Universität Passau
Fakultät für Informatik und Mathematik

Arbeit zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

Prefetching in Datenbanken

Stefan Ganser

16. April 2014

Masterarbeit
am Lehrstuhl für Informatik mit Schwerpunkt Informationsmanagement
(Prof. Dr. Burkhard Freitag)
der Fakultät für Informatik und Mathematik
der Universität Passau

Erstgutachter:	Prof. Dr. Ralf Schenkel
Zweitgutachter:	Prof. Dr. Harald Kosch
Betreuer:	Dipl.-Inf. Univ. Christoph Ehlers

Abstract

This master's thesis describes the prefetcher WMoCache, which is a possible adoption of the WM_o algorithm [1] for client side prefetching of parameterized SQL queries. It uses ideas from the Scalpel prefetcher [2] and C-Miner [3]. The evaluation with two OLTP benchmarks shows promising results.

Inhaltsverzeichnis

Einleitung

1 Einleitung

Bei einfachen Anfragen an einen Server hat die Interprozesskommunikation einen bedeutenden Anteil an der durch einen Benutzer wahrgenommenen Latenz [4]. Dieser Trend nimmt durch schnellere Prozessoren, größere Speicher und die durch verbesserte Auswertungsalgorithmen wachsende Effizienz bei der Verarbeitung von Anfragen zu [4]. Abbildung 1.1 illustriert die oft ungleichen Anteile von Kommunikationsaufwand und der Verarbeitung einer Anfrage durch einen Server an der Gesamtzeit, die ein Client damit verbringt, auf das Ergebnis einer Anfrage zu warten [1].

Aber auch durch die Auslastung der Server und des Netzwerks steigt die wahrgenommene Verzögerung [1, 5, 6]. Hier kann zwar durch die Steigerung von Bandbreiten und Rechenleistung zeitweise entgegengewirkt werden; Das Problem wird durch weiteres Wachstum der Anforderungen jedoch erneut auftreten. Zudem kann die Ausbreitungsgeschwindigkeit von Daten nicht unter einen bestimmten Wert reduziert werden, da sie (letztlich) von der physikalischen Distanz der Kommunikationspartner abhängt [1, 6]. Der Trend zu WLANs und WANs mit hohen Latenzen und niedriger Bandbreite verstärkt das Problem weiter [7].

Caching [8] kann teilweise Abhilfe schaffen, es ist jedoch ungeeignet, wenn Anfragen selten wiederkehren [7], d.h. wenn der Anfragestrom keine temporale Lokalität enthält [1, 5, 9]. Des Weiteren ist Caching ungeeignet, wenn Datenobjekte sich häufig ändern [1, 5, 10]. Der mit langen Wiederverwendungszyklen einhergehende Bedarf für eine große Cachegröße wird verschlimmert, wenn die durchschnittliche Größe von Anfrageergebnissen ebenfalls hoch ist [9]. In diesen Fällen ist eine frühzeitige Vorhersage von Anfragen und vorzeitiges Abrufen, genannt Prefetching [7, 8, 10, 11, 12],

[1] Das Diagramm ist eine Reproduktion von Abbildung 1.1 in [4]

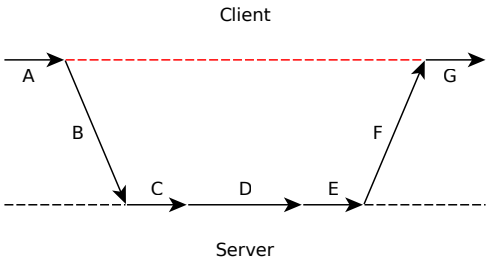


Abbildung 1.1: Schritte der Beantwortung einer Anfrage Q an einen Server: A: Vorbereitung von Q durch den Client – B: Übertragung der Anfrage zum Server – C: Interpretation der Anfrage durch den Server – D: Beantwortung der Anfrage durch den Server – E: Vorbereiten der Antwort durch den Server – F: Übertragung der Antwort zum Client – G: Interpretation des Ergebnisses durch den Client

vorteilhaft [7, 10]. Einen Sonderfall stellen semantische Caches wie IQCache [2] dar. Diese führen ein Rewriting einer Anfrage durch, falls eine Teilmenge von deren Ergebnis bereits lokal verfügbar ist. Sowohl Caching als auch Prefetching können Datenzugriffe signifikant beschleunigen [7, 8]. Prefetching kann entweder asynchron erfolgen und Latenzen somit verbergen [1, 5, 6, 10, 12, 13], oder durch kombiniertes Ausführen von Anfragen [4, 7]. In ersterem Fall kann Prefetching eine konstantere Nutzung der Bandbreite unterstützen [1, 5]. Durch Letzteres kann Latenz bei Kombination mehrerer einfacher Anfragen in einer einzigen reduziert werden [4, 7]. Die Zahl der RTTs (Round Trