathcal{D}| - 1)$, wobei jedes Bit auf 1 gesetzt ist;
// Transaktionsindex zur Anzeige, ob eine Transaktion betrachtet werden muß
// (Bitnummer der Transaktion = 1), oder nicht (Bitnummer der Transaktion = 0)
// Am Anfang werden alle Bitnummern auf 1 gesetzt.
(06) $L_1 := \{\{i\} \mid support(assoctab(i)) \geq s_{min}\}$
(07) global_items := Bitvektor mit den Bits $(0, \ldots, |assoctab| - 1)$,
wobei jedes Bit auf 0 gesetzt ist;
// Jeder Itemnummer wird eine Bitnummer zugeordnet, die angibt, ob ein Item noch
// gültig ist, d.h. in den Large Itemsets vorkommt (Bit = 1), oder nicht (Bit = 0)
(08) **for all** {i} $\in$ L_1 **do** global_items(i) = 1;
// Die Bits der Itemnummern der Large 1-Itemsets werden auf 1 gesetzt.
(09) k := 2;
(10) **while** ($L_{k-1} \neq \emptyset$) **do begin**
(11) new_global_items := Bitvektor mit den Bits (0, ..., |assoctab|-1) ,
wobei jedes Bit auf 0 gesetzt wird;
// Jeder Itemnummer wird eine Bitnummer zugeordnet, die angibt, ob ein Item noch
// in den Large k-Itemsets vorkommt (dient der Aktualisierung von global_items).
(12) C_k := generate_candidates(L_{k-1}, $\mathcal{B}$, b_index, t_index,
global_items, new_global_items);
// Generierung der Hypothesenmenge der Candidate k-Itemsets
(13) determine_support_values($\mathcal{B}$, b_index, t_index, C_k, global_items);
(14) L_k := find_large_itemsets(s_{min}, C_k, new_global_items);
// Herausfiltern der Large k-Itemsets
(15) global_items := new_global_items;
(16) k++;
(17) **end**;
(18) rule_discovery(c_{min}, $\bigcup_{i=1}^{k-1} L_i$);

Abbildung 4.1: Der Apriori-Algorithmus.

4.2 Die Vorverarbeitung

```
Procedure preprocessing (𝒟, ℬ, b_index, assoctab, s_min)

𝒟: Datenmenge von Transaktionen
ℬ: Menge von Blöcken
b_index: Blockindex
assoctab: Assoziationstabelle zur Zuordnung der Items auf Itemnummern
s_min: minimaler Supportwert

  (01) ℐ := ∅;
  (02) while not eof(𝒟) do begin
  (03)    lese t ∈ 𝒟;
  (04)    for all item ∈ t do begin
  (05)       support(item)++;
  (06)       ℐ := ℐ ∪ item;
  (07)    end;
  (08) end;
  (09) n := |ℐ|;
  (10) sortiere ℐ nach den Supportwerten der darin enthaltenen Items;
  (11) i := 0;
  (12) while i < n do begin
  (13)    assoctab := assoctab ∪ (i, i.tes Item aus ℐ);
          // Zuordnung von Itemnummern für Items wird in einer
          // Assoziationstabelle eingetragen. Das am häufigsten
          // vorkommende Item erhält die Nummer 0, das nächste 1, usw.
  (14)    i++;
  (15) end;
  (16) i := 0;
  (17) reserviere Block b_i;
  (18) b_index(i) := 0;
  (19) while not eof(𝒟) do begin
  (20)    lese t ∈ 𝒟;
  (21)    ersetze in t jedes Item durch die zugehörige Itemnummer;
  (22)    sortiere t;
  (23)    transformiere t in eine Bitvektor-Repräsentation t';
  (24)    schreibe t' in den Block i;
  (25)    b_index(i)++;
  (26)    if (Block b_i ist voll) then do begin
  (27)       schreibe Block b_i nach ℬ;
  (28)       i++;
  (29)       reserviere Block b_i;
  (30)       b_index(i) := 0;
  (31)    end;
  (32) end;
```

Abbildung 4.2: Die Prozedur zum Preprocessing der Datenmenge $\mathcal{D}$.

Innerhalb der Vorverarbeitung werden verschiedene Arbeiten erledigt, um die Anwendung des Apriori-Algorithmus auf großen Datenmengen zu ermöglichen. Diese Vorarbeiten bestehen aus

- dem Preprocessing der Datenmenge $\mathcal{D}$ (siehe Abbildung 4.2)
  - zur Angabe einer eindeutigen Abbildung aller Items in der Datenmenge $\mathcal{D}$ auf Itemnummern (vgl. Abb. 4.2, Zeile 01–15) und

– zur Umwandlung aller Transaktionen in Bitvektoren, die blockweise in die Datei $\mathcal{B}$ geschrieben werden (vgl. Abb. 4.2, Zeile 16–32) .

- der Bestimmung der Large 1-Itemsets (vgl. Abb. 4.1, Zeile 06),
- der Erstellung einer globalen Liste zur Aufnahme aller in den Large Itemsets vorkommenden Itemnummern (vgl. Abb. 4.1, Zeile 07–08) und
- der Errichtung eines Indexes sowohl für die Blöcke in $\mathcal{B}$ als auch für die darin enthaltenen Transaktionen (vgl. Abb. 4.1, Zeile 05 und Abb. 4.2, Zeile 18, 25 und 30).

4.2.1 Repräsentationswechsel

Zunächst werden die Häufigkeiten der Items in $\mathcal{D}$ ermittelt (vgl. Abb. 4.2, Zeile 01–08). Dann wird eine sogenannte Assoziationstabelle aufgebaut (vgl. Abb. 4.2, Zeile 11–15), in der die Items absteigend nach den Häufigkeiten sortiert werden (vgl. Abb. 4.2, Zeile 09–10). Die in $\mathcal{D}$ am häufigsten erscheinenden Items stehen in dieser Liste ganz oben, während die am geringsten vorkommenden Items am Ende der Liste zu finden sind. Für das Beispiel aus Kapitel 2 (vgl. 2.1) hat die Assoziationstabelle folgende Gestalt:

Itemnummer	Item	Support
0	Cola	5
1	Saft	4
2	Bier	4
3	Brot	2
4	Wasser	2
5	Schokolade	2
6	Wein	2
7	Chips	1
8	Schinken	1

Abbildung 4.3: Die Assoziationstabelle für das Eingangsbeispiel aus Kapitel 2.

Jedem Item wird jetzt die Positionsnummer, unter der das Item in der Assoziationstabelle abgelegt ist, als eindeutige Itemnummer zugeordnet. Das Item *Bier* erhält z.B. die Itemnummer 0. Nun wird in der Datenmenge $\mathcal{D}$ in jeder Transaktion t jedes vorkommende Item durch die korrespondierende Itemnummer ersetzt (vgl. Abb. 4.2, Zeile 21). Die Transaktion t_7 mit $\{Schokolade, Schinken, Brot\}$ wird z.B. durch die Itemnummern 3, 5 und 8 dargestellt.

Als nächstes wird jede Transaktion durch einen Bitvektor repräsentiert, indem jede Itemnummer als Bitnummer interpretiert wird (vgl. Abb. 4.2, Zeile 23). Die Bitvektor-Transaktion wird in die Datei $\mathcal{B}$ geschrieben und der entsprechende Indexeintrag für den jeweiligen Block erhöht(vgl. Abb. 4.2, Zeile 25–31). Ist ein bestimmtes Item in der Transaktion vorhanden, so wird das Bit, dessen Bitnummer mit der entsprechenden Itemnummer korrespondiert, auf 1, ansonsten auf 0 gesetzt. Da z.B. in der Transaktion t_7 die Items mit den Itemnummern 3, 5 und 8 präsent sind, werden in der Bitvektor-Repräsentation t'_7 von t_7 die Bits 3, 5 und 8 der insgesamt 9 Bits für 9 Items auf 1 gesetzt:

Bitnummer	0	1	2	3	4	5	6	7	8
t'_7	0	0	0	1	0	1	0	0	1

Die Sortierung der Items nach den Häufigkeiten sorgt dafür, daß die Bits der häufigen Items kompakt nebeneinander gespeichert werden.

Enthält die Datenmenge $\mathcal{D}$ insgesamt m Items und können insgesamt n Bits zur Anzeige, ob ein bestimmtes Item in der jeweiligen Transaktion vorhanden ist oder nicht, am Stück in einem Speicherwort gespeichert werden, so können beispielsweise zwei Verfahren zum Aufbau der Speicherwörter zur Bitvektor-Repräsentation für Transaktionen, die sich aus Item- bzw. Bitnummern zusammensetzen, eingesetzt werden:

1. Die zu den Itemnummern in der Datenmenge $\mathcal{D}$ korrespondierenden Bitnummern werden kontinuierlich und fest in Speicherwörter der Länge n eingeteilt. Jede Transaktion aus $\mathcal{D}$ kann durch einen Bitvektor repräsentiert werden, der aus $\lceil \frac{m}{n} \rceil$ durchnumerierten Speicherwörtern besteht, deren Bits anfänglich den Wert 0 haben. Jedes der ab 0 durchnumerierten Speicherwörter s_i enthält dabei die Bits $i, \ldots, (i \cdot n) - 1$. So enthält Speicherwort s_0 die Bits $0, \ldots, n-1$, Speicherwort s_1 die Bits $n, \ldots, 2n-1$, Speicherwort s_2 die Bits $2n, \ldots, 3n-1$, usw.

   Die Bits innerhalb der Speicherwörter werden gesetzt, indem jede Item- bzw. Bitnummer einer beliebigen Transaktion t_i durch n geteilt wird, um mit dem ganzzahligen Ergebnis der Division das Speicherwort zu selektieren, in dem das entsprechende Bit gesetzt wird, dessen Nummer sich aus dem Rest der vorangegangenen Division ergibt. Jedes der $\lceil \frac{m}{n} \rceil$ durchnumerierten Speicherwörter, in dem mindestens ein Bit gesetzt ist, wird zur Bitvektor-Repräsentation von t_i abgespeichert.

2. Wie in 1., aber innerhalb des Bitvektors jeder Transaktion werden Speicherwörter derart verschoben, daß sie insgesamt alle gesetzten Bits überdecken und sich dabei nicht gegenseitig überlagern.

Die Bitvektor-Repräsentation der Transaktion mit den Itemnummern 0, 2, 11, 12, 13 und 14 wird beispielsweise bei einer angenommenen Speicherwortlänge von vier Bits wie folgt mit Hilfe der beiden oben angesprochenen Verfahren in Speicherwörter, die hier unterstrichen dargestellt werden, abgelegt:

Itemnummer	0 15
Bitmuster nach dem 1. Verfahren	$\underline{1\ 0\ 1\ 0}\ 0\ 0\ 0\ 0\ \underline{0\ 0\ 0\ 1}\ \underline{1\ 1\ 1\ 0}$
Bitmuster nach dem 2. Verfahren	$\underline{1\ 0\ 1\ 0}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ \underline{1\ 1\ 1\ 1}\ 0$

Der Vorteil der Anwendung des zweiten Verfahrens gegenüber dem ersten ist, daß weniger Speicherwörter für die Bitvektor-Repräsentation der Transaktionen abgespeichert werden müssen. Die Anwendung des ersten Verfahrens hat allerdings den entscheidenden Vorteil der effizienteren Verknüpfung der nach diesem Verfahren abgespeicherten Bitvektoren untereinander. So können beispielsweise zwei Bitvektoren logisch verknüpft werden, indem jeweils die Speicherwörter beider Bitvektoren mit derselben Numerierung direkt bitweise verknüpft werden. Dieses ist bei den nach dem zweiten Verfahren abgespeicherten Bitvektoren nicht so einfach möglich, denn aufgrund der variablen Verschiebung der Speicherwörter sind Bits mit derselben Bitnummer innerhalb verschiedener Bitvektoren im allgemeinen nicht auf dieselbe Position innerhalb der jeweiligen Speicherwörter ausgerichtet. Um dennoch eine logische Verknüpfung zu ermöglichen, müssen die betreffenden Bits derart verschoben werden, daß sie „untereinanderstehen“.

Die als Bitvektor repräsentierte Transaktion t'_7 wird dann nach dem 1. Verfahren wie folgt in zwei Speicherwörter mit jeweils 8 Bits abgespeichert:

Speicherwort	Bit 0, ..., Bit 8
s_0	0001 0100
s_1	1000 0000

In Abbildung 4.4 wird der Repräsentationswechsel der Datenmenge $\mathcal{D}$ aus Abbildung 2.1 (siehe Kapitel 2) gemäß der in Abbildung 4.3 aufgestellten Assoziationstabelle von der Item- über die Itemnummer- zur Bitvektor-Repräsentation dargestellt.

Transaktion	in $\mathcal{D}$	mit Itemnummern	in $\mathcal{B}$
t_0	Saft, Cola, Bier	0, 1, 2	s_0: 1110 0000
t_1	Saft, Cola, Wein	0, 1, 6	s_0: 1100 0010
t_2	Saft, Wasser	1, 4	s_0: 0100 1000
t_3	Cola, Bier	0, 2	s_0: 1010 0000
t_4	Saft, Cola, Bier, Wein	0, 1, 2, 6	s_0: 1110 0010
t_5	Wasser	4	s_0: 0000 1000
t_6	Schokolade, Cola, Chips	0, 5, 7	s_0: 1000 0101
t_7	Schokolade, Schinken, Brot	3, 5, 8	s_0: 0001 0100
			s_1: 1000 0000
t_8	Brot, Bier	2, 3	s_0: 0011 0000

Abbildung 4.4: Repräsentationswechsel der Datenmenge $\mathcal{D}$ zur Datei $\mathcal{B}$, in der alle Transaktionen als Bitvektoren abgelegt sind, wobei in einem Speicherwort s_i 8 Bits einer als Bitvektor repräsentierten Transaktion abgespeichert sind.

4.2.2 Aufbau einer globalen Itemliste

Die globale Itemliste *global_items* ist eine als Bitvektor repräsentierte Liste, die zu jedem Item in der Datenmenge $\mathcal{D}$ ein Bit enthält. Für die relevanten Items, die in den Large Itemsets vorkommen, wird das entsprechende Bit auf 1, ansonsten auf 0 gesetzt. Der Bitvektor wird ebenfalls nach dem 1. Verfahren, das zur Umrepräsentation von Transaktionen in Bitvektoren verwendet wurde, abgespeichert. Deshalb kann jede Transaktion leicht mit der globalen Itemliste logisch verknüpft werden, indem jeweils die Speicherwörter der beiden entsprechenden Bitvektoren mit derselben Numerierung direkt miteinander verknüpft werden. Die globale Itemliste wird zu Anfang mit den Large 1-Itemsets initialisiert und wird dann nach jeder Iteration des Algorithmus aktualisiert.

Eine logische „und"-Verknüpfung zwischen einer Transaktion und der globalen Itemliste hat beispielsweise zur Folge, daß all diejenigen Items aus der Transaktion herausgefiltert werden, die nicht in den Large Itemsets der vorherigen Iteration präsent sind. Diese Verknüpfung wird bei der Ermittlung der Supportwerte der Candidate Itemsets beim Scannen der Transaktionen eingesetzt (siehe Seite 55).

Für das Eingangsbeispiel aus Kapitel 2 (siehe Abb. 2.1, $s_{min} = 2$) hat die als Bitvektor repräsentierte globale Itemliste *global_items* gemäß der in Abbildung 4.3 dargestellten Assoziationstabelle folgende Gestalt:

	Globale Itemliste *global_items*								
Bitnummer	0	1	2	3	4	5	6	7	8
Zustand	1	1	1	1	1	1	1	0	0

Die globale Itemliste *global_items* kann in zwei 8 Bit lange Speicherwörter $s_0 = 11111110$ und $s_1 = 00000000$ abgespeichert werden.

4.2.3 Einsatz von Indizes

Zur effizienten Verwaltung der zu analysierenden Transaktionen werden zwei Indizes verwendet. Zum einen wird ein Index für die Blöcke in $\mathcal{B}$ verwendet, um nur diejenigen Blöcke einzulesen und zu verarbeiten, in denen Transaktionen enthalten sind, die potentielle Candidate Itemsets enthalten. Damit genau diese Transaktionen angesteuert werden können, wird zum anderen ein weiterer Index für Transaktionen eingesetzt.

In der in [Park et al., 1995] beschriebenen Implementierung zur Entdeckung von Assoziationsregeln werden in jeder Iteration diejenigen Transaktionen aus der Datenmenge $\mathcal{D}$ gelöscht, in denen keine Candidate Itemsets mehr enthalten sind. Darüberhinaus werden Transaktionen um diejenigen Items gekürzt, die in keinem der darin enthaltenen Candidate Itemsets vorkommen. Aufgrund der kontinuierlichen Verkleinerung der Datenmenge $\mathcal{D}$ wird eine stetige Verringerung des lesenden I/O-Aufwandes erreicht. Gleichzeitig sinken die Kosten für die Supportwertermittlung der Candidate Itemsets, da sich die Menge der zu betrachtenden Daten aus $\mathcal{D}$ verringert.

In der hier beschriebenen Implementierung sorgt das Zusammenspiel der beiden Indizes, die zur Verwaltung der Blöcke und der darin enthaltenen Transaktionen eingesetzt werden, für eine Reduzierung des lesenden I/O-Aufwandes, um eine Kostensenkung innerhalb der Supportwertermittlung der Candidate Itemsets zu erzielen. Im Gegensatz zu [Park et al., 1995] ist kein schreibender I/O-Zugriff zur Modifikation der als Datei $\mathcal{B}$ repräsentierten Datenmenge $\mathcal{D}$ vorgesehen, der gerade bei größeren Datenmengen hohe I/O-Zugriffskosten verursacht. Der Blockindex *b_index* sorgt dafür, daß nur diejenigen Blöcke eingelesen werden, die noch relevante Transaktionen für die Supportwertermittlung der Candidate k-Itemsets enthalten. Sobald ein Block eingelesen wird, wird der Transaktionsindex *t_index* eingesetzt, um innerhalb des Blockes nur die relevanten Transaktionen anzusteuern. Die anderen werden nicht betrachtet.

4.2.3.1 Index für Blöcke

Die in der Datei $\mathcal{B}$ enthaltenen Blöcke werden mit Hilfe eines Blockindexes *b_index* verwaltet, der zu jedem Block b_i mit $i \in \{0, \ldots, |\mathcal{B}| - 1\}$ die Anzahl $|b_i|$ der darin befindlichen und bei der Supportwertermittlung der Candidate Itemsets zu betrachtenden Transaktionen angibt, wobei sich $|\mathcal{B}|$ auf die Anzahl der Blöcke in $\mathcal{B}$ bezieht:

	Blockindex *b_index*						
Blocknummer	0	1	...	...	...	$\|\mathcal{B}\| - 2$	$\|\mathcal{B}\| - 1$
Anzahl Transaktionen	$\|b_0\|$	$\|b_1\|$	...	...	...	$\|b_{\|\mathcal{B}\|-2}\|$	$\|b_{\|\mathcal{B}\|-1}\|$

Zu Beginn wird jeder Indexeintrag mit der Anzahl der in dem jeweiligen Block gespeicherten Transaktionen initialisiert. Die Werte unter den Indexeinträgen werden dekrementiert, wenn Transaktionen in den jeweiligen Blöcken nicht mehr weiter bei der Supportwertermittlung der Candidate Itemsets berücksichtigt werden müssen. Diejenigen Blöcke, deren zugehörige Indexeinträge den Wert 0 erreichen, werden in der nächsten Iteration des Algorithmus nicht mehr zur Bearbeitung in den Hauptspeicher geladen.

4.2.3.2 Index für Transaktionen

Zur Verwaltung der Transaktionen innerhalb der Blöcke in $\mathcal{B}$ wird zusätzlich ein Index *t_index* (Transaktionsindex) verwendet, der zu jeder Transaktion einen Eintrag enthält. Sind in der ursprünglichen Datenmenge $\mathcal{D}$ insgesamt n Transaktionen enthalten, so besteht der als Bitvektor repräsentierte Index aus n Bits:

	Transaktionsindex *t_index*						
Bitnummer	0	1	...	...	...	$\|\mathcal{D}\| - 2$	$\|\mathcal{D}\| - 1$
Transaktion	t_0	t_1	...	...	...	$t_{\|\mathcal{D}\|-2}$	$t_{\|\mathcal{D}\|-1}$

Diejenigen Transaktionen, deren zugehörige Bits im Index gesetzt sind, werden bei der Ermittlung der Supportwerte der Candidate Itemsets berücksichtigt, während die anderen Transaktionen nicht weiter untersucht werden. Bei der Initialisierung des Indexes, der nach dem 1. Verfahren aufgebaut worden ist (s.o.), werden alle Bits des Bitvektors auf 1 gesetzt. Im Laufe des Algorithmus können einzelne Bits auf 0 gesetzt werden, wenn die betreffenden Transaktionen keine oder keine weiteren Candidate Itemsets mehr enthalten. Zusätzlich werden die Indexeinträge der Blöcke, in denen diese Transaktionen enthalten sind, entsprechend dekrementiert.

4.3 Bestimmung und Verwaltung der Large Itemsets

Aus den Large 1-Itemsets wird die Hypothesenmenge der Candidate 2-Itemsets gebildet, deren Häufigkeiten anschließend in der Datenmenge $\mathcal{D}$, die durch die Datei $\mathcal{B}$ repräsentiert wird, ermittelt werden. Danach werden die Large 2-Itemsets aus der Hypothesenmenge herausgefiltert und dienen als Basis zur Generierung der Candidate 3-Itemsets. Es folgen wiederum die Schritte zur Ermittlung der Supportwerte und zum Herausfiltern der Large 3-Itemsets. Dieser Vorgang wiederholt sich n-mal, wenn aus den Large $(n-1)$-Itemsets noch Candidate n-Itemsets generiert werden können.

Wenn eine Itemnummer γ Bytes an Speicherplatz benötigt, dann nimmt ein k-Itemset $k \cdot \gamma$ Bytes in Anspruch. Zusätzlich wird jedem k-Itemset ein Supportzähler, der δ Bytes benötigt, zugeordnet. Somit muß für einen Candidate k-Itemset c_k in der Hypothesenmenge C_k ein Speicherplatz von

$$k \cdot \gamma + \delta$$

Bytes reserviert werden. Vor jeder Iteration des Algorithmus stellt sich die Frage, ob die Menge der neu zu generierenden Kandidaten komplett im zur Verfügung stehenden Hauptspeicher von $\mathcal{M}$ Bytes verwaltet werden kann oder nicht.

4.3.1 Die Kandidatengenerierung

Die Datenmenge $\mathcal{D}$ enthalte m Items, die durch die Itemnummern $i_1, \ldots, i_m$ repräsentiert werden. Ferner wird durch die „$<$"-Relation eine Ordnung über die Itemnummern definiert. Es gilt:

$$i_1 < i_2 < \ldots < i_{m-1} < i_m$$

Jeder Itemset wird durch die den Items entsprechenden Itemnummern repräsentiert, die in aufsteigend sortierter Ordnung abgespeichert werden. Folglich wird durch die „$<$"Relation über die Itemnummern eine Ordnung über die Itemsets induziert.

Definition 12 *Es seien zwei k-Itemsets, $is_k^1 = \{i_1^1, i_2^1, \ldots, i_{k-1}^1, i_k^1\}$ und $is_k^2 = \{i_1^2, i_2^2, \ldots, i_{k-1}^2, i_k^2\}$, gegeben. Dann ist is_k^1 kleiner als is_k^2, wenn gilt*

$$\begin{aligned} is_k^1 < is_k^2 \quad \Leftrightarrow \quad & i_1^1 < i_1^2 \\ & \vee\ i_1^1 = i_1^2 \wedge i_2^1 < i_2^2 \\ & \vdots \\ & \vee\ i_1^1 = i_1^2 \wedge \ldots \wedge i_{k-1}^1 = i_{k-1}^2 \wedge i_k^1 < i_k^2 \ . \end{aligned}$$

Bei der Generierung der Hypothesenmenge C_k der Candidate k-Itemsets c_k lassen sich für k drei Fälle unterscheiden:

1. $k = 1$: Die Menge der Candidate 1-Itemsets ist identisch mit der Menge der Itemnummern in der Datenmenge $\mathcal{D}$. Es wird vorausgesetzt, daß diese Menge vollständig im verfügbaren Hauptspeicher gehalten werden kann.

2. $k = 2$: Die Menge der Candidate 2-Itemsets entsteht dadurch, daß der Reihe nach jeweils zwei Large 1-Itemsets, $li_1^1 = \{\, i_1^1 \,\}$ und $li_1^2 = \{\, i_1^2 \,\}$ mit $li_1 < li_2$, zu einem Candidate 2-Itemset $ci_2 = \{\, i_1^1, i_1^2 \,\}$ zusammengefaßt werden. Wenn die Anzahl der Large 1-Itemsets n beträgt, dann besteht die Hypothesenmenge C_2 aus
$$\binom{n}{2}$$
Candidate 2-Itemsets, die in der aufsteigend sortierten Folge
$$\{0,1\}, \{0,2\}, \ldots, \{n-2, n\}, \{n-1, n\}$$
erzeugt und in C_2 eingetragen werden und für die allein ein Speicherbedarf von
$$\binom{n}{2} \cdot (2\gamma + \delta)$$
Bytes besteht.

3. $k \geq 3$: Ein Candidate k-Itemset ci_k wird gebildet, indem der Reihe nach jeweils zwei Large $(k-1)$-Itemsets, $li_{k-1}^1 = \{i_1^1, \ldots, i_{k-1}^1\}$ und $li_{k-1}^2 = \{i_1^2, \ldots, i_{k-1}^2\}$, vereinigt werden, so daß gilt (vgl. Abb. 4.5):

   (a) $i_1^1 = i_1^2, i_2^1 = i_2^2, \ldots, i_{k-2}^1 = i_{k-2}^2$ und $i_{k-1}^1 < i_{k-1}^2$

   (b) $ci_k = \{i_1^1, i_2^1, \ldots, i_{k-1}^1, i_{k-1}^2\}$

   (c) $\forall j \in \{1, \ldots, k-2\} : \ \{i_1^1, i_2^1, \ldots, i_{k-1}^1, i_{k-1}^2\} \ \setminus \ i_j^1) \in L_{k-1}$

Aus den Punkten a) und b) geht hervor, daß neue Candidate k-Itemsets aus den ersten $k-2$ übereinstimmenden und den übrigen beiden unterschiedlichen Itemnummern zweier Large $(k-1)$-Itemsets gebildet werden. Punkt c) stellt sicher, daß gemäß Satz 1 neben den beiden Large $(k-1)$-Itemsets, li^1_{k-1} und li^2_{k-1}, alle übrigen $(k-1)$-elementigen Teilmengen von ci_k in L_{k-1}, der Menge der Large $(k-1)$-Itemsets, enthalten sein müssen. Die Candidate k-Itemsets werden ebenfalls in einer aufsteigend sortierten Folge erzeugt und in C_k eingetragen. Für x Candidate k-Itemsets muß ein Speicherbereich von

$$x \cdot (k \cdot \gamma + \delta)$$

Bytes reserviert werden.

Damit für $k \geq 2$ die Hypothesenmenge C_k von x Candidate k-Itemsets mittels des in Kapitel 3 vorgestellten internen Hashings verwaltet werden kann, muß zusätzlich eine hinreichend große Hashtabelle, die maximal zu α Prozent[1] gefüllt sein darf, bereitgestellt werden. Die Größe der Hashtabelle ergibt sich somit zu

$$x \cdot \frac{1}{\alpha} \ .$$

Ferner werden für jeden Hashtabelleneintrag β Bytes reserviert, um zwei Verweise unterzubringen. Während ein Verweis auf einen Candidate k-Itemset ci_k in C_k zeigt, dient der andere gemäß der in Kapitel 3 dargestellten Erweiterung zum internen Hashing zur Verkettung der Candidate k-Itemsets in Einfügereihenfolge und zeigt deshalb auf denjenigen Hashtabelleneintrag, in dem sich der Verweis auf den Candidate k-Itemset befindet, der nach ci_k generiert worden ist. Daraus folgt, daß für x Candidate k-Itemsets ein zusätzlicher Speicherbedarf von

$$x \cdot \frac{1}{\alpha} \cdot \beta$$

Bytes besteht. Wenn die Beziehung

$$(\underbrace{x \cdot (k \cdot \gamma + \delta)}_{Hauptspeicher} + \underbrace{x \cdot \frac{1}{\alpha} \cdot \beta}_{Hashtabelle}) \leq \mathcal{M}$$

gilt, dann kann die Hypothesenmenge C_k komplett im verfügbaren Hauptspeicher mittels internem Hashing verarbeitet werden, ansonsten bieten sich zwei Implementierungsalternativen an:

1. Die Hypothesenmenge C_k wird komplett erzeugt und mit Hilfe von externem Hashing (siehe Kapitel 3, Seite 65) verwaltet, damit die Supportwerte der Candidate k-Itemsets durch ein einmaliges Scannen der Bitvektor-Transaktionen in $\mathcal{B}$ bestimmt werden können. Der gesamte zur Verfügung

[1] Aus Performanzgründen ist darauf zu achten, daß der Auslastungsfaktor α der Hashtabelle nicht mehr als 80% beträgt [Knuth, 1973].

stehende Hauptspeicher dient dabei als Puffer zur Ein- und Auslagerung von Blöcken der externen Hashdatei.

2. Die Hypothesenmenge C_k wird Stück für Stück erzeugt, so daß jeweils ein Teilstück komplett im verfügbaren Hauptspeicher mit Hilfe von internem Hashing verarbeitet werden kann, um die Supportwerte der darin enthaltenen Candidate k-Itemsets durch ein Scannen der Bitvektor-Transaktionen in $\mathcal{B}$ zu ermitteln (siehe Abb. 4.5, Zeile 08–12). Nachdem die Transaktionen einmal zur Bestimmung der Supportwerte der in dem Teilstück befindlichen Candidate k-Itemsets gescannt worden sind, werden alle in dem jeweiligen Teilstück befindlichen Large k-Itemsets zusammen mit ihren Supportwerten in einen Präfix-B^+-Baum (siehe Kapitel 3, Seite 35) eingetragen.

```
Procedure generate_candidates (L_{k-1}, B, b_index, t_index, global_items, new_global_items)

L_{k-1}: Menge der (k-1)-Itemsets
B: Menge von Blöcken
b_index: Blockindex
t_index: Transaktionsindex
global_items: globale Itemliste (Bitvektor)
new_global_items: aktualisierte globale Itemliste (Bitvektor)

(01) C_k = ∅;
(02) for all li_{k-1} ∈ L_{k-1} do begin
(03)    for all li'_{k-1} ∈ L_{k-1} mit li'_{k-1} > li_{k-1} do begin
(04)       if (|li_{k-1} ∩ li'_{k-1}| = k - 2) then do begin
(05)          ci_k := li_{k-1} ∪ li'_{k-1};
(06)          if (∀is_{k-1} ⊂ ci_k : is_{k-1}) ∈ L_{k-1}) then do begin
(07)             C_k := C_k ∪ ci_k;
(08)             if ((|C_k| ≥ MAIN_MEMORY_LIMIT) ∧ NOT_EXTERN) then do begin
                    // MAIN_MEMORY_LIMIT spezifiziert die verfügbare
                    // Hauptspeichermenge zur Speicherung der Candidate k-Itemsets.
                    // NOT_EXTERN gibt an, ob im Falle eines Speicherengpaßes
                    // die die Hypothesenmenge C_k komplett erzeugt und extern
                    // verwaltet werden soll oder nicht.
(09)                determine_support_values(B, db_index, global_items);
(10)                L_k := find_large_itemsets(C_k, new_global_items);
(11)                C_k := ∅;
(12)             end;
(13)          end;
(14)       end;
(15)    end;
(16) end;
(17) return C_k;
```

Abbildung 4.5: Die Prozedur zur Generierung der Candidate Itemsets.

Um zu Beginn bei n Large 1-Itemsets die daraus resultierende Hypothesenmenge der Candidate 2-Itemsets mit Hilfe von internem Hashing komplett im verfügbaren Hauptspeicher zu verwalten, muß die Bedingung

$$(\binom{n}{2} \cdot (2 \cdot \gamma + \delta) + \binom{n}{2} \cdot \frac{1}{\alpha} \cdot \beta) \leq \mathcal{M}$$

gelten. Mit Hilfe der von

$$\binom{n}{2} = \frac{n \cdot (n-1)}{2}$$

kann die obige Bedingung nach n aufgelöst werden. Man erhält dann mit

$$n < \sqrt{\frac{2\mathcal{M}}{(2\gamma + \delta) + \frac{\beta}{\alpha}} + \frac{1}{4}} + \frac{1}{2}$$

eine Bedingung, die angibt, wie groß die Menge der Large 1-Itemsets maximal sein darf, damit die Hypothesenmenge C_2 der Candidate 2-Itemsets komplett im verfügbaren Hauptspeicher mit Hilfe von internem Hashing verarbeitet werden kann. Falls die Hypothesenmenge C_2 komplett im Hauptspeicher verarbeitet werden konnte, können nach dem Scannen der Transaktionen und dem Herausfiltern der Large 2-Itemsets die Candidate 3-Itemsets direkt aus der eingesetzten Hashtabelle erzeugt werden, ansonsten werden sie aus einem Präfix-B^+-Baum generiert, in dem die Large 2-Itemsets nach der Verarbeitung von C_2 abgespeichert worden sind. Somit werden für $k \geq 3$ die Candidate k-Itemsets entweder hauptspeicher- oder externspeicherbasiert erzeugt.

4.3.1.1 Hauptspeicherbasierte Generierung

Bei der hauptspeicherbasierten Kandidatengenerierung wird vorausgesetzt, daß die Hypothesenmenge C_{k-1} mit Hilfe einer internen Hashtabelle vollständig im Hauptspeicher verwaltet wird. Jede belegte Hashtabellenposition der internen Hashtabelle enthält einen Verweis $v^i_{itemset}$ auf einen Candidate $(k-1)$-Itemset ci_{k-1} und einen Verweis v^i_{next} auf die Hashtabellenposition, in der sich der Verweis $v^{i+1}_{itemset}$ auf denjenigen Candidate $(k-1)$-Itemset befindet, der gemäß Sortierordnung nach ci_{k-1} erzeugt worden ist. Die Verkettung der belegten Hashtabellenpositionen mit Hilfe der Verweise v^i_{next} $(1 \leq i \leq |C_{k-1}|)$ ermöglicht somit eine effiziente sequentielle Verarbeitung der Candidate $(k-1)$-Itemset in Sortierreihenfolge in $O(|C_{k-1}|)$, die unmittelbar bei der Generierung der Candidate k-Itemsets ausgenutzt wird.

Nachdem die Supportwerte der Candidate $(k-1)$-Itemsets ermittelt worden sind, können die Candidate k-Itemsets direkt aus der internen Hashtabelle generiert werden. Die Large $(k-1)$-Itemsets und die neu generierten Candidate k-Itemsets werden zunächst jeweils in hauptspeicherbasierten Listen eingefügt. Sobald die Kapazität des verfügbaren Hauptspeichers ausgeschöpft ist, werden die bis dahin generierten Candidate k-Itemsets und alle im Hauptspeicher befindlichen Large Itemsets in externe Dateien ausgelagert, um im Hauptspeicher Platz für die weitere Kandidatengenerierung zu schaffen.

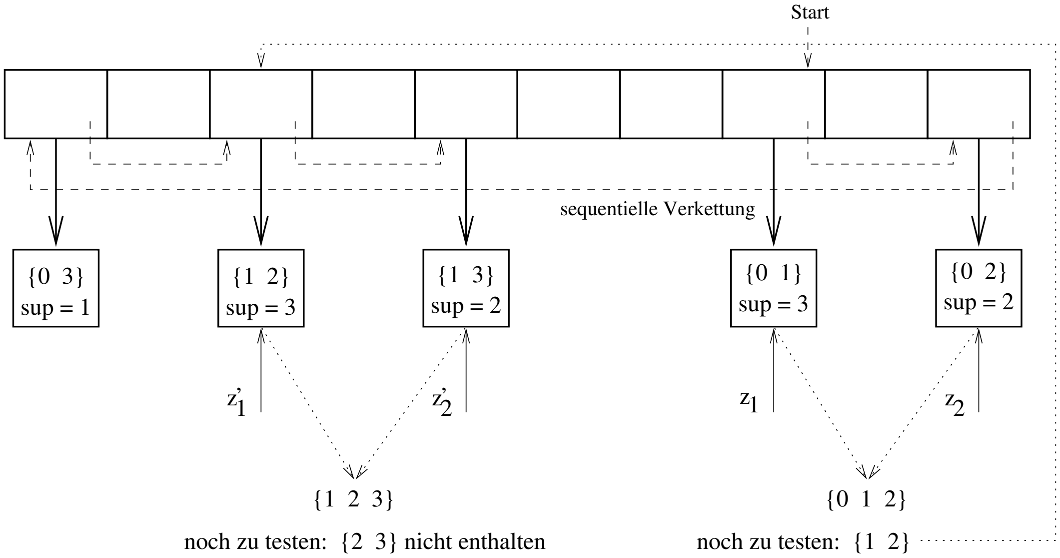


Abbildung 4.6: Hauptspeicherbasierte Generierung von Candidate 3-Itemsets (s_{min} = 2). Nach der Generierung des Candidate 3-Itemsets { 0 1 2 } werden die beiden Zeiger, z_1 und z_2, gemäß der sequentiellen Verkettung zu den Positionen unter z_1' bzw. z_2' weitergeschaltet, um den nächsten Candidate 3-Itemset { 1 2 3 } zu generieren. Er wird aber verworfen, da es mit { 2 3 } eine 2-elementige Teilmenge enthält, die nicht in der Hashtabelle enthalten und damit kein Large 2-Itemset ist.

Zur Generierung der Candidate k-Itemsets werden zwei Zeiger, z_1 und z_2, zum sequentiellen Durchlauf über die belegten Hashtabellenpositionen reserviert (siehe Abb. 4.6). Der Zeiger z_1 wird zu Beginn auf diejenige Hashtabellenposition gesetzt, die den Verweis $v^1_{itemset}$ auf den zuerst generierten Candidate $(k-1)$-Itemset enthält und steuert danach jede Hashtabellenposition genau einmal an, indem z_1 der Reihe nach auf die Verweise v^1_{next} bis $v^{|C_k|}_{next}$ gesetzt wird. Sobald sich z_1 an einer Position befindet, unter der ein Verweis $v^i_{itemset}$ $(1 \leq i \leq |C_{k-1}|)$ auf einen Large $(k-1)$-Itemset $li^1_{k-1} = \{i^1_1, \ldots, i^1_{k-1}\}$ abgespeichert ist, wird z_2 eingesetzt, um über die Verweise v^i_{next} $(i < |C_{k-1}|)$ der Reihe nach jeweils eine Hashtabellenposition anzusteuern, die einen Verweis $v^j_{itemset}$ $(i < j \leq |C_{k-1}|)$ auf einen weiteren Large $(k-1)$-Itemset $li^2_{k-1} = \{i^2_1, \ldots, i^2_{k-1}\}$ enthält, der sich durch $i^1_{k-1} < i^2_{k-1}$ nur in dem $(k-1)$.ten Item von li_1 unterscheidet und zusammen mit li^1_{k-1} einen k-Itemset $ci_k = \{i^1_1, i^1_2, \ldots, i^1_{k-1}, i^2_{k-1}\}$ bildet (vgl. Fall 3, Seite 49).

Darüberhinaus muß der Reihe nach getestet und sichergestellt werden, daß neben li^1_{k-1} und li^2_{k-1} jeder weitere in ci_k enthaltene $(k-1)$-Itemset in C_{k-1} mit einem Supportwert von mindestens s_{min} enthalten ist, damit ci_k als Candidate k-Itemset in die Hypothesenmenge C_k eingetragen werden kann (vgl. Punkt 3). Zu diesem Zweck werden Suchoperationen innerhalb der die Menge C_{k-1} verwaltenden Hashtabelle aufgerufen, die jeweils in konstanter Zeit von $O(1)$ ausgeführt werden können.

Existiert kein weiterer Large $(k-1)$-Itemset li^2_{k-1}, der auf diese Weise mit li^1_{k-1} kombiniert werden kann, dann wird z_1 auf die nächste Hashtabellenposition, die einen Verweis auf einen Large $(k-1)$-Itemset enthält, weitergeschaltet. Nachdem z_1 den Verweis $v_{next}^{|C_{k-1}|}$ erreicht hat, sind die Candidate $(k-1)$-Itemsets in aufsteigend sortierter Folge abgearbeitet worden und die Generierung der Candidate k-Itemsets ist damit abgeschlossen.

4.3.1.2 Externspeicherbasierte Generierung

Die Large $(k-1)$-Itemsets für $k > 2$ werden in einem Präfix-B^+-Baum abgespeichert, wenn bereits die Menge der Candidate $(k-1)$-Itemsets nicht komplett im Hauptspeicher gehalten werden kann. In diesem Fall werden die Candidate k-Itemsets direkt aus dem Präfix-B^+-Baum (vgl. Kapitel 3) generiert.

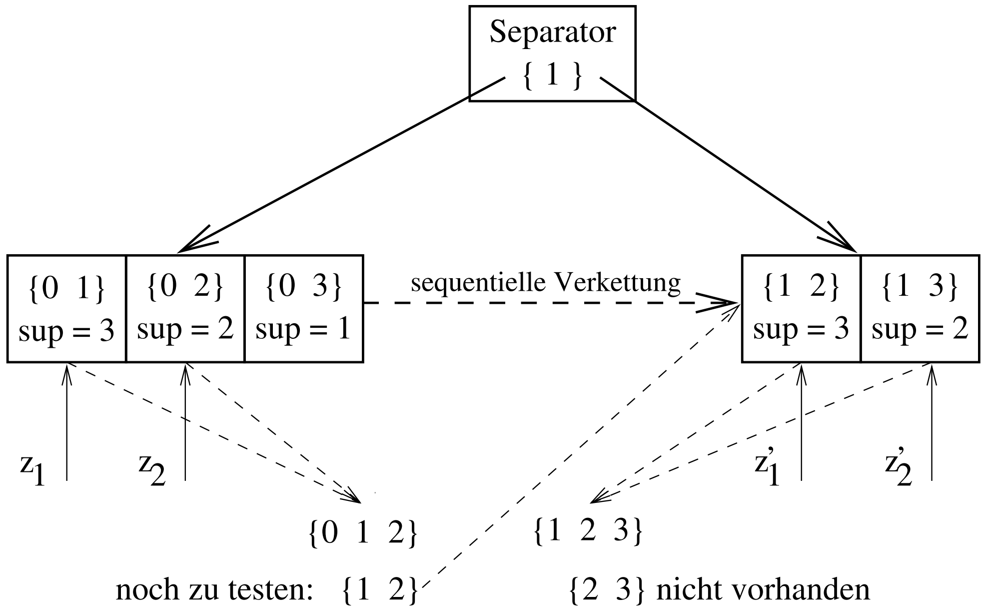


Abbildung 4.7: Externspeicherbasierte Generierung von Candidate 3-Itemsets mit Hilfe eines Präfix-B^+-Baumes ($s_{min} = 2$). Nach der Generierung des Candidate 3-Itemsets { 0 1 2 } werden die beiden Zeiger, z_1 und z_2, gemäß der sequentiellen Verkettung der Blätter zu den Positionen unter z'_1 bzw. z'_2 weitergeschaltet, um das nächste Candidate 3-Itemsets { 1 2 3 } zu generieren. Es wird aber verworfen, da es mit { 2 3 } eine 2-elementige Teilmenge enthält, die nicht im Präfix-B^+-Baum enthalten und damit kein Large 2-Itemset ist.

Bei der Generierung der Candidate k-Itemsets wird zum einen die sequentielle Verkettung der Blätter des Präfix-B^+-Baumes und zum anderen die aufsteigende Sortierung der in jedem Blatt befindlichen Large $(k-1)$-Itemsets ausgenutzt. Daher können hier in Anlehnung an die hauptspeicherbasierte Generierung der Kandidaten wiederum zwei Zeiger, z_1 und z_2, reserviert werden, mit deren Hilfe die Large $(k-1)$-Itemsets in den Blättern sequentiell durchlaufen werden. Der Zeiger z_1 durchläuft die Large $(k-1)$-Itemsets in jedem Blatt genau einmal, während z_2 jeweils auf nachfolgende Large $(k-1)$-Itemsets verweist, um zusammen jeweils auf Paare von Large $(k-1)$-Itemsets, li^1_{k-1} und li^2_{k-1}, zu verweisen, die zu einem Candidate k-Itemset ci_k kombiniert werden können (siehe Abb. 4.7).

Außer li^1_{k-1} und li^2_{k-1} werden systematisch alle $(k-1)$-elementigen Teilmengen des erzeugten Candidate k-Itemset ci_k daraufhin getestet, ob sie als Large $(k-1)$-Itemset in dem Präfix-B^+-Baum gespeichert sind. Dieser Test verursacht im Gegensatz zur hauptspeicherbasierten Generierung für jede Teilmenge einen Zeitaufwand von $O(log_d n)$, wobei d die Ordnung (Verzweigungsgrad) des Präfix-B^+-Baumes spezifiziert. Nur wenn der Test für alle zu untersuchenden Teilmengen erfolgreich durchgeführt worden ist, wird ci_k in die Hypothesenmenge C_k aufgenommen, ansonsten wird ci_k verworfen.

4.3.2 Supportwertermittlung der Candidate Itemsets

Bevor die Large k-Itemsets aus der Hypothesenmenge C_k herausgefiltert werden können, müssen zunächst die Supportwerte der Candidate k-Itemsets bestimmt werden, indem für jeden Candidate k-Itemset die Anzahl der Transaktionen, die dieses Itemset enthalten, bestimmt wird. Die Prozedur zur Ermittlung der Supportwerte für die Candidate k-Itemsets läuft nach der in Abbildung 4.8 dargestellten Prozedur in den folgenden Schritten ab:

- Der Blockindex *b_index* wird sequentiell durchlaufen, um jeden Block b_i, dessen Indexeintrag einen Wert größer als 0 aufweist, einzulesen, damit die darin als Bitvektoren gespeicherten Transaktionen gescannt werden können (vgl. Abb. 4.8, Zeile 01–04).

- Jede Transaktion t' in b_i, dessen zugehöriges Bit im Transaktionsindex *t_index* gesetzt ist, wird weiter untersucht (vgl. Abb. 4.8, Zeile 05), um mit der globalen Itemliste durch „und" verknüpft zu werden (vgl. Abb. 4.8, Zeile 06). Das Ergebnis dieser Verknüpfung ist ein Bitvektor t'', in dem alle Bits aus t', die für Items stehen, die nicht in den Large $(k-1)$-Itemsets und damit auch nicht in den Candidate k-Itemsets vorkommen, gelöscht sind. Mit dieser Verknüpfung wird das Ziel verfolgt, die Kosten für die Bestimmung der in t' präsenten Candidate k-Itemsets zu senken, denn es werden nur die gesetzten Bits in t'' in Itemnummern umrepräsentiert und in t''' gespeichert (vgl. Abb. 4.8, Zeile 07).

```
Procedure determine_support_values (𝓑, b_index, t_index, C_k, global_items)

𝓑: Menge der Blöcke
b_index: Blockindex
t_index: Transaktionsindex
C_k: Hypothesenmenge der Candidate k-Itemsets
global_items: globale Itemliste als Bitvektor

for all b_i ∈ 𝓑 mit i ∈ {0,...,|𝓑| − 1} do begin
   if (b_index(i) > 0 ) then do begin
      lese b_i;
      for all t' ∈ b_i do begin
         // t' ist eine Transaktion in Bitvektordarstellung
         if (t_index(t') = 1) then do begin
            t'' := t' and global_items;
            t''' := Itemnummer-Repräsentation von t'';
            if (|t'''| ≥ k) then do begin
               for all ci_k ⊆ t''' do
                  if (ci_k ∈ C_k) then support(ci_k)++;
               if (|t'''| = k) then do begin
                  t_index(t') := 0;
                  b_index(i)--;
               end;
            end;
            else do begin
               t_index(t') := 0;
               b_index(i)--;
            end;
         end;
      end;
   end;
end;
```

Abbildung 4.8: Die Prozedur zur Ermittlung der Supportwerte für eine Hypothesenmenge C_k.

In Iteration 2 des Apriori-Algorithmus ergibt sich t''' aus den in $\mathcal{B}$ als Bitvektoren repräsentierten Transaktionen (siehe Abb. 4.4) bei einer globalen Itemliste *global_items*, die sich zusammensetzt aus den beiden Bitstrings, $s_0 = 11111110$ und $s_1 = 00000000$ (siehe Seite 46) wie folgt:

Nach der „und"-Verknüpfung mit der globalen Itemliste *global_items* ist in dem Ergebnis-Bitvektor zu der Transaktion t_6 das Bit mit der Nummer 7 und zu der Transaktion t_7 das Bit mit der Nummer 8 gelöscht, weil die Bitnummern 7 und 8 in der globalen Itemliste *global_items* auf 0 gesetzt sind, denn sie repräsentieren mit *Chips* und *Schinken* zwei Items, die nur einmal bei einem minimalen Supportwert von $s_{min} = 2$ in $\mathcal{D}$ vorkommen und daher keine Large 1-Itemsets darstellen.

- Wenn in t''' mindestens k Itemnummern enthalten sind, wird der Reihe nach jeder darin enthaltene k-Itemset ci_k aufgezählt und getestet, ob er sich als Candidate k-Itemset in der Hypothesenmenge C_k befindet, um den entsprechenden Supportwert $support(ci_k)$ um 1 zu inkrementieren (vgl. Abb. 4.8,

t_i	t'	t''	t'''
t_0	s_0: 1110 0000	1110 0000	{ 0, 1, 2 }
t_1	s_0: 1100 0010	1100 0010	{ 0, 1, 6 }
t_2	s_0: 0100 1000	0100 1000	{ 1, 4 }
t_3	s_0: 1010 0000	1010 0000	{ 0, 2 }
t_4	s_0: 1110 0010	1110 0010	{ 0, 1, 2, 6 }
t_5	s_0: 0000 1000	0000 1000	{ 4 }
t_6	s_0: 1000 0101	1000 0100	{ 0, 5 }
t_7	s_0: 0001 0100 s_1: 1000 0000	0001 0100 0000 0000	{ 3, 5 }
t_8	s_0: 0011 0000	0011 0000	{ 2, 3 }

Abbildung 4.9: Repräsentationswechsel der als Bitvektoren gespeicherten Transaktionen aus $\mathcal{B}$ in Mengen von Itemnummern zum Zwecke der Ermittlung der Supportwerte für Candidate 2-Itemsets.

Zeile 08–09).

Die Aufzählung der in t''' enthaltenen k-Itemsets geschieht mit Hilfe von „Gray-Codes“, die das Prinzip des „Minimal Change“ realisieren [Nijenhuis und Wilf, 1978], indem der jeweils nächste aufzuzählende k-Itemset dadurch entsteht, daß aus dem direkt zuvor aufgezählten k-Itemset genau eine Itemnummer durch eine andere ersetzt wird. In der folgenden Beispieltabelle werden zur Illustration alle 3-Itemsets aus der Transaktion $t''' = \{1, 2, 3, 4, 5\}$ sowohl in aufsteigend sortierter Ordnung als auch nach dem „Minimal Change“-Prinzip aufgezählt:

	Aufsteigend sortierte Aufzählung	„Minimal Change“-Aufzählung
1.	{ 1 2 3 }	{ 1 2 3 }
2.	{ 1 2 4 }	{ 1 3 4 }
3.	{ 1 2 5 }	{ 2 3 4 }
4.	{ 1 3 4 }	{ 1 2 4 }
5.	{ 1 3 5 }	{ 1 4 5 }
6.	{ 1 4 5 }	{ 2 4 5 }
7.	{ 2 3 4 }	{ 3 4 5 }
8.	{ 2 3 5 }	{ 1 3 5 }
9.	{ 2 4 5 }	{ 2 3 5 }
10.	{ 3 4 5 }	{ 1 2 5 }

Während sich zwei benachbarte 3-Itemsets in der „Minimal Change“-Aufzählung nur in jeweils einer Itemnummer unterscheiden, gibt es in der aufsteigend sortierten Aufzählung zwischen dem 3. und 4. und 6. und 7. 3-Itemset zwei Übergänge, in denen jeweils zwei Itemnummern ausgetauscht werden.

In Anlehnung an Abbildung 4.9 werden die folgenden zu jeder Transaktion t_i in t''' enthaltenen k-Itemsets gemäß der „Minimal Change“-Ordnung aufgezählt und getestet, ob sie als Candidate 2-Itemsets in C_2 enthalten sind, um die entsprechenden Supportwerte zu erhöhen:

t_i	t'''	zu untersuchende 2-Itemsets
t_0	{0 1 2}	{0 1}, {1 2}, {0 2}
t_1	{0 2 3}	{0 1}, {1 6}, {0 6}
t_2	{1 4}	{1 4}
t_3	{0 2}	{0 2}
t_4	{0 1 2 6}	{0 1}, {1 2}, {0 2}, {2 6}, {1 6}, {0 6}
t_5	{4}	–
t_6	{0 5 7}	{0 5}
t_7	{3 5 8}	{3 5}
t_8	{2 3}	{2 3}

Aus der Transaktion t_5 kann kein 2-Itemset aufgezählt werden. Ferner wird aus der Transaktion t_6 bzw. t_7 kein 2-Itemset ci_2 aufgezählt, der das Item mit der Itemnummer 7 bzw. 8 enthält, weil die zugehörigen Items *Chips* bzw. *Schinken* keinen Large 1-Itemset darstellen (vgl. Abb. 4.3), denn ci_2 kann laut Definition 8 (siehe Seite 9) kein Candidate 2-Itemset sein.

