Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Integration eines hochperformanten Apriori-Operators in einer Hauptspeicherdatenbank

Maximilian E. Schüle

Masterarbeit im Elitestudiengang Software Engineering

Software Engineering
Elite Graduate Program


Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Integration eines hochperformanten Apriori-Operators in einer Hauptspeicherdatenbank

Matrikelnummer:	1399777
Beginn der Arbeit:	15. September 2016
Abgabe der Arbeit:	15. März 2017
Erstgutachter:	Prof. Alfons Kemper, Ph.D.
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	Linnea Passing, M.Sc. w. honours

Software Engineering
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 27. Juli 2017 Maximilian E. Schüle

Abstract

Die Speicherung hochdimensionaler Datensätze erfolgt effizient in Datenbanksystemen, jedoch ist deren Analyse nur mit Programmen außerhalb möglich und bedingt einen Transfer der Daten. Deshalb stellt die Integration von Algorithmen zur Assoziationsanalyse in Datenbanksysteme den nächsten Schritt dar. Kommerzielle Anbieter wie freie Implementierungen unterstützen bereits Data-Mining-Algorithmen in gewöhnlichen plattenbasierten Datenbanksystemen. Die Entwicklung von Hauptspeicherdatenbanken bietet neue Möglichkeiten, weshalb die vorliegende Arbeit einen Weg zeigt, einen hochperformanten Apriori-Operator in einer Hauptspeicherdatenbank zu integrieren.

Die Arbeit untersucht den Apriori-Algorithmus, einen Algorithmus im Bereich der Assoziationsanalyse, der aus Warenkorbdaten Assoziationsregeln generiert, um Korrelationen zwischen Elementen in Mengen zu finden und diese auf zukünftige Mengen zu projezieren. Die Arbeit beurteilt existierende Verbesserungen des Algorithmus auf ihre Effizienz im Zusammenspiel mit Hauptspeicherdatenbanken unter Berücksichtigung der Anfragezeit als zu optimierende Größe.

Mit einem daraus abgeleiteten Konzept, um einen Apriori-Algorithmus als Operator in das relationale Hauptspeicherdatenbanksystem HyPer einzubinden, wird ein Operator in zwei Versionen implementiert und evaluiert. Die Evaluation mit verschiedenen Datensätzen zeigt, dass ein Apriori-Algorithmus in paralleler Variante auf moderner Hardware mit Anzahl der Rechenkerne skaliert und eine effiziente Integration möglich ist. Außerdem ermutigt die Arbeit, weitere Algorithmen aus dem Bereich der Assoziationsanalyse in Hauptspeicherdatenbanken zu integrieren, da mit neuen Technologien unerforschte Leistungen möglich sind.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Der Apriori-Algorithmus ist ein Algorithmus zum Finden von Assoziationsregeln, also Regeln, die bei Auftreten von Elementen in einer Menge auf das Auftreten eines anderen Elementes schließen. Die Idee ist aus Warenkorbdaten, Daten über gleichzeitig gekaufte Produkte, die Assoziationsregeln

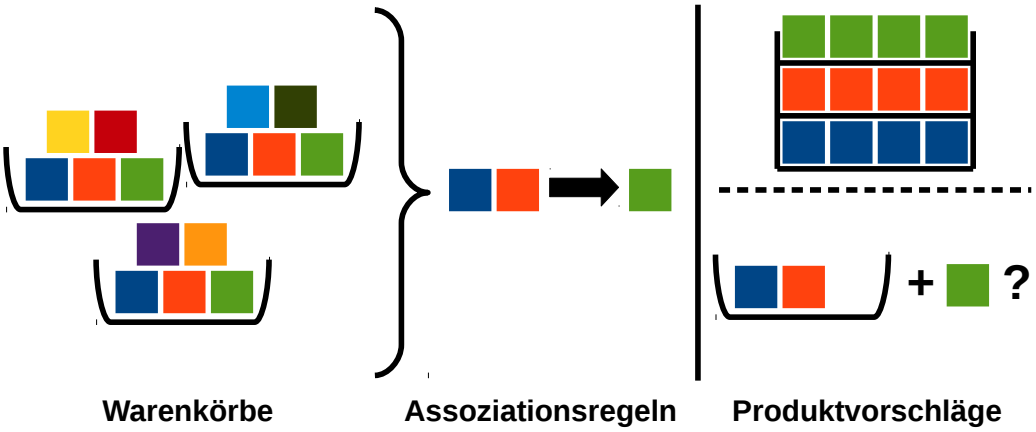

Abbildung 1.1.: Erzeugen von Assoziationsregeln aus Warenkorbdaten um Produkte vorzuschlagen

zu generieren (s. Abb. 1.1). Dazu nimmt man die anonymisierten Daten bisheriger Kunden, die in der relationalen Form aus Tupeln mit einem Transaktionsbezeichner (TID) und dem Produkt an sich bestehen, und gruppiert diese zu Warenkörben, also Mengen aus Produkten, die unter der gleichen TID gekauft sind. Darunter sind Mengen, die häufiger auftreten, ab einem gewissen Schwellenwert häufig auftretende Mengen von Elementen, Frequentitemsets, genannt. Wenn ein neuer Kunde nun ein Produkt kauft, was Teil eines Frequentitemsets ist, schlägt ein Algorithmus weitere Produkte des Frequentitemsets vor (bei einem Online-Warenhaus) oder es ist sinnvoll, häufig gekaufte Produkte in einem Warenhaus nebeneinander zu platzieren. Um auch die richtigen Produkte zu finden, existieren verschiedene Kennzahlen, die angeben, wie oft Produkte miteinander gekauft werden oder wie wahrscheinlich ein bestimmtes weiteres Produkt gekauft wird.

1.1. Motivation

Die Analyse hochdimensionaler Datensätze erfolgt mittels auf Data-Mining spezialisierten Anwendungen (s. Abb. 1.2), wobei die Datensätze effizient in Datenbanken gespeichert sind, was einen

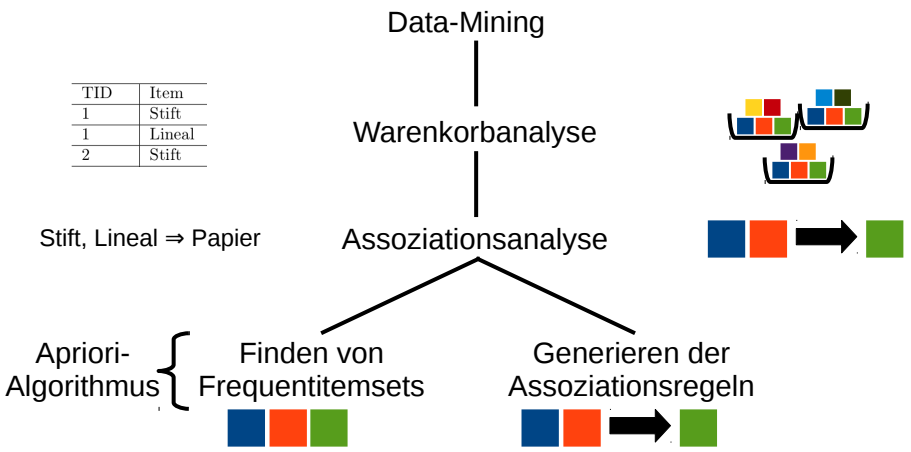

Abbildung 1.2.: Taxonomie zur Einordnung der Assoziationsanalyse

Transfer der Daten außerhalb der Datenbank und wieder zurück bedingt. Erste Ansätze kommerzieller Datenbanken integrieren nach dem Motto, nicht die Daten sondern den Algorithmus zu verschieben [3], Data-Mining-Algorithmen in relationale Datenbanksysteme, um mittels relationaler Anfragesprache die Analyse der Datensätze zu ermöglichen. Bei Hauptspeicherdatenbanken entfällt die Limitierung des Speichers, womit mehr Daten analysiert und, im Falle von Assoziationsregeln, mehr Regeln produziert werden können. Aus diesem Grund ist die Integration von Data-Mining-Algorithmen, wie des in dieser Arbeit beschriebenen Apriori-Algorithmus, notwendig. Das Ziel dieser Arbeit ist erstmals einen Apriori-Algorithmus mit den Vorteilen einer Hauptspeicherdatenbank zu verbinden, um die Daten effektiv und schnell zu bearbeiten und aus der Datenbank heraus die Assoziationsregeln zu generieren. Um den Vergleich zu anderen Apriori-Algorithmen herzustellen, sollen in einer Hauptspeicherdatenbank zwei verschiedene Versionen eines Apriori-Operator implementiert werden, deren Performanz nach Skalierung und Größe des Ergebnisses mit verschienden Datenbasen getestet wird. Anschließend vergleicht die Arbeit die Ergebnisse mit anderen Apriori-Algorithmen.

1.2. Vorgehen

Die vorliegende Arbeit führt im folgenden Kapitel zunächst Grundbegriffe im Bereich der Assoziationsregeln und Frequentitemsets ein, um anschließend die Generierung von Assoziationsregeln anhand des Apriori-Algorithmus vorzustellen. Anschließend erfolgt ein Überblick über Verbesserungen des Apriori-Algorithmus, unter Verwendung einer anderen Datenstruktur, verschiedenen Tricks, um Speicherplatz zu sparen, und die Möglichkeit der Parallelisierung, der mit alternativen Algorithmen zum Apriori-Algorithmus abschließt. Am Ende des Kapitels folgt eine Übersicht über die bisherige Nutzung des Apriori-Algorithmus in anderen Anwendungen. Das darauffolgende Kapitel liefert die Grundlagen über die Hauptspeicherdatenbank, in der der Apriori-Algorithmus integriert werden soll, die Datenbank HyPer vom Lehrstuhl für Datenbanksysteme der TU München. Dazu liefert das Kapitel Einblicke in die Systemarchitektur und Anfrageverarbeitung in HyPer, um anschließend zu erklären, wie andere Data-Mining-Algorithmen in dieser Hauptspeicherdatenbank integriert sind.

Die eigentliche Arbeit erläutert das vierte Kapitel über die Konzeption des zu integrierenden Algorithmus, sowohl die zugrundeliegende Datenstruktur, ein Präfixbaum um häufig auftretende Mengen zu speichern, wie der zweigeteilte Algorithmus an sich, der zuerst häufige Mengen sucht und daraus die Assoziationsregeln generiert. Der Konzeption folgt ein Kapitel über die Implementierungsdetails mit verwendeten Bibliotheken und der Anpassung an die Architektur der Hauptspeicherdatenbank HyPer.

Der Evaluation widmet sich ein eigenes Kapitel, das anhand diverser geeigneter Datensätze zeigen soll, dass eine Integration des Apriori-Algorithmus in einer Hauptspeicherdatenbank für verschieden große Ergebnisse gelingt und dass ein Apriori-Algorithmus mit der Anzahl der verfügbaren Rechenkerne skaliert. Im weiteren Ausblick soll die Arbeit ermutigen, die Integration von Data-Mining-Algorithmen zu fördern sowie andere Algorithmen zur Generierung von Assoziationsregeln in der Kombination mit Hauptspeicherdatenbanken zu erforschen, die bei ausreichend großem Hauptspeicher unerforschte Leistungen bringen.

2. Apriori-Algorithmus für Assoziationsregeln

Ein Algorithmus, um Assoziationsregeln zu erzeugen, ist der Apriori-Algorithmus, der sukzessive aus kleineren Frequentitemsets auf größere schließt und somit von vorherigen Mengen auf größere, also a priori (lat. *ab*: von; lat. *prior*: vorherig) Mengen ausschließt oder erweitert. Dieses Kapitel gliedert sich in fünf Teile: der erste Teil erläutert Grundbegriffe zu Assoziationsregeln, der zweite Teil den Apriori-Algorithmus. Darauf folgt ein Kapitel zu Verbesserungen des Algorithmus mit Ausblick auf verwandte Algorithmen zur Erzeugung von Assoziationsregeln und eines zur Komplexitätsabschätzung, um im letzten Teil auf existierende Anwendungen hinzuweisen, die Assoziationsregeln generieren.

2.1. Assoziationsregeln

Assoziationsregeln sind Folgerungen, um Korrelationen in einer Menge Elemente zu finden. Dabei sollen Regeln mit hoher Aussagekraft gefunden werden, sogenannte *starke Regeln*, deren Kennzahlen ein gewisses Minimum übersteigen und sich somit auf zukünfige Mengen übertragen lassen.

2.1.1. Einführung

Die Assoziationsanalyse ist ein Teilgebiet des Data Minings, um in großen Datenmengen Zusammenhänge verschiedener Elemente zueinander zu finden. Aus zusammengehörigen Elementen werden Implikationen abgeleitet, die sogenannten Assoziationsregeln: Wenn sich Objekt A im Warenkorb befindet, so liegt Objekt B mit bestimmter Wahrscheinlichkeit daneben. [21, S. 556] Eine Regel besteht aus zwei Teilmengen, einer Prämisse oder Antezedens genannt und einer Folgerung, der Konklusion. Mit einer gewissen Wahrscheinlichkeit sind die Elemente der Konklusion in einer Menge enthalten, wenn bereits die Elemente der Prämisse darin enthalten sind. Beispielsweise verwenden Warenhäuser Assoziationsregeln, um zu einem Produkt, das ein Kunde bereits im Warenkorb liegen hat, zugehörige Produkte zu bewerben. Ein Beispiel für ein Warenhaus könnte sein: „Wer einen Stift kauft, kauft auch ein Lineal."

TID	Item
1	Stift
1	Lineal
2	Stift
2	Lineal
2	Papier
3	Stift
3	Lineal

TID	Items
1	{Stift, Lineal}
2	{Stift, Lineal, Papier}
3	{Stift, Lineal}

Items
{Stift, Lineal}

Tabelle 2.1.: Beispiel einer Ausprägung, aggregierte Form, Frequentitemsets mit $\sigma(X) > 1$

Die Warenkorb-Analyse findet Assoziationsregeln basierend auf den Einkäufen bisheriger Kunden. Die Einkäufe werden anonymisiert, aber zusammengehörig in Warenkörben gespeichert, die eine Transaktionsnummer erhalten.

2.1.2. Grundbegriffe

Die Literatur trifft folgende Definitionen zur Bestimmung starker Assoziationsregeln: [24]

- **Items** $I = \{i_1, ..., i_m\}$: Elemente, zum Beispiel das Produkt $Stift$
- **Item-Menge** X: Menge aus verschiedenen Items, $X \subseteq I$, z.B. $X = \{Stift, Lineal\}$
- **k-Item-Menge** X_k: Item-Menge aus k Items, $X_k \subseteq I, |X_k| = k$
- **Transaktionen** $\mathcal{D} = \{T_1, ..., T_n\}$: Warenkörbe von Items, identifiziert durch $TID \in [n], T_i \subseteq I$
- Transaktion T **enthält** eine Item-Menge X gdw. $X \subseteq T$
- **Support-Anzahl** $\sigma(X) := |\{T | T \in \mathcal{D} \wedge X \subseteq T\}|$: Absolute Häufigkeit einer Item-Menge
- **Support** $s(X) := \frac{\sigma(X)}{|\mathcal{D}|}$: relative Häufigkeit einer Item-Menge
- **Schwellenwert** s_0: minimaler Support für eine große Item-Menge
- **absoluter Schwellenwert**: $\sigma_0 = s_0 * |\mathcal{D}|$: minimale absolute Häufigkeit
- Item-Menge X **häufig auftretend** gdw. $s(X) \geq s_0$
- **k-Frequentitemsets** $\mathcal{L}_k := \{L | L \subseteq I \wedge |L| = k \wedge s(L) \geq s_0\}$
- **Frequentitemsets** $\mathcal{L} := \bigcup_{k=1}^{\infty} \mathcal{L}_k = \{L | L \subseteq I \wedge s(L) \geq s_0\}$
- **Kandidatenmenge** $\mathcal{C}_k$: potentielle k-Frequentitemsets, $\mathcal{L}_k \subseteq \mathcal{C}_k \subseteq \{X | X \subseteq I \wedge |X| = k\}$
- **Assoziationsregel** $X \Rightarrow Y$ mit $X \subset T, Y \subset T, X \cap Y = \emptyset$, z.B. $Stift, Lineal \Rightarrow Papier$
- **Konfidenz** k einer Assoziationsregel $X \Rightarrow Y$: Häufigkeit der Transaktionen die Y enthalten, wenn diese bereits X enthalten: $k(X \Rightarrow Y) := P(Y|X) = \frac{\sigma(X \cup Y)}{\sigma(X)} = \frac{s(X \cup Y)}{s(X)}$
- **Vertrauenswert** k_0: minimale Konfidenz einer starken Assoziationsregel
- Assoziationsregel $X \Rightarrow Y$ eine **starke Assoziationsregel** gdw. $k(X \Rightarrow Y) \geq k_0$
- **Lift** bzw. **Interest**: Abweichung von der Unabhängigkeit, also wie relevant sich X auf das Eintreten von Y auswirkt [10]
  $lift(X \Rightarrow Y) := \frac{P(X \cup Y)}{P(X) * P(Y)} = \frac{s(X \cup Y)}{s(X) * s(Y)} = \frac{\sigma(X \cup Y) * |D|}{\sigma(X) * \sigma(Y)}$
- **Conviction**: Abweichung der negierten Assoziationsregel von der Unabhängigkeit
  $conv(X \Rightarrow Y) := \frac{P(A) * P(\neg B)}{P(A \backslash B)} = \frac{1 - s(X)}{1 - k(X \Rightarrow Y)}$

2.2. Apriori

Zum Finden von Assoziationsregeln stellt Rakesh Agrawal 1993 erstmals einen Algorithmus vor [1], den er später als Apriori-Algorithmus näher definiert [2].

2.2.1. Algorithmus zum Erzeugen von Assoziationsregeln

In der ersten Veröffentlichung schlägt Agrawal vor, Warenkorbdaten für Unternehmensfragen zu nutzen und regt an, Datenbanksysteme um eine Funktion zur Erzeugung von Assoziationsregeln zu erweitern. [1]

Definition 1. ***Warenkorbdaten*** *sind Daten über miteinander gekaufte Elemente pro Einkauf oder pro Kunde, die Warenkörbe oder die Transaktionen, die anonymisiert aber zusammenhängend abgelegt sind. Sie sind entweder gruppiert abgespeichert oder verfügen über einen Bezeichner, der Transaktions-ID (TID), mit der sich verschiedene Elemente zu einem Warenkorb zuordnen lassen.*

Beispiel 1. *Ein Schema für Warenkorbdaten in relationaler Form ist das Schema Transaktionen: {[TID, Item]}*

Definition 2. *Ein* ***Frequentitemset*** *($L \in \mathcal{L}$), auch genannt* ***häufig auftretende*** *oder* ***große*** *Item-Menge (engl.: large itemset), ist eine Menge von Elementen, genannt* ***Items****, die in einer bestimmten Anzahl von Warenkörben gemeinsam auftreten. Ein Frequentitemset heißt* ***maximal****, wenn keine Supermenge davon ein Frequentitemset ist.*

Definition 3. *Der* ***Support*** *ist die relative Häufigkeit einer Item-Menge, die sich als Quotient aus der absoluten Häufigkeit einer Item-Menge zur Anzahl aller Transaktionen berechnet. Die absolute Häufigkeit gibt an, in wie vielen Transaktionen eine Item-Menge enthalten ist. Ein beliebig definierter Schwellenwert, der minimale Support, ist der Punkt, ab welchem eine Item-Menge zu einem Frequentitemset wird.*

Sein vorgeschlagener Algorithmus generiert in jedem Schritt Item-Mengen, die möglicherweise häufig auftreten, die Kandidaten. Pro Iteration vergleicht er alle Transaktionen mit jedem Kandidaten. Der Algorithmus generiert die Kandidaten, für die er einen hohen Support erwartet, aus Frequentitemsets, also Item-Mengen, mit einem Support über dem Schwellenwert. Denn „[w]enn eine Kandidaten-Item-Menge X einen niedrigen erwarteten Support aufweist, so ist keine Erweiterung $X + I_j$ eine Kandidaten-Item-Menge“ [1], das spätere Apriori-Prinzip.
Der Algorithmus schätzt zuerst den Support, im nächsten Schritt zählt er ihn. Dabei werden alle möglichen k-Item-Mengen berücksichtigt, sofern deren Teilmengen einen hohen Support erwarten. Von Item-Mengen mit geringem erwarteten Support überprüft er nur die kleinste Item-Menge. Wenn diese häufig auftritt, so überprüft er deren Erweiterungen in den nächsten Schritten. Darin kommt das Apriori-Lemma zu tragen: Wenn eine Item-Menge einen niedrigen Support aufweist, so sind auch alle Erweiterungen nicht häufig auftretend.
Eine gute Schätzfunktion schätzt den Support einer Item-Menge zu über 96 % richtig ein. [1] Eine Überlegung, den Algorithmus zu verbessern, ist, bereits im Voraus nach Frequentitemsets zu filtern, die bestimmte Items enthalten müssen. Dennoch bleibt das Problem exponentiell in Abhängigkeit der Anzahl an Items, enthält jede Transaktion alle m Items, so ist Potenzmenge mit 2^m Frequentitemsets zu bilden, bei n Transaktionen sind $2^m * n$ Vergleiche notwendig.

2.2.2. Verbesserung als Apriori-Algorithmus

Ein Jahr später verfeinert Agrawal den Algorithmus zum Finden von Assoziationsregeln als Apriori-Algorithmus [2], indem in jedem Iterationsschritt die Item-Mengen um genau ein Item jeweils wachsen. Dazu teilt er den Algorithmus in zwei Teilprobleme, dem Finden von häufig auftretenden Item-Mengen und dem Ableiten der Assoziationsregeln daraus.

Frequentitemsets

Die häufig auftretenden Item-Mengen, die Frequentitemsets, erzeugt der Algorithmus in jeder Iteration aus den um ein Element kleineren Mengen, bis alle gefunden sind. Zuerst zählt der Apriori-Algorithmus die Häufigkeit der einzelnen Items, also den Support der 1-Item-Mengen, filtert diese, dass nur die häufig auftretenden übrig bleiben, und nutzt diese Frequentitemsets als Basis für die Iteration. In jeder Iteration bildet er zuerst die Kandidatenmenge $\mathcal{C}_k$ der aus um ein Element erweiterten k-1-Frequentitemsets der vorherigen Iteration (`apriori-gen()`). Anschließend durchlaufen alle Transaktionen die generierten Item-Mengen, ist eine in der Transaktion enthalten, so wird der Support-Zähler einer Item-Menge inkrementiert. Die Item-Mengen mit einem Support größer als der Schwellenwert s_0 qualifizieren sich für die nächste Iteration. Sobald eine Iteration keine Frequentitemsets mehr produziert, terminiert die Schleife. Frequentitemsets sind alle Item-Mengen jeder Iteration mit einem Support höher als s_0 (s. Algorithmus 1).

$\mathcal{L}_1 = \{\text{large 1 itemsets}\}$;
for $(k = 2; \mathcal{L}_{k-1} \neq \emptyset; k++)$ **do**
  $\mathcal{C}_k = \text{apriori-gen}(\mathcal{L}_{k-1})$;
  forall $t \in \mathcal{D}$ **do**
    $\mathcal{C}_t = \text{subset}(\mathcal{C}_k, t)$;
    forall $c \in \mathcal{C}_t$ **do**
      $c.\text{count}++$;
    end
  end
  $\mathcal{L}_k = \{c \in \mathcal{C}_k | c.\text{count} \geq \sigma_0\}$;
end
return $\bigcup_k \mathcal{L}_k$;

Algorithmus 1 : Apriori-Algorithmus

Nach dem Apriori-Prinzip [29] ist der Support einer Item-Menge Y maximal genauso groß wie der Support ihrer Teilmengen. Daraus lassen sich die Apriori-Eigenschaft [18], wonach alle Teilmengen von Frequentitemsets mindestens genauso häufig auftreten, und deren Umkehrung, die Anti-Monotonie der Apriori-Heuristik [17], ableiten.

Definition 4. ***Apriori-Prinzip:*** *Der Support einer Item-Menge ist maximal so groß wie der Support ihrer Teilmengen.*

$$\forall X, Y \subseteq I.(X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

Definition 5. ***Apriori-Eigenschaft:*** *Wenn Y ein k-Frequentitemset ($Y \in \mathcal{L}_k$) ist, so ist jedes $X \subseteq Y$ mit $k' := |X|$ ein k'-Frequentitemset.*

Definition 6. ***Anti-Monotonie der Apriori-Heuristik:*** *Wenn eine k-Item-Menge kein Frequentitemset ist, so ist keine Übermenge mit $k+1$ Items ein Frequentitemset.*

Beweis 1. *Für $X, X', Y \subseteq I$ mit $Y = X \cup Y'$ folgt aus der Definition des Supports:*

$$\frac{\sigma(X \cup X')}{|\mathcal{D}|} = s(X \cup X') \leq s(X) = \frac{\sigma(X)}{|\mathcal{D}|} \tag{2.1}$$

$$\sigma(X \cup X') \leq \sigma(X) \tag{2.2}$$

$$|\{T|T \in \mathcal{D} \wedge X \cup Y \subseteq T\}| \leq |\{T|T \in \mathcal{D} \wedge X \subseteq T\}| \tag{2.3}$$

$$\{T|T \in \mathcal{D} \wedge X \cup Y \subseteq T\} \subseteq \{T|T \in \mathcal{D} \wedge X \subseteq T\} \tag{2.4}$$

Die letzte Formel folgt aus

$$\forall S \in \{T|T \in \mathcal{D} \wedge X \cup Y \subseteq T\}.X \subseteq S \Rightarrow S \in \{T|T \in \mathcal{D} \wedge X \subseteq T\}$$

und aus dem Apriori-Prinzip folgt die Apriori-Eigenschaft

$$s(X \cup X') \geq s_0 \Rightarrow s(X) \geq s_0$$

und damit auch die Umkehrung

$$s(X) < s_0 \Rightarrow s(X \cup X') < s_0$$

Die Kandidaten für die nächste Iterationen generiert `apriori-gen()` in zwei Schritten basierend auf den Frequentitemsets der vorherigen Iteration. Zunächst erzeugt `apriori-join()` aus um ein Element verschiedenen k-1-Frequentitemset die k-Item-Mengen (1):

$$\mathcal{C}'_k = \{X \cup X'.X, X' \in \mathcal{L}_{k-1} \wedge |X \cap X'| = k-1\}$$

Oder in SQL ausgedrückt entspricht die Formel folgender Abfrage:

insert into $\mathcal{C}_k$
select $p.item_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}$
from $\mathcal{L}_{k-1}$ p, $\mathcal{L}_{k-1}$ q
where $p.item_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$;

Algorithmus 2 : Kandidatengenerierung (1)

Anschließend entfernt `apriori-prune()` die k-Item-Mengen, deren Teilmengen keine Frequentitemsets sind (2):

$$\mathcal{C}_k = \{X \in \mathcal{C}'_k.X' \subset X \wedge |X'| = k-1 \Rightarrow X' \in \mathcal{L}_{k-1}\}$$

forall *itemsets* $c \in \mathcal{C}_k$ **do**
  forall *(k-1)-subsets* s *of* c **do**
    if($s \notin L_{k-1}$) **Delete** c from $\mathcal{C}_k$;
  end
end

Algorithmus 3 : Aussortieren der Kandidaten (2)

Die Generierung der Kandidaten lässt sich als Netz darstellen (jede Ebene entspricht einer Iteration, die Knoten entsprechen den Item-Mengen mit Kanten zu den Item-Mengen, aus denen `apriori-gen()` sie generiert). Darin lässt sich die Anti-Monotonie der Apriori-Heuristik ablesen: Sobald eine Item-Menge einen zu geringen Support aufweist, grau markiert, so sind alle darauf aufbauenden Item-Mengen keine Kandidaten für ein Frequentitemset (s. Abb. 2.1). Für jede Transaktion liefert die Funktion `subset()` eine Menge der Item-Mengen, die in der Transaktion enthalten sind, um deren Häufigkeit zu inkrementieren. Am Ende bilden die Item-Mengen mit einem Support höher als der Schwellenwert die Frequentitemsets. Alle Frequentitemsets sind gefunden (s. Tab. 2.2), wenn die k-te Iteration keine weiteren Frequentitemsets mehr produziert (alle weiteren Knoten im Gitter scheiden aus) oder wenn die Menge mit allen Items erreicht ist (alle Items sind Teil eines Frequentitemsets).

TID	Items
1	{Stift, Lineal}
2	{Stift, Lineal, Papier}
3	{Stift, Lineal}
4	{Lineal, Papier}

Iteration	Item-Menge	Support
1	{Stift}	3
1	{Lineal}	4
1	{Papier}	2
2	{Stift, Lineal}	3
2	*{Stift, Papier}*	1
2	{Lineal, Papier}	2
3	~~{Stift, Lineal, Papier}~~	

Tabelle 2.2.: Apriori-Algorithmus mit $\sigma_0 = 2$: `subset()` ermittelt Häufigkeit und disqualifiziert Item-Mengen (*kursiv*), `apriori-prune()` streicht Item-Mengen

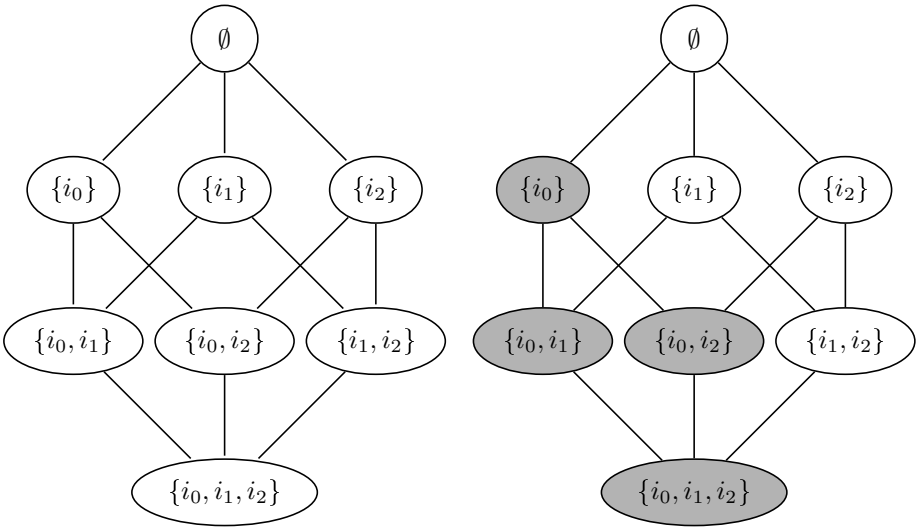

Abbildung 2.1.: Generierung der potentiellen Frequentitemsets (Kandidaten-Item-Mengen)

Assoziationsregeln

Definition 7. *Eine **Assoziationsregel** $X \Rightarrow Y$ beschreibt eine Korrelation zwischen Elementen. Sie besteht aus einer Menge von Items auf der linken Seite (X), auch **Prämisse** oder **Antezedens** genannt, und einer nicht leeren Menge auf der rechten Seite (Y), auch **Folge** oder **Konsequenz** genannt. Eine Assoziationsregel besagt, wenn eine Menge alle Elemente der linken Seite enthält, dann enthält diese mit einer Wahrscheinlichkeit, genannt **Konfidenz**, die Elemente der rechten Seite.*

Aus den Frequentitemsets $L \in \mathcal{L}$ entstehen die Assoziationsregeln $\mathcal{R}$. Aus jeder nicht leeren Teilmenge eines Frequentitemsets entsteht eine Assoziationsegel mit dieser Teilmenge als Prämisse und den verbleibenden Elementen als Folge. [1]

$$\mathcal{R} = \{X \Rightarrow Y | \exists L \subset \mathcal{L}. X \subseteq L \wedge Y = L \setminus X \wedge Y \neq \emptyset\}$$

Eine Assoziationsregel ist eine **starke** Assoziationsregel ($(X \Rightarrow Y) \in \mathcal{R}^+$), wenn ihre Konfidenz

höher als der entsprechende Schwellenwert ist:

$$k(X \Rightarrow Y) = \frac{s(X \cup Y)}{s(X)} \geq k_0 \Rightarrow (X \Rightarrow Y) \in \mathcal{R}^+$$

Somit sind zwei Kennzahlen für eine starke Assoziationsregel nötig: der minimale Support s_0, den Item-Mengen erreichen müssen, um Frequentitemsets zu sein, und die minimale Konfidenz k_0, die Assoziationsregeln erreichen müssen, um starke Assoziationsregeln zu sein. Mit Ermittlung aller starken Assoziationsregeln terminiert der Apriori-Algorithmus.

Die Assoziationsregeln lassen sich ebenfalls als Abhängigkeitsgraph skizzieren (s. Abb. 2.2). Da die Konfidenz einer Assoziationsregel steigt, umso mehr Elemente die Prämisse enthält, stünde die Regel mit der leeren Folge analog zum Abhängigkeitsnetz der Frequentitemsets an der Spitze, denn sie hat die höchste Konfidenz (analog zum höchsten Support), aber sie ist keine gültige Assoziationsregel.

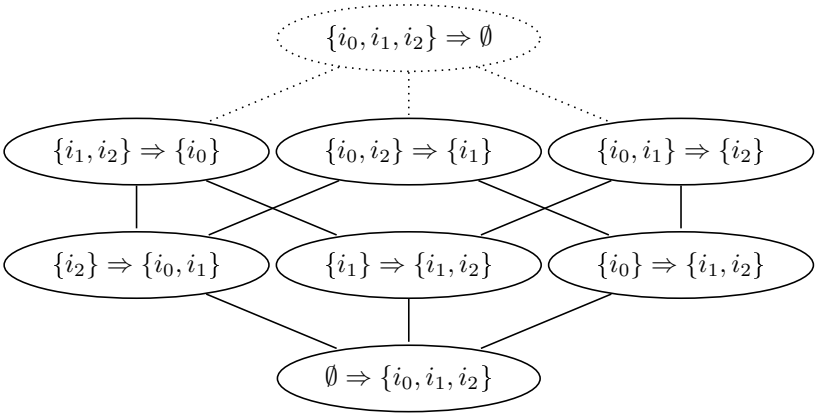

Abbildung 2.2.: Abhängigkeitsnetz der Assoziationsregeln

Das Abhängigkeitsnetz ist möglich, weil der Support der Item-Mengen einer Assoziationsregel deren Konfidenz bestimmt und sie steigt durch Vergrößerung der Prämisse, da der Support einer Item-Menge mit Erweiterung um Elemente zunimmt.

Definition 8. ***Apriori-Prinzip der Assoziationsregeln:*** *[21] Sei L ein Frequentitemset und seien X, X^+, Y, Y^- Teilmengen mit $X \cup Y = X^+ \cup Y^- = L, X \cap Y = \emptyset, X^+ \cap Y^- = \emptyset$. So steigt die Konfidenz einer Assoziationsregel, wenn mehr Items in die Prämisse aufgenommen werden: $\forall X \subseteq X^+, Y^- \subseteq Y.\ k(X^+ \Rightarrow Y^-) \geq k(X \Rightarrow Y)$*

Beweis 2. *Das Apriori-Prinzip wirkt sich auf Assoziationsregeln aus:*

$$c(X \Rightarrow Y) = \frac{s(X \cup Y)}{s(X)} = \frac{s(L)}{s(X)} \leq \frac{s(L)}{s(X^+)} = \frac{s(X^+ \cup Y^-)}{s(X^+)} = c(X^+ \Rightarrow Y^-) \tag{2.5}$$

$$s(X^+) \leq s(X) \tag{2.6}$$

Zweite Gleichung entspricht mit $X \subseteq X^+$ dem Apriori-Prinzip, womit es bewiesen ist.

Dementsprechend lassen sich die Regeln analog zur Kandidatengenerierung für Frequentitemsets erzeugen, aus zwei Regeln $(X \Rightarrow Y, X' \Rightarrow Y')$ desselben Frequentitemsets $(X \uplus Y = X' \uplus Y')$, deren Konklusionen Y, Y' um ein Element verschieden sind $(\exists_1 y \in Y.y \notin Y' \wedge \exists_1 y' \in Y'.y' \notin Y)$, erzeugen. So erzeugen die Regeln $i_1, i_5 \Rightarrow i_2, i_3, i_4$ und $i_1, i_4 \Rightarrow i_2, i_3, i_5$ die Regel $i_1 \Rightarrow i_2, i_3, i_4, i_5$.

AprioriTID

Eine Verbesserung des Apriori-Algorithmus, genannt AprioriTID [2], sieht vor, dass nur die relevanten Transaktionen die jeweiligen generierten Item-Mengen durchlaufen. Dazu referenzieren oder speichern die Item-Mengen die zugehörigen Transaktionen und reichen sie weiter. Wenn die Transaktionen unterschiedliche oder sogar disjunkte Item-Mengen enthalten, läuft der Algorithmus schneller, benötigt jedoch mehr Speicher.

$L_1 = \{\text{large 1 itemsets}\}$;
$\overline{\mathcal{C}_1} = \mathcal{D}$;
for $(k = 2; L_{k-1} \neq \emptyset; k++)$ **do**
  $\mathcal{C}_k =$apriori-gen$(\mathcal{L}_{k-1})$;
  $\overline{\mathcal{C}_k} = \emptyset$;
  forall $t \in \overline{\mathcal{C}_{k-1}}$ **do**
    $\mathcal{C}_t = \{c \in \mathcal{C}_k | (c - c[k]) \in t.\text{setOfItemsets} \wedge (c - c[k-1]) \in t.\text{setOfItemsets}\}$;
    forall $c \in \mathcal{C}_t$ **do**
      $c.\text{count}++$;
    end
    if $\mathcal{C}_t \neq \emptyset$ **then**
      $\overline{\mathcal{C}_k} += < t.TID, C_t >$;
    end
  end
  $L_k = \{c \in \mathcal{C}_k | c.\text{count} \geq \sigma_0\}$;
end
return $\bigcup_k \mathcal{L}_k$;

Algorithmus 4 : AprioriTid-Algorithmus

Der Algorithmus speichert in der Menge $\overline{C}$ die Items einer Transaktion als Menge von Mengen (s. Tab. 2.3), um effizient Kombinationen aus um ein Element größeren Item-Mengen daraus zu generieren.

TID	Items
1	{Stift, Lineal}
2	{Stift, Lineal, Papier}
3	{Stift, Lineal}

TID	Items
1	{{Stift}, {Lineal}}
2	{{Stift}, {Lineal}, {Papier}}
3	{{Stift}, {Lineal}}

TID	Items
1	{{Stift, Lineal}}
2	{{Stift, Lineal}, {Stift, Papier}, {Lineal,Papier}}
3	{{Stift, Lineal}

Tabelle 2.3.: Hilfstabellen für AprioriTID: $\mathcal{D}$, $\overline{C_1}$, $\overline{C_2}$

2.3. Verwandte Arbeiten

Basierend auf dem beschriebenen Apriori-Algorithmus schlagen verschiedene Veröffentlichungen Verbesserungen vor, die in diesem Unterkapitel vorgestellt werden. Erste wichtige Verbesserung ist die Nutzung eines Präfixbaumes, um schnell vergleichen zu können, ob Transaktionen Item-Mengen enthalten. Zweitgenannte Verbesserungen optimieren die Speicherplatzausnutzung, wenn der Haupt-

speicher nicht ausreicht. Weitere Veröffentlichung beziehen sich auf die Möglichkeit, den Apriori-Algorithmus parallel auszuführen. Anschließend folgt die Vorstellung verwandter Algorithmen der Assoziationsanalyse, des FP-Growth-Algorithmus und des Eclat-Algorithmus.

2.3.1. Präfixbäum zur Reduktion der Vergleiche

Um die Frequentitemsets schnell zu finden, hilft die Reduktion der Kandidaten (mit m Items maximal 2^m potentielle Item-Mengen) wie bei dem Apriori-Prinzip, die Reduktion der Anzahl an Transaktionen n wie im AprioriTID-Algorithmus sowie die Reduktion der Vergleiche (maximal $n * 2^m$ Vergleiche). [29] Ferenc Bodon sieht Verbesserungsmöglichkeiten des Algorithmus vor allem in der Datenstruktur und den Implementierungsdetails. [4] Effektive Datenstrukturen wie Präfixbäume helfen, die Anzahl der Vergleiche pro Transaktion konstant zu halten. Die Knoten der Präfixbäume entsprechen den Items und speichern den zugehörigen Support (s. Abb. 2.3), die Pfade entsprechen den Item-Mengen (anstatt, wie bei Präfixbäumen üblich, den Sequenzen von Wörtern). Mit jeder Iteration wächst der Präfixbaum um eine Ebene, die Pfade bis zu einer Ebene k entsprechen k-Item-Mengen. Wenn der Support eines Knotens der Ebene k größer als der Schwellenwert ist, so entspricht der Pfad dahin einem k-Frequentitemset. [8]

Knoten sind entweder Vektoren fester Größe unter Verwendung der Items als Index, Vektoren variabler Größe mit expliziter Speicherung der Items oder Hashtabellen. Hashtabellen und Vektoren fester Größe erlauben Zugriff in konstanter Zeit, benötigen jedoch mehr Speicher, da sie selten komplett gefüllt sind. Vektoren variabler Größe besitzen minimale Länge, benötigen zusätzlich Speicher für die Items und erlauben Zugriff bei binärer Suche nur in logarithmischer Zeit. Bei Implementierungen, die auf die erwartete Performanz achten und der Speicherverbrauch in den Hintergrund tritt, ist die erste Variante mit Vektoren fester Größe zu empfehlen. [8]

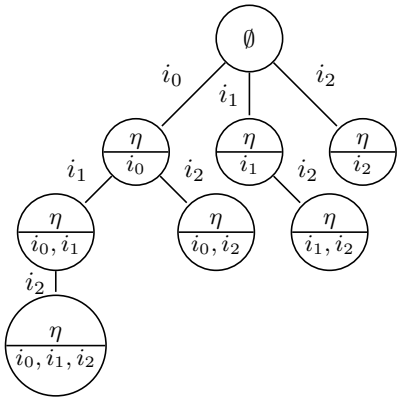

Abbildung 2.3.: Größter duplikatfreier Präfixbaum bei drei verschiedenen Items (Supportzähler η)

Pro Iteration durchläuft jede Transaktion als Item-Menge den Baum. Dabei rufen die jeweiligen Knoten rekursiv ihre relevanten Kindknoten mit der um das kleinste Element gekürzten Item-Menge auf, der Knoten der letzten Ebene inkrementiert seinen Support-Zähler. Anschließend folgt der rekursive Aufruf identisch für alle Teilmengen der Transaktion (s. Abb. 2.4 und Abb. 2.5). Wenn die Transaktionen bereits nach Items sortiert sind, so entspricht der rekursive Aufruf der vorne um eins gekürzten Item-Menge. [5] Außerdem reduziert sich die Anzahl an Durchläufen pro Iteration, wenn gleiche Transaktionen zu einer gewichtigen Transaktion zusammengefasst sind und der Supportzähler um den Betrag des Gewichts erhöht wird.

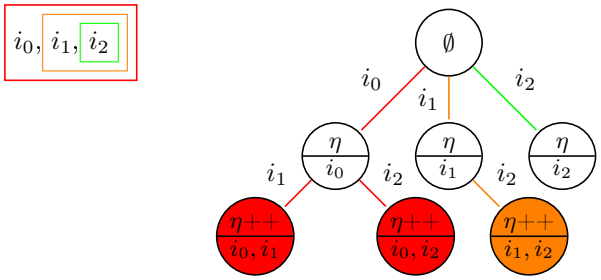

Abbildung 2.4.: Transaktion $\{i_0, i_1, i_2\}$ in der zweiten Iteration

Jeder innere Knoten kennt beim Abstieg seinen Support, ist dieser zu gering, bricht er gemäß der Anti-Monotonie der Apriori-Heuristik ab und schließt damit Item-Mengen von Unterpfaden wegen zu niedrigen Supports aus. Eine weitere Verbesserung ist, Transaktion, die weniger als k Items enthalten, bei der k-ten Iteration auszuschließen. [6]

Weitere Vorteile bei der Nutzung von Präfixbäumen als Datenstruktur sind [4]:

- die schnelle Generierung der Kandidatenmengen,
- die schnelle Generierung der Assoziationsregeln, da eine schnelle Bestimmung des Supports von Item-Mengen möglich ist,
- dass nur eine Datenstruktur notwendig ist und
- dass eine leichte Ermittlung der *negativen Grenze* [4] möglich ist, wenn der Support zu gering ist.

Sowohl lineare als auch binäre Suche finden Elemente im Präfixbaum mit Vektoren variabler Länge wieder. Lineare Suche ist schneller [4], wenn stellvertretend für die Items *Frequency Codes* stehen. Items werden nach ihrer Häufigkeit sortiert und kodiert, das häufigste erhält die kleinste Zahl. So sind Treffer im ersten Teil des Vektors wahrscheinlicher. Eine allgemeine Kodierung, *Item Coding*, verringert den benötigten Speicher bei Vektoren fester Größe, indem eine dichte Kodierung Lücken vermeidet.

2.3.2. Speicherausnutzung

Wenn der Hauptspeicher klein ist, kann der Apriori-Operator nicht ausschließlich im Hauptspeicher ausgeführt werden, da er entweder zu viele Item-Mengen erzeugt oder die Transaktionen zu viele sind, als dass sie zusammen in den Hauptspeicher passen. Deshalb zielen nachfolgende Algorithmen auf die Reduktion des Speicherverbrauches ab.

PCY-Algorithmus

Der Algorithmus von Park, Chen und Yu (PCY-Algorithmus) nutzt freibleibenden Speicher aus, um die Anzahl der Kandidaten für Frequentitemsets zu reduzieren. [30] Die erste Phase der ersten Iteration zählt die Häufigkeit einzelner Items und fügt Paare aus 2-Item-Mengen in denselben Korb

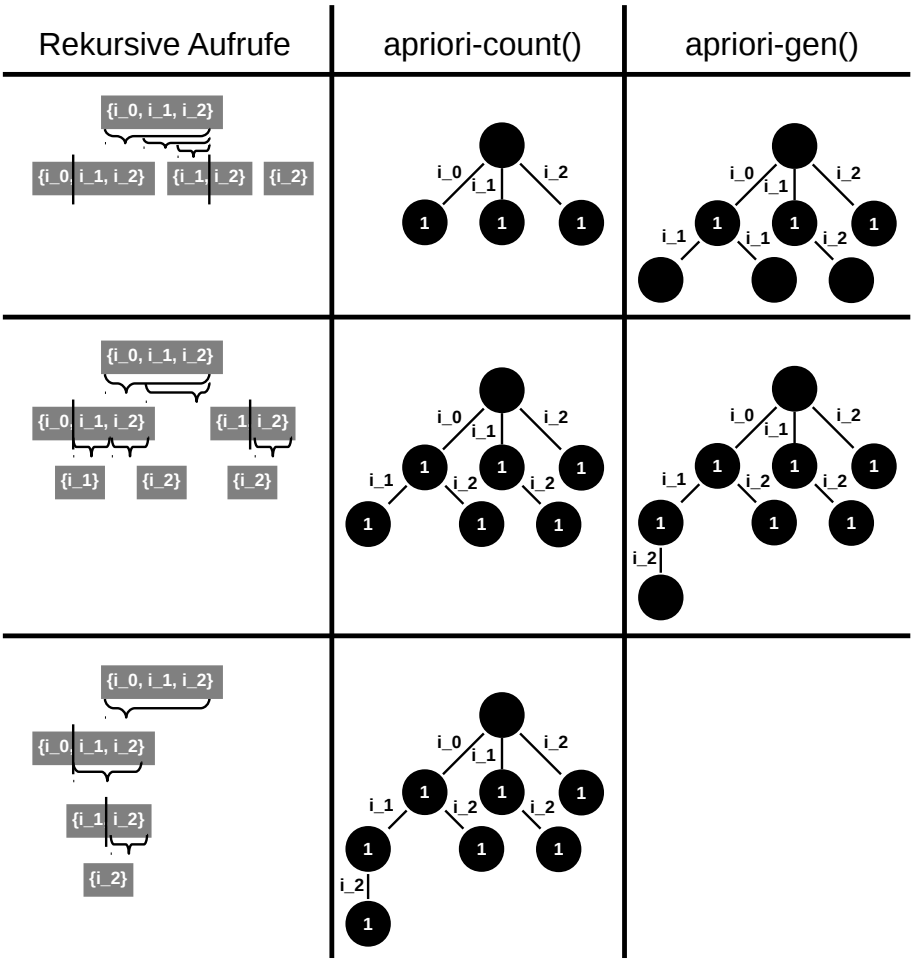

Abbildung 2.5.: Zählen der Häufigkeit und Generieren der Kindknoten

einer Hashtabelle ein und zählt im selben Schritt die Häufigkeit der Paare mit selbem Hashwert (s. Algorithmus 5).

forall $T \in D$ **do**
  forall $i \in T$ **do**
    i.count++;
  end
  forall $i_1, i_2 \in T$ **do**
    hashmap$[i_1, i_2]$++;
  end
end

Algorithmus 5 : PCY-Algorithmus, erste Phase

Aus der Hashtabelle entsteht eine Bitmap, die für jeden Hashwert aus zwei verschiedenen Items angibt, ob die Häufigkeit größer als der Schwellenwert ist (hashmap$[i_1, i_2] \geq s_0$) und es sich bei dem

Paar um ein Frequentitemset handeln kann. Die zweite Phase zählt nur die 2-Item-Mengen (die Kandidaten), in denen jedes Item an sich häufig auftritt (analog zum Apriori-Algorithmus) und bei denen es sich laut Bitmap um ein Frequentitemset handeln kann. Der Algorithmus ist eine Verbesserung bei Datensätzen mit einem hohen Anteil einelementiger Item-Mengen und dementsprechend geringerem Anteil zweielementiger Item-Mengen.

Multistage-Algorithmus

Der Multistage-Algorithmus [11] verbessert den PCY-Algorithmus, indem er durch eine zweite Hashmap die Anzahl möglicher Kandidaten für Frequentitemsets weiter reduziert. Eine Phase nach der ersten fügt alle Paare in die zweite Hashmap ein, die laut der ersten häufig auftreten können (s. Abb. 2.6). Die dritte Phase zählt dann die 2-Item-Mengen, in denen sowohl jedes Item an sich

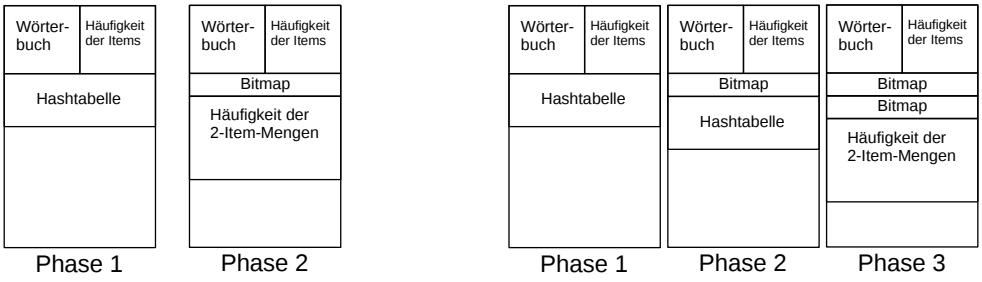

Abbildung 2.6.: PCY-Algorithmus (links) im Vergleich zu Multistage-Algorithus

häufig auftritt als auch bei denen es sich laut der ersten Bitmap (beide Algorithmen stimmen darin überein) **und** zusätzlich auch laut der zweiten Bitmap um ein Frequentitemset handeln kann. Ein Nachteil des Algorithmus ist die weitere Phase, um alle Transaktionen durchzugehen, der Vorteil ist der geringere Speicherplatzverbrauch, weil er weniger Kandidaten erzeugt. Alternativ kann die zweite Phase beide Hashtabellen füllen auf die Gefahr hin, dass die zweite Hashtabelle die Größe der ersten erreicht.

Toivonen-Algorithmus

Der Toivonen-Algorithmus erzeugt die Kandidaten für Frequentitemsets aus einem Teil aller Transaktionen und erzeugt daraus die negative Grenze. Anschließend zählt er mit allen Warenkörben die Häufigkeit der Item-Mengen, sowohl die der negativen Grenze wie die der Kandidaten, und ermittelt so die Frequentitemsets. [36]

Definition 9. *Die **negative Grenze** ist die Menge an Item-Mengen, die selbst nicht häufig auftreten, aber deren alle direkte Teilmengen (mit einem Item weniger) Frequentitemsets sind.*

Beispiel 2. *Seien 5 Items gegeben (i_0, i_1, i_2, i_3, i_4) mit Frequentitemsets $\{i_0\}, \{i_1\}, \{i_2\}, \{i_3\}, \{i_0, i_2\}$, so besteht die negative Grenze aus den Elementen $\{i_4\}$ mit der leeren Menge als direkte, häufig auftretende Teilmenge und den Item-Mengen $\{i_0, i_1\}, \{i_0, i_3\}, \{i_1, i_2\}, \{i_1, i_3\}, \{i_2, i_3\}$.*

Der geeignete Ausschnitt der Daten mit f Transaktionen berechnet sich wie folgt, um für eine Item-Menge X gute Ergebnisse zu erzielen ($\epsilon \leq 1, w \leq 1$ sind zwei Werte, zum Einstellen der Genauigkeit): [40]

$$f = \frac{-2ln(w)}{s(X)\epsilon^2}$$

Der Algorithmus arbeitet mit einer Teilmenge der Transaktionen bei der Kandidatengenerierung und mit allen Transaktionen zur Ermittlung der Häufigkeit. Somit werden nur korrekte Frequentitemsets gefunden, aber bei einem schlecht gewählten Ausschnitt werden mögliche Frequentitemsets bei der Kandidatengenerierung übergangen, sogenannte falsche negative sind möglich.

2.3.3. Parallele und verteilte Apriori-Algorithmen

Um den Apriori-Algorithmus zu parallelisieren, sieht ein Ansatz die Aufteilung der Transaktionen $\mathcal{D}$ auf mehrere identische Präfixbaume vor. Jeder Thread erhält $\frac{|\mathcal{D}|}{|Threads|}$ Transaktionen, die er in einer Iterations in seinen Präfixbaum einfügt. Nach jeder Iteration werden die Bäume synchronisiert, dabei ist ein Präfixbaum der Master, dem alle weiteren Präfixbäume (Slaves) die ermittelten Häufigkeiten berichten. Der Master generiert die Kandidaten (`apriori-gen()`) und synchronisiert die Häufigkeiten der Item-Mengen sowie die Kandidaten mit den Slaves, die anschließend mit der nächsten Iteration fortfahren (s. Abb. 2.7). [18]

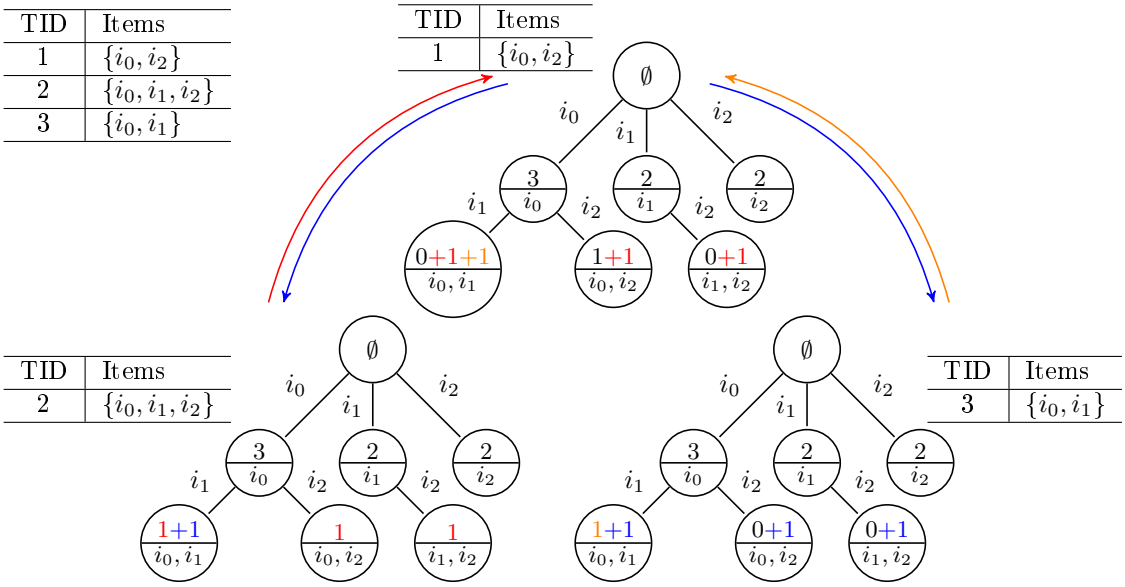

Abbildung 2.7.: Parallele Verarbeitung der Transaktionen und Synchronisation

Weitere existierende verteilte Apriori-Algorithmen nutzen das Hadoop MapReduce-Framework. Das Programmiermodell sieht eine Map-Phase vor, die Paare aus Element und einem Wert bildet, und eine Reduce-Phase, die auf den Paaren Funktionen aufruft. Eine Möglichkeit ist, den Algorithmus auf drei Map-Reduce-Schritte aufzuteilen: Der erste Schritt kombiniert 1-Item-Mengen mit derer Häufigkeit (Map) und filtert solche mit zu geringem Support heraus (Reduce). Der zweite Schritt erzeugt Frequentitemsets, indem er rekursiv die Kandidatenmenge produziert (Map) und nur die

häufig auftretenden weiterreicht (Reduce). Der letzte Schritt erzeugt alle Assoziationsregeln (Map) und filtert anhand deren Konfidenz die relevanten heraus (Reduce). Auf einem System funktionert der Algorithmus, auf verteilten Systemen sind aufgrund von fehlender Sichtbarkeit von Variablen bisher keine Tests erfolgreich gewesen. [41]

Ein theoretischer Ansatz verwendet zwei Map-Reduce-Schritte, einen um die 1-Item-Mengen zu generieren, einen um die weiteren Frequentitemsets zu erzeugen. Auch hier generiert die Map-Phase jeweils die Item-Menge, die Reduce-Phase filtert diese nach dem Apriori-Prinzip, sodass die Frequentitemsets übrig bleiben. Die Generierung der Item-Mengen erfolgt iterativ. Die theoretische Studie berechnet einen Geschwindigkeitsvorteil um die Anzahl verwendeter Knoten zur Parallelisierung. Die Laufzeit skaliert mit der Anzahl der Kerne p und hängt ab von der Anzahl der Transaktionen n, der Anzahl der Items m und der Größe der Frequentitemsets mit $t >> k$ und $m >> k$ [38]:

$$\mathcal{O}(k^3 + \frac{k * n * m}{p}) = \mathcal{O}(\frac{k * n * m}{p})$$

Eine weitere Möglichkeit, von lokalen auf globale Frequentitemsets zu schließen, zerteilt die Datenbasis in gleiche Teile:

$$\mathcal{D} = \mathcal{D}_1 \uplus \mathcal{D}_2 \uplus \cdots \uplus \mathcal{D}_p$$

Sie basiert auf der Annahme, dass ein Frequenitemset zur Datenbasis $\mathcal{D}$ in mindestens einer ihrer Partition $\mathcal{D}_k$ häufig auftritt. So liegt der globale Support einer Item-Menge zwischen dem Minimum und Maximum des lokalen Support der Item-Menge in dessen Partitionen, für einen Beweis sei auf die Literatur verwiesen [18]:

$$\forall X \subseteq I. min(s_{\mathcal{D}_1}(X), \ldots, s_{\mathcal{D}_p}(X)) \leq s(X) \leq max(s_{\mathcal{D}_1}(X), \ldots, s_{\mathcal{D}_p}(X))$$

Dadurch entfällt die Synchronisation am Ende jeder Iteration, dafür muss der lokale minimale Support einer Partition berechnet werden.

2.3.4. Systematik und verwandte Algorithmen

Eine Systematik kategorisiert die Algorithmen in Bezug auf die Suchstrategie und der Strategie zur Ermittlung der Häufigkeit zur Bestimmung der Frequentitemsets. [19] Als Suchstrategie ist entweder eine Breitensuche (engl.: *breadth-first search*, BFS), die k-1-Item-Mengen vor den k-Item-Mengen bestimmt, oder eine Tiefensuche (engl.: *depth-first search*, DFS), die Item-Mengen einer Baumstruktur folgend einfügt, möglich. Die Strategien, um die Häufigkeit zu ermitteln, sind entweder das Zählen der Transaktionen, die eine Item-Menge enthalten, oder die Schnittmengenbildung von Transaktionsvektoren, die zu einer Transaktion angeben, welche Items sie enthält (s. Tab. 2.4).

	Zählend	Schnittmengen
BFS	Apriori	SON
DFS	FP-Growth	Eclat

Tabelle 2.4.: Systematik der Algorithmen zur Assoziationsanalyse

2.3.5. FP-Growth

Aus nur zwei Datenbank-Scans erzeugt der Frequent-Pattern-Growth-Algorithmus (FP-Growth) alle Frequentitemsets, der die gesamte Datenbasis in einem Präfixbaum, Frequent-Pattern-Tree (FP-Tree), speichert. Anstelle iterativ Kandidatenmengen zu generieren und mit der Datenbasis abzugleichen erzeugt FP-Growth einen mit jeder eingelesenen Transaktion wachsenden Baum (Growth).

Im ersten Durchlauf misst er das Auftreten aller 1-Item-Mengen, streicht alle Item-Mengen mit zu geringem Support und sortiert die Items nach Häufigkeit, damit häufige in der Ordnung vorne erscheinen. Im zweiten fügt er pro Transaktion einen Pfad in den Präfixbaum ein, der seine vollständige Tiefe bereits im zweiten Durchlauf erreicht, wobei er Transaktionen als Ganzes und nicht deren Teilmengen separat speichert. Jeder Knoten entspricht einem Item und speichert dessen Häufigkeit, ein Pfad entspricht einer Item-Menge. Mehrere gleiche Transaktion teilen sich denselben Pfad und inkrementieren jeweils die Häufigkeit der Items auf diesem Pfad, Transaktionen mit demselben Präfix teilen sich einen Pfad und zweigen vor dem ersten unterschiedlichen Item ab. Um nun doch alle Teilmengen zu berücksichtigen, hat jeder Knoten einen Zeiger auf den nächsten Knoten für dasselbe Item eines anderen Pfades, eine Tabelle referenziert jeweils den ersten eingefügten Knoten (s. Abb. 2.8). [17] Die Projektion des Präfixbaumes erzeugt die Frequentitemsets zu einem gewünschten Item i. Wurzel des projezierten Baumes ist der von der Tabelle referenzierte Knoten. Da von diesem Referenzen zu allen weiteren Knoten mit Item i ausgehen, spannt dieser einen Baum auf, der alle Frequentitemsets enthält. Die Projektion erfordert, je nach angewandtem Algorithmus, im Baum doppelt-verkettete Elemente. [7] Die Assoziationsregeln entstehen analog zum Apriori-Algorithmus.

TID	Items
1	$\{i_0, i_1, i_2\}$
2	$\{i_0, i_1, i_2\}$
3	$\{i_0, i_1\}$
4	$\{i_0, i_2\}$
6	$\{i_0\}$
7	$\{i_1\}$
8	$\{i_2\}$

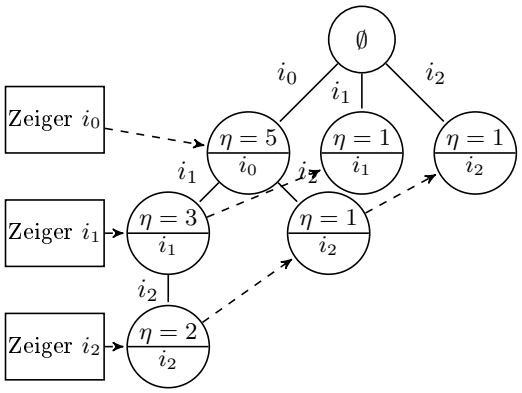

Abbildung 2.8.: FP-Tree bei gegebenen Transaktionen

Der FP-Growth-Algorithmus ist im Vorteil, wenn der minimale Support gering ist und somit sehr viele Item-Mengen häufig autreten. Dann benötigt der Apriori-Algorithmus mehrere Iterationen, der FP-Growth-Algorithmus nur zwei Schritte. Außerdem entfällt beim FP-Growth-Algorithmus die Kandidatengenerierung und somit überprüft dieser keine nicht existenten Item-Mengen, die in keiner Transaktion enthalten sind. Der Nachteil des FP-Growth-Algorithmus ist dessen hoher Speicherverbrauch. Im zweiten Schritt wird der Baum gebaut, der alle Item-Mengen enthält. Bei zu vielen verschiedenen Items und voneinander unterschiedlichen Transaktionen würde der Baum zu groß um in den Hauptspeicher zu passen. Somit ist bei wenig zu erwartetenden, kleinelementigen Frequentitemsets der Apriori und bei vielen Frequentitemsets der FP-Growth der geeignetere Algorithmus. [16]

2.3.6. Eclat

Der 1997 vorgestellte Equivalence-Class-Transformation-Algorithmus (Eclat) erzeugt aus Äquivalenzklassen Frequentitemsets, deren Support er durch Schnittmengenbildung über verschiedene Transaktionen ermittelt. [39] Äquivalenzklassen enthalten zu einer Item-Menge das letzte Item jeweils einer Item-Menge, mit der sie dasselbe Präfix teilt und nur um das letzte Item verschieden ist. Die ersten

Äquivalenzklassen entstehen aus 2-Frequentitemsets.

Definition 10. *Die Äquivalenzklasse zur Menge X ist definiert als* $[X] = \{Y[k] | X[1:k-1] = Y[1:k-1]\}$.

Beispiel 3. $L_2 = \{\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{2,3\},\{2,5\},\{2,7\},\{3,4\},\{3,5\}\}$
Daraus folgen die Äquivalenzklassen: $[1] = \{2,3,4,5\}$, $[2] = \{3,5,7\}$ *und* $[3] = \{4,5\}$

Aus den generierten Äquivalenzklassen lassen sich die Frequentitemsets bestimmen, jede Äquivalenzklasse $[i]$ entspricht dem maximalen potentiellen Frequentitemset, das i enthält. Alle Teilmengen, die i enthalten, sind ebenso Kandidaten, doch ein Support größer als der Schwellenwert macht sie erst zu einem Frequentitemset. Die Ermittlung des Supports beginnt entweder bei der maximalen Item-Menge und endet bei den minimalen (Bottom-Up) oder umgekehrt (Top-Down), ein Präfixbaum indexiert die Item-Mengen. Im Gegensatz zum Apriori-Algorithmus, der die Transaktionen *horizontal* ablegt, sodass die Transaktionen aus einer Kennung und ihren Items bestehen, speichert der Eclat-Algorithmus die Transaktionen *vertikal* ab, sodass jedes Item i seine zugehörigen Transaktionen im Vektor $t(i)$ kennt. Die Schnittmenge über die zugehörigen Transaktionen aller Items einer Item-Menge X ($\bigcap_{i \in X} t(i)$) enthält alle Transaktionen, die X enthalten, die Anzahl dieser Transaktionen entspricht der Support-Anzahl der Item-Menge $\sigma(X)$ (s. Tab. 2.5).

	$t(i_0)$	$t(i_1)$	$t(i_2)$	$t(i_0) \cap t(i_1)$	$t(i_0) \cap t(i_2)$	$t(i_1) \cap t(i_2)$	$t(i_0) \cap t(i_1) \cap t(i_2)$
T_0	1	1	1	1	1	1	1
T_1	0	1	1	0	0	1	0
T_2	0	0	1	0	0	0	0
σ	1	2	3	1	1	2	1

Tabelle 2.5.: Supportermittlung durch Schnittmenge der zugehörigen Transaktionsvektoren

2.3.7. SON-Algorithmus

Der Algorithmus von Savasere, Omiecinski und Navathe [33] teilt die Transaktionen in kleinere Partitionen auf und sucht darin nach Frequentitemsets. Der Algorithmus basiert auf der Idee, dass eine Item-Menge für alle Transaktionen nur häufig auftreten kann, wenn sie auch in mindestens einer Teilmenge häufig auftritt. Pro Iteration werden für jede Partition die lokalen Frequentitemsets bestimmt und anschließend zu globalen vereint und der globale Support ermittelt. Die Idee, Transaktionen auf mehrere Partitionen aufzuteilen, eignet sich auch für eine Parallelisierung des Apriori-Algorithmus.

2.4. Komplexitätsabschätzung

Die obere Grenze für die Laufzeit des Apriori-Algorithmus entspricht dem naiven Ansatz (Brute Force). Er berechnet bei m Items im ersten Schritt, der Generierung der Frequentitemsets, 2^m mögliche Item-Mengen. Jede generierte Item-Menge wird mit jeder Transaktion verglichen, um zu überprüfen, in wie vielen sie enthalten ist. Die Kosten für einen Vergleich betragen maximal m (Anzahl der Items), folglich ist die Laufzeit bei n Transaktionen:

$$\mathcal{O}(2^m * m * n)$$

Zur Erzeugung der Assoziationsregeln kann ein Algorithmus für eine Prämisse k aus m Elementen auswählen, für die Konklusionen wählt er j aus den verbleibenden $m - k$ Elementen aus. Damit

folgen, ohne Berücksichtigung einer leeren Prämisse (k beginnt bei 1) und einer leeren Konklusion ($j = 1$, Prämisse lässt mind. ein Element übrig: k aus $m - 1$), so viele mögliche Assoziationsregeln aus m Items:

$$\sum_{k=1}^{m-1}(\binom{m}{k} * \sum_{j=1}^{m-k}\binom{m-k}{j}) = 3^m - 2^{m+1} + 1$$

Wenn bedingungslose Assoziationsregeln (leere Menge als Prämisse) ebenfalls möglich sein sollen, ergeben sich mehr mögliche Asszіationsregeln:

$$\sum_{k=0}^{m-1}(\binom{m}{k} * \sum_{j=1}^{m-k}\binom{m-k}{j}) = 3^m + 2^m$$

Die Idee des Apriori-Algorithmus ist, die Anzahl der möglichen Item-Mengen zu reduzieren, die auf ihre Häufigkeit überprüft werden sollen. Dadurch liegt die Laufzeit darunter und hängt vielmehr von der Anzahl generierter Kandidaten $\mathcal{C}$ mit $|\mathcal{C}| \leq 2^m$ ab. So werden nur die Kandidaten mit den Transaktionen verglichen ($|\mathcal{C}| * n$). Die Anzahl der Assoziationsregeln hängt von den gefundenen Frequentitemsets $\mathcal{L}$ mit $|\mathcal{L}| \leq |\mathcal{C}| \leq 2^m$ ab. Jede Assoziationsregel ($X \Rightarrow Y$) entsteht aus einem Frequentitemset ($L \in \mathcal{L}, L = X \uplus Y$), sodass die Anzahl generierter Assoziationsregeln von der Anzahl der Items in einem Frequentitemset abhängt, aus denen j für die Prämisse gewählt werden:

$$\sum_{j=1}^{|L|-1}\binom{|L|}{j} = 2^{|L|-1} - 1$$

Somit hängt die Anzahl generierter Assoziationsregeln von der Anzahl der erzeugten Frequentitemsets pro Iteration ab sowie von der Anzahl der Iterationen (bis zur letzten Iteration, in der mind. eine häufig auftretende Menge gefunden wird: $max(\{o|\mathcal{L}_o \neq \emptyset\})$). Die k-te Iteration erzeugt k-Item-Mengen und somit enthält ein Frequentitemset $L \in \mathcal{L}_k$ genau k Items. Die genaue Zahl generierter Assoziationsregeln beträgt:

$$\sum_{k=1}^{max(\{o|\mathcal{L}_o \neq \emptyset\})} |\mathcal{L}_k| * \sum_{j=1}^{k-1}\binom{k}{j} = \sum_{L \in \mathcal{L}}\sum_{j=1}^{|L|-1}\binom{|L|}{j} \leq 3^m - 2^{m+1} + 1$$

Die Laufzeit des Algorithmus hängt ebenso von der Anzahl der Kandidaten ab, jeder Kandidat wird mit jeder Transaktion einmal verglichen ($|\mathcal{C}| * n$), die Laufzeit verringert sich auf:

$$\mathcal{O}(|\mathcal{C}| * m * n)$$

Ein Präfixbaum reduziert die Anzahl der Vergleiche, sodass jede Transaktion nur die Item-Mengen mit demselben Präfix vergleicht. Die Anzahl der rekursiven Aufrufe pro Knoten hängt von der Größe der Transaktion t und von der Größe der zu testenden Item-Menge k ab; k entspricht initial dem Iterationsschritt und ist abhängig von der Position des Knotens im Baum, umso weiter unten desto kleiner die verbleibende Teilmenge. Der rekursive Aufruf erfolgt iterativ pro Knoten mit der um ein Element gekürzten Teilmenge, bis die verbleibende Item-Menge weniger als k Items enthält; somit beträgt die Anzahl rekursiver Aufruf pro Knoten $t - k + 1$. Jeder rekursive Aufruf hat wieder $(m') - (k - 1) + 1$ Aufrufe zur Folge, bis die Tiefe k erreicht ist. Somit lässt sich die Anzahl der Aufrufe als rekursive Formel beschreiben:

$$f(t,k) = \sum_{i=t-k+1}^{t-1} f(i, k-1) \tag{2.7}$$

$$f(t,0) = 1 \tag{2.8}$$

	generierte Kandidaten c	Laufzeit T (nur Frequentitemsets)
Brute-Force	$c = 2^m$	$T = 2^m * m * n$
Apriori	$m \leq c \leq 2^m$	$T = n * m * c$
AprioriTID	$m \leq c \leq 2^m$	$T \leq n * m * c$
Apriori (Präfixbaum)	$m \leq c \leq 2^m$	$T \leq n * m * c$

Tabelle 2.6.: Komplexitätsabschätzung und theoretischer Vergleich

2.5. Nutzung des Apriori-Algorithmus in Datenbanksystemen

Um hochdimensionale Datensätze ohne Umweg innerhalb des Datenbanksystems auf Assoziationsregeln hin zu analysieren, ist das mittels SQL nur unter Verwendung von rekursiven Konstrukten und Feldern als Erweiterung von SQL möglich. Oracle bietet seit 2003 den Apriori-Algorithmus als Funktionalität in der Datenbank an, das freie Apache MADlib Framework bietet diesen ebenso als Funktion aus der Datenbank heraus an. Eine Alternative zu integrierten Apriori-Algorithmen stellt das Paket `arules` für die Programmiersprache R dar, das verschiedene Data-Mining-Algorithmen beherrscht.

2.5.1. Verwendung bisheriger SQL-Syntax

Data-Mining-Algorithmen sind in keinem SQL-Standard enthalten [34], weshalb ein Apriori-Algorithmus auf Behelfskonstruktionen zurückgreifen muss, wenn er nicht als Funktion integriert ist. Die beschriebene Abfrage stellt einen Weg dar, Frequentitemsets mit einer PostgreSQL-Datenbank zu finden (s. Abb. 2.9).

Häufig auftretende 1-Item-Mengen lassen sich mit einer einfachen SQL-Abfrage durch Gruppierung nach Elementen und einer Aggregatsfunktion, die deren Auftreten zählt, bestimmen (`sales_supp`). Um alle Frequentitemsets zu finden, sind für die nächsten Iterationsschritte bereits Felder und eine Rekursion nötig: Die Größe des größtmöglichen Frequentitemsets entspricht maximal der Anzahl der unterschiedlichen Items, sie liegt allerdings meist darunter. Folglich müsste das Schema abhängig von der Ausprägung der Relation mit den Warenkorbdaten sein. Die einzige Möglichkeit, ein Schema unabhängig von der Ausprägung festzulegen, ist unter Verwendung von Feldern durch Verletzung der Normalform, denn diese entsprechen einem Attribut, das variabel viele Elemente enthalten kann. Die Rekursion ist nötig, da vorab die Anzahl der Iterationen nicht bekannt ist. Da die Rekursion in SQL nicht doppelt von derselben rekursiven Tabelle abhängen darf, entsteht die Kandidatengenerierung aus den bisheriegen Frequentitemsets um alle einelementigen Frequentitemsets erweitert.

In der Erweiterung von PostgreSQL gibt es einen eigenen Teilmengen-Operator ($\subseteq$ entspricht `<@`). Wenn die Transaktionen zu Warenkörben gebündelt vorliegen (`transactions`), kann auf diesen zurückgegriffen werden, um zu überprüfen, wie viele Transaktionen die gebildeten Item-Mengen enthalten.

Zusammenfassend stellt dies einen Weg dar, um Frequentitemsets zu finden, funktioniert allerdings nur mit Erweiterungen von SQL und ist sehr umständlich; weitere Informationen wie der Support, um daraus Assoziationsregeln zu generieren, müssen redundant abgefragt werden.

2.5.2. Oracle Data Mining

Oracle integriert den Apriori-Algorithmus als Funktion erstmals 2003 in seiner Datenbank `Oracle10g` [25], die für Grid-Computing entworfen ist, einer Form des verteilten Rechnens. Als Teil von Oracle Data Mining stellt das System eine Funktion bereit, um Frequentitemsets zu finden, aus denen

```
-- gegeben: Tabelle mit Warenkorbdaten
create table sales (tid integer, item text);

-- Erzeugung von Warenkoerben
create table transactions (tid integer, bucket text[]);
insert into transactions select tid, array_agg(item) from sales group by
    tid;

-- hauefig auftretende 1-Item-Mengen
create view sales_supp as (
   select item
   from sales
   group by item
   having count(*) >=2); --items with minSupp >= 2

with recursive frequentitemsets as(
      -- Basis: 1-Item-Mengen, L_1
      select distinct array[p.item]::text[] as items
      from sales_supp p
   union all
      select distinct array_append(t.items,p.item::text)::text[]
      -- apriori-gen(): L_{k-1} -> C_k
      from frequentitemsets t, sales_supp p
      where
         -- zaehle Support: c |in C_k. c.count >= 2 -> c |in L_k
         ( select count(*)
            from transactions
            -- c |in C_k, t |in D. c |subseteq t -> c |in subset(C_k,t)
            where array_append(t.items,p.item::text)::text[] <@ bucket )
                >= 2
         and t.items[cardinality(t.items)]<p.item)
select * from frequentitemsets;
```

Abbildung 2.9.: SQL-Abfrage um Frequentitemsets zu finden

```
#!/usr/bin/Rscript
# Paket
library("arules")
# Einlesen der Warenkorbdaten: Zeile = Warenkorb, durch Leerzeichen
    getreent
tr<-read.transactions("retail.dat",format="basket",sep="␣")
# Assoziationsregeln generieren
rules <- apriori(tr, parameter=list(supp=0.01, conf=0.1))
# Anzeigen
inspect(rules)
```

Abbildung 2.10.: R-Skript zur Bestimmung der Assoziationsregeln

sich Assoziationsregeln ableiten lassen. In der aktuellen Version seiner Datenbank *Oracle 12c* bietet die Komponente *Oracle Data Mining* als Teil der Option *Oracle Advanced Analytics* den Apriori-Operator als native SQL-Funktion an. Der Operator wird im Rahmen einer Reihe von anderen Data-Mining-Algorithmen angeboten, aber als einziger Algorithmus zur Warenkorbanalyse und zur Bestimmung von Frequentitemsets. Auch hier ist die Idee, anstatt die Daten zu exportieren und mit externen Programmen zu bearbeiten, die Algorithmen in der Datenbank anzubieten, um dort die Daten zu analysieren. [3]

2.5.3. Apache MADlib

Den gleichen Ansatz nur mit freier Software verfolgt Apache MADlib, um Analysen auf hochdimensionalen Datensätzen innerhalb einer Datenbank ausführen zu können. Technisch nimmt der als Funktion `assoc_rules()` implementierte Apriori-Operator Warenkorbdaten als eine Relation an (mit den Attributen Transaktions-ID und Items), generiert daraus die Assoziationsregeln und gibt pro Regel ein Tupel mit Prämisse, Konklusion und den Kennzahlen aus. [37]

2.5.4. R-Bibliothek - arules

Außerhalb von Datenbanksystemen bietet das Paket `arules` für die Programmiersprache R einen nach [5] implementierten Apriori- wie Eclat-Algorithmus als Funktion für die Sprache R an. [32] [13] Die Funktionen des Pakets entsprechen Schnnittstellen zu den in C programmierten Algorithmen um Datensätze zu verarbeiten, wobei der Eclat-Algorithmus nur häufige Item-Mengen ausgeben kann, der Apriori-Algorithmus zusätzlich die Assoziationsregeln. Desweiteren unterstützt das Paket ein Argument `appearance` beim Funktionsaufruf, das angibt, welche Items in den Regeln enthalten sein müssen.

Eine Funktion liest die Transaktionen in beliebigem Datenformat ein (gruppiert: `bucket`), auf deren Ergebnis die Data-Mining-Algorithmen ausgeführt werden (s. Abb. 2.10). Das Ergebnis sind die Assoziationsregeln in aggregiertem Format mit einer Zeile pro Assoziationsregel. Die R-Bibliothek eignet sich gut, um die Ergebnisse eigener Implementierungen zu überprüfen.

Außerdem stellt die R-Bibliothek einen Datengenerator bereit, der basierend auf zwei verschiedenen Methoden ([2], [15]) Warenkorbdaten mit `random.transactions()` erzeugt. Die Entwickler haben die Bibliothek zusätzlich um Clustering-Algorithmen ergänzt, die Partitionen bilden, auf denen der Apriori-Algorithmus läuft, um so die Häufigkeit der Items clusterübergreifend zu vergleichen. [14]

3. Einführung in HyPer

Der Apriori-Operator wird in die Datenbank HyPer integriert, der Hauptspeicherdatenbank, die am Lehrstuhl für Datenbanksysteme der TU München unter Federführung von Professor Neumann und Professor Kemper enwickelt wird. HyPer ist ein hochperformantes hyprides OLTP und OLAP Datenbanksystem (**Hy**prid OLTP & OLAP High-**Per**formance Database System), welches parallel mehrfache Echtzeit-Transaktionen verarbeiten (Online Transaction Processing, OLTP) sowie mehrfache analysierende Anfragen (Online Analytical Processing, OLAP) ausführen kann. Außerdem unterscheidet sich HyPer von klassischen plattenbasierten Datenbanken, da es die Transaktionen ausschließlich im Hauptspeicher verarbeitet, und berücksichtigt als komplette Neuentwicklung den neuesten Stand der Technik von Hardware und nutzt Funktionalitäten des Betriebssystems aus.

Die nachfolgenden Unterkapitel erklären HyPer im Detail, sie erläutern die Systemarchitektur, um parallele Anfragen zu verarbeiten, die Semantik der Transaktionen und den Wiederherstellungsprozess, um die AKID-Bedingungen zu erfüllen, die Verarbeitung von Anfragen und schließen mit der effizienten Integration von Data- und Graph-Mining-Algorithmen ab.

3.1. Systemarchitektur

Da die Daten vollständig im Hauptspeicher liegen, geht keine Zeit durch Zugriff auf die Festplatte verloren. Die Anweisungen einer OLTP-Transaktion können seriell ausgeführt werden. Um parallel OLAP-Anfragen starten zu können, erhält jede OLAP-Anfrage einen eigenen virtuellen Adressraum. Der Unix-Systemaufruf `fork()` auf einem OLTP-Prozess, wenn die Datenbank in einem konsistenten Zustand ist, liefert eine Kopie des virtuellen Adressraumes. Die Speichersegmente werden nur bei Bedarf geändert, wenn sich der Inhalt ändert (*copy-on-demand*), bis dahin zeigen beide virtuellen Adressen auf denselben physischen Speicherbereich. Da OLAP-Anfragen nur lesend auf den Inhalt zugreifen, kann jede weitere OLAP-Anfrage denselben virtuellen Speicherbereich nutzen. Wenn jedoch jede Transaktionen auf den jeweils zu Anfragestart aktuellen Stand der Datenbank zugreifen soll, so muss jede Transaktion den jeweils aktuellen Adressraum mittels `fork()` kopiert bekommen und sie arbeitet auf ihrem eigenen virtuellen Adressraum. [23] Mehrere OLTP-Anfragen parallel zu verarbeiten kommt nicht ohne Schutzmechanismen aus, sobald eine Anfrage gemeinsam genutzte Ressourcen schreibend ändern möchte. Daher arbeiten OLTP-Anfragen auf demselben Adressraum, wobei der Zugriff auf kritsche Ressourcen nur synchronisiert erfolgen darf. [22]

3.2. AKID-Eigenschaften in HyPer

HyPer erfüllt alle AKID-Eigenschaften, **A**tomarität, **K**onsistenz, **I**solation, **D**auerhaftigkeit.

Die Atomarität der Transaktionen, das heißt eine Transaktion entweder ganz oder gar nicht auszuführen, gewährleistet ein Undo-Log im Hauptspeicher, um abgebrochene Transaktionen wiederherzustellen (R1-Recovery), und ein Redo-Log, das erst bei erfolgreichem Commit geschrieben wird, um die archivierte Kopie der Datenbank immer in einem konsistenten Zustand zu halten. Ein R3-Recovery, wegen Hauptspeicherverlustes und deshalb abgebrochener, nicht abgeschlossener Transaktionen (Verlierer-Transaktionen), ist nicht nötig, da die Datenbank im Hauptspeicher dabei ebenso

verloren ginge und die Verlierer-Transaktion nicht wiederhergestellt werden müssten. Folglich genügt für die Atomarität ein persistentes Redo-Log und ein unbeständiges Undo-Log. Da das Schreiben ins Redo-Log ausschließlich nach erfolgreichen Transaktionen erfolgt, ist eine Datenbank, die daraus wiederhergestellt wird, in einem konsistenten Zustand. Das Undo-Log setzt die Datenbasis im Hauptspeicher nach abgebrochene Transaktionen in einen konsistenten Zustand zurück.

Die Isolation, „ausgeführte Transaktionen [sollen] sich nicht gegenseitig beeinflussen“ [21], ist durch eine Schnappschussisolation (Snapshot Isolation) gegeben, bei HyPer durch einen eigenen virtuellen Speicher (Virtual Memory Snapshots) [22] für jede analysierende Anfragen, den der `fork()`-Befehl erzeugt. Das persistente Redo-Log gewährleistet die Dauerhaftigkeit der Daten. Nach einem Systemfehler lässt sich aus dem Redo-Log in chronologischer Reihenfolge der letzte konsistente Zustand im Hauptspeicher wiederherstellen. Wenn dies zu lange dauert, kann der HyPer-Server einfach dupliziert werden. Damit das Redo-Log aktuell ist, wird es kontinuierlich über eine Bandbreite zwischen 1 bis 10 Gb/s [22] auf verschiedene Server geschrieben.

3.3. Anfrageverarbeitung

Im Gegensatz zu bisherigen Modellen übersetzt HyPer eine Anfrage direkt in kompakten Maschinencode, sodass eine Anfrage nicht mehr aus mehreren überladenen Funktionsaufrufen einzelner Operatoren besteht, sondern aus den auszuführenden Anweisungen an sich. Dazu generiert HyPer zu jeder Anfrage den Maschinencode, den jedes Tupel in der Ausführung durchläuft. Im gängigen Iterator-Modell fragt der oberste Operator die Kindoperatoren, die dazu eine überladene Funktion `next()` bereitstellen, nach Tupeln. Das entspricht einem Baum von Funktionsaufrufen, die die Anfragen bearbeiten und für jedes Tupel aufgerufen werden. HyPer baut während der Codegenerierung einen bidirektionalen Baum, der den Code erzeugt (s. Abb. 3.1). Zentraler Bestandteil bilden die Funktionen `produce()` und `consume()`. Ein Eltern-Operator bittet die Kindoperatoren, ihm die Tupel zu bringen. Also anstatt, wie im Iterator-Modell üblich, die Tupel von oben zu ziehen, steckt ein Kindknoten dem Elternknoten die Tupel zu. Dabei generiert `produce()` den Code, um die Tupel zu produzieren. In `produce()` folgt ein Aufruf von `consume()` auf dem Elternknoten, der den Code produziert, um diesem die Tupel zuzustecken. Die beiden Methoden generieren ein imperatives Programm, durch das die Tupel dann laufen. Um ein Programm in Maschinencode zu erhalten,

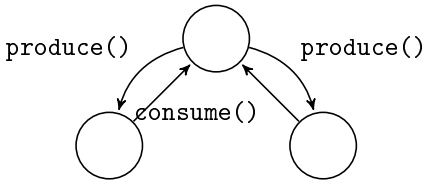

Abbildung 3.1.: Reihenfolge der Codegenerierung bei einem binären Operator

erfolgt die Codegenerierung mittels virtuellem Assembler (Low Level Virtual Machine, LLVM). Ein JIT-Übersetzer (Just-in-time-Compiler) übersetzt den Code zur Laufzeit in Maschinencode. LLVM kann C++-Methoden aufrufen und umgekehrt, sodass eine Mischung aus beiden Sprachen, komplexe Teile in C++ und zeitkritische Pfade, die für jedes Tupel aufgerufen werden, in LLVM, geeignet ist. [28]

3.4. Integration von Data- und Graph-Mining-Algorithmen

Um die Analyse hochdimensionaler Datensätze zu vereinfachen, schlägt eine Veröffentlichung der TU München [35] vor, Data- und Graph-Mining-Algorithmen in Datenbanksysteme zu integrieren, eine weitere Veröffentlichung führt dies anhand des K-Means- wie PageRank-Algorithmus vor. [31] Datenbanken eignen sich für eine effiziente Speicherung von Daten, doch große Datensätze lassen sich in SQL nur über Umwege, durch Behelfskonstrukte mit Hilfstabellen, analysieren. Applikationen, die auf die Analyse hochdimensionaler Datensätze zugeschnitten sind, müssen die Datensätze aus Datenbanken auslesen, weshalb es sinnvoll ist, die Funktionalitäten solcher Applikation gleich in Datenbanksysteme zu integrieren. Die Veröffentlichung der TU München [35] zeigt eine erfolgreiche vierschichtige Integration von Data-Mining-Algorithmen in das Datenbanksystem HyPer auf, wo die Algorithmen entweder extern laufen mit HyPer als Datenbasis, in der Datenbank als temporäre Funkionen ausgeführt werden, in ein erweiteres SQL formuliert oder direkt als Funktionen im Datenbankkern implementiert sind. Die Implementierung des K-Means-Algorithmus in HyPer ist um Faktor zehn schneller als eine in SQL-geschriebene Variante und doppelt so schnell wie im Vergleichssystem Apache Spark 1.4.0, der implementierte PageRank-Algorithmus mehr als zehnfach so schnell wie das Vergleichssystem und knapp doppelt so schnell wie eine in SQL-geschrieben Version. Damit lohnt sich die Adaption weiterer Data-Mining-Algorithmen in Datenbanksysteme.

4. Konzept

Diese Kapitel erklärt das Konzept, wie ein Apriori-Operator in die Hauptspeicherdatenbank HyPer integriert ist. Der erste Teil des Kapitels handelt von der Konzeption des Algorithmus an sich, der zweite Teil erklärt, wie mehrere Kerne die Verarbeitung verteilt übernehmen. Ein Unterkapitel gibt Aufschluss über Entscheidungen, die für die Konzeption getroffen worden sind. Ein Unterkapitel ist einem universellen Operator für Assoziationsregeln und eines einem möglichen Operatorbaum gewidmet.

4.1. Apriori-Algorithmus

Der Operator basiert auf dem ursprünglichen Apriori-Algorithmus [2] mit der Datenstruktur eines Präfixbaumes [8]. Folglich besteht der Algorithmus aus zwei Teilen, dem Generieren der Frequentitemsets und dem Ableiten der Assoziationsregeln daraus. Um die Transaktionen in der relationalen Normalform zu Warenkörben zu aggregieren und die für den Präfixbaum nötige Sortierung der Transaktionen nach Items herzustellen, ist dem Apriori-Operator ein Sort-Operator vorangestellt, der die Eingangsdaten nach der Transaktions-ID und nach den Items lexikographisch sortiert. Der Apriori-Operator materialisiert die Daten und gruppiert die Items gemäß ihrer Transaktions-ID. Bei der Gruppierung führt der Operator ein Item-Coding durch, um eine lückenlose Speicherung der Items in Vektoren fester Länge des Präfixbaumes zu ermöglichen. Die Vergabe der Kodewörter erfolgt fortlaufend nach Auftreten der Items. Anschließend generiert er aus den Warenkörben die Frequentitemsets und leitet die Assoziationsregeln ab (s. Abb. 4.1).

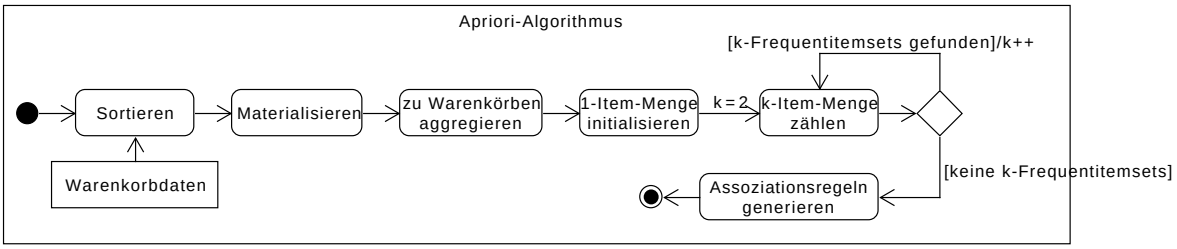

Abbildung 4.1.: Aktivitätsdiagramm des Apriori-Algorithmus

4.1.1. Datenstruktur

Die Frequentitemsets verwaltet eine Baumstruktur (s. Abb. 4.3), in der jeder Knoten ein Item repräsentiert und die zugehörige Häufigkeit zählt. Knoten verwalten Kindknoten in Vektoren fester Größe, die Position im Vektor entspricht implizit dem dem Kindknoten zugehörigen Item. Über den Knoten steht ein expliziter Wurzelknoten (s. Abb. 4.2). Die Knoten eines Pfades beliebiger Länge entsprechen einer Item-Menge, diese Item-Menge ist ein Frequentitemset, wenn die absolute

Häufigkeit des letzten Knotens des Pfades über dem absoluten Schwellenwert σ_0 liegt (gleich dem relativen Schwellenwert s_0 multipliziert mit der Anzahl aller Warenkörbe $|\mathcal{D}|$, den Transaktionen). Der Zugriff auf eine beliebige Item-Menge zur Bestimmung der Häufigkeit ist in linearer Zeit abhängig von der Länge der Item-Menge k möglich.

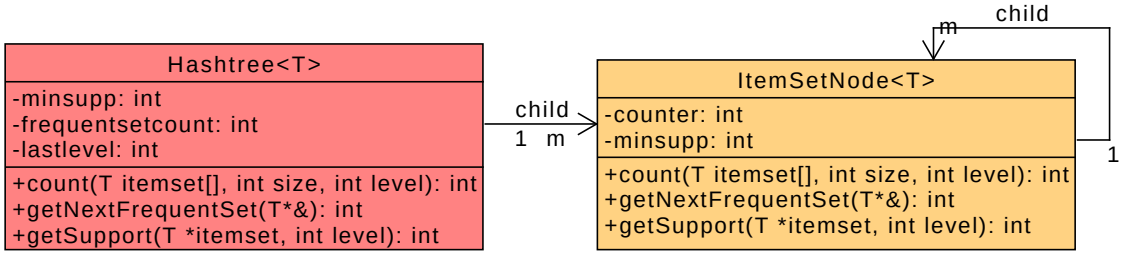

Abbildung 4.2.: Aufbau der Klasse Hashtree mit Kindknoten

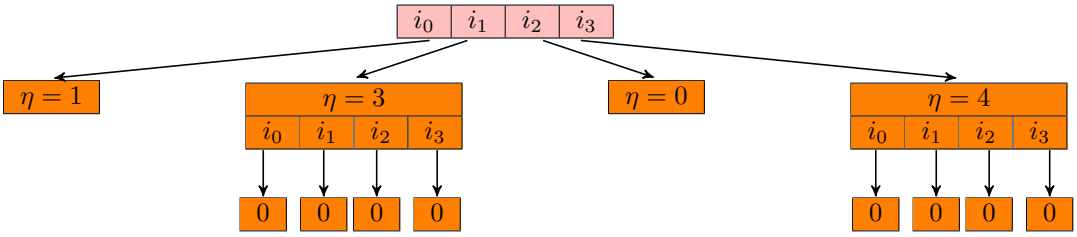

Abbildung 4.3.: Präfixbaum für Item-Mengen der Länge 2 bei vier möglichen Items, Duplikate möglich

4.1.2. Erzeugung der Frequentitemsets

Der Apriori-Operator erzeugt Kandidaten für Frequentitemsets der Länge k dem Apriori-Prinzip folgend sukzessive aus Frequentitemsets der Länge $k-1$. Die Kandidatengenerierung erfolgt implizit während der Zählphase, sobald der Support den absoluten Schwellenwert überschreitet. Der Wurzelknoten enthält initial alle Kandidaten für Item-Mengen der Länge 1, da er einen Vektor mit Kindknoten für jedes mögliche Item enthält. Die Präfixbaumstruktur begrenzt die Anzahl der Aufrufe pro Transaktion, da die Teilmengen einer Transaktionen nur mit den Item-Mengen, mit denen sie dasselbe Präfix teilen, verglichen werden und nicht mehr mit allen. Für das Zählen der Häufigkeit einer Item-Mengen sind zwei Konzepte möglich. Entweder die Transaktionen durchlaufen in jeder Iteration den Präfixbaum (Apriori), oder nur die relevanten Transaktionen, die im vorangegangenen Schritt ein Frequentitemset erzeugt haben (ähnlich AprioriTID).

Die klassische Variante des Apriori-Algorithmus ruft in jeder Iteration pro Transaktion eine Zählmethode auf, genannt `count()`, die der Methode `subset()` entspricht und die Häufigkeit der Item-Mengen bestimmt. Die Methode wird für jedes Element i einer Transaktion T auf dem entsprechenden Kindknoten aufgerufen und bekommt als Parameter die restlichen Item-Menge einer Transaktion

$(T \setminus i)$ und die verbleibenden Anzahl an Schritten. Dem initialen Aufruf im Wurzelknoten folgen rekursive Aufrufe entlang der Knoten eines Pfades, denn ein Pfad entspricht einer Item-Menge.

```
steps ← 1;
repeat
    found ← 0;
    forall T ∈ 𝒟 do
        T' ← T;
        forall i ∈ T' do
            T' ← T' \ i;
            found ← found + child[i].count(T', steps);
        end
    end
    steps ← steps + 1;
until found = 0;
```

Algorithmus 6 : Iteration des Apriori-Algorithmus in der Wurzel

Der rekursive Aufruf terminiert im letzten Schritt des Rekursionsbaums, also nach t Aufrufen in der t-ten Iteration bei einer t-Item-Menge und inkrementiert im dortigen Knoten den Zähler für die Häufigkeit (*count*). Gemäß der Antimonotonie der Apriori-Heuristik terminiert ein Aufruf bereits eher, wenn ein mittlerer Knoten eine zu geringe Häufigkeit aufweist ($count < \sigma_0$) und die zugehörige Item-Menge kein Frequentitemset ist. Jeder Aufruf liefert die Anzahl der Inkrementierungen ($found$), folglich die berücksichtigten Item-Mengen, für die die Rekursion nicht zu früh terminiert. Sobald alle Frequentitemsets ($found = 0$) gefunden sind und keine weiteren Item-Mengen als Frequentitemsets mehr in Frage kommen, endet der Algorithmus.

```
if counter < σ0 then
    return 0;
end
if s = 0 then
    counter ← counter + 1;
    if counter = σ0 then
        apriori-gen();
    end
    return 1;
else
    T' ← T;
    found ← 0;
    forall i ∈ T' do
        T' ← T' \ i;
        found ← found + child[i].count(T', s − 1);
    end
    return found;
end
```

Algorithmus 7 : Methode `count(Transaktion t, Schritte s)` der Kindknoten

Die Kandidaten werden implizit generiert, denn sobald der Zähler den Schwellenwert ($counter = \sigma_0$) überschritten hat, wird für jedes mögliche Element ein Kindknoten angelegt (s. Abb. 4.4). Da ein Item implizit die Position bestimmt, legt `apriori-gen()`, wenn Vektoren für die Kindknoten genutzt werden, einen Vektor mit der Größe aller Items $|I| = m$ an. Bei Nutzung einer Hashmap entfällt die Funktion, da der Kindknoten beim ersten Bedarf erzeugt wird.

TID	Items
1	$\{i_0, i_2\}$

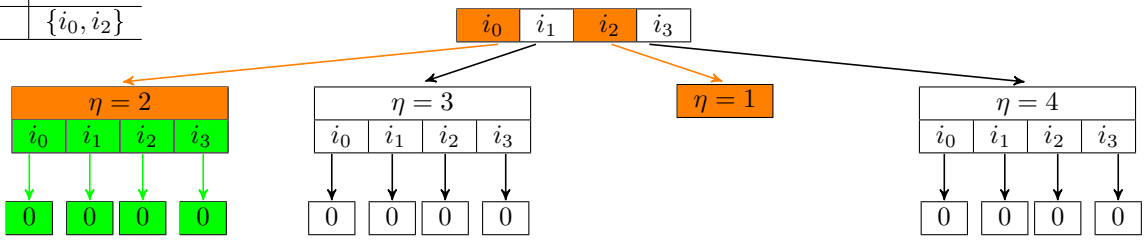

Abbildung 4.4.: Implizite Kandidatengenerierung bei Durchlauf einer Transaktion und $\sigma_0 = 2$

4.1.3. Generieren der Assoziationsregeln

Assoziationsregeln $X \Rightarrow Y$ entstehen aus den Frequentitemsets $L \in \mathcal{L}$, indem jeweils eine Teilmenge $X \subseteq L$ die Prämisse bildet, der Rest die Konklusion $Y := L \backslash X$. Relevant sind die Assoziationsregeln, deren Vertrauenswert den Schwellenwert übersteigt ($k(X \Rightarrow Y) = \frac{s(X \cup Y)}{s(X)} \geq k_0$). Um systematisch alle Regeln zu erhalten, hilft eine Sortierung: Jedes Item $i_j \in L$ erhält eine binäre Variable p_j, die anzeigt, ob das Item in der Prämisse enthalten ist ($i_j \in X \Rightarrow p_j = 1$). Der Vektor p wird als Binärzahl interpretiert und inkrementiert, daraus ergeben sich bei einer k-Item-Menge schrittweise alle $2^k - 1$ möglichen Assoziationsregeln, die Regel abgezogen mit allen Items in der Prämisse, denn so bliebe kein Item für die Folge übrig. Die Binärzahl kann rückwärts interpretiert werden, um mit dem ersten Item i_0 in der Prämisse zu beginnen (s. Tab. 4.1).

i_2	i_1	i_0	X	$\Rightarrow$	Y
0	0	0	$\emptyset$	$\Rightarrow$	$\{i_0, i_1, i_2\}$
0	0	1	$\{i_0\}$	$\Rightarrow$	$\{i_1, i_2\}$
0	1	0	$\{i_1\}$	$\Rightarrow$	$\{i_0, i_2\}$
0	1	1	$\{i_0, i_1\}$	$\Rightarrow$	$\{i_2\}$
1	0	0	$\{i_2\}$	$\Rightarrow$	$\{i_0, i_2\}$
1	0	1	$\{i_0, i_2\}$	$\Rightarrow$	$\{i_1\}$
1	1	0	$\{i_1, i_2\}$	$\Rightarrow$	$\{i_0\}$
1	1	1	$\{i_0, i_1, i_2\}$	$\Rightarrow$	$\emptyset$

Tabelle 4.1.: Generierung der Assoziationsregeln aus einem Frequentitemset mit 3 Items

Gemäß dem Apriori-Prinzip der Assoziationsregeln wird zuerst die Konfidenz der Assoziationsregel mit der leeren Prämisse bestimmt. Folglich ist eine Sortierung nach Anzahl gesetzter Bits sinnvoll, um aus Assoziationsregeln mit einelementiger Prämisse die Prämissen mit zwei Elementen zu erzeugen (s. Tab. 4.2).

Bis zu Frequentitemsets aus 64 Elementen lässt sich die Sortierung leicht mittels Ganzzahlen bei einer 64-Bit-Architektur angehen, soll ein Frequentitemset mehr Elemente enthalten können, so ist ein Bitvektor nachzubauen, der entsprechende Arithmetik unterstützt. Dennoch spielt das Problem der zu generierenden Regeln im Vergleich zum Finden der Frequenitemset nur eine nachgelagerte Rolle, denn die Assoziationsregel mit den meisten Items enthält genauso viele wie das zugehörige Frequentitemset.

i_0	i_1	i_2	X	$\Rightarrow$	Y
0	0	0	$\emptyset$	$\Rightarrow$	$\{i_0, i_1, i_2\}$
1	0	0	$\{i_0\}$	$\Rightarrow$	$\{i_1, i_2\}$
0	1	0	$\{i_1\}$	$\Rightarrow$	$\{i_0, i_2\}$
0	0	1	$\{i_2\}$	$\Rightarrow$	$\{i_0, i_2\}$
1	1	0	$\{i_0, i_1\}$	$\Rightarrow$	$\{i_2\}$
1	0	1	$\{i_0, i_2\}$	$\Rightarrow$	$\{i_1\}$
0	1	1	$\{i_1, i_2\}$	$\Rightarrow$	$\{i_0\}$
1	1	1	$\{i_0, i_1, i_2\}$	$\Rightarrow$	$\emptyset$

Tabelle 4.2.: Sortierung nach Anzahl gesetzter Bits

4.2. Parallelisierung

Um den Apriori-Algorithmus zu parallelisieren stehen mehrere Möglichkeiten zur Auswahl. Der klassische Apriori-Algorithmus wird parallelisiert, indem entweder jedem Thread mehrere Transaktionen pro Iteration zugewiesen werden, wie in Kapitel 2.3.3 beschrieben, womit die Geschwindigkeit bei vielen Transaktionen durch Anzahl verwendeter Threads linear steigt. Alternativ können die rekursiven Aufrufe aller Teilmengen einer Item-Menge parallel erfolgen, der Grad der Parallelisierung ist hierbei auf die Kardinalität einer Item-Menge beschränkt. Verarbeiten mehrere Threads parallel verschieden Transaktionen, so schlägt [18] hierfür die Spiegelung und anschließende Synchronisation des Präfixbaumes vor, damit sich Threads beim Zählen identischer Item-Mengen nicht in die Quere kommen. Da in jeder Iteration nur die Blätter, also die untersten Knoten des Baumes, beschrieben werden und auf alle anderen Knoten nur lesend zugegriffen wird, ist eine Spiegelung des gesamten Baumes nicht nötig, es genügt, für jeden Thread eine eigene Zählervariable im Baum zur Verfügung zu stellen, die am Ende jeder Iteration aufsummiert werden. Da die Kandidatengenerierung gemäß der Apriori-Antimonotonie von der Häufigkeit der kürzeren Item-Mengen und somit vom aufsummierten Ergebnis abhängt, bietet es sich an, das Aufsummieren und die Kandidatengenerierung in einem gemeinsamen Schritt nach jeder Iteration zu vollziehen (genannt `consolidate()`). Damit ähnelt der Apriori-Algorithmus wieder der ursprünglichen Version, die zwei Phasen kennt, die Teilmengenfunktion um Kandidaten zu überprüfen und die Kandidatengenerierung (s. Abb. 4.5).

4.3. Zielkonflikte

Diese Unterkapitel gibt Aufschluss über Entscheidungen, die es bei einer Implementierung abzuwägen gilt. Dies beginnt bei der Datenstruktur, ob Duplikate zugelassen sind, führt weiter zur Relevanz von Itemcoding und zum AprioriTID-Algorithmus und endet beim Format der Ausgabe der Assoziationsregeln.

4.3.1. Umgang mit Duplikaten

Die Datenstruktur des Präfixbaumes unterstützt den Umgang mit duplizierten Items in einer Item-Menge. Ob es sinnvoll ist, Duplikate zu registrieren und Assoziationsregeln zu generieren, wo entweder ein Item sich selber folgert oder doppelt gekaufte Items zu einem Item führen oder der Kauf eines Items zum Kauf mehrerer gleicher anderer Items führen (s. Abb. 4.6), hängt von der Anwendung ab. Für ein Online-Warenhaus scheint es unsinnig zu sein, auf dasselbe Produkt in einer Vorschlagsliste hinzuweisen, wenn es der Kunde sowieso ausgewählt hat. Wiederum kann es sinnvoll sein, ein

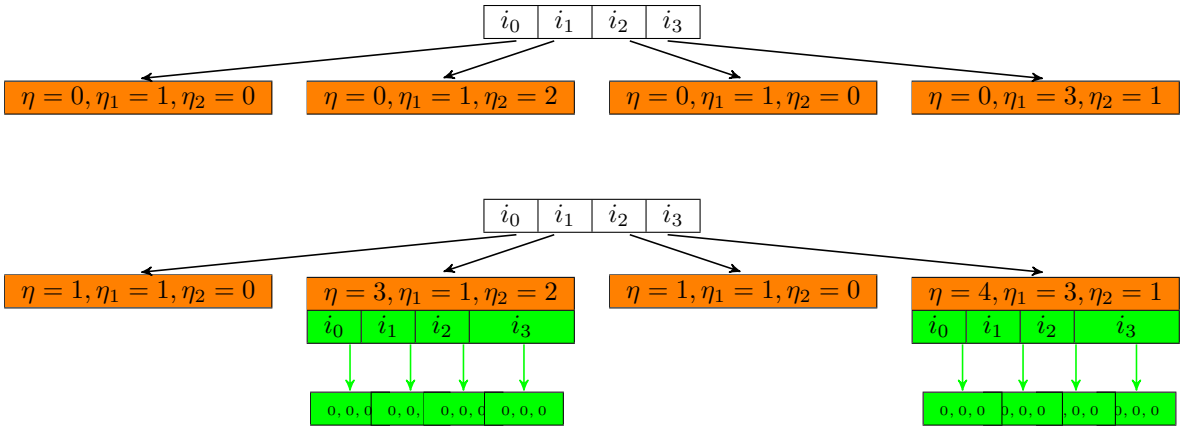

Abbildung 4.5.: Paralleler Apriori-Algorithmus: Zählen und Zusammenfügen

bestimmtes Produkt erst vorzuschlagen, wenn ein anderes in ausreichender Stückzahl eingekauft ist (wie einen Kaugummi-Spendeapparat, den ein Kunde erst nutzen kann, wenn er ausreichend Kaugummis gekauft hat). Der umgekehrte Fall, hinzuweisen, dass ein Produkt in größerer Menge beliebt, sollte nicht nötig sein, da die Evidenz dem Kunden klar sein sollte (so passen in einen Kaugummi-Spendeapparat mehrere Kaugummis), aber kann für Angebote genutzt werden (zehn Kaugummis für den Preis von fünf, wenn ein Kaugummi-Spendeapparat bereits im Warenkorb liegt). Leider enthalten verfügbare Warenkorbdaten nur distinkte Werte ohne die Stückzahl eines gekauften Items, weshalb eine Analyse, wie sich mehrfach auftretende Items auf das Kaufverhalten auswirken, nicht möglich ist.

Bisherige Veröffentlichungen beschäftigen sich nur rudimentär mit dem Umgang von Duplikaten. Das ursprüngliche Papier von Agrawal [1] kennt keine Duplikate, da es einen Warenkorb als einen Vektor von Indikatorvariablen sieht und somit nur prüft, ob ein Element enthalten ist. Allerdings ließe sich das Konzept auf Duplikate übertragen, wenn der Vektor statt Indikatorvariablen Zählvariablen enthält. Wiederum als sinnlos erachtet [41] Duplikate in Frequentitemsets, mit dem Ziel diese als Teil des Operators im Voraus herauszufiltern, obwohl die gewählte Datenstruktur genauso Duplikate aufnehmen könne. Ein Nachteil, wenn Duplikate erlaubt sind, ist der höhere Speicherverbrauch. Jeder Knoten referenziert für jedes lexikographisch größere und mögliche Item, einen Kindknoten. Bei distinkten Werten sinkt die Anzahl der Kindknoten in jeder Ebene um eins. Wenn Duplikate erlaubt sind, so muss der Knoten einen weiteren Kindknoten für das Item referenzieren, das er selbst repräsentiert. Da der Algorithmus in einer Hauptspeicherdatenbank mit ausreichend Hauptspeicher integriert wird und Duplikate sich durch ein `select distinct` bereits vorher eliminieren lassen, unterstützt der integrierte Algorithmus Duplikate. Um Duplikate in Item-Mengen richtig zu zählen, ist eine Modifikation des Algrithmus dahingehend nötig, den rekursiven Aufruf der Zählmethode nur für einen der duplizierten Elemente zu tätigen. Da für eine Assoziationsregel zählt, auf wie viele Warenkörbe sie zutrifft, darf ein Warenkorb mit Duplikaten ($\{i_0, i_0, i_1\}$ nur einfach zählen, ein Aufruf für jedes duplizierte Element entspräche einer Gewichtung des Warenkorbes um die Anzahl der Duplikate.

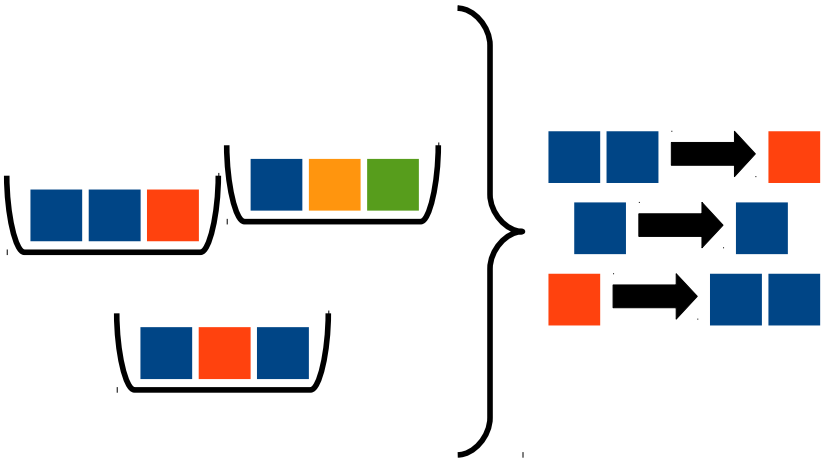

Abbildung 4.6.: Assoziationsregeln aus Duplikaten

4.3.2. Itemcoding

Ein weiterer Zielkonflikt ist die Speicherung der Originalwerte oder eigener Itemcodes im Präfixbaum. Bei der Speicherung der originalen Werte entfällt der Aufwand, ein Wörterbuch zu verwenden (s. Abb. 4.7). Jedoch identifiziert ein Hashwert jedes Item in der Datenstruktur, sodass Kollisionen möglich sind und lineare Suche über alle Werte desselben Hashwertes ist die Folge. Bei in der Suche schnellen Vektoren ist eine dichte Bepackung nur möglich, wenn ein Wörterbuch verwendet wird, um Kollisionen zu vermeiden. Letztendlich ist der Aufwand, ein Wörterbuch zu erstellen konstant in der Anzahl der Elemente, es abzurufen konstant in der Anzahl der Ergebnisse, demgegenüber verlängert sich die Laufzeit ohne Wörterbuch mit der Anzahl der Iterationen und der Kollisionen. Somit relativiert sich der Aufwand zur Erstellung des Wörterbuches und die schnellere Datenstruktur eines Vektors findet Anwendung.

4.3.3. AprioriTID-Algorithmus

Wenn sich jeder Knoten die zugehörigen Transaktionen merkt, entfällt der Aufruf von Transaktionen, die zu kurz sind oder die sicher kein Frequentitemset mehr erzeugen. Dazu speichert jeder Knoten eine Referenz zu der entsprechenden Transaktion und ruft selbst diese auf. Dies entspricht dem AprioriTID-Algorithmus [2]. Da der Präfixbaum bereits die Transaktionen nur mit Item-Mengen mit demselben Präfix vergleicht, reduziert der AprioriTID-Algorithmus nur begrenzt die Anzahl notwendiger Vergleiche: Es fallen die Vergleiche von Item-Mengen weg, die wegen der Apriori-Antimonotonie kein Frequentitemset sein können. Deshalb ist ein Performanz-Vergleich zwischen gewöhnlichem Apriori-Algorithmus mit dem AprioriTID-Algorithmus interessant, jeweils unter Nutzung von Präfixbäumen.

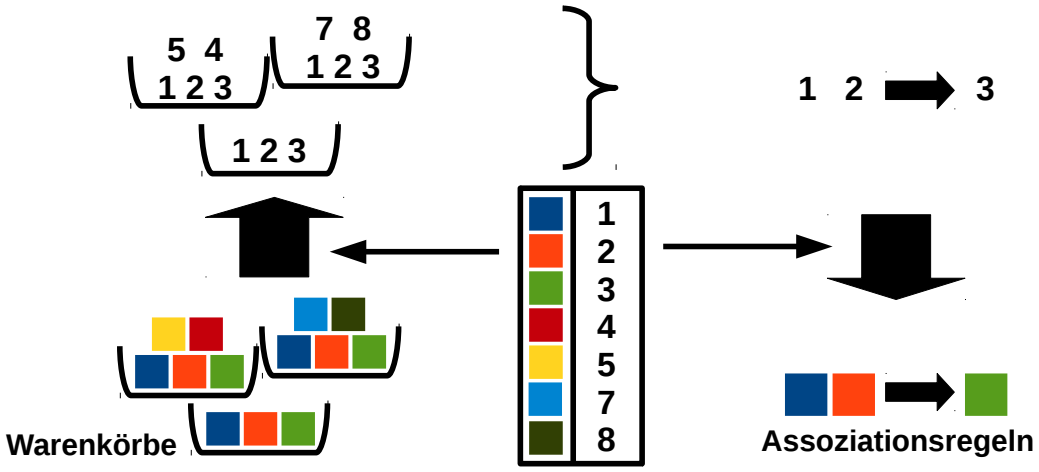

Abbildung 4.7.: Wörterbuch für interne Repräsentation

4.3.4. Verletzung der Normalform

Der integrierte Apriori-Algorithmus gibt die Items der Konklusion und der Prämisse einer Assoziationsregeln in Mengen an und verstößt damit gegen die erste Normalform. Eine Assoziationsregel besteht aus einer Bezeichnung, der Prämisse und der Konklusion, sowie den Kennzahlen für Support, Konfidenz, Lift und Conviction. Da Prämisse und Konklusion eine Menge von Items sind, zu jeweils einem Feld von Items zusammengefasst entspricht jeweils eine Assoziationsregel genau einem Tupel, aber verletzt die erste Normalform.

- Regeln: {[ID, Prämisse, Konklusion, Support, Konfidenz, Lift, Conviction]}

Die Relation ist in erster Normalform, wenn pro Item ein Tupel ausgegeben wird, mit dem Bezeichner der Assoziationsregel und der Angabe, ob das Item Teil der Prämisse oder der Konklusion ist. Wenn jedes Tupel darüberhinaus Informationen über die Kennzahlen der Assoziationsregel enthalten, verletzt die Relation die zweite Normalform.

- Regeln: {[ID, Zugehörigkeit, Item, Konklusion, Support, Konfidenz, Lift, Conviction]}

Um ein Schema in der vierten Normalform zu erhalten, sind mindestens zwei Relationen nötig, eine die zu jeder Assoziationsregel die Kennzahlen angibt (mit dem Bezeichner als Schlüssel und den Kennzahlen als Attribute) und mindestens eine bis zwei Relationen, die zu jeder Assoziationsregel die Items angibt, die in der Prämisse oder der Konklusion stehen (mit Schlüsselattributen Bezeichner, das Item selbst und, wenn nur eine Relation, einem Attribut um Konklusion oder Prämisse zu unterscheiden).

- Prämissen: {[ID, Item]}
- Konklusionen: {[ID, Item]}
- Kennzahlen: {[ID, Support, Konfidenz, Lift, Conviction]}

Da andere Data-Mining-Algorithmen (wie Apriori-Algorithmen) pro Assoziationsregeln eine Zeile ausgeben mit Mengen von Items, wählt dieser Apriori-Algorithmus die kompaktere Form mit einem Tupel pro Assoziationsregel. Obiges, genormtes Schema lässt sich daraus ableiten.

4.3.5. Bedingter Apriori-Algorithmus

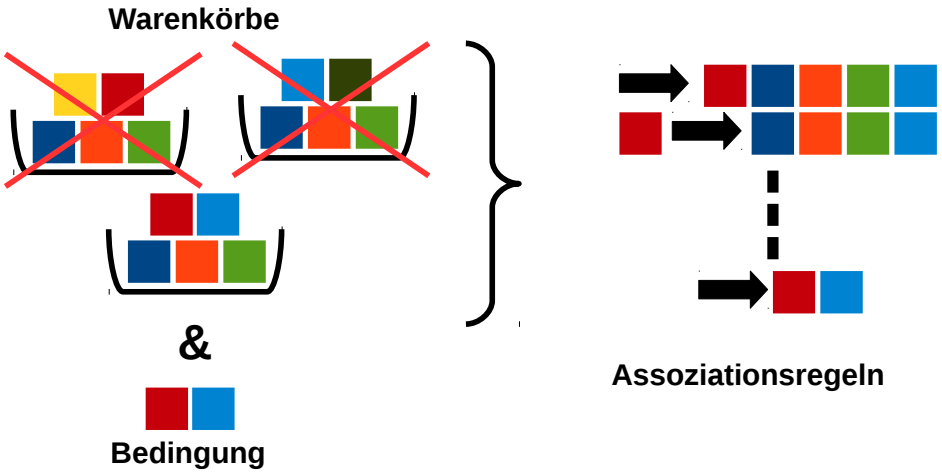

Abbildung 4.8.: Bedingter Apriori-Algorithmus durch Vorgabe bestimmter Items

Um die Ergebnismenge einzugrenzen, ist es gewünscht, Bedingungen an den Apriori-Operator zu stellen (s. Abb. 4.8), sodass er Assoziationsregeln ausgibt, in denen ein bestimmtes Item i_s enthalten ist. Das bedeutet nur solche Item-Mengen zu berücksichtigen, die das Element enthalten ($\mathcal{D}_s = \{T | T \in \mathcal{D} \wedge i_s \in T\}$). An der Baumstruktur selbst ändert sich nichts, da die lexikographische Ordnung der Items den Pfad bestimmt. Eine einfach SQL-Abfrage schränkt die berücksichtigten Ergebnisse ein (s. Abb. 4.9). Allerdings erzeugt der Apriori-Operator alle Assoziationsregeln aus den gefilterten Transaktionen, auch Regeln, die die Elemente, die gesucht sind, nicht enthalten. Diese Regeln müssen anschließend herausgefiltert werden, die Laufzeit des Algorithmus beschleunigt sich.

```
select * from apriori((
        select *
        from sales s1
        where exists(
                select *
                from sales s2
                where s1.tid=s2.tid and s2.item='Stift'
        )),0.01,0.01);
```

Abbildung 4.9.: Bedingungen im Vorfeld des Apriori-Operators

Leider ändert sich dadurch die relative Häufigkeit und der Support muss manuell berechnet werden, die Konfidenz bleibt gleich. Die Rückberechnung ist mit der Anzahl der Transaktionen für ein beliebiges Frequentitemset L möglich:

$$s(L) = s_s(L) * \frac{|\mathcal{D}_s|}{|\mathcal{D}| - |\mathcal{D}_s|}$$

4.3.6. Apriori-Algorithmus für große Item-Mengen

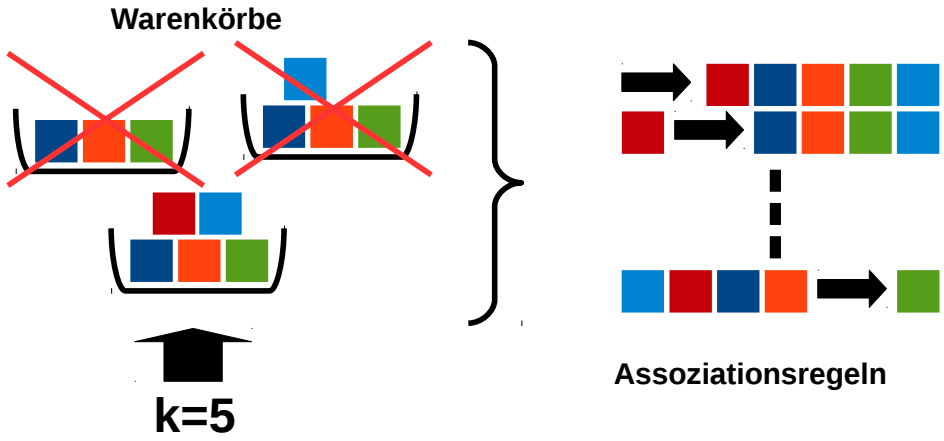

Abbildung 4.10.: Vordefinierte Größe der Assoziationsregeln

Wenn vorab klar ist, dass große Frequentitemsets gesucht sind (s. Abb. 4.10), lässt sich die Eingabe durch eine SQL-Abfrage beschränken (s. 4.11), die nur Transaktionen mit mehr als k Elementen berücksichtigt. Denn für ein k-Frequentitemset sind nur k-Item-Mengen relevant.

```
select * from apriori((
        select *
        from sales s1
        where 5<(
                select count(*)
                from sales s2
                where s1.tid=s2.tid
        )),0.01,0.01);
```

Abbildung 4.11.: Einschränkung auf k-Item-Mengen (hier: $k = 5$)

4.4. Universeller Operator für Assoziationsregeln

Um effizient neue Algorithmen auszuprobieren, ohne eigens einen Operator zu implementieren, gibt es die Idee einer rechenbetonten Datenbank (engl.: *computational database*). Mit dieser soll es möglich sein, dass Endanwender universal Algorithmen einpflegen, die auf Daten in der Datenbank ausgeführt werden können. Dazu sind zwei Überlegungen relevant, die dieses Unterkapitel erläutert, nämlich welche Abstraktionen sich vornehmen lassen, um flexibel den dahinterstehenden Algorithmus auszuwechseln, und welche Bausteine für die Assoziationsanalyse unabdingbar sind.

4.4.1. Abstraktion der Assoziationsanalyse

Die Assoziationsanalyse besteht bei allen vorgestellten Algorithmen aus zwei Teilen, dem Finden von Frequentitemsets und dem Generieren der Assoziationsregeln. Jeder Teil erwartet die gleiche Form der Eingabe und erzeugt die gleiche Form der Ausgabe unabhängig von der Art des gewählten Algorithmus. Um die Frequentitemsets zu finden, müssen Warenkorbdaten eingegeben werden, als Ausgabe werden die Item-Mengen jeweils mit dem zugehörigen Support erwartet. Der zweite Teil nimmt die Item-Mengen, generiert daraus die Assoziationsregeln und berechnet anhand des Supports die Konfidenz. Beides zusammen stellt das Ergebnis der Assoziationsanalyse dar. Somit müssen die Algorithmen nur eine Schnittstelle erfüllen, der Algorithmus darunter kann beliebig ausgetauscht werden. Das Entwurfsmuster *Strategie* entspricht genau dieser Idee, eine abstrakte Superklasse `Frequentitemsetsfinder` definiert die Schnittstelle für den konkreten Algorithmus, analog eine abstrakte Superklasse `Regelgenerator` die Schnittstelle für den zweiten Teil (s. Abb. 4.12).

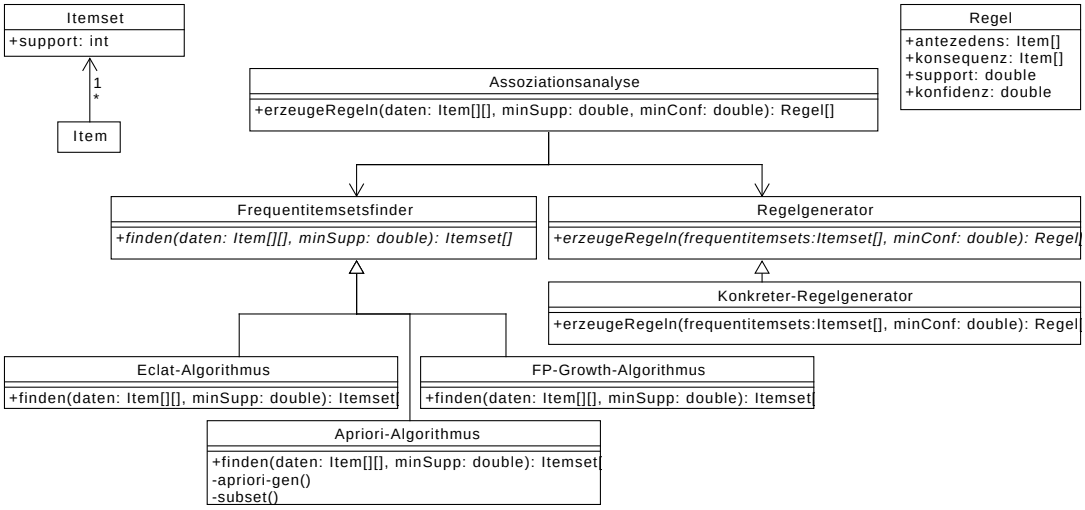

Abbildung 4.12.: Entwurfsmuster Strategie in der Assoziationsanalyse

Die konkrete Implementierung zur Superklasse `Regelgenerator` muss eine Möglichkeit haben, die Frequentitemset inklusive des zugehörigen Supports übergeben zu bekommen. Entweder werden die Daten als Objekt übergeben oder die Schnittstelle des `Frequentitemsetsfinder` bietet eine Funktion an, um über alle Frequentitemsets zu iterieren und eine Funktion, um den Support zu einer Item-Menge zu erhalten. Letztere Variante entspricht dem in dieser Arbeit vorgestelltem Konzept

und bietet den Vorteil, dass eine effiziente Datenstruktur, wie hier der Präfixbaum, genutzt wird um den Support auszugeben und keine Objekte transferiert werden.

Unabhängig davon sind die Formate der Ein- und Ausgabe zu definieren. Die Eingabe in relationaler Normalform und die Ausgabe in aggregierter Form stellt eine Möglichkeit hierzu dar. Dabei ist eine Funktionalität nötig, die die Warenkorbdaten zu Feldern zusammenfasst, die alle Algorithmen in gleicher Weise benötigen.

4.4.2. Bausteine der Assoziationsanalyse

Die nötigen Bausteine, um als Endanwender Algorithmen selbst entwerfen zu können, orientieren sich an an den vorhin definierten Abstraktionen. Als Vorbild für die Verwendung der Bausteine liefert das vorgestellte Paket `arules` für die Programmiersprache R, das für die vorgestellten Abstraktionen jeweils eine Funktion bereitstellt, deren Ergebnis gespeichert und weiterverarbeitet werden kann. Um nun die Algorithmen freier entwerfen zu können, müssen Ein- wie Ausgabe genormt sein. Wenn für die Eingabe die relationale Form erlaubt sein soll, so muss ein Baustein existieren, um die relationale Form in Felder umzuformen. Schleifen ermöglichen dann das Verarbeiten der Elemente der Felder. Für die Ausgabe sollte ein Format definiert sein, das der selbst zu schreibende Algorithmus zu erfüllen hat, analog zur Apriori-Funktion aus Apache MADlib.

4.5. Operatorbaum für Apriori-Algorithmus

Um die Funktionalitäten des Apriori-Algorithmus mit einem Operatorbaum zu beschreiben, um auf bekannte Operatoren zurückzugreifen, sind ein Operator zur Sortierung nötig und ein Gamma-Operator zur Gruppierung und Aggregation. Der Gamma-Operator bündelt die Warenkörbe zu Feldern, wenn eine Funktion `array-agg()` implementiert ist. Der Operator zur Sortierung ist nur noch nötig für die Sortierung innerhalb eines Warenkorbes und eine Sortierung könnte somit auch nach der Aggregierung auf den gebündelten Warenkörben lokal ausgeführt werden. Anschließend arbeitet ein neudefinierter Apriori-Operator auf den Warenkörben in aggregierter Form (s. Abb. 4.13, links).

Ein Problem, wenn für die Implementierung des Apriori-Algorithmus auf existierende Operatoren weitestgehend zugegriffen wird, ist ein Wörterbuch. Ein Wörterbuch zu erstellen ist verhältnismäßig einfach, denn es entspricht einer Gruppierung nach allen möglichen Items (oder einem `select distinct item`) und jedes Item erhält ein Kodewort durch eine Sequenz zugewiesen (s. Abb. 4.13, rechts). Allerdings müsste die Pipeline für das Wörterbuch von der Pipeline für den Apriori-Algorithmus abgespalten werden, vor der Generierung der Warenkörbe müssen die Items durch die Kodewörter ersetzt werden, um am Ende die Kodewörter in den Assoziationsregeln wieder durch die Items zu ersetzen. Da dies einem unüblichen Operatorbaum entspräche, wird zwar die Logik der einzelnen Operatoren aufgegriffen (Sortierung, Aggregierung), dennoch werden die Funktionalitäten in einem Apriori-Operator zusammengefasst.

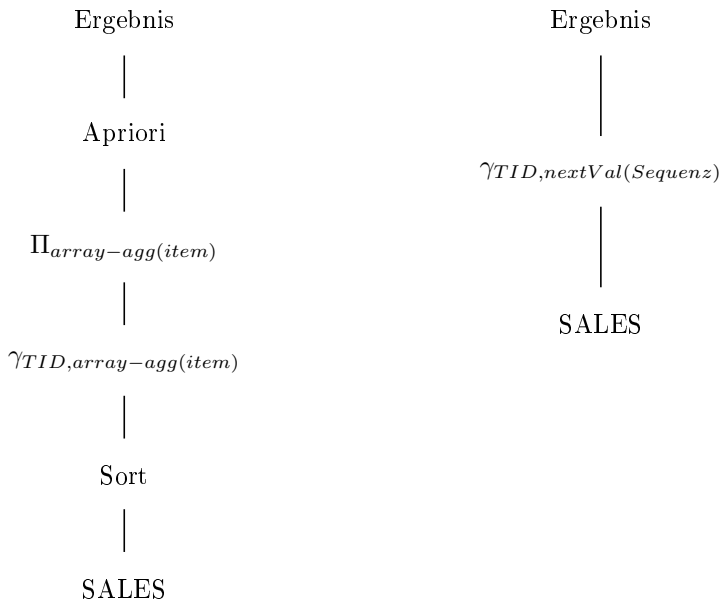

Abbildung 4.13.: Operatorbaum für Apriori-Algorithmus und das Wörterbuch

5. Implementierung

Der Algorithmus fügt sich der Architektur von HyPer und erzeugt Maschinencode. Der generierte Code, den `produce()` produziert, ruft den darunter stehenden Operator auf und materialisiert die übergebenen Warenkorbdaten. Anschließend werden die Assoziationsregeln erzeugt und der generierte Code von `consume()` reicht die Tupel an den darüberstehenden Operator weiter.

Der generierte Code, um Assoziationsregeln zu erzeugen, ist Maschinencode und ruft C++-Methoden auf, da ein Großteil der Datenstrukturen in C++ modelliert sind. Maschinencode materialisiert zuerst die Daten, anschließend bündelt C++-Code die Warenkörbe für die interne Repräsentation. Die Schleife für die Iteration ist in Maschinencode geschrieben. Dieser ruft eine C++-Methode auf, um die Transaktionen in den Baum einzufügen. C++-Code erzeugt die Assoziationsregeln, LLVM-Code packt diese dann zu Datentypen, um sie dem Elternknoten weiterzureichen (s. Abb. 5.1).

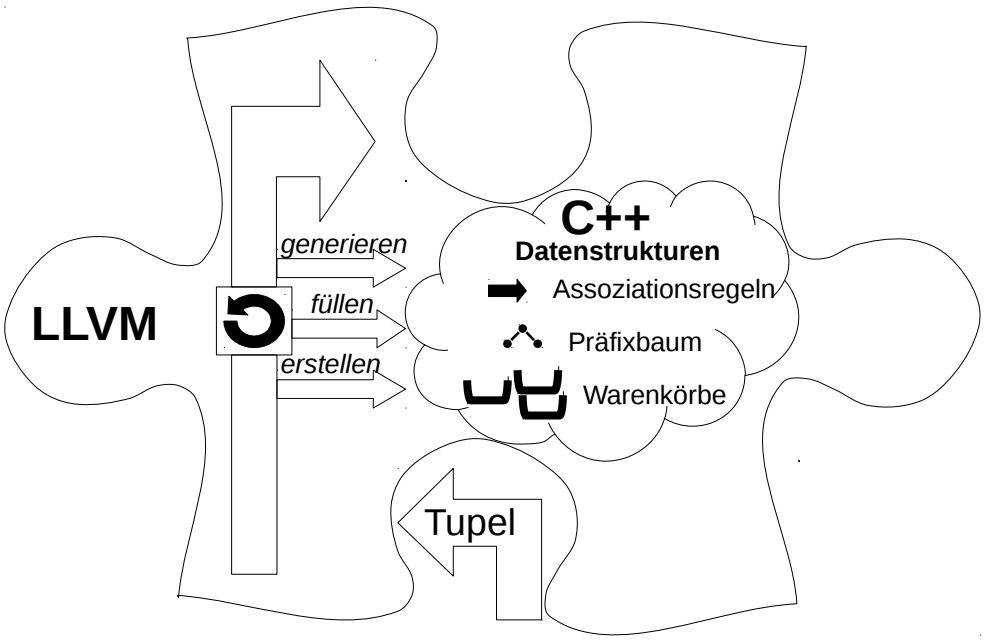

Abbildung 5.1.: Zusammenspiel LLVM mit C++-Code

Der Präfixbaum, um die Frequentitemsets zu bestimmen, besteht aus zwei Klassen: einer für die Wurzel und einer für die inneren Knoten bzw. die Blätter, die die Items repräsentieren. Jede Klasse

verwaltet die Kindknoten in einem Vektor der C++-Standardbibliothek (`std::vector`) mit eigenem Speicherallokator und einer initialen Größe von null Elementen. Innere Knoten und Blätter enthalten einen Zähler um den absoluten Support zu zählen. Der Baum wächst durch die Kandidatengenerierung, implementiert durch eine einmalige Anpassung der Größe des Vektors durch `resize()`, wenn der gezählte absolute Support den Schwellenwert übersteigt. Der absolute Schwellenwert wird gesetzt sobald die Anzahl der Warenkörbe bekannt ist und wird aus dem relativen Schwellenwert berechnet.

Die Wurzel des Präfixbaumes gibt sequentiell alle Frequentitemsets aus, die Standardfunktion `std::next_permutation()` liefert alle Permutationen, eine Schleife spaltet die Menge in je eine linke und rechte Teilmenge auf und erzeugt dadurch die Assoziationsregeln.

Ein zweisprachiges Wörterbuch bilden zwei ungeordnete Streuwerttabellen (`std::unordered_map`), die die Items in Zahlen für eine interne Repräsentation übersetzen und wieder zurück. Eine Alternative bildet eine in LLVM-geschriebene Streuwerttabelle aus HyPer, die bei Kollisionen bessere Laufzeit bietet. Zwei Varianten für das Wörterbuch sind möglich: Entweder werden die Items codiert, während sie zu Warenkörben aggregiert werden, oder die Codierung wird vorgezogen, alle Items werden codiert und im Anschluss werden die Warenkörbe aggregiert. Das bietet den Vorteil, dass der letzte Schritt parallel ausgeführt werden kann. Alle Items einer Transaktion (TID) sind in einem Vektor abgespeichert, was einem Warenkorb entspricht. Ein weiterer Vektor referenziert alle Warenkörbe.

Aufgerufen wird der Operator als Funktion `apriori` und erwartet die Warenkorbdaten in relationaler Form; optional kann der minimale Support und die minimale Konfidenz angegeben werden, die Ausgabe erfolgt in aggregierter Form mit einem Tupel pro Assoziationsregel (s. Abb. 5.2).

```
> select * from apriori((select * from sales),0.1,0.69);
ID Pre Post Support Confidence Lift Conviction
0 {41} {39} 0.129466209931717 0.76373369019739 1.32870823078801
    1.79968896693493
1 {48} {39} 0.330550577346249 0.691634033463866 1.2032726128908
    1.37890012891666
```

Abbildung 5.2.: Aufruf des Apriori-Operators mit Beispielausgabe

6. Evaluierung

Diese Kapitel erklärt die Evaluation des Apriori-Algorithmus, zuerst den Versuchsaufbau mit der verwendeten Konfiguration, dann den Grund für die gewählten Experimente und interpretiert die Ergebnisse der Experimente im vorletzten Unterkapitel. Anschließend folgen die Ergebnisse unter Verwendung einer anderen Hashtabelle.

6.1. Versuchsaufbau

Die Versuche laufen auf einer Maschine mit 4 Prozessoren des Typs Intel Xeon E7-4870 v2 (2.30GHz) mit jeweils 15 Kernen, zusammen stehen folglich 60 Kerne und 120 Hyperthreads zur Verfügung. Der Hauptspeicher der Maschine beträgt ein Terabyte, das Betriebssystem ist ein Ubuntu 16.4 LTS. Aus der Literatur entstammen folgende Datensätze (s. Tab. 6.1):

- **accidents**: Unfalldaten unter Angabe verschiedener Faktoren und Schwere des Unfalls [12]
- **kosarak**: Daten eines ungarischen Nachrichtenportals [4]
- **retail**: Daten eines belgischen Onlinewarenhauses [9]
- **T10I4D100K, T40I10D100K**: generierte Daten [20]
- Amazon Review Daten mit **metadata.json**: Metadaten zu 142,8 Mio. Kommentaren zwischen Mai 1996 bis Juli 2014 auf Amazon [27] [26]

Die generierten Daten entstammen einem nicht mehr verfügbaren Generator der *IBM Almaden Quest Forschungsgruppe* und generiert Datensätze der Form `T<avgI>I<avgF>D<#objects>`, wobei `<avgI>` für die durchschnittlichen Items pro Warenkorb, `<avgF>` für die durchschnittlichen Größe eines Frequentitemsets und `<#objects>` für die Anzahl an Warenkörben steht.

Die Amazon Review Daten bestehen aus zwei Datensätzen, den Reviews, also den Rezensionen zu Produkten, und den Metadaten, die Informationen zu den Produkten enthalten. Ein Feld steht für die Produkte, die gemeinsam mit dem zugehörigen gekauft werden (*bought_together*, bt), ein weiteres für die Produkte, die gekauft werden, nach dem Kunden das zugehörige angeschaut haben (*also_bought*, ab), und ein letztes Feld entspricht den Produkten, die gemeinsam mit dem zugehörigen angeschaut werden (*also_viewed*, av). Wenn Frequentitemsets aus allen Produkten, die zwischen Mai 1996 und Juli 2014 eingekauft worden sind, berechnet werden sollen, so benötigt der Operator 107 Stunden (unter Verwendung von `std::unordered_map`). Außerdem sind Versuche mit einer Teilmenge der Amazondaten möglich, wie die auf Amazon verkauften Bücher (*books*). Um die Daten in relationale Form zu überführen, werden die JSON-Daten geparst und gefiltert, anschließend mit einer Transaktions-ID versehen und als SQL-Insert-Statement abgespeichert. Die anderen Datensätze liegen als Warenkorbdaten vor, woraus SQL-Insert-Statements erzeugt werden.

6.2. Experimente

Sinnvolle Laufzeitmessungen sind Skalierungstests, wie sich die Laufzeit in Abhängigkeit verfügbarer Rechenkerne verändert, und Lasttests mit Variierung des minimalen Supports, um die Laufzeit bei

	Warenkörbe	vers. Items	Items gesamt	Durchschnittsgröße	Maximum
accidents	340 183	468	11 500 870	33	51
kosarak	990 002	41 270	8 019 015	8	2498
retail	88 162	16 470	908 576	10	76
T10I4D100K	100 000	870	1 010 228	10	29
T40I4D100K	100 000	942	3 960 507	39	77
Books bt	547 430	648 584	1 392 110	2	14
Books av	303 550	1 153 313	2 882 620	9	61
Books ab	1 268 996	2 672 055	51 628 616	40	101
Amazon bt	2 869 523	3 283 874	8 018 025	2	16
Amazon av	4 021 445	9 698 670	130 943 487	32	61
Amazon ab	4 126 717	7 063 924	152 520 523	36	101

Tabelle 6.1.: Übersicht der Datensätze

verschieden großen Ergebnissen zu testen. Folgende Faktoren beeinflussen die Laufzeit des Algorithmus:

- Datenbasis mit
  - Anzahl der Transaktionen: Umso mehr Transaktion, desto mehr durchlaufen pro Iteration den Präfixbaum
  - Anzahl verschiedener Items: Umso größer, desto mehr Kinder besitzt ein Baum und desto größer wird das Wörterbuch
- Gewählter Support: Umso größer, desto tiefer ist der Präfixbaum und desto mehr Iterationen sind nötig
- Anzahl Rechenkerne: Mit jedem weiteren Rechenkern reduziert sich die Last pro Rechenkern

Die Tests sollen alle drei Faktoren abdecken, Skalierungstests messen die Laufzeit in Abhängigkeit der verfügbaren Rechenkerne bei einem gleichbleibenden festgesetzten minimalen Support. Lasttests messen die Laufzeit in Abhängigkeit des minimalen Supports und konstanter Rechenleistung. Um den Einfluss unterschiedlicher Datenbasen zu testen, erfolgen die Messungen auf unterschiedlichen Datensätzen, da die Verteilung der Items auf Warenkörbe pro Datenbasis variiert.

Um die Effizienz der vier verschiedenen Implementierungen zu testen (Apriori und AprioriTID, jeweils in einer seriellen und einer parallelen Variante), misst ein weiterer Test auf den Datenbasen *kosarak* und *retail* die Laufzeit in Abhängigkeit des Supports, wenn nur ein Kern zur Verfügung steht.

Ein Bash-Skript sorgt für die Reproduzierbarkeit der Messungen. Es ruft dazu ein universelles SQL-Skript auf den jeweiligen Datensätzen auf, das jeweils einem Skalierungs-, Lasttest oder einem Vergleich der Implementierungen entspricht. Um aussagekräftige Ergebnisse zu erhalten, wiederholt das Skript alle Messungen zehnmal. Im Nachfolgenden wird der Mittelwert über alle Ergebnisse als Vergleichswert genommen. Nach sechs Stunden terminiert ein Aufruf wegen einer gesetzten Zeitbegrenzung, um Ressourcen nicht zu lange zu belegen. Bei zehn Abfragen eines SQL-Skripts darf eine maximal 36 Minuten dauern.

Damit die Abfragen der Tests sinnvolle (weder zu viele noch zu wenige) Ergebnisse liefern, aber dennoch nicht zu lange laufen und nicht zu viel Speicherplatz benötigen, wird der minimale Support jeweils in Abhängigkeit erwarteter Ergebnisse pro Datenbasis gewählt (s. Abb. 6.5).

6.3. Ergebnisse

Im nachfolgenden Teil werden die Ergebnisse der Skalierungs-, der Lasttests und des Vergleichs der Implementierungen, erklärt, analysiert und interpretiert.

6.3.1. Skalierungstests

Sowohl die Skalierungstests bei einem Support von $0,1$ wie bei einem Support $0,001$ (s. Abb. 6.1) zeigen, wie die Laufzeit mit Anzahl verfügbarer Rechenkerne skaliert. Bis zur Verteilung auf sieben Kerne reduziert sich die Laufzeit deutlich, bei den Daten *kosarak* verbessert sich die Laufzeit bei der Nutzung von zwei Kernen um 16,5 %, in den nächsten drei Schritten jeweils mit der Hinzunahme eines weiteren Rechenkernes um 10 %. Mit der Annäherung an alle 60 Rechenkerne nähert sich die Laufzeit einem Grenzwert an. Dennoch sollte sich die Rechenzeit im Optimum um die Anzahl verfügbarer Kerne halbieren, was offensichtlich nicht passiert.

Die Vermutung liegt nahe, dass der Grenzwert der Laufzeit entspricht, die der Apriori-Operator benötigt, um alle Datenstrukturen zu initialisieren. Ein späterer Vergleich mit den Lasttests bestätigt die Vermutung, da mit Abnahme der produzierten Ergebnisse sich die Laufzeit ebenfalls an diesen Grenzwert annähert. Wenn man den Grenzwert von der Laufzeit abzieht, entspricht die Skalierung anfangs einer Halbierung je weiterem verfügbaren Kern.

Ein Problem bezüglich der Skalierung sind die seriellen Abschnitte: Das Erstellen der Warenkörbe läuft seriell ab, da gleichzeitig ein Wörterbuch erzeugt wird, damit dasselbe Element nicht zwei Kodewörter parallel zugewiesen bekommt. Zweitens läuft das Generieren der Assoziationsregeln noch seriell ab, da dessen Parallelisierung nicht so entscheidend ist wie die des Findens der Frequentitemsets. Das Optimum, mit jedem weiteren verfügbaren Kern die Laufzeit zu halbieren, würde dennoch nicht erreicht werden, da eine Synchronisation nach jedem Schritt eine kurzfristige serielle Ausführung bedingt. Die Messungen ergeben, dass ab ungefähr zehn verfügbaren Kernen der Grenzwert erreicht wird. Die Struktur der Warenkörbe variiert und somit auch die Anzahl der Elemente pro Transaktion. Mit jeder Iteration k scheiden Warenkörbe mit weniger als k Elementen aus, ab einem gewissen Punkt verdichten sich die relevanten auf wenige Warenkörbe. Dadurch reduziert sich die Laufzeit nur in den ersten Iterationen.

6.3.2. Lasttests

Die Lasttests (s. Abb. 6.2) ergeben die erwarteten Messwerte. Umso niedriger der Support ist, desto mehr Assoziationsregeln produziert der Algorithmus, wodurch sich die Laufzeit verlängert. Alle Ergebnisse liegen unter den Zeiten bisheriger Veröffentlichungen ([4]). In den Ergebnissen lässt sich die unterschiedliche Struktur der Datensätze erkennen, manche Datensätze liefern bereits für einen minimalen Support von $0,5$ erste Ergebnisse, während andere erst bei einem minimalen Support von $0,1$ einsetzen.

6.3.3. Vergleich der Implementierungen

Bei allen Versuchen liefert die klassische Implementierung des Apriori-Algorithmus schneller die Ergebnisse und ist um ca. 30 % schneller als der AprioriTID-Algorithmus, der sich pro Knoten die relevanten Transaktionen merkt. Der Aufbau der Struktur, um die Transaktionen zu referenzieren, benötigt mehr Zeit, als was durch die Reduktion der Vergleiche eingespart wird. Dies liegt an der Struktur des Präfixbaumes, der Transaktionen nur mit darin enthaltenen Item-Mengen vergleicht und Item-Mengen mit zu wenig Elementen bereits weglässt.

Der serielle Apriori-Algorithmus ist etwas schneller als die parallel implementierte Variante, was an der zweigeteilten Iteration liegt. Erfolgt die Kandidatengenerierung bei der seriellen Variante implizit, sobald der Schwellenwert überschritten ist, so teilt die parallele Variante das Zählen der Häufigkeit auf zwei Schritte auf, zuerst zählt jeder Thread die Häufigkeit der Item-Mengen seiner Transaktionen, anschließend bildet er die Summe und generiert bei Bedarf die Kandidaten.

Der parallele AprioriTID-Algorithmus ist etwas schneller die serielle Variante, doch unterscheidet sich diese nur durch eine parallele Schleife in der letzten Ebene der Rekursion. Einzige Erklärung wäre eine effizientere Implementierung der parallelen Schleife.

6.4. Verbesserungen

Die Laufzeit von über vier Tagen mit den Amazon-Daten (bt) zeigt Verbesserungsbedarf, wenn es zu Kollisionen in der Hashtabelle kommt. Abhilfe schafft hierbei eine Hashtabelle aus HyPer, die eine deutlich bessere Zugriffszeit ermöglicht, wenn zwei Items denselben Hashwert besitzen. Außerdem kann anstelle eines Zählers pro Thread, die anschließend zusammengerechnet werden, auch der entsprechende Abschnitt exklusiv gesetzt werden. Ein Mutex (`std::mutex`) schützt den kritischen Bereich, der aus der Zählvariable und dem Generieren der Kindknoten besteht. Wenn nur wenige Warenkörbe identisch sind, finden auch wenige Zugriffe gleichzeitig statt. Dadurch bleiben unnötige Zählvariablen erspart, die nebeneinander liegen und deswegen ein schlechtes Cache-Verhalten aufweisen. Alternativ kann die Variable auch atomar gesetzt werden (`std::atomic`), um sie vor gleichzeitigem Zugriff zu schützen. Um diese Änderungen erweitert, folgen die gleichen Messungen auf denselben Datensätzen.

Die Skalierungs- (s. Abb. 6.6) und Lasttests (s. Abb. 6.7) mit den Datenbasen *accidents, kosarak, retail, T10I4D100K, T40I4D100K* ergeben ähnliche Ergebnisse wie in der ursprünglichen Implementierung: Umso mehr Kerne, desto geringer die Laufzeit. Dennoch nähert sich die Laufzeit gegen 60 verfügbaren Kernen einem Grenzwert an. Ein Vergleich mit den Lasttests, die sich mit steigendem Support ebenso diesem Grenzwert annähern, zeigt, dass bei keinen Kollisionen die Laufzeit unverändert bleibt. Der Grenzwert entspricht der Laufzeit um den Operator zu initialisieren.

Ein anderes Bild zeichnet sich ab, wenn, wie bei den Amazon-Review-Datensätzen, Items einen gleichen Hashwert besitzen und kollidieren. Mit der Hashtabelle aus HyPer sind nun Laufzeiten von einigen Sekunden bis zu wenigen Minuten möglich (s. Abb. 6.10), während zuvor noch über vier Tage nötig gewesen wären. Damit ist der Operator nun fähig, große Mengen von Daten wie alle verkaufte Produkte innerhalb von acht Jahren eines Onlinewarenhauses zu analysieren.

Die Implementierungen, in der der kritische Bereich mittels eines Mutex gesichert wird, hat im Vergleich zu der Implementierung mit mehreren Zählern, die anschließend addiert werden, eine längere Laufzeit (s. Abb. 6.8) in der parallelen Ausführung. Seriell ausgeführt ist die Variante mit nur einem Zähler schneller. Das bedeutet, dass ein anschließender Additionsschritt weniger zeitintensiv ist als das Sicheren des kritischen Bereiches bei gleichzeitigem Zugriff. Wenn die Variable als atomar gesetzt ist, ergibt sich das gleiche Ergebnis (s. Abb. 6.9): In der seriellen Ausführung ist die Implementierung mit nur einem Zähler schneller, parallel die Variante mit mehreren Zählern.

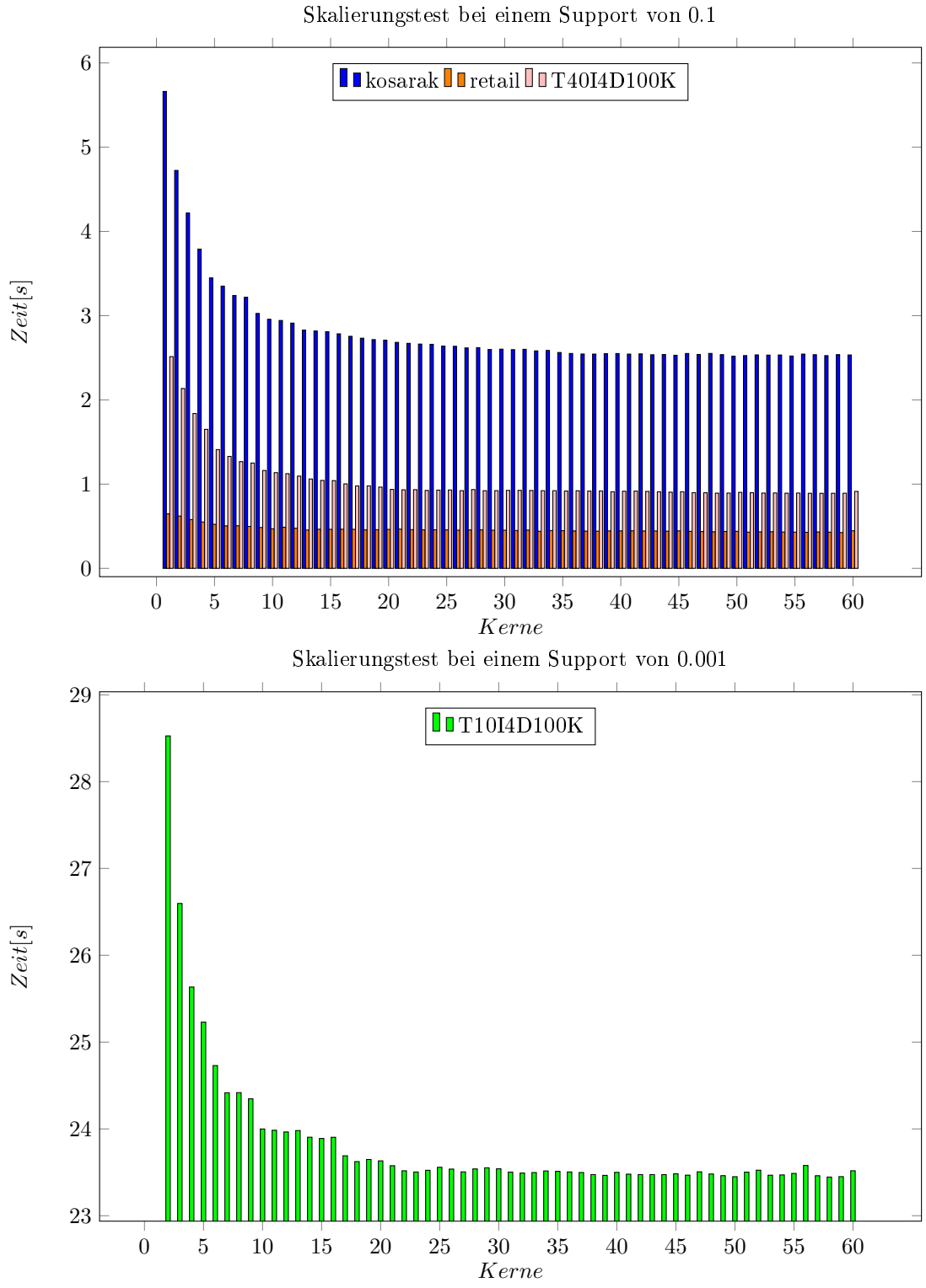

Abbildung 6.1.: Skalierungstest (1)

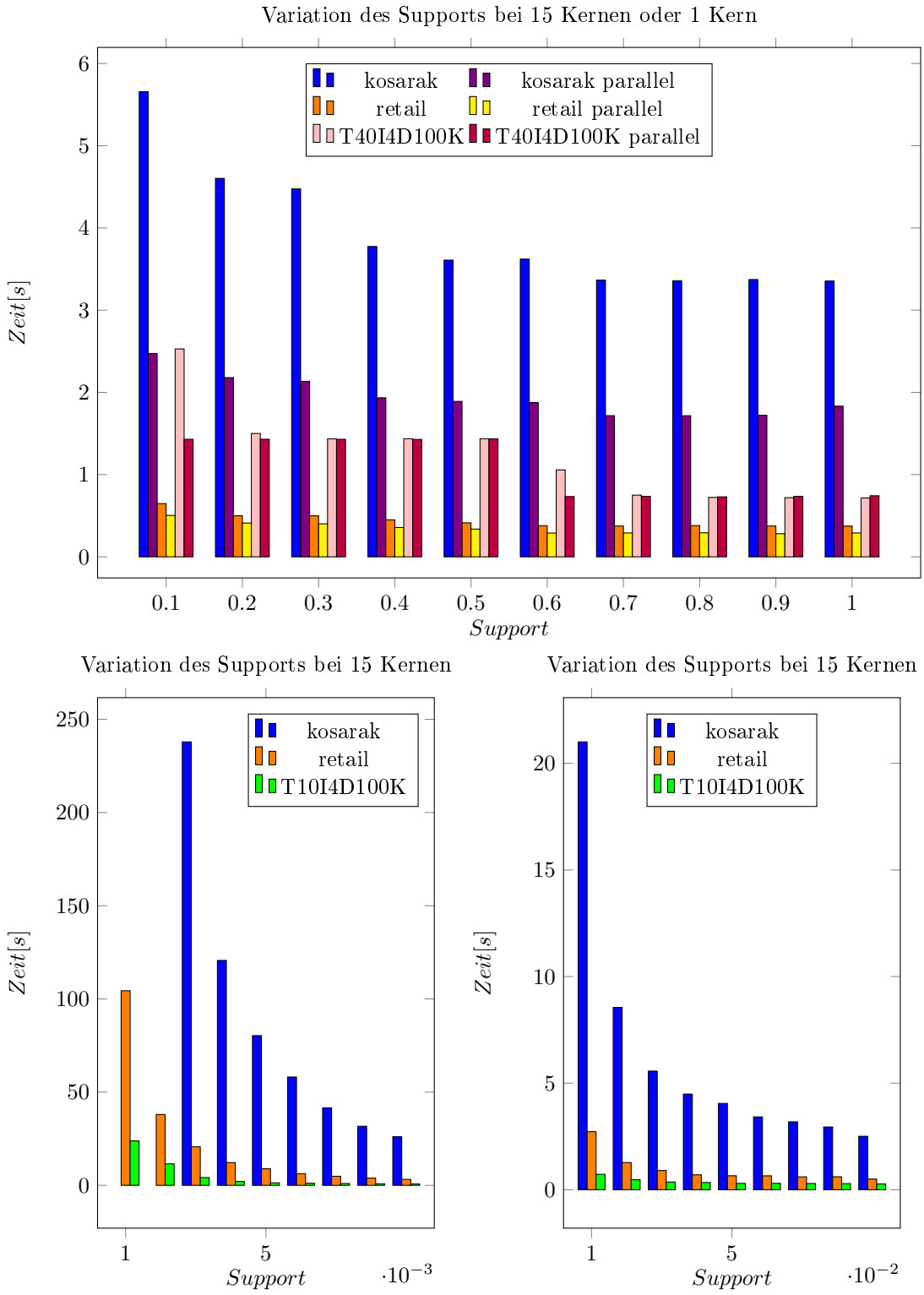

Abbildung 6.2.: Lasttests (1)

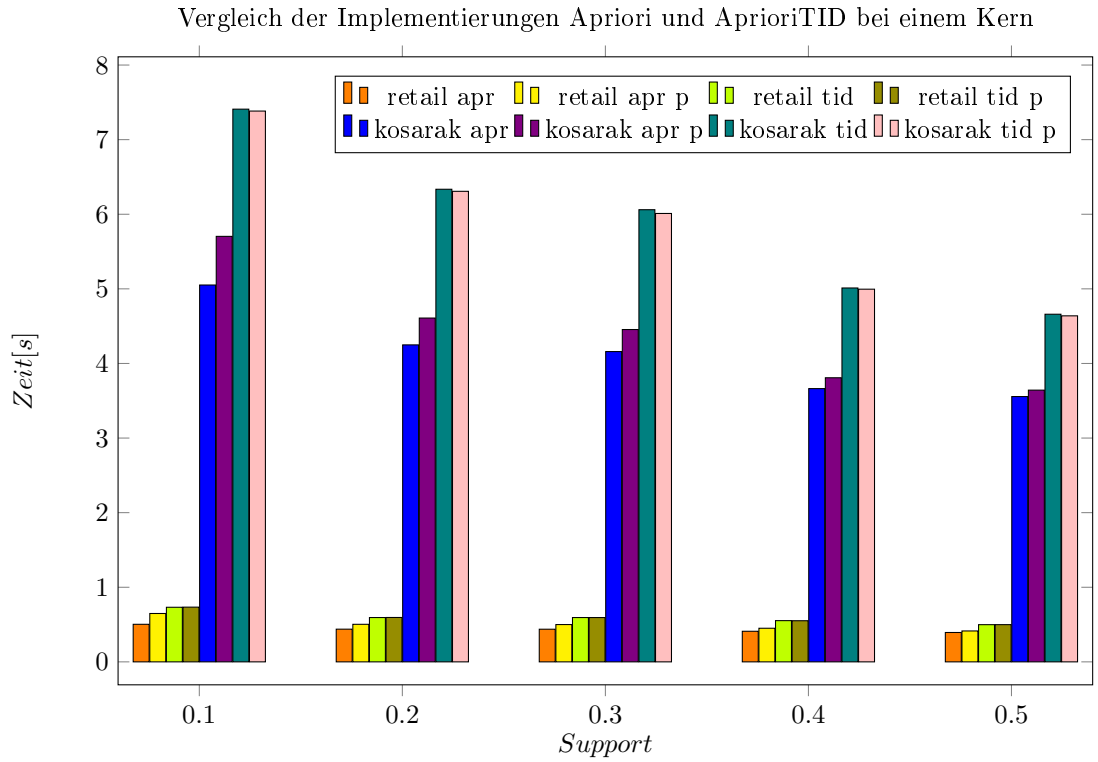

Abbildung 6.3.: Vergleich der Implementierungen von Apriori (apr) und AprioriTID (tid) in der seriellen wie parallelen (p) Implementierung (1)

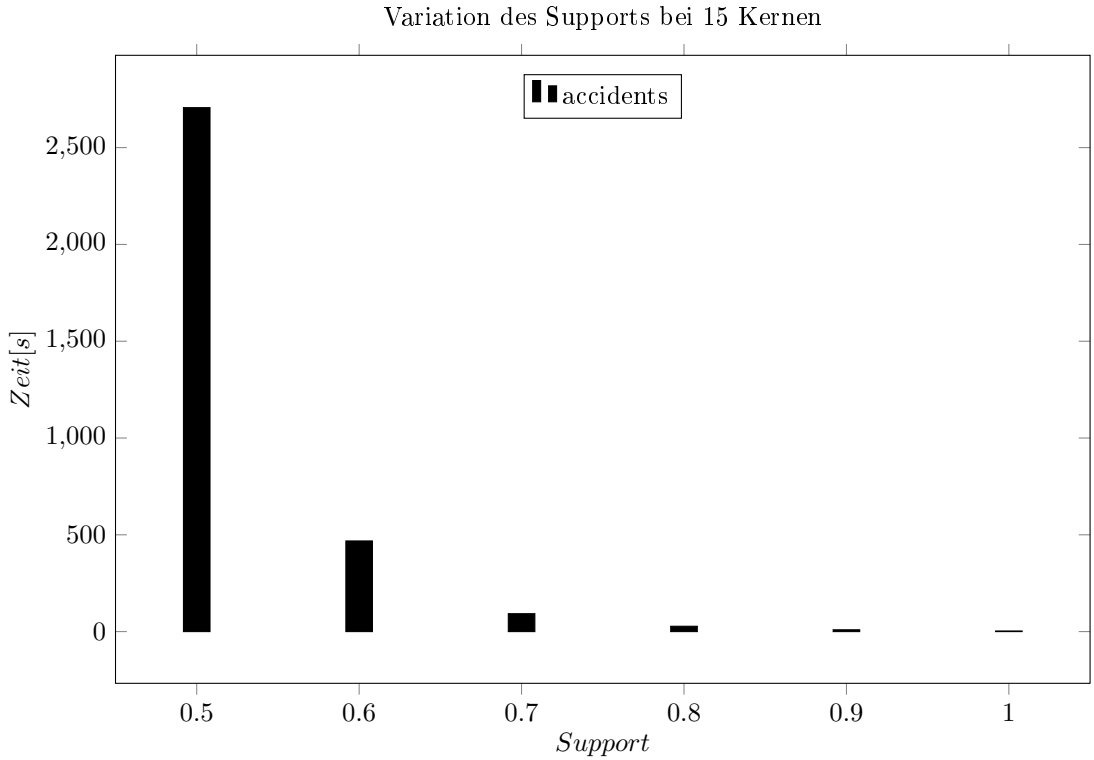

Abbildung 6.4.: Lasttest mit Unfalldaten (accidents)

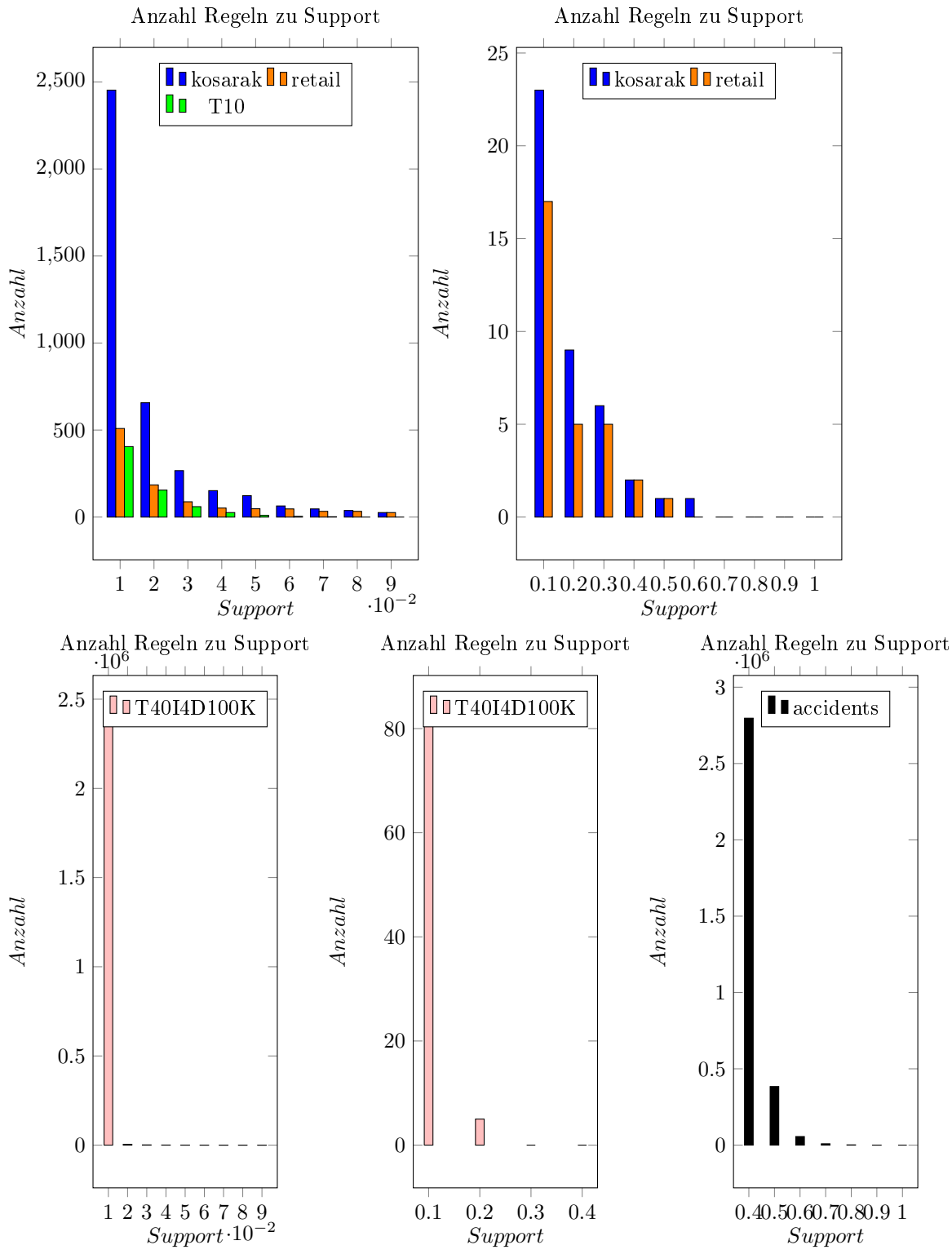

Abbildung 6.5.: Erzeugte Regeln in Abhängigkeit des Supports bei $k_0 = 0$

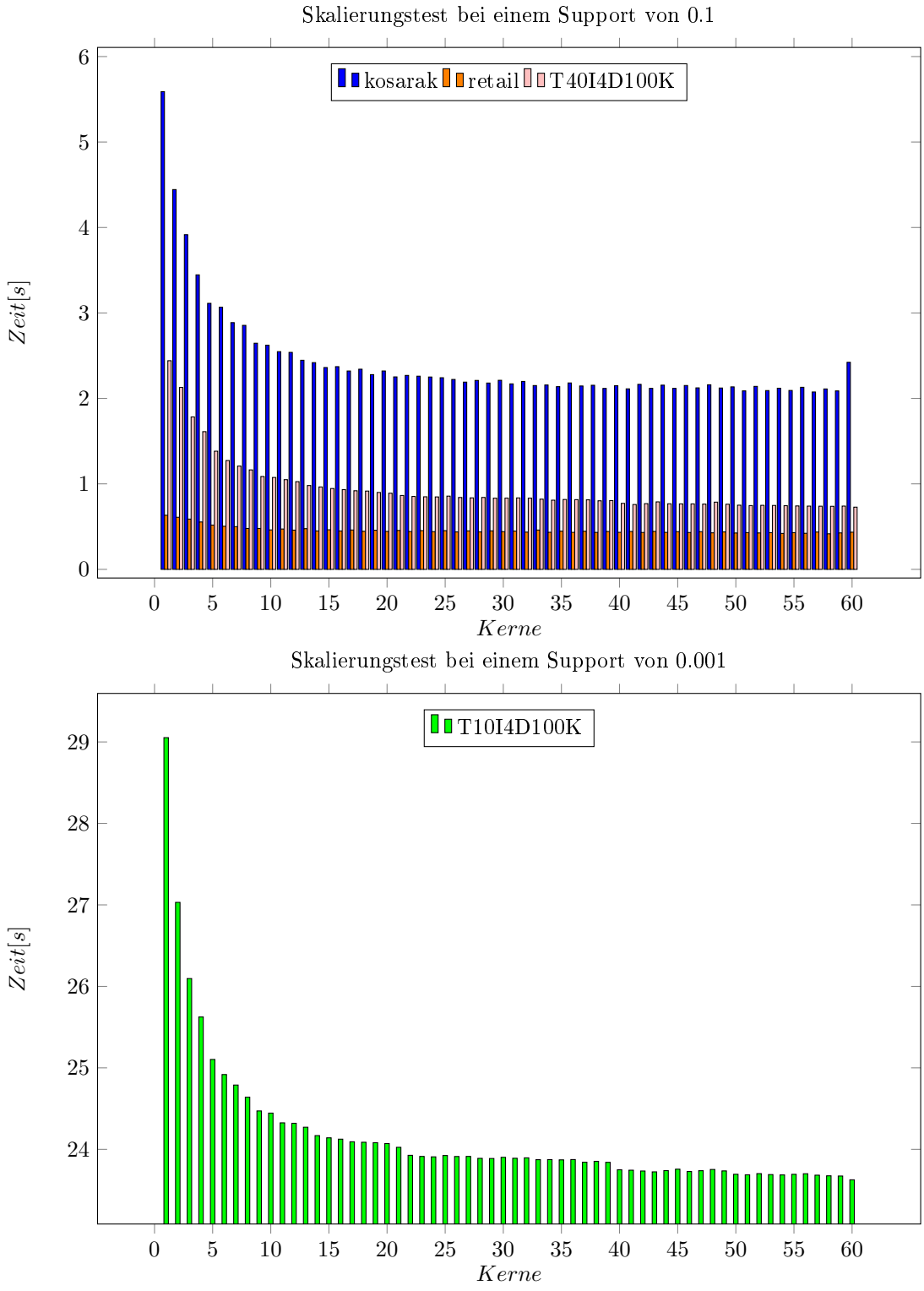

Abbildung 6.6.: Skalierungstest (2)

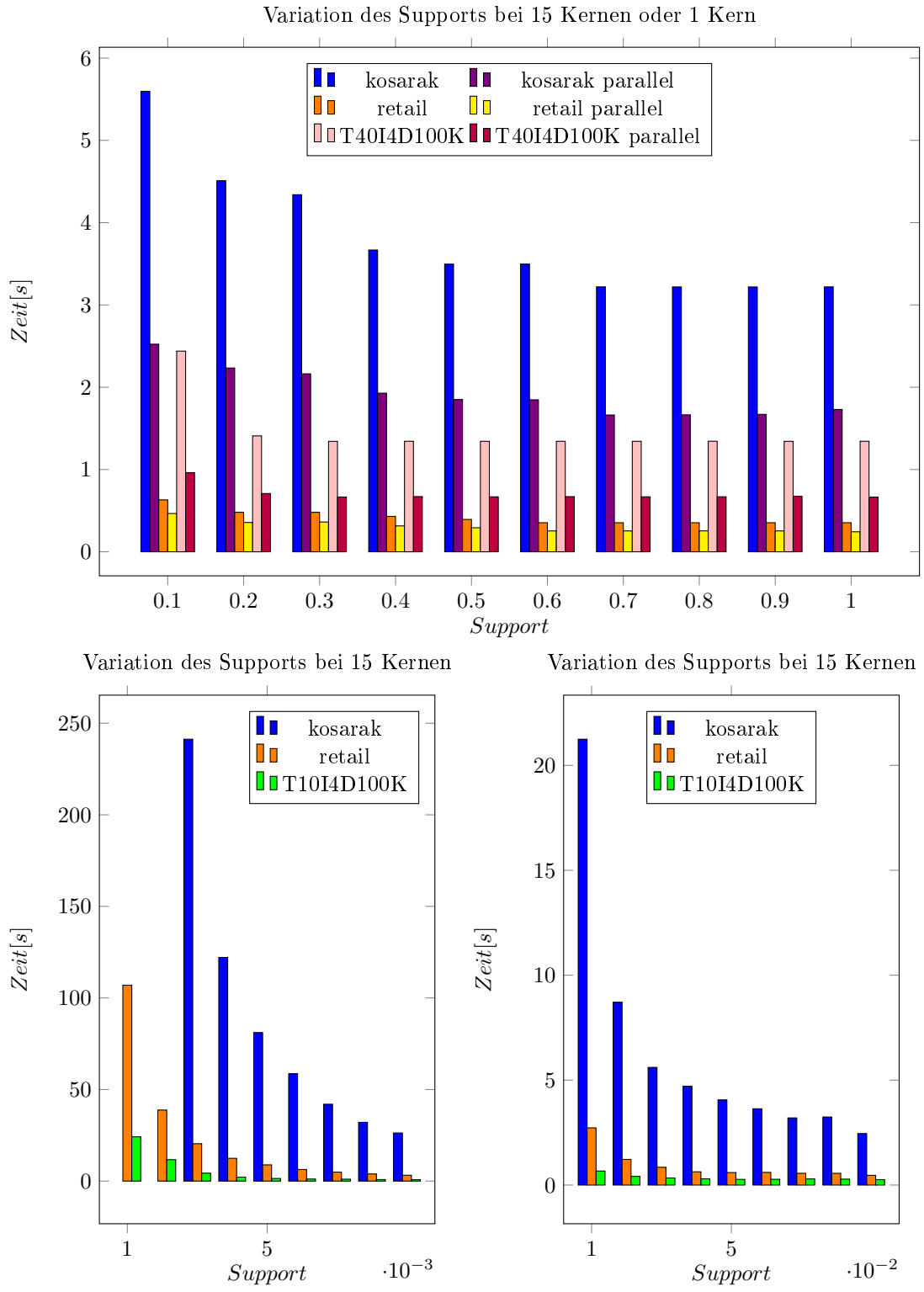

Abbildung 6.7.: Lasttests (2)

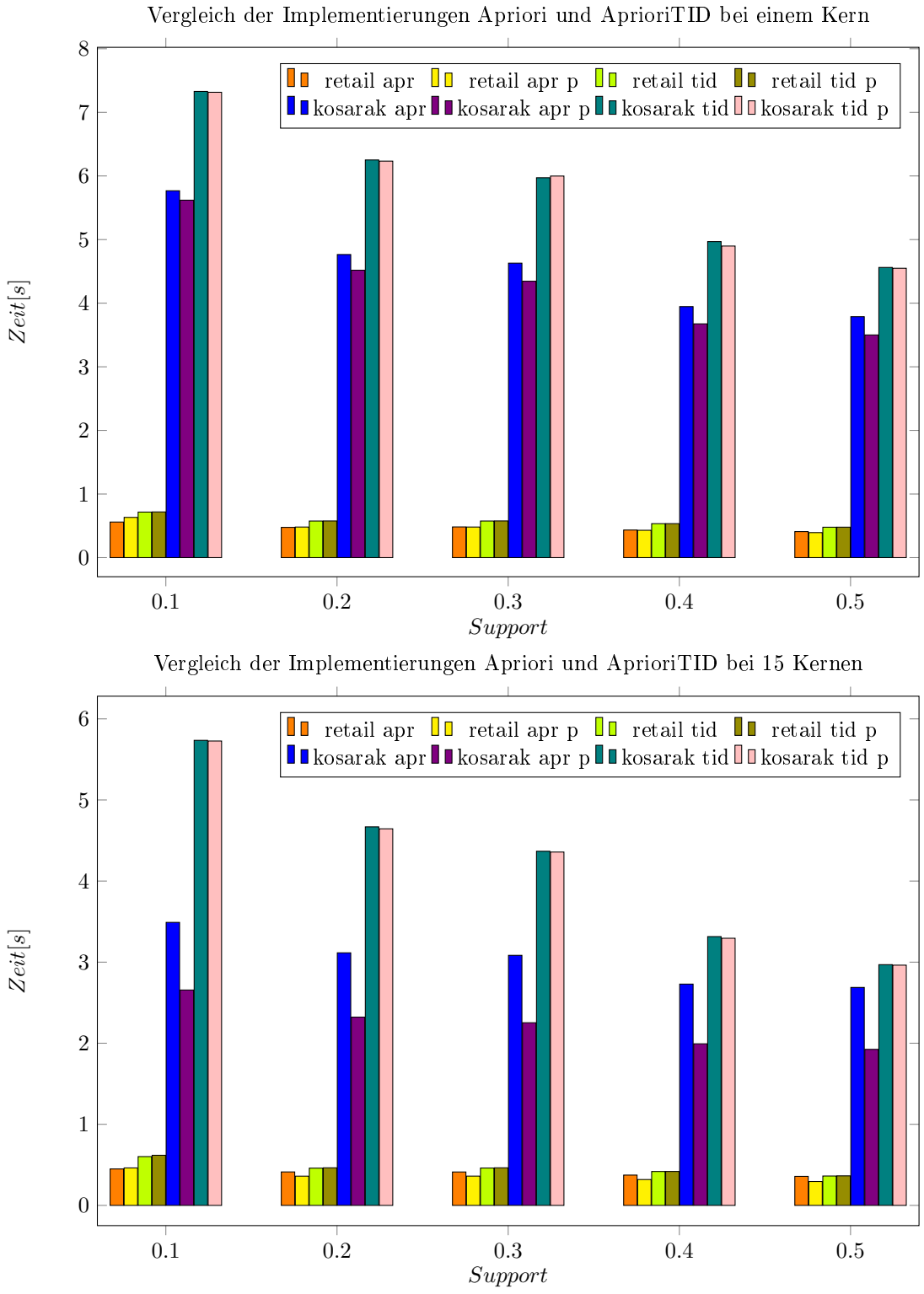

Abbildung 6.8.: Vergleich der Implementierungen von Apriori (apr) und AprioriTID (tid) in der seriellen wie parallelen (p) Implementierung (2)

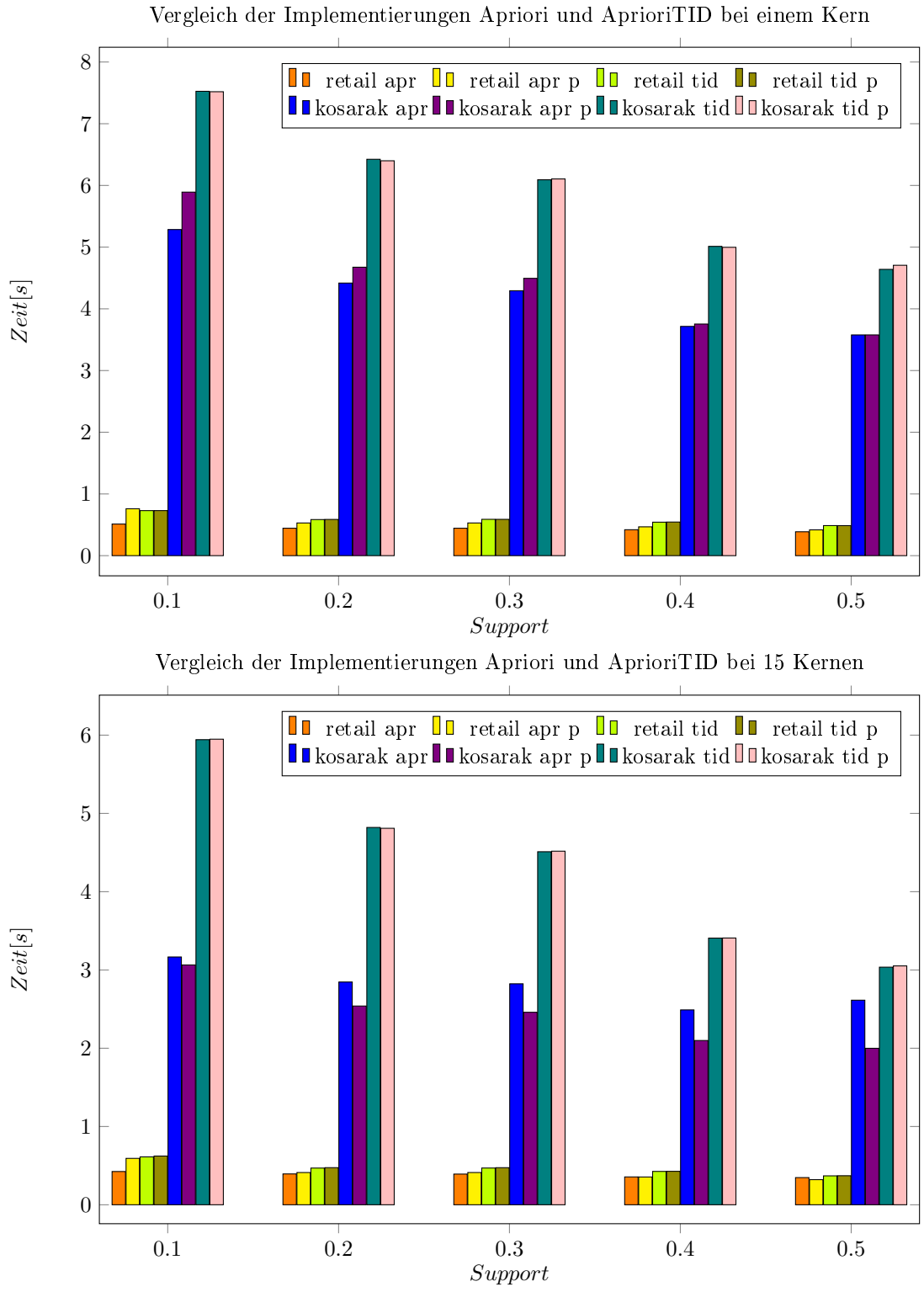

Abbildung 6.9.: Vergleich der Implementierungen von Apriori (apr) und AprioriTID (tid) in der seriellen wie parallelen (p) Implementierung (3)

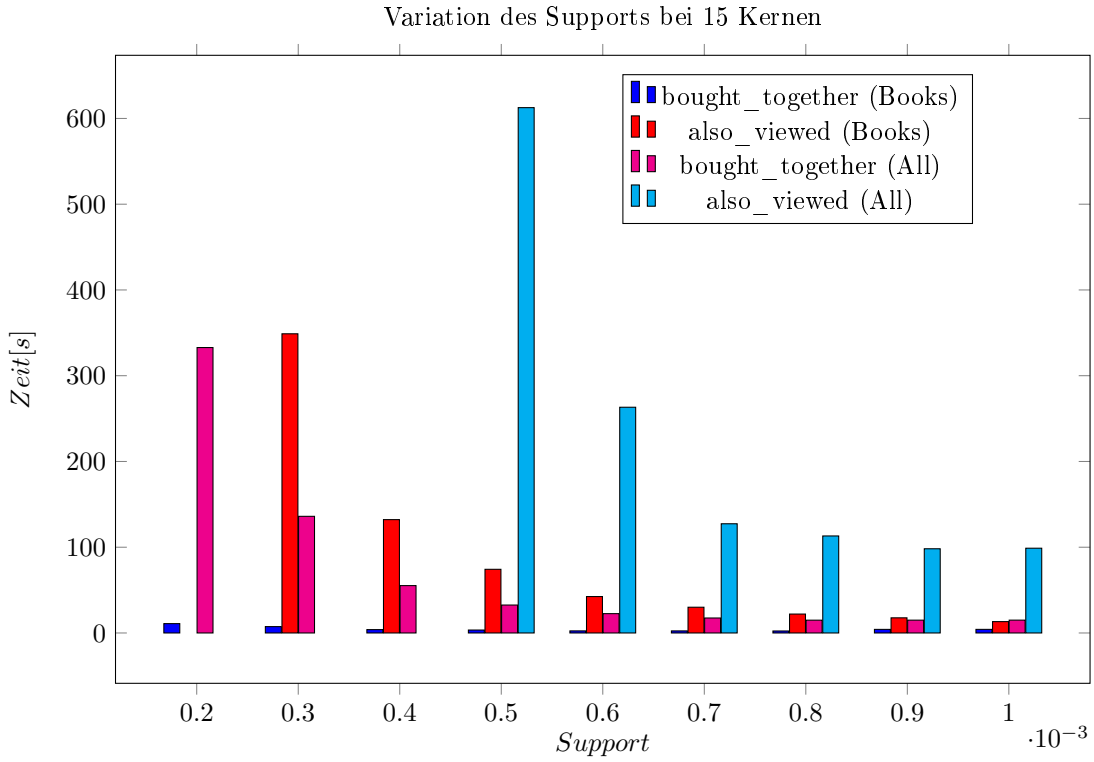

Abbildung 6.10.: Lasttest mit Amazondaten

7. Zusammenfassung und Ausblick

Das letzte Kapitel fasst das Vorgehen und die Ergebnisse dieser Arbeit zusammen und zeigt auf, wie an Algorithmen zur Assoziationsanalyse weiter geforscht werden kann.

7.1. Zusammenfassung

Die vorliegende Arbeit zeigte einen Weg auf, den Apriori-Algorithmus als Operator in einer Hauptspeicherdatenbank zu integrieren, um so das Gebiet der Assoziationsanalyse mit der effizienten Speicherung der Daten in Datenbanksystemen zusammenzuführen. Um einen Operator aufbauend auf bestehenden Arbeiten zu entwickeln, behandelte die Arbeit zunächst die Grundlagen der Assoziationsanalyse, um Begriffe wie Frequentitemsets, also Mengen aus Transaktionen, die besonders häufig vorkommen, und Assoziationsregeln, die eine Korrelation von Elementen in Warenkörben beschreiben, zu definieren. Zugehörige Kennzahlen, wie der Support, die Häufigkeit und die Konfidenz, die Wahrscheinlichkeit einer Assoziationsregel, wurden ebenso eingeführt. Die erste Veröffentlichung zum Apriori-Algorithmus wurde behandelt, um darauf basierend Verbesserungsmöglichkeiten zu diskutieren und in das vorgestellte Konzept zu übernehmen. Übernommene Verbesserungen waren die Datenstruktur eines Präfixbaumes, um Frequentitemsets zu verwalten, und weitere Detailfragen, wie Itemcoding. Ein Vergleich mit bestehenden produktreifen Systemen grenzte die Anforderungen an einen Apriori-Operator ein und ein Vergleich mit anderen Algorithmen benannte die Vor- und Nachteile.

Dass das Hauptspeicherdatenbanksystem HyPer, das die Transaktionen ausschließlich im Hauptspeicher verarbeitete und pro Anfrage Maschinencode generierte, bereits fähig war, Data-Mining-Algorithmen innerhalb der Datenbank auszuführen, zeigte eine Veröffentlichung zur Integration von Data- und Graph-Mining-Algorithmen. Das darauf aufbauend entwickelte Konzept schlug einen Operator vor, der auf dem klassischen Apriori-Algorithmus unter Nutzung eines Präfixbaumes basierte und Duplikate akzeptierte. Ein davon abweichender Vorschlag basierte auf dem AprioriTID-Algorithmus und nutzte einen Präfixbaum, wie es ihn in dieser Kombination nicht gegeben hatte. Anhand dieses Konzeptes wurde der Operator unter Berücksichtigung der Systemarchitektur in der Hauptspeicherdatenbank implementiert, sodass wichtige Teile in Maschinensprache und die Datenstruktur in Hochsprache geschrieben waren.