Aufgrund der Tatsache, daß die Menge der Candidate k-Itemsets C_k entweder hauptspeicherbasiert mit Hilfe einer internen Hashtabelle oder externspeicherbasiert mit Hilfe einer Hashdatei verwaltet werden kann, können zwei Vorgehensweisen zur Inkrementierung der Supportwerte der in t''' enthaltenen Candidate k-Itemsets unterschieden werden:

1. Wenn C_k mit Hilfe einer internen Hashtabelle verwaltet wird, so wird *Double Hashing* (siehe Kapitel 3, Seite 27) eingesetzt, um für jeden in t''' enthaltenen k-Itemset ci_k nach einem Verweis in der Hashtabelle zu suchen, der auf einen entsprechenden Candidate k-Itemset ci_k zeigt. Der Supportwert $support(ci_k)$ von ci_k in C_k wird im Falle einer erfolgreichen Suche um 1 inkrementiert. Die Suche kann erfolglos abgebrochen werden, falls
   - eine Hashtabellenposition angesteuert wird, unter der sich ein „leerer“ Verweis befindet oder
   - innerhalb der Sondierungfolge ein Verweis betrachtet wird, unter dem sich ein Candidate k-Itemset befindet, der größer als ci_k ist. Dieses folgt daraus, daß alle Candidate k-Itemsets in einer aufsteigend sortierten Ordnung erzeugt und die Verweise darauf in einer entsprechenden Reihenfolge in die interne Hashtabelle eingetragen

worden sind und demnach die Candidate k-Itemsets, auf die entlang einer Sondierungsfolge zugegriffen wird, in einer aufsteigend sortierten Reihenfolge betrachtet werden. In diesem Fall kann also gefolgert werden, daß kein Verweis auf einen Candidate k-Itemset ci_k in der internen Hashtabelle enthalten ist.

2. Wenn eine externe Hashdatei zur Verwaltung von C_k verwendet wird, so wird für jeden in t''' enthaltenen k-Itemset ci_k mit Hilfe einer Hashfunktion h auf einen bestimmten Block und gegebenenfalls auf dessen Überlaufblöcke zugegriffen, um dort nach einem entsprechenden Candidate k-Itemset ci_k zu suchen. Falls ci_k gefunden werden kann, wird der Supportwert $support(ci_k)$ in C_k um 1 inkrementiert.

   Bei dieser Vorgehensweise werden laufend Blöcke ein- und ausgelagert, da sich diejenigen Blöcke, auf die über die Hashfunktion h zugegriffen werden muß, in der Regel nicht permanent im Hauptspeicher befinden. Wird ein bestimmter Block aus dem Hauptspeicher ausgelagert, so werden die darin enthaltenen inkrementierten Supportwerte der betreffenden Candidate k-Itemsets in die Hashdatei zurückgeschrieben. Damit entsteht nicht nur ein lesender sondern auch ein hoher schreibender I/O-Aufwand.

- Der Blockindex *b_index* und der Transaktionsindex *t_index* werden wie folgt aktualisiert: Wenn die Anzahl der in t''' enthaltenen Itemnummern kleiner oder gleich k ist, dann wird das entsprechende Bit für die Transaktion t' im Transaktionsindex *t_index* gelöscht (vgl. Abb. 4.8, Zeile 10–11 und Zeile 15–16). Darüberhinaus wird im Blockindex *b_index* der Indexeintrag für den Block b_i, in dem die Transaktion t' gespeichert ist, um 1 dekrementiert (vgl. Abb. 4.8, Zeile 12 und 17).

Nach der Ermittlung der Supportwerte für die Candidate 2-Itemsets in den Transaktionen in Abbildung 4.9 besitzen die beiden Indizes, *t_index* und *b_index*, den folgenden aktualisierten Zustand. Jeder Block b_i mit $i \in \{0, 1, 2\}$ beinhaltet jeweils drei Transaktionen.

Transaktion	Block	$t\_index(t_i)$	$b\_index(b_i)$
t_0	b_0	1	2
t_1		1	
t_2		0	
t_3	b_1	0	1
t_4		1	
t_5		0	
t_6	b_2	0	0
t_7		0	
t_8		0	

Zu beachten ist, daß der Indexeintrag für Block b_2 einen aktualisierten Wert von 0 aufweist. Zur Ermittlung der Supportwerte für die Candidate 3-Itemsets wird der Block b_2 nicht mehr in den Hauptspeicher eingelesen und bearbeitet, denn alle darin gespeicherten Transaktionen enthalten keine Candidate 3-Itemsets, weil die den Transaktionen entsprechenden Bits im Transaktionsindex gelöscht sind.

4.3.3 Selektion der Large Itemsets aus der Kandidatenmenge

```
Procedure find_large_itemsets (s_min, C_k, global_items)

s_min: minimaler Supportwert
C_k: Hypothesenmenge der Candidate k-Itemsets
global_items: globale Itemliste als Bitvektor

(01) L_k = Ø;
(02) for all ci_k ∈ C_k begin
(03)     if (support(ci_k) ≥ s_min) then do begin
(04)         L_k = L_k ∪ ci_k ;
(05)         for all i ∈ ci_k do global_items(i) := 1;
             // Für jede Itemnummer i, die in einem Large k-Itemset vorkommt,
             // wird das entsprechende Bit in der globalen Itemliste gesetzt.
(06)     end;
(07) end;
(08) return L_k;
```

Abbildung 4.10: Die Prozedur zum Selektieren der Large k-Itemsets aus der Menge der Candidate k-Itemsets nach der Ermittlung der Supportwerte.

Nach der Supportwertermittlung wird jeder Candidate k-Itemset, der einen Supportwert von mindestens s_{min} vorweisen kann, als Large k-Itemset in die Menge L_k eingetragen (vgl. Abb. 4.10, Zeile 02–04). Außerdem wird die globale Itemliste *global_items* aktualisiert, indem diejenigen Bits gesetzt werden, deren zugehörige Bitnummern mit den entsprechenden Itemnummern der Large k-Itemsets korrespondieren (vgl. Abb. 4.10, Zeile 05). Folglich werden in der nächsten Iteration des Algorithmus innerhalb der Supportwertermittlung nur diejenigen Itemnummern einer Transaktion betrachtet und daraus $(k+1)$-Itemsets aufgezählt, die in Candidate $(k+1)$-Itemsets vorkommen können, denn die Menge der Candidate $(k+1)$-Itemsets wird aus der Menge der Large k-Itemsets generiert und enthält folglich nur Itemnummern, die in Large k-Itemsets vorkommen.

4.4 Die Regelgenerierung

Nachdem die Large Itemsets in n Iterationen des Apriori-Algorithmus bestimmt worden sind, kann mit der Regelgenerierung begonnen werden, dessen Implementierung sich an dem in Kapitel 2 vorgestellten schnelleren Algorithmus orientiert (siehe Seite 17).

```
Procedure rule_discovery (c_min, ⋃_{i=1}^{k} L_i);
c_min: minimaler Konfidenzwert
⋃_{i=1}^{k} L_i: Menge aller Large Itemsets

(01) H_1 := ∅;
     // H_1 ist die Menge der 1-elementigen Konklusionen aller für
     // den Benutzer relevanten Regeln
(02) k := n;
     // n steht für die Anzahl der Items, die maximal in einem Large Itemset
     // enthalten sind
(03) while k > 1 do begin
(04)   for all li_k ∈ L_k do begin
(05)     A := { a_{k-1} | a_{k-1} ⊂ li_k };
         // A ist die Menge aller (k − 1)-Teilmengen aus einem Large k-Itemset
(06)     for all a_{k-1} ∈ A do begin
(07)       conf := support(li_k) / support(a_{k-1});
(08)       if (conf ≥ c_min) then do begin
(09)         print a_{k-1} → (li_k \ a_{k-1}) mit
(10)         confidence = conf ∧ support = support(li_k);
(11)         H_1 = H_1 ∪ (li_k \ a_{k-1});
(12)       end;
(13)     end;
(14)     generate_more_rules(c_min, li_k, H_1);
(15)   end;
(16)   k--;
(17) end;
```

Abbildung 4.11: Die Prozedur zur Generierung von Assoziationsregeln mit einer 1-elementigen Konklusion.

Jede Menge L_k der Large k-Itemsets mit $k \geq 2$ wird sequentiell durchlaufen, um aus den darin enthaltenen Large k-Itemsets li_k mit dem Supportwert $support(li_k)$ Assoziationsregeln zu generieren und auszugeben, die in den Konklusionen sowohl ein als auch mehrere Items enthalten. Zuerst wird k auf n gesetzt (vgl. Abb. 4.11, Zeile 02), damit die Regelgenerierung mit der Menge der längsten Large Itemsets, die sich entweder in einer hauptspeicherbasierten Liste, einer internen Datei oder in einem Präfix-B^+-Baum befindet, gestartet werden kann. Danach wird die Menge L_{n-1} abgearbeitet, usw.

Zur Generierung der Assoziationsregeln $a_{k-1} \rightarrow (li_k \setminus a_{k-1})$ mit einer 1-elementigen Konklusion $(li_k \setminus a_{k-1})$ und dementsprechend mit einer $(k-1)$-elementigen Prämisse a_{k-1} werden zunächst alle $(k-1)$-elementigen Teilmengen von li_k in A gespeichert (vgl. Abb. 4.11, Zeile 05). Dann wird auf die Menge L_{k-1} der Large $(k-1)$-Itemsets zugegriffen, die sich entweder in einem Prafix-B^+-Baum befindet oder hauptspeicherbasiert mit Hilfe einer internen Hashtabelle verwaltet wird, um der Reihe nach für jeden Large $(k-1)$-Itemset a_{k-1} den Supportwert

$support(a_{k-1})$ abzurufen und mit

$$conf = \frac{support(li_k)}{support(a_{k-1})}$$

den Konfidenzwert der Assoziationsregel $a_{k-1} \rightarrow (li_k \setminus a_{k-1})$ zu berechnen (vgl. Abb. 4.11, Zeile 06–07). Wenn der Konfidenzwert $conf$ mindestens einen minimalen Konfidenzwert von c_{min} aufweist, dann wird die für den Benutzer relevante Regel $a_{k-1} \rightarrow (li_k \setminus a_{k-1})$ zusammen mit dem Konfidenzwert $conf$ und dem Supportwert $support(li_k)$ ausgegeben und die 1-elementige Konklusion $(li_k \setminus a_{k-1})$ in die Menge H_1 abgelegt (vgl. Abb. 4.11, Zeile 08–12), die den Ausgangspunkt zur Generierung von Assoziationsregeln mit mehrelementigen Konklusionen darstellt (vgl. Abb. 4.11, Zeile 14).

```
Procedure generate_more_rules (c_min, li_k, H_m)
 c_min: minimaler Konfidenzwert
 li_k: Large k-Itemset
 H_m: Menge von m-elementigen Konklusionen

 (01) if (k > m+1) then do begin
 (02)    H_{m+1} = Ø;
 (03)    for all h_m ∈ H_m do begin
 (04)       for all h'_m ∈ H_m mit h'_m > h_m do begin
 (05)          if (|h_m ∩ h'_m| = m − 1) then do begin
 (06)             h_{m+1} := h_m ∪ h'_m;
 (07)             if (∀ h''_m ⊂ h_{m+1} : h''_m ∈ H_m) then do begin
 (08)                conf := support(li_k) / support(li_k \ h_{m+1});
 (09)                if (conf ≥ minconf) then do begin
 (10)                   print (li_k \ h_{m+1}) → h_{m+1} mit
                        confidence = conf ∧ support = support(li_k);
 (11)                   H_{m+1} = H_{m+1} ∪ h_{m+1};
 (12)                end;
 (13)             end;
 (14)          end;
 (15)       end;
 (16)    end;
 (17)    generate_more_rules(li_k, H_{m+1});
 (18) end;
```

Abbildung 4.12: Die Prozedur zur Generierung von Assoziationsregeln mit mehrelementigen Konklusionen.

Die Generierung von Assoziationsregeln $(li_k \setminus h_{m+1}) \rightarrow h_{m+1}$ aus li_k, die aus mehrelementigen Konklusionen h_{m+1} mit $1 \leq m \leq k-1$ bestehen, beginnt damit, daß aus der Menge H_1 in Analogie zur Kandidatengenerierung bei der Bestimmung der Large Itemsets der Reihe nach 2-Itemsets h_2 gebildet werden (vgl. Abb. 4.12, Zeile 03–07), die als 2-elementige Konklusionen in Assoziationsregeln der Form $(li_k \setminus h_2) \rightarrow h_2$ eingesetzt werden. Zur Bestimmung des Konfidenzwertes $conf$ einer Regel $(li_k \setminus h_2) \rightarrow h_2$ mit

$$conf = \frac{support(li_k)}{support(li_k \setminus h_2)}$$

muß auf die Menge L_{k-2} zugegriffen werden, die sich entweder in einem Präfix-B^+-Baum befindet oder hauptspeicherbasiert mit Hilfe einer internen Hashtabelle verwaltet wird, um den Supportwert $support(li_k \setminus h_2)$ der Prämisse bzw. des Large $(k-2)$-Itemsets $(li_k \setminus h_2)$ zu erhalten (vgl. Abb. 4.12, Zeile 08). Wenn der Konfidenzwert $conf$ mindestens c_{min} beträgt, dann wird die Assoziationsregel zusammen mit dem Konfidenz- und Supportwert ausgegeben und h_2 der Menge H_2 hinzugefügt.

Sobald die Generierung von Regeln mit 2-elementigen Konklusionen beendet ist, wird die Menge H_2 benutzt, um daraus 3-elementige Konklusionen für weitere aus li_k zu generierende Assoziationsregeln zu bilden (vgl. Abb. 4.12, Zeile 17), wenn die Kardinalität von li_k größer als 3 $(= 2 + 1)$ ist (vgl. Abb. 4.12, Zeile 01). Zur Berechnung der Konfidenzwerte der entsprechenden Regeln muß dann auf die Menge L_{k-3} zugegriffen werden, die mit den Large $(k-3)$-Itemsets die $(k-3)$-elementigen Prämissen der zu generierenden Assoziationsregeln enthält. Die Regelgenerierung wird für li_k beendet, wenn die Menge H_m mit $1 \leq m < k$ leer ist oder nachdem Assoziationsregeln mit $(k-1)$-elementigen Konklusionen und 1-elementigen Prämissen erzeugt worden sind.

Zu beachten ist, daß bei der hier eingesetzten Implementierung zur Generierung von Assoziationsregeln $(li_k \setminus h_{m+1}) \rightarrow h_{m+1}$ mit mehrelementigen Konklusionen h_{m+1} ein Speicherbereich $\mathcal{S}$ zur Aufnahme der Mengen H_m und H_{m+1} im voraus reserviert werden muß. Die maximale Größe von $\mathcal{S}$ läßt sich wie folgt berechnen:

Aus einem Large k-Itemset können maximal $\binom{k}{m}$ Assoziationsregeln der Form $(li_k \setminus h_m) \rightarrow h_m$ mit einer m-elementigen Konklusion generiert werden. Zur konkreten Berechnung des Binomialkoeffizienten $\binom{k}{m}$ kann das sogenannte *Pascalsche Dreieck* angegeben werden, in dessen k.ter Zeile $(k \geq 1)$ von links nach rechts die Werte von $\binom{k}{m}$ für $m = 0, 1, \ldots, k$ stehen (vgl. [Dörfler und Peschek, 1988]):

$$\begin{array}{c} 1\ 1 \\ 1\ 2\ 1 \\ 1\ 3\ 3\ 1 \\ 1\ 4\ 6\ 4\ 1 \\ 1\ 5\ 10\ 10\ 5\ 1 \\ \text{usw.} \end{array}$$

Mit Ausnahme der Zahlen am Rande ist jede Zahl die Summe der beiden links und rechts oberhalb stehenden Zahlen. Der maximale Wert für $\binom{k}{m}$ liegt jeweils in der Mitte oder in den beiden mittleren Positionen einer Zeile k, d.h. bei $\lceil \frac{k}{2} \rceil$.

Daraus folgt, daß für die Regelgenerierung aus den in n Iterationen des Apriori-Algorithmus bestimmten Large Itemsets maximal ein Speicherbereich $\mathcal{S}$ für

$$\binom{n}{\lceil n/2 \rceil}$$

$\lceil n/2 \rceil$-elementige und

$$\binom{n}{\lceil n/2 \rceil + 1}$$

$(\lceil n/2 \rceil + 1)$-elementige Konklusionen im voraus reserviert werden muß, der im folgenden nicht mehr zur Verfügung steht.

Während der Generierung von Assoziationsregeln der Form $a_{k-1} \rightarrow (li_k \backslash a_{k-1})$ bzw. $(li_k \setminus h_{m+1}) \rightarrow h_{m+1}$ mit $1 < m < k$ wird genau einmal auf jedes Large k-Itemset li_k zur Abfrage des Supportwertes $support(li_k)$ und mehrfach auf die Mengen $L_1, L_2, \ldots, L_{k-1}$ zur Abfrage der Supportwerte der jeweiligen Prämissen, a_{k-1} aus der Menge der Large $(k-1)$-Itemsets (vgl. Abb. 4.11, Zeile 07) bzw. $(li_k \backslash h_m)$ mit $1 < m < k$ aus der Menge der Large $(k-m)$-Itemsets (vgl. Abb. 4.12, Zeile 08), zugegriffen. Solange der zur Verfügung stehende Hauptspeicher ausreicht, wird daher der Reihe nach für $k = n$ jede Menge L_i mit $1 < i < k$, intern mit Hilfe einer Hashtabelle verwaltet und gegebenenfalls aus einer externen Datei oder einem Präfix-B^+-Baum dorthin transferiert. Sobald der verfügbare Hauptspeicher ausgeschöpft ist, wird jede weitere Menge L_i in einem Präfix-B^+-Baum verwaltet und gegebenfalls aus einer externen Datei dorthin transferiert.

Problematisch ist die Regelgenerierung für den Fall, in dem relativ „lange" Large Itemsets entdeckt worden sind, weil dafür ein enorm großer Speicherbedarf zur Aufbewahrung der Mengen H_m und H_{m+1} zur Generierung von Assoziationsregeln mit mehrelementigen Konklusionen entsteht. Wenn also der dafür vorgesehene Speicherbereich $\mathcal{S}$ derart groß ist, daß nahezu der gesamte Hauptspeicher zur Aufnahme der Mengen H_m und H_{m+1} in Anspruch genommen werden müßte, dann wird jede Menge L_i mit $1 < i < k$, die sich im Hauptspeicher oder in einer externen Datei befindet, in einen Präfix-B^+-Baum übertragen. Die Menge L_1 der Large 1-Itemsets verbleibt aufgrund ihrer relativ geringen Kardinalität im Hauptspeicher, während L_k gegebenenfalls in eine externe Datei ausgelagert wird, denn L_k muß lediglich einmal sequentiell durchlaufen werden, um der Reihe nach für jeden Large k-Itemset li_k die jeweiligen Assoziationsregeln zu generieren.

4.5 Komplexitätsbetrachtung

Die Komplexitätsbetrachtung des Apriori-Algorithmus in der hier vorgestellten Implementierung ergibt sich unter Berücksichtigung der Phasen: Vorverarbeitung, Bestimmung der Large k-Itemsets und Regelgenerierung:

1. Vorverarbeitung: Das zweimalige Scannen der Datenmenge $\mathcal{D}$, um einerseits die Items in $\mathcal{D}$ aufsteigend nach den zuvor ermittelten Supportwerten zu sortieren, die Large 1-Itemsets zu selektieren und anschließend die Items in Itemnummern umzurepräsentieren und andererseits jede Transaktion in $\mathcal{D}$ in einen Bitvektor umzuwandeln und in die Datei $\mathcal{B}$ abzuspeichern, erfordert jeweils einen Zeitaufwand von $O(|\mathcal{D}| \cdot |t_{max}|)$, wobei $|\mathcal{D}|$ die Anzahl der Transaktionen in $\mathcal{D}$ und $|t_{max}|$ die Anzahl der Items, die in der längsten Transaktion t_{max} in $\mathcal{D}$ vorkommen, angibt.

   Der Platzbedarf beträgt durch die Verwendung zweier Puffer, $buffer\_in$ und $buffer\_out$, zur jeweiligen Pufferung der zu scannenden Transaktionen aus $\mathcal{D}$ und der in die Datei $\mathcal{B}$ zu schreibenden Transaktionen, die in Bitvektoren umgewandelt worden sind, $O(|buffer\_in| + |buffer\_out|)$, wobei sich $|buffer\_in|$ bzw. $|buffer\_out|$ auf die jeweiligen Puffergrößen beziehen.

2. Bestimmung der Large k-Itemsets für $k \geq 2$: Der Zeitaufwand zur Bestimmung der Large k-Itemsets wird bestimmt durch die Generierung der Candidate k-Itemsets, der Ermittlung der Supportwerte für jeden Candidate k-Itemset durch ein Scannen der Transaktionen in $\mathcal{B}$ und durch die anschließende Selektion der Large k-Itemsets aus der Menge der Candidate k-Itemsets.

   (a) Kandidatengenerierung: Da die Menge C_k der Candidate k-Itemsets aus der Menge L_{k-1} der Large $(k-1)$-Itemsets erzeugt wird, ergibt sich ein Zeitaufwand von $O(|L_{k-1}| \cdot |L_{k-1}|)$, wobei $|L_{k-1}|$ die Kardinalität der Menge L_{k-1} spezifiziert.

   Der Platzbedarf zur Aufbewahrung der Menge C_k von x Candidate k-Itemsets wird durch den Term

   $$(\underbrace{x \cdot (k \cdot \gamma + \delta)}_{Hauptspeicher} + \underbrace{x \cdot \frac{1}{\alpha} \cdot \beta}_{Hashtabelle})$$

   bestimmt (siehe Seite 50), wenn die Menge C_k komplett im zur Verfügung stehenden Hauptspeicher der Kardinalität $\mathcal{M}$ mit Hilfe einer Hashtabelle verwaltet werden kann, ansonsten wird für die beiden Implementierungsalternativen zur Verwaltung von Candidate k-Itemsets, die nicht komplett im Hauptspeicher verwaltet werden können, der gesamte verfügbare Hauptspeicher zur Unterbringung von Candidate k-Itemsets benutzt (siehe Seite 50). Wenn die Implementierungsalternative, die auf externem linearen Hashing basiert, verwendet wird, so wird für die externe Hashdatei ein zusätzlicher Platzbedarf von $O(|C_k|)$ benötigt, wobei sich $|C_k|$ auf die Kardinalität der Menge C_k bezieht.

   (b) Ermittlung der Supportwerte: Aufgrund der Tatsache, daß bei der Supportwertermittlung der Candidate k-Itemsets die Blöcke aus $\mathcal{B}$ gelesen und aus jeder Transaktion alle darin enthaltenen k-Itemsets aufgezählt werden, kann der dafür benötigte Zeitaufwand durch

$$O(|\mathcal{B}|) + \mathcal{O}(|\mathcal{D}| \cdot \binom{|t_{max}|}{k})$$

abgeschätzt werden, wobei $|\mathcal{B}|$ die Anzahl der Blöcke, $|\mathcal{D}|$ die Anzahl der Transaktionen in $\mathcal{D}$ und $|t_{max}|$ die Kardinalität der längsten Transaktion angibt. Falls aufgrund der Verwendung der 2. Implementierungsalternative (siehe Seite 50) die Transaktionen n-mal zur Ermittlung der Supportwerte der Candidate k-Itemsets gescannt werden müssen, muß der obige Term mit n multipliziert werden.

(c) Selektion der Large k-Itemsets: Nach der Supportwertermittlung für die Candidate k-Itemsets wird die Menge C_k in einer Zeit von $O(|C_k|)$ sequentiell durchlaufen, um alle Large k-Itemsets herauszufiltern. Falls die Large k-Itemsets in einen Präfix-B^+-Baum ausgelagert werden müssen, weil die Hauptspeicherkapazität nicht ausreicht, ergibt sich ein zusätzlicher Zeitaufwand von $O(|L_k| \cdot log_d|L_k|)$. Werden die Large k-Itemsets hingegen in eine externe Datei ausgelagert, so entsteht ein zusätzlicher Zeitaufwand von $O(|L_k|)$.

3. Regelgenerierung: Für die Generierung von Assoziationsregeln mit 1-elementigen Konklusionen aus einem Large k-Itemset li_k mit $k \geq 2$ ist ein Zeitaufwand von $O(k \cdot access_{k-1})$ erforderlich. Der Term $access_{k-1}$ gibt den Zeitaufwand an, um innerhalb der Konfidenzberechnung auf eine $(k-1)$-elementige Prämisse zuzugreifen. So besitzt $access_{k-1}$ den Wert

   - 1, wenn die Menge L_{k-1}, in der sich die $(k-1)$-elementigen Prämissen befinden, intern mit Hilfe einer Hashtabelle verwaltet wird oder
   - $log_d\,|L_{k-1}|$, wenn sich L_{k-1} in einem Präfix-B^+-Baum befindet.

   Folglich ergibt sich ein Zeitaufwand von $O(\sum_{i=2}^{k} |L_i| \cdot i \cdot access_{i-1})$, um alle Assoziationsregeln mit 1-elementigen Konklusionen aus den Large Itemsets zu generieren.

   Um aus einem Large k-Itemset li_k mit $k \geq 3$ Assoziationsregeln mit m-elementigen Konklusionen für $m \geq 2$ zu erzeugen, wird die Menge H_m benutzt, aus der die m-elementigen Konklusionen der zu generierenden Regeln entnommen werden. Da H_m aus den $(m-1)$-elementigen Konklusionen der Menge H_{m-1} gebildet wird, ergibt sich bei der Generierung von Assoziationsregeln mit m-elementigen Konklusionen aus li_k ein Zeitaufwand von $O(\sum_{m=2}^{k-1}(|H_{m-1}| \cdot |H_{m-1}| \cdot access_{k-m}))$, wobei $|H_{m-1}|$ die Kardinalität der Menge H_{m-1} bezeichnet und $access_{k-m}$ die beiden folgenden Werte aufweisen kann (s.o.):

- 1, wenn die Menge L_{k-m}, in der sich die $(k-m)$-elementigen Prämissen befinden, intern mit Hilfe einer Hashtabelle verwaltet wird oder
- $log_d\,|L_{k-m}|$, wenn sich L_{k-m} in einem Präfix-B^+-Baum befindet.