Die Evaluierung des Operators erfolgte auf generierten wie echten Datensätzen, die teilweise aus der Literatur übernommen wurden, um eine Vergleichbarkeit der Ergebnisse zu schaffen. Die Messungen zeigten sowohl eine gute Skalierung in Abhängigkeit verfügbarer Kerne wie den erwarteten Anstieg der Laufzeit, umso kleiner der minimale Support gewählt wurde. Durch Parallelisierung des Operators konnte eine Halbierung der Laufzeit bei ausreichend Kernen festgestellt werden. Den Vergleich beider Konzepte gewann der gewöhnliche Apriori- im Vergleich zum AprioriTID-Algorithmus mit deutlichem Vorsprung. Ursächlich war die Datenstruktur eines Präfixbaumes, die die erforderlichen Vergleiche auf die nötigsten reduzierte. Damit führte die vorliegende Arbeit die erfolgreiche Integration eines Apriori-Algorithmus in einer Hauptspeicherdatenbank vor und zeigte, dass Assoziationsanalyse in Hauptspeicherdatenbanksystemen betrieben werden kann und soll.

7.2. Ausblick

Dennoch bietet die Assoziationsanalyse im Zusammenspiel mit Hauptspeicherdatenbanken weitere Varianten und Möglichkeiten, die es wert sind, erforscht zu werden. Ein Vergleich mit Eclat- und FP-Growth-Algorithmus ist bisher unerforscht in Zusammenhang mit Hauptspeicherdatenbanken. Gerade letzterer könnte bei ausreichend großem Hauptspeicher, wie es hier der Fall ist, zu unerforschten Ergebnissen führen, da die Restriktion der Begrenztheit des Speichers entfällt.

Eine Verbesserung wäre das Wörterbuch parallel zu erstellen. Wenn die Items zu Warenkörben gebündelt werden, wird auch das Wörterbuch erstellt. Damit für das Itemcoding ein Item nicht mehrere Kodewörter zugewiesen bekommt, läuft das seriell ab. Eine Möglichkeit, ein Wörterbuch parallel zu befüllen, ist zuerst die Menge verschiedener Items abzufragen und anschließend aus einer Abfrage, die eine duplikatfreien Menge aller Items liefert, das Wörterbuch parallel zu erstellen. Die Anzahl aller Items ist bekannt, entsprechend können mehrere Threads für einen disjunkten Bereich das Wöterbuch erstellen. Das Bündeln zu Warenkörbe läuft anschließend und unabhängig ab.

Das Gebiet der Assoziationsanalyse erfordert eine Form um Mengen von Attributen darstellen, zum Beispiel Felder, die sich nur schwer in den SQL-Standard aufbrechen lassen. Aus diesem Grund wäre es einfacher, Assoziationsalgorithmen in Datenbanksysteme zu integrieren, wenn diese Felder oder eine alternative Repräsentation unterstützen. So könnten Warenkorbdaten bereits aggregiert als Eingabe übergeben werden. Die Materialisierung der Daten basiert im Moment auf deren relationalen Schemata, interessant wäre der Vergleich, wenn die Daten aggregiert materialisert werden.

Aus gutem Grund fokussieren die meisten Verbesserungen und alternativen Ansätze das Finden der Frequentitemsets und vernachlässigen die Generierung der Assoziationsregeln daraus, denn die Kardinalität eines Frequentitemsets bestimmt die Anzahl möglicher Assoziationsregeln daraus. Umso mehr Hauptspeicher zur Verfügung steht, desto größere Item-Mengen können analysiert werden und desto speziellere Vorhersagen können getroffen werden. Doch aus größeren Item-Mengen folgen mehr Assoziationsregeln, weshalb sich dieser Schritt zu optimieren lohnt, gerade da er unabhängig vom Algorithmus zum Finden der Frequentitemsets ist. Basierend auf dem Apriori-Prinzip für Assoziationsregeln ist eine Idee hierfür die Adaption des Präfixbaumes für Assoziationsregeln und deren iterative Generierung eine Möglichkeit, die Generierung zu beschleunigen.

Ein Filtern nach Elemente, damit nur solche Assoziationsregeln mit bestimmten Items in der Prämisse oder in der Konklusion erscheinen, ist eine potentielle Erweiterung des Apriori-Operators. Zwar ist es aktuell möglich, den Apriori-Operator mit einer vorgestellten Abfrage nur mit den relevanten Warenkörben aufzurufen, allerdings muss die relative Häufigkeit angepasst und nachberechnet werden und die Assoziationsregeln müssten noch gesondert gefiltert werden. Diese beiden Abfragen könnten in den Apriori-Operator integriert sein, am besten durch eine Erweiterung der SQL-Sprache.

Appendix

A. Algorithmus

```
L = ∅;
F = {∅};
while F ≠ ∅ do
    C = ∅;
    forall T ∈ D do
        forall F ∈ F do
            if F ⊆ T then
                C_f = candidates(F, T);
                forall C_f ∈ C_f do
                    if C_f ∈ C then
                        C_f.count+ = 1;
                    else
                        C_f.count = 0;
                        C = C ∪ {C_f}
                    end
                end
            end
        end
        F = ∅;
        forall C ∈ C do
            if count(C) / |D| ≥ s_0 then
                L = L ∪ {C};
            end
            if asFrontier(C) then
                F = F ∪ C;
            end
        end
    end
end
```

Algorithmus 8 : Erster Algorithmus zum Finden von Frequentitemsets [1]

B. Hilfsskript

B.1. Parsen der Daten

```
#!/bin/bash
src=$1
dest="$2"
usage="$0 [amazon review data] [destination folder]"

if [[ $# -lt 2 ]]; then
    echo $usage
    exit 1
fi

# transforms basket data to sql insert statements
function dat2sql {
    destSqlFile=$1
    echo "create table sales (TID INTEGER, Item VARCHAR(26));" > "
        $destSqlFile"
    echo -n "insert into sales values " >> "$destSqlFile"
    i=1
    komma=""
    while read -r line
    do
        for item in $line
        do
            echo -n $komma >> "$destSqlFile"
            echo -n "($i,'$item')" >> "$destSqlFile"
            komma=","
        done
        i=`expr $i + 1`
        komma=",";
    done
    echo -n ";" >> "$destSqlFile"
}

#### WORK BEGINS ###
sed -nr "s/\{'asin': '([A-Z0-9]*)'.*bought_together.:.\[([A-Z0-9, \']*)
    \].*/\1, \2/p" $1 | tr -d \', | tee "$dest"/bought_together.dat |
    dat2sql "$dest"/bought_together.sql &
sed -nr "s/\{'asin': '([A-Z0-9]*)'.*also_bought.:.\[([A-Z0-9, \']*)
    \].*/\1, \2/p" $1 | tr -d \', | tee "$dest"/also_bought.dat |
    dat2sql "$dest"/also_bought.sql &
```

```
sed -nr "s/\{'asin': '([A-Z0-9]*)'.*also_viewed.:.\[([A-Z0-9,\ ']*)
    \].*/\1,\2/p" $1 | tr -d \', | tee "$dest"/also_viewed.dat |
    dat2sql "$dest"/also_viewed.sql &
wait
```

B.2. Ausführen der SQL-Skripte

```
#!/bin/bash
DEST=$1
echo Zielordner fuer Messungen ist $1

echo implementation ST
echo retail
timeout 300m hyper/bin/sql base/retail_n aprioritests/
    apriori_implementations.test > $1/vs_retail.dat
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/
    apriori_implementations.test > $1/vs_kosarak.dat

echo implementation MT
echo retail
timeout 300m hyper/bin/sql base/retail_n aprioritests/
    apriori_implementations_mt.test > $1/vs_15_retail.dat
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/
    apriori_implementations_mt.test > $1/vs_15_kosarak.dat

echo scale tests
echo retail
timeout 300m hyper/bin/sql base/retail_n aprioritests/apriori_scale.test
     | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/scale_retail.dat
echo accidents
timeout 300m hyper/bin/sql base/accidents aprioritests/apriori_scale.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/scale_accidents.dat
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/apriori_scale.test
    | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/scale_kosarak.dat
echo T10 scale
timeout 300m hyper/bin/sql base/T10I4D100K aprioritests/apriori_scale.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/scale_T10.dat
echo T40 scale
timeout 300m hyper/bin/sql base/T40I4D100K aprioritests/apriori_scale.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/scale_T40.dat

echo supp tests
echo retail
timeout 300m hyper/bin/sql base/retail_n aprioritests/
    apriori_dezisupp_10.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    supp_retail.dat
```

```
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/apriori_dezisupp_10
    .test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/supp_kosarak.dat
echo T40
timeout 300m hyper/bin/sql base/T40I4D100K aprioritests/
    apriori_dezisupp_10.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    supp_T40.dat

echo dezi supp
echo accidents
timeout 300m hyper/bin/sql base/accidents aprioritests/
    apriori_dezisupp_3.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    supp_accidents.dat

echo centiscale tests
echo retail
timeout 300m hyper/bin/sql base/retail_n aprioritests/apriori_centiscale
    .test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/centiscale_retail.dat
echo T10
timeout 300m hyper/bin/sql base/T10I4D100K aprioritests/
    apriori_centiscale.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    centiscale_T10.dat
echo T40
timeout 300m hyper/bin/sql base/T40I4D100K aprioritests/
    apriori_centiscale.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    centiscale_T40.dat

echo centi supp
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/apriori_centisupp.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/singlesupp_kosarak.dat
echo T10
timeout 300m hyper/bin/sql base/T10I4D100K aprioritests/
    apriori_centisupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    singlesupp_T10.dat
echo T40
timeout 300m hyper/bin/sql base/T40I4D100K aprioritests/
    apriori_centisupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    singlesupp_T40.dat
echo retail_n
timeout 300m hyper/bin/sql base/retail_n aprioritests/apriori_centisupp.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/singlesupp_retail.dat

echo milli supp
echo kosarak
timeout 300m hyper/bin/sql base/kosarak aprioritests/apriori_millisupp.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/millisupp_kosarak.dat
echo T10
timeout 300m hyper/bin/sql base/T10I4D100K aprioritests/
```

```
    apriori_millisupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    millisupp_T10.dat
echo T40
timeout 300m hyper/bin/sql base/T40I4D100K aprioritests/
    apriori_millisupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    millisupp_T40.dat
echo retail_n
timeout 300m hyper/bin/sql base/retail_n aprioritests/apriori_millisupp.
    test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/millisupp_retail.dat

echo mikro supp
echo bt
timeout 150m hyper/bin/sql base/books_bought_together aprioritests/
    apriori_mikrosupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    books_bt.dat
echo ab
timeout 150m hyper/bin/sql base/books_also_viewed aprioritests/
    apriori_mikrosupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    books_av.dat
echo av
timeout 150m hyper/bin/sql base/books_also_bought aprioritests/
    apriori_mikrosupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/
    books_ab.dat
echo all bt
timeout 150m hyper/bin/sql base/amazon_bought_together aprioritests/
    apriori_mikrosupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/a_bt.
    dat
echo all av
timeout 150m hyper/bin/sql base/amazon_also_viewed aprioritests/
    apriori_mikrosupp.test | sed -nr 's/.*avg (.*)\)/\1/g p' > $1/a_av.
    dat
```

C. SQL-Test-Skripte

C.1. Zählen

apriori_count.test

```
select count(*) from apriori((select * from sales), 1,0,1);
select count(*) from apriori((select * from sales), 0.9,0,1);
select count(*) from apriori((select * from sales), 0.8,0,1);
select count(*) from apriori((select * from sales), 0.7,0,1);
select count(*) from apriori((select * from sales), 0.6,0,1);
select count(*) from apriori((select * from sales), 0.5,0,1);
select count(*) from apriori((select * from sales), 0.4,0,1);
select count(*) from apriori((select * from sales), 0.3,0,1);
select count(*) from apriori((select * from sales), 0.2,0,1);
select count(*) from apriori((select * from sales), 0.1,0,1);
select count(*) from apriori((select * from sales), 0.09,0,1);
select count(*) from apriori((select * from sales), 0.08,0,1);
select count(*) from apriori((select * from sales), 0.07,0,1);
select count(*) from apriori((select * from sales), 0.06,0,1);
select count(*) from apriori((select * from sales), 0.05,0,1);
select count(*) from apriori((select * from sales), 0.04,0,1);
select count(*) from apriori((select * from sales), 0.03,0,1);
select count(*) from apriori((select * from sales), 0.02,0,1);
select count(*) from apriori((select * from sales), 0.01,0,1);
select count(*) from apriori((select * from sales), 0.009,0,1);
select count(*) from apriori((select * from sales), 0.008,0,1);
select count(*) from apriori((select * from sales), 0.007,0,1);
select count(*) from apriori((select * from sales), 0.006,0,1);
select count(*) from apriori((select * from sales), 0.005,0,1);
select count(*) from apriori((select * from sales), 0.004,0,1);
select count(*) from apriori((select * from sales), 0.003,0,1);
select count(*) from apriori((select * from sales), 0.002,0,1);
select count(*) from apriori((select * from sales), 0.001,0,1);
```

C.2. Skalierungstests

apriori_scale.test

```
\set repeat 10
\o /dev/null
\set limitworkers 60
select * from apriori((select * from sales), 0.1,1,1);
```

```
\set limitworkers 59
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 58
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 57
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 56
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 55
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 54
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 53
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 52
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 51
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 50
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 49
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 48
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 47
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 46
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 45
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 44
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 43
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 44
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 41
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 40
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 39
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 38
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 37
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 36
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 35
```

```
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 34
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 33
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 32
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 31
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 30
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 29
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 28
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 27
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 26
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 25
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 24
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 23
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 22
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 21
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 20
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 19
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 18
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 17
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 16
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 15
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 14
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 13
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 12
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 11
select * from apriori((select * from sales), 0.1,1,1);
```

```
\set limitworkers 10
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 9
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 8
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 7
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 6
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 5
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 4
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 3
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 2
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 1
select * from apriori((select * from sales), 0.1,1,1);
```

apriori_centiscale.test

```
\set repeat 10
\o /dev/null
\set limitworkers 60
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 59
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 58
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 57
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 56
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 55
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 54
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 53
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 52
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 51
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 50
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 49
select * from apriori((select * from sales), 0.001,1, 1);
```

```
\set limitworkers 48
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 47
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 46
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 45
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 44
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 43
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 44
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 41
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 40
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 39
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 38
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 37
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 36
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 35
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 34
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 33
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 32
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 31
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 30
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 29
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 28
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 27
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 26
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 25
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 24
```

```
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 23
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 22
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 21
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 20
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 19
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 18
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 17
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 16
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 15
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 14
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 13
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 12
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 11
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 10
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 9
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 8
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 7
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 6
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 5
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 4
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 3
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 2
select * from apriori((select * from sales), 0.001,1, 1);
\set limitworkers 1
select * from apriori((select * from sales), 0.001,1, 1);
```

C.3. Lasttests

apriori_dezisupp_10.test

```
\set repeat 10
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 1.0,1,1);
select * from apriori((select * from sales), 0.9,1,1);
select * from apriori((select * from sales), 0.8,1,1);
select * from apriori((select * from sales), 0.7,1,1);
select * from apriori((select * from sales), 0.6,1,1);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 1
select * from apriori((select * from sales), 1.0,1,1);
select * from apriori((select * from sales), 0.9,1,1);
select * from apriori((select * from sales), 0.8,1,1);
select * from apriori((select * from sales), 0.7,1,1);
select * from apriori((select * from sales), 0.6,1,1);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
```

apriori_dezisupp_3.test

```
\set repeat 3
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 1,1,1);
select * from apriori((select * from sales), 0.9,1,1);
select * from apriori((select * from sales), 0.8,1,1);
select * from apriori((select * from sales), 0.7,1,1);
select * from apriori((select * from sales), 0.6,1,1);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
\set limitworkers 1
select * from apriori((select * from sales), 1,1,1);
select * from apriori((select * from sales), 0.9,1,1);
select * from apriori((select * from sales), 0.8,1,1);
select * from apriori((select * from sales), 0.7,1,1);
```

```
select * from apriori((select * from sales), 0.6,1,1);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
```

apriori_centisupp.test

```
\set repeat 3
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 0.09,1,1);
select * from apriori((select * from sales), 0.08,1,1);
select * from apriori((select * from sales), 0.07,1,1);
select * from apriori((select * from sales), 0.06,1,1);
select * from apriori((select * from sales), 0.05,1,1);
select * from apriori((select * from sales), 0.04,1,1);
select * from apriori((select * from sales), 0.03,1,1);
select * from apriori((select * from sales), 0.02,1,1);
select * from apriori((select * from sales), 0.01,1,1);
```

apriori_millisupp.test

```
\set repeat 3
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 0.009,1,1);
select * from apriori((select * from sales), 0.008,1,1);
select * from apriori((select * from sales), 0.007,1,1);
select * from apriori((select * from sales), 0.006,1,1);
select * from apriori((select * from sales), 0.005,1,1);
select * from apriori((select * from sales), 0.004,1,1);
select * from apriori((select * from sales), 0.003,1,1);
select * from apriori((select * from sales), 0.002,1,1);
select * from apriori((select * from sales), 0.001,1,1);
```

apriori_mikrosupp.test

```
\set repeat 3
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 0.0010,1,1);
select * from apriori((select * from sales), 0.0009,1,1);
select * from apriori((select * from sales), 0.0008,1,1);
select * from apriori((select * from sales), 0.0007,1,1);
select * from apriori((select * from sales), 0.0006,1,1);
select * from apriori((select * from sales), 0.0005,1,1);
select * from apriori((select * from sales), 0.0004,1,1);
```

```
select * from apriori((select * from sales), 0.0003,1,1);
select * from apriori((select * from sales), 0.0002,1,1);
```

C.4. Vergleich der Implementierungen

apriori_implementations.test

```
\set repeat 10
\o /dev/null
\set limitworkers 1
select * from apriori((select * from sales), 0.5,1,0);
select * from apriori((select * from sales), 0.4,1,0);
select * from apriori((select * from sales), 0.3,1,0);
select * from apriori((select * from sales), 0.2,1,0);
select * from apriori((select * from sales), 0.1,1,0);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
select * from apriori((select * from sales), 0.5,1,2);
select * from apriori((select * from sales), 0.4,1,2);
select * from apriori((select * from sales), 0.3,1,2);
select * from apriori((select * from sales), 0.2,1,2);
select * from apriori((select * from sales), 0.1,1,2);
select * from apriori((select * from sales), 0.5,1,3);
select * from apriori((select * from sales), 0.4,1,3);
select * from apriori((select * from sales), 0.3,1,3);
select * from apriori((select * from sales), 0.2,1,3);
select * from apriori((select * from sales), 0.1,1,3);
```

apriori_implementations_mt.test

```
\set repeat 10
\o /dev/null
\set limitworkers 15
select * from apriori((select * from sales), 0.5,1,0);
select * from apriori((select * from sales), 0.4,1,0);
select * from apriori((select * from sales), 0.3,1,0);
select * from apriori((select * from sales), 0.2,1,0);
select * from apriori((select * from sales), 0.1,1,0);
select * from apriori((select * from sales), 0.5,1,1);
select * from apriori((select * from sales), 0.4,1,1);
select * from apriori((select * from sales), 0.3,1,1);
select * from apriori((select * from sales), 0.2,1,1);
select * from apriori((select * from sales), 0.1,1,1);
select * from apriori((select * from sales), 0.5,1,2);
select * from apriori((select * from sales), 0.4,1,2);
select * from apriori((select * from sales), 0.3,1,2);
```

```
select * from apriori((select * from sales), 0.2,1,2);
select * from apriori((select * from sales), 0.1,1,2);
select * from apriori((select * from sales), 0.5,1,3);
select * from apriori((select * from sales), 0.4,1,3);
select * from apriori((select * from sales), 0.3,1,3);
select * from apriori((select * from sales), 0.2,1,3);
select * from apriori((select * from sales), 0.1,1,3);
```

Literaturverzeichnis

[1] Agrawal, Rakesh, Tomasz Imielinski und Arun Swami: *Mining Association Rules between Sets of Items in Large Databases.* In: *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)*, 1993.

[2] Agrawal, Rakesh und Ramakrishnan Srikant: *Fast Algorithms for Mining Association Rules.* In: *Proc. of Intl. Conf. on Very Large Databases (VLDB'94)*, 1994.

[3] Berger, Charlie: *Big Data Analytics with Oracle Advanced Analytics*, 2015.

[4] Bodon, Ferenc: *A fast APRIORI implementation.* In: *Proc. of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003.

[5] Borgelt, Christian: *Efficient Implementations of Apriori and Eclat.* In: *Proc. of the 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI'03)*, 2003.

[6] Borgelt, Christian: *Recursion Pruning for the Apriori Algorithm.* In: *Proc. of the 2nd IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI'04)*, 2004.

[7] Borgelt, Christian: *An Implementation of the FP-growth Algorithm.* In: *Proc. Workshop Open Software for Data Mining (OSDM'05 at KDD'05)*, 2005.

[8] Borgelt, Christian und Rudolf Kruse: *Induction of Association Rules: Apriori Implementation.* In: *Proc. in Computational Statistics (Compstat'02)*, 2002.

[9] Brijs, Tom, Gilbert Swinnen, Koen Vanhoof und Geert Wets: *Using Association Rules for Product Assortment Decisions: A Case Study.* In: *Knowledge Discovery and Data Mining (KDDM'99)*, Seiten 254–260, 1999.

[10] Brin, Sergey, Rejeev Motwani, Jeffrey D. Ullman und Shalom Tsur: *Dynamic Itemset Counting and Implication Rules for Market Basket Data.* In: *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, 1997.

[11] Fang, Min, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani und Jeffrey D. Ullman: *Computing iceberg queries efficiently.* In: *Intl. Conf. on Very Large Databases (VLDB'98)*, 1998.

[12] Geurts, Karolien, Geert Wets, Tom Brijs und Koen Vanhoof: *Profiling High Frequency Accident Locations Using Association Rules.* In: *Proc. of the 82nd Annual Transportation Research Board*, Seite 18 ff., 2003.

[13] Hahsler, Michael, Bettina Grün und Kurt Hornik: *arules – A Computational Environment for Mining Association Rules and Frequent Item Sets.* Journal of Statistical Software, 14(15):1–25, October 2005, ISSN 1548-7660. `http://dx.doi.org/10.18637/jss.v014.i15`.

[14] Hahsler, Michael und Kurt Hornik: *Building on the arules Infrastructure for Analyzing Transaction Data with R.* In: Decker, R. und H. J. Lenz (Herausgeber): *Advances in Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, Seiten 449–456. Springer-Verlag, 2007.

[15] Hahsler, Michael, Kurt Hornik und Thomas Reutterer: *Implications of Probabilistic Data Modeling for Mining Association Rules.* In: Spiliopoulou, M., R. Kruse, C. Borgelt, A. Nürnberger und W. Gaul (Herausgeber): *From Data and Information Analysis to Knowledge Engineering*, Studies in Classification, Data Analysis, and Knowledge Organization, Seiten 598–605. Springer-Verlag, 2006. `http://www.springerlink.com/content/978-3-540-31314-4/`.

[16] Han, Jiawei und Jian Pei: *Mining Frequent Patterns by Pattern-Growth: Methodology and Implications.* In: *ACM SIGKDD Explorations (Special Issue on Scaleble Data Mining Algorithms)*, Dezember 2000.

[17] Han, Jiawei, Jian Pei und Yiwen Yin: *Mining Frequent Patterns without Candidate Generation.* In: *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, Mai 2000.

[18] Hegland, Markus: *The Apriori Algorithm – a Tutorial.* WSPC/Lecture Notes Series, 2005.

[19] Hipp, Jochen, Ulrich Güntzer und Gholamreza Nakhaeizadeh: *Algorithms for Association Rule Mining &Mdash; a General Survey and Comparison.* SIGKDD Explor. Newsl., 2(1):58–64, Juni 2000, ISSN 1931-0145. `http://doi.acm.org/10.1145/360402.360421`.

[20] Intelligent Information Systems, IBM Almaden Research Center: *Synthetic Data Generation Code for Associations and Sequential Patterns.* `http://www.almaden.ibm.com/`.

[21] Kemper, Alfons und André Eickler: *Datenbanksysteme - Eine Einführung.* Oldenburg Verlag, 2011.

[22] Kemper, Alfons und Thomas Neumann: *HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots.* In: *International Council for Open and Distance Education (ICDE'11)*, 2011.

[23] Kemper, Alfons, Thomas Neumann, Florian Funke, Viktor Leis und Henrik Mühe: *HyPer: Adapting Main-Memory Data Management for Transactional AND Query Processing.* In: *Institute of Electrical and Electronics Engineers (IEEE'12)*, 2012.

[24] Kudraß, Thomas: *Taschenbuch Datenbanken.* Fachbuchverlag Leipzig im Carl Hanser Verlag, 2007.

[25] Lumpkin, George: *Frequent Itemsets in Oracle10g*, 2003.

[26] McAuley, Julian, Rahul Pandey und Jure Leskovec: *Inferring networks of substitutable and complementary products.* Knowledge Discovery and Data Mining (KDDM'15), 2015.

[27] McAuley, Julian, Christopher Targett, Javen Shi und Anton van den Hengel: *Image-based recommendations on styles and substitutes.* Annual ACM Special Interest Group on Information Retrieval (SIGIR'15), 2015.

[28] Neumann, Thomas: *Efficiently Compiling Query Plans for Modern Hardware.* In: *Intl. Conf. on Very Large Databases (VLDB'11)*, 2011.

[29] Nordman, Aida: *Association rule mining: Apriori algorithm.* Lecture Data Mining, 2011.

[30] Park, Jong Soo, Ming syan Chen und Philip S. Yu: *An effective hash-based algorithm for mining association rules.* In: *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'97)*, 1995.

[31] Passing, Linnea, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper und Thomas Neumann: *SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases.* In: *EDBT/ICDT, EDBT-36*, 2017.

[32] R Development Core Team: *R: A Language and Environment for Statistical Computing*, 2005. `http://www.R-project.org`.

[33] Savasere, Ashok, Edward Omiecinski und Shamkant Navathe: *An efficient algorithm for mining association rules in large databases.* In: *Intl. Conf. on Very Large Databases (VLDB'95)*, 1995.

[34] Standardization, International Organization for und International Electrotechnical Commission: *ISO/IEC 9075-1:2011*, 2011.

[35] Then, Manuel, Linnea Passing, Nina Hubig, Stephan Günnemann, Alfons Kemper und Thomas Neumann: *Effiziente Integration von Data- und Graph-Mining-Algorithmen in relationale Datenbanksysteme.* In: *Proc. of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB*, 2015.

[36] Toivonen, Hannu: *Sampling large databases for association rules.* In: *Intl. Conf. on Very Large Databases (VLDB'96)*, 1996.

[37] User Documentation for MADlib: *Apriori Algorithm*, 2016. `http://madlib.incubator.apache.org/docs/latest/group__grp__assoc__rules.html`.

[38] Woo, Jongwook: *Apriori-Map/Reduce Algorithm.* In: *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, Juli 2012.

[39] Zaki, M. J., S. Parthasarathy, M. Ogihara und W. Li: *New Algorithms for Fast Discovery of Association Rules.* In: *Proc. 3rd Int. Conference on Knowledge Discovery and Data Mining (KDD'97)*, 1997.

[40] Zaki, Mohammed Javeed, Srinivasan Parthasarathy, Wei Li und Mitsunori Ogihara: *Evaluation of Sampling for Data Mining of Association Rules.* In: *Proc. of the 7th International Workshop on Research Issues in Data Engineering (RIDE'97) High Performance Database Management for Large-Scale Applications*, 1997.

[41] Zhao, Shulei und Rongxin Du: *Distributed Apriori in Hadoop MapReduce Framework.* 2012.