Insgesamt ergibt sich bei der Generierung von Assoziationsregeln mit mehrelementigen Konklusionen aus den Large Itemsets ein Zeitaufwand von

$$O(\sum_{i\geq 3}^{k}\sum_{m=2}^{k-1}(|H_{m-1}|\cdot|H_{m-1}|\cdot access_{k-m}))$$

Assoziationsregeln mit mehrelementigen Konklusionen generiert werden.

Der Platzbedarf für die Regelgenerierung wird durch die Kardinalität des Speicherbereichs $\mathcal{S}$ bestimmt (siehe Seite 63).

Kapitel 5

Experimente

In diesem Kapitel werden sowohl Tests mit synthetischen als auch mit reellen Daten einer Drogeriekette durchgeführt. Darüberhinaus wird ein Vergleich der in Kapitel 4 angesprochenen Implementierungsalternativen durchgeführt.

5.1 Generierung von synthetischen Daten

Das in [Agrawal et al., 1996] beschriebene Verfahren zur Generierung von synthetischen Daten generiert Transaktionen, die die tatsächlichen Transaktionen im Einzelhandel imitieren. Das Modell der „realen" Welt ist, daß Kunden dazu tendieren, bestimmte Artikel zusammen einzukaufen. Deshalb kann jede mögliche Kombination unter den Artikeln als ein potentieller Large Itemset angesehen werden. Transaktionen können in der Regel mehrere Large Itemsets enthalten, wobei die Transaktionsgrößen typischerweise um einen Mittelwert gestreut sind und nur wenige Transaktionen viele Items enthalten. Genauso ist es mit den Large Itemsets, deren Größen ebenfalls um einen Mittelwert gestreut sind, und nur wenige können eine große Anzahl an Items aufweisen.

Das Programm zur Generierung synthetischer Daten bekommt als Eingabe die folgenden Parameter, um Datenmengen zu generieren:

$\lvert\mathcal{D}\rvert$	Anzahl der Transaktionen
$\lvert T\rvert$	durchschnittl. Größe der Transaktionen
$\lvert I\rvert$	durchschnittl. Größe der Large Itemsets
$\lvert L\rvert$	Anzahl der Large Itemsets
N	Anzahl der Items

Die Größe der nächsten Transaktion wird aus einer Poisson-Verteilung mit dem Mittelwert μ, der gleich $|T|$ ist, entnommen. Wenn bei N Items jedes Item mit derselben Wahrscheinlichkeit p ausgewählt wird, dann ist die erwartete Anzahl der Items in einer Transaktion gegeben durch die Binomialverteilung mit den Parametern N und p und ist approximiert durch die Poisson-Verteilung mit dem Mittelwert $N\,p$.

Dann werden der Transaktion Items und eine Reihe von Large Itemsets zugewiesen. Die Large Itemsets werden aus einer speziellen Menge $\mathcal{T}$ ausgewählt. Die Anzahl der Itemsets in $\mathcal{T}$ ist $|L|$. Ein Itemset in $\mathcal{T}$ wird generiert, indem zuerst die Größe des Itemsets aus einer Poisson-Verteilung mit einem Mittelwert μ gleich $|I|$ gewählt wird. Die Items für den ersten Itemset werden zufällig gewählt. Um das Phänomen zu modellieren, daß Large Itemsets oft Items gemeinsam haben, wird ein Teil der Items in nachfolgenden Itemsets aus den davor generierten Itemsets gewählt. Deshalb wird eine exponentiell verteilte Zufallsvariable mit einem Mittelwert, der gleich dem *Correlation Level* ist, benutzt, um diesen gemeinsamen Anteil für jeden Itemset zu bestimmen. Die restlichen Items werden zufällig gewählt.

Jedes Itemset in $|\mathcal{T}|$ besitzt ein Gewicht, das sich auf die Wahrscheinlichkeit bezieht, mit der das Itemset ausgewählt wird. Dieses Gewicht wird einer exponentiellen Verteilung mit einem einheitlichen Mittelwert entnommen, und ist normalisiert, so daß die Summe der Gewichte aller Itemsets in $\mathcal{T}$ gleich 1 ist. So wird das nächste für die Transaktion auszuwählende Itemset aus $\mathcal{T}$ ausgewählt, indem eine $|L|$-seitig gewichtete Münze geworfen wird, wobei das Gewicht einer Seite die Wahrscheinlichkeit ist, mit der das entsprechende Itemset ausgewählt wird.

Um das Phänomen zu modellieren, daß nicht alle Items eines Large Itemsets immer zusammen eingekauft werden, wird jedem Itemset in $\mathcal{T}$ ein *Corruption Level* c zugeordnet. Wenn ein Large Itemset zu einer Transaktion hinzugefügt wird, dann wird solange ein Item aus dem jeweiligen Itemset herausgenommen, bis eine uniform verteilte Zufallszahl, zwischen 0 und 1, kleiner als c ist. So werden für einen Itemset der Größe l $(1-c)$-mal l Items zur Transaktion hinzugefügt, $c(1-c)$-mal $l-1$ Items, $c^2(1-c)$-mal $l-2$ Items, usw. Der *Corruption Level* ist für ein Itemset fest und man erhält ihn aus einer Normalverteilung mit einem Mittelwert von 0.5 und einer Varianz von 0.1.

Jetzt werden Datenmengen generiert, indem N auf 1000 und $|L|$ auf 2000 gesetzt werden. Dann werden für $|T|$ die beiden Werte 10 und 20 gewählt und für $|I|$ die Werte 2, 4 und 6. Die Anzahl der Transaktionen wird auf 100.000 gesetzt. Die folgende Tabelle zeigt die Parameter für eine Testanordnung mit synthetischen Datenmengen. Für dieselben Werte für $|T|$ und $|\mathcal{D}|$ sind die Datenmengen ungefähr gleich groß für unterschiedliche Werte von $|I|$.

Name	$\|T\|$	$\|I\|$	$\|\mathcal{D}\|$	Größe in MByte
T10.I2.D100K	10	2	100K	4.0
T10.I4.D100K	10	4	100K	
T20.I2.D100K	20	2	100K	8.0
T20.I4.D100K	20	4	100K	
T20.I6.D100K	20	6	100K	

5.2 Tests mit synthetischen Daten

Die Laufzeiten für die Vorverarbeitung der synthetischen Datenmengen werden in der folgenden Tabelle gezeigt.

Datenmenge	Zeit (in Sek.)
T10.I2.D100K	12
T10.I4.D100K	12
T20.I2.D100K	24
T20.I4.D100K	24
T20.I6.D100K	62

Die Laufzeiten ergeben sich aus der Umrepräsentation der Items durch Itemnummern und der Umwandlung der ursprünglichen Transaktionen in Bitvektoren.

In Abbildung 5.1 und 5.2 werden die Laufzeiten zur Bestimmung der Large Itemsets aus den synthetischen Datenmengen in Abhängigkeit vom Supportwert dargestellt. Die Testläufe wurden auf einer Sun ULTRA mit 128 MByte Hauptspeicher durchgeführ. Die beiden Abbildungen zeigen, daß sich die Laufzeit proportional zum minimalen Supportwert verhält. Je geringer der Supportwert, desto höher ist der programminterne Aufwand, beispielsweise zur Verwaltung der Candidate Itemsets, so daß sich dieses direkt auf die Laufzeit auswirkt.

Vergleicht man die beiden Abbildungen miteinander, so kann man feststellen, daß die Laufzeit ebenfalls von der durchschnittlichen Länge der Transaktionen abhängt, denn bei sehr langen Transaktionen entsteht ein verhältmismäßig großer Aufwand zur Bestimmung der darin enthaltenen Candidate Itemsets.

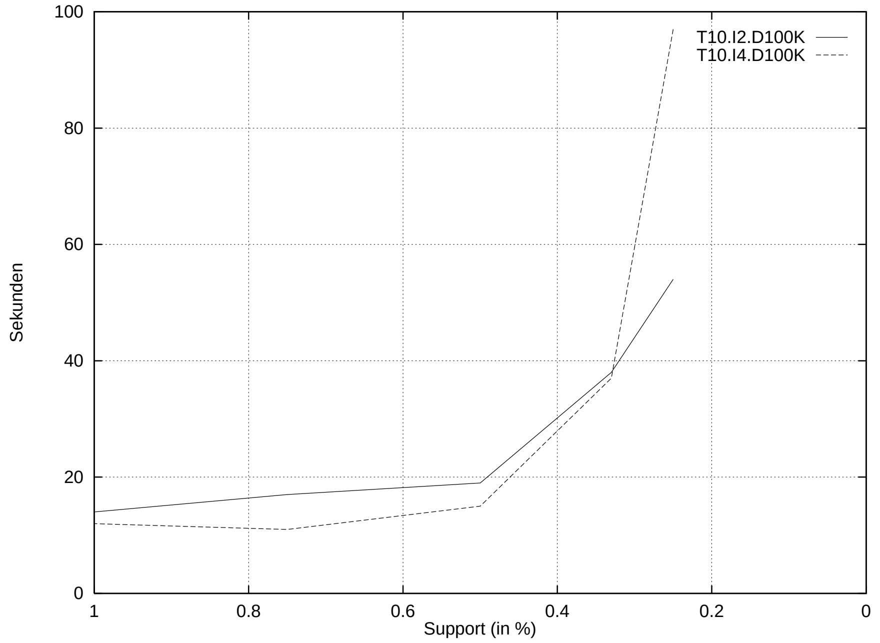


Abbildung 5.1: Abhängigkeit der Laufzeit vom Supportwert für die beiden Testdatenmengen mit $|T|$ gleich 10.

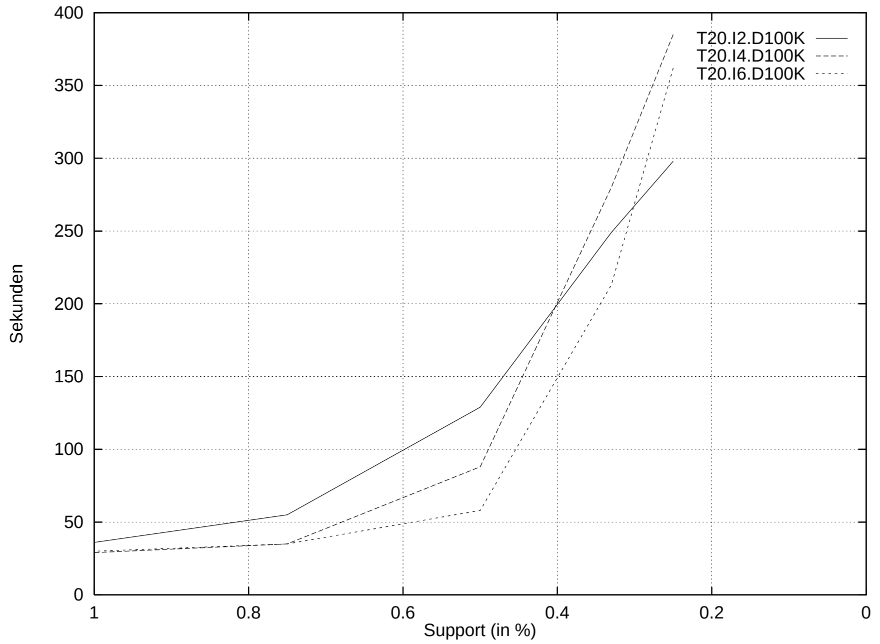


Abbildung 5.2: Abhängigkeit der Laufzeit vom Supportwert für die drei Testdatenmengen mit $|T|$ gleich 20

5.3 Tests mit reellen Daten

Die in den Abbildungen 5.3, 5.4 und 5.5 dargestellten Datenbanktabellen bilden einen Teilausschnitt aus einer Datenbank eines Drogeriemarktes. Die in der Tabelle in Abbildung 5.3 gespeicherten Daten wurden über alle Filialen (über 400 Filialen) und über alle Artikel in zwei Monaten gesammelt. Die Daten sind auf Wochenebene pro Filiale und Artikel verdichtet. Die Tabelle enthält insgesamt ca. 6 Millionen Zeilen.

Bewegungsdaten			
Spalte	Spaltenbezeichnung	Datentyp	Länge
FNR	Filialnummer	num	3
DAN	dm-Artikelnummer	num	5
BJ	Berichtsjahr	num	2
BM	Berichtmonat	num	2
BW	Berichtswoche	num	2
AKZ	Aktionskennzeichen	alpha	1
Menge	Menge	num	7
Umsatz	Umsatz	num	11,2

Abbildung 5.3: Die Datenbanktabelle *Bewegungsdaten*.

Artikel			
Spalte	Spaltenbezeichnung	Datentyp	Länge
...	...	...	...
DAN	dm-Artikelnummer	num	5
...	...	...	...
WG	Warengruppe	num	3
Marke	Markennummer	num	3
...	...	...	...

Abbildung 5.4: Die Datenbanktabelle Artikel.

Warengruppen			
Spalte	Spaltenbezeichnung	Datentyp	Länge
WB	Warenbereichsnummer	num	2
WG	Warengruppennummer	num	3
WGKBEZ	Warengruppenbezeichnung	alpha	12

Abbildung 5.5: Die Datenbanktabelle *Warengruppen*.

Mit Hilfe dieser Datenbanktabellen wurden zwei Datenmengen von Transaktionen generiert, auf die der Apriori-Algorithmus angewendet wurde:

1. Eine Transaktion bestand aus allen Artikelnummern DAN, die in einer bestimmten Filiale FNR in einer Berichtswoche BW eines Berichtsmonat BM eines Berichtsjahrs BJ verkauft wurden.

2. Eine Transaktion bestand aus den unterschiedlichen Warengruppenbezeichnungen WGKBEZ der Artikel, die in einer Filiale FNR in einer Berichtswoche BW eines Berichtsmonats BM eines Berichtsjahrs BJ verkauft wurden.

Es ergeben sich für die beiden Datenmengen, die beide ca. 80 MBytes groß sind, folgende Parameter:

	1. Datenmenge	2. Datenmenge
Anzahl Transaktionen	3702	3509
max. Transaktionslänge	5560	289
durschschnittl. Transaktionslänge	3485	232
Anzahl Items	14920	398

Der Apriori-Algorithmus wurde auf der 1. Datenmenge mit einem minimalen Supportwert von 94% ausgeführt. Die Laufzeit betrug für dieses Beispiel 6 Tage und 8 Stunden, wobei 7 Iterationen durchgeführt worden sind. Hier lag die relativ lange Laufzeit darin begründet, daß die Transaktionen sehr lang waren.

Beim Test der 2. Datenmenge, die ebenfalls relativ lange Transaktionen enthält, lag bei einem minimalen Supportwert von 86% nach der 3. Iteration des Apriori-Algorithmus folgende Situation vor: Aus den 182 Large 1-Itemset wurden 16471 Candidate 2-Itemsets generiert. In einer Zeit von ca. 7 Minuten wurden die Supportwerte ermittelt und aus den Candidate 2-Itemsets 16366 Large 2-Itemsets selektiert, aus denen 970594 Candidate 3-Itemsets erzeugt wurden. Hier dauerte die Supportwertermittlung knapp 10 Stunden. Insgesamt wurden 969451 Large 3-Itemsets herausgefiltert. Daraus enstand eine externe Datei mit einer Größe von 340 MBytes, in der die Candidate 4-Itemsets zur weiteren Bearbeitung ausgelagert worden sind. Darüberhinaus wurden die Large 2-Itemset und Large 3-Itemset in externe Dateien ausgelagert, um Platz zur Bearbeitung der Candidate 4-Itemsets zu schaffen. Dieser Programmverlauf läßt sich aus der Gleichverteilung der Ausgangsdaten erklären.

5.4 Vergleich hauptspeicher- mit externspeicherbasierter Kandidatenverarbeitung

Um die beiden Implementierungsalternativen, hauptspeicher- und externspeicherbasierte Kandidatenverarbeitung, zu testen, werden nach dem auf Seite 68 beschriebenen Verfahren zur Generierung synthetischer Datenmengen drei Testdatenmengen mit den folgenden Parametern generiert:

Name	$\|T\|$	$\|I\|$	$\|\mathcal{D}\|$	Größe in MByte
T10.I4.D10K	10	4	10K	0.5
T10.I4.D5000K	10	4	5000K	240

Für die beiden Testdatenmengen, T10.I6.D10K und T10.I4.D5000K, wurde der Support derart gewählt, daß die Kandidatenmenge C einmal eine relativ kleine und ein anderes Mal eine relativ große Kardinalität $|C|$ aufwies, aber in beiden Fällen nicht komplett im Hauptspeicher gehalten werden konnte. Es ergaben sich dann für die Implementierungsalternative, die die Kandidatenmenge mit Hilfe von externem Hashing verarbeitet (EX_HASH), und für die Implementierungsalternative, die die Kandidatenmenge stückweise im Hauptspeicher mit internem Hashing verarbeitet und die Zwischenergebnisse in einem Präfix-B^+-Baum abspeichert (B_BAUM), die folgenden Laufzeiten:

T10.I6.D10K	$\|C\|$ klein	$\|C\|$ groß
EX_HASH	28 Min., 26 Sec.	58 Min., 12 Sec.
B_BAUM	1 Min., 30 Sek.	2 Min., 39 Sek.

T10.I4.D5000K	$\|C\|$ klein	$\|C\|$ groß
EX_HASH	ca. 13 Std.	-
B_BAUM	50 Min., 54 Sek.	2 Std., 21 Min., 46 Sek.

Bei der Variante EX_HASH ergaben sich in den hier durchgeführten Experimenten insgesamt schlechtere Laufzeiten, weil im Gegensatz zur Variante B_BAUM ein ständiger I/O-Zugriff auf die Blöcke der Hashdatei nötig war. Die Hashdatei für den Fall, T10.I4.D5000K, EX_HASH und $|C|$ klein, hatte beispielsweise eine Größe von 168 MBytes. Aufgrund der deutlich schlechteren Laufzeit gegenüber der Variante B_BAUM wurde auf den Test - T10.I4.D5000K, EX_HASH und $|C|$ groß - verzichtet.

Insgesamt lieferte also die Implementierungsvariante B_BAUM die besseren Ergebnisse, obwohl mehrmals über die Transaktionen gescannt werden mußte, aber die Verarbeitung der Candidate Itemsets im Gegensatz zur Implementierungsalternative EX_HASH komplett hauptspeicherbasiert vonstatten ging.

Kapitel 6

Schluß

Die in dieser Arbeit vorgestellte Implementierungsalternative, wobei die Candidate Itemsets ausschließlich im Hauptspeicher verarbeitet und Zwischenergebnissse in Form von Large Itemsets in einen Präfix-B^+-Baum abgespeichert werden, erlaubt die Verarbeitung einer beliebig großen Zahl von Candidate Itemsets, die lediglich durch die Größe des zur Verfügung stehenden Filesystems beschränkt wird. Es hat sich gezeigt, daß diese Implementierungsalternative der stückweisen Verarbeitung einer Menge zu testender Candidate Itemsets, die in ihrer Gesamtheit nicht komplett im Hauptspeicher gehalten werden kann, gegenüber der Verarbeitung der Candidate Itemsets am Stück Laufzeitvorteile mit sich bringt, denn die Supportwerte der Candidate Itemsets werden ausschließlich im Hauptspeicher gezählt. Ein schreibender I/O-Zugriff ist nicht vorgesehen.

Das Problem bei dieser Implementierungsalternative ist, daß die Transaktionen in einer Iteration des Algorithmus für eine Candidatenmenge, die nicht komplett im zur Verfügung stehenden Hauptspeicher gehalten werden kann, mehrfach gescannt werden müssen. Die Lösung für dieses Problem könnte beispielsweise so aussehen, daß man versucht, für die Menge der Candidate Itemsets eine kompakte Repräsentation zu finden, um möglichst viele Candidate Itemsets im Hauptspeicher zu verarbeiten.

Die Regeln, die dieses System entdeckt, können einem darauf aufbauenden System als Eingabe dienen, um beispielsweise eine Analyse aller zuvor entdeckten Assoziationsregeln vorzunehmen.

Literaturverzeichnis

[Agrawal et al., 1993] Agrawal, R., Imielinski, T., und Swami, A. (1993). Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Washington, D. C.

[Agrawal et al., 1996] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., und Verkamo, A. I. (1996). Fast Discovery of Association Rules. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., und Uthurusamy, R., Hrsg., *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science, Kapitel 12, Seiten 277–296. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England.

[Agrawal und Srikant, 1994] Agrawal, R. und Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Data Bases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, Seiten 478–499, Santiago, Chile.

[Bayer und Unterauer, 1977] Bayer, R. und Unterauer, K. (1977). Prefix B-Trees. *ACM Transaktions on Database Systems*, 2(1):11–26.

[Brachman und Anand, 1996] Brachman, R. J. und Anand, T. (1996). The Process of Knowledge Discovery in Databases: A Human-Centered Approach. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., und Uthurusamy, R., Hrsg., *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science, Kapitel 2, Seiten 33–51. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England.

[Breiman et al., 1984] Breiman, L., Friedman, J., Olshen, R., und Stone, C. (1984). *Classifikation and regression trees.* Wadsworth and Brooks, Pacific Grove.

[Brockhausen und Morik, 1996] Brockhausen, P. und Morik, K. (1996). Direct Access of an ILP Algorithm to a Database Management System. In Pfaringer, B. und Fürnkranz, J., Hrsg., *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming (ILP for KDD)*, MLnet Sponsored Familiarization Workshop, Seiten 95–110, Bari, Italy.

[Cheeseman und Stutz, 1996] Cheeseman, P. und Stutz, J. (1996). Bayesian Classification (AutoClass): Theory and Results. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., und Uthurusamy, R., Hrsg., *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science, Kapitel 2, Seiten 153–180. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England.

[Chen et al., 1996] Chen, M.-S., Han, J., und Yu, P. S. (1996). Data Mining: An Overview from a Database Perspektive. *IEEE Transaktions On Knowledge and Data Engineering*, 8(6).

[Comer, 1979] Comer, D. (1979). The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137.

[Cooper und Herskovits, 1992] Cooper, G. F. und Herskovits, E. (1992). A Bayesian Method for the Induction of Probabilistic Networks from Data. *Machine Learning*, 9:309–347.

[Cormen et al., 1990] Cormen, T. H., Leiserson, C. E., und Rivest, R. C. (1990). *Introduction to Algorithms*. The MIT Press.

[Dörfler und Peschek, 1988] Dörfler, W. und Peschek, W. (1988). *Einführung in die Mathematik für Informatiker*. Hanser Verlag München Wien.

[Elder und Pregibon, 1996] Elder, J. F. und Pregibon, D. (1996). A Statistical Perspective on Knowledge Discovery in Databases. In Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., und Uthurusamy, R., Hrsg., *Advances in Knowledge Discovery and Data Mining*, Seiten 75 – 101. The MIT Press, Cambridge, Massachusetts.

[Enbody und Du, 1988] Enbody, R. J. und Du, H. C. (1988). Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(2):85–113.

[Fagin et al., 1979] Fagin, R., Nivergelt, J., Pippenger, N., und Strong, H. R. (1979). Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–344.

[Fayyad et al., 1996] Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., und Uthurusamy, R., Hrsg. (1996). *Advances in Knowledge Discovery and Data Mining*. AAAI Press Series in Computer Science. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England.

[Feller, 1968] Feller, W. (1968). *An Introduction to Probability Theory and its Applications*, Jgg. I. John Wiley & Sons New York.

[Frawley et al., 1992] Frawley, W., Piatetsky-Shapiro, G., und Matheus, C. (1992). Knowledge Discovery in Databases: An Overview. *AI Magazine*, 14(3):57–70.

[Gonnet und Baeza-Yates, 1990] Gonnet, G. H. und Baeza-Yates, R. (1990). *Handbook of Algorithms and Data Structures*. Addison-Wesley Publishing Company, 2. Auflage.

[Houtsma und Swami, 1993] Houtsma, M. und Swami, A. (1993). Set-oriented mining of association rules. Research Report 9567. Technical report, IBM Almaden Research Center, San Jose, California.

[Keim und Kriegel, 1996] Keim, D. A. und Kriegel, H.-P. (1996). Visualization Techniques for Mining Large Data Bases: A Comparison. *IEEE Transactions On Knowledge and Data Engineering*, 8(6):923–938.

[Knuth, 1973] Knuth, D. E. (1973). *The Art of Computer Programming*, Jgg. 3. Addison-Wesley Publishing Company.

[Larson, 1978] Larson, P. A. (1978). Dynamic Hashing. *BIT*, 18(2):184–201.

[Larson, 1980] Larson, P. A. (1980). Linear Hashing with partial Expansions. In *Proceedings of the 6th Conference on Very Large Data Bases*, Seiten 224–232. Montreal.

[Larson, 1988] Larson, P. A. (1988). Dynamic Hash Tables. *Communications of the ACM*, 31(4):448–457.

[Lockemann et al., 1993] Lockemann, P. C., Krüger, G., und Krumm, H. (1993). *Telekommunikation und Datenhaltung*. Carl Hanser Verlag München Wien.

[Lum et al., 1971] Lum, V. Y., Yuen, P. S. T., und Dodd, M. (1971). Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files. *Communications of the ACM*, 14(4):228–239.

[Morik und Brockhausen, 1997] Morik, K. und Brockhausen, P. (1997). A Multi-strategy Approach to Relational Knowledge Discovery in Databases. *Machine Learning Journal*, 27(3):287–312.

[Ng und Han, 1994] Ng, R. und Han, J. (1994). Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proceedings of the International Conference on Very Large Data Bases*, Seiten 144–155, Santiago, Chile.

[Nijenhuis und Wilf, 1978] Nijenhuis, A. und Wilf, H. S. (1978). *Combinatorical Algoritms for Computers and Calculators*. Academic Press, 2. Auflage.

[Ottmann und Widmayer, 1990] Ottmann, T. und Widmayer, P. (1990). *Algorithmen und Datenstrukturen.* BI Wissenschaftsverlag Reihe Informatik.

[Park et al., 1995] Park, J. S., Chen, M.-S., und Yu, P. S. (1995). An Effective Hash-Based Algorithm for Mining Association Rules. In *Proceedings of the ACM SIGMOD*, Seiten 175–186, San Jose, California, USA.

[Quinlan, 1990] Quinlan, J. (1990). Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5 Programs for Machine Learning.* Morgan Kaufmann, San Mateo.

[Ramamohanarao und Sacks-Davis, 1984] Ramamohanarao, K. und Sacks-Davis, R. (1984). Recursive linear Hashing. *ACM Transactions on Database Systems*, 9(3):369–391.

[Seltzer und Yigit, 1991] Seltzer, M. und Yigit, O. (1991). A New Hashing Package for UNIX. *USENIX, Winter '91, Dallas, TX.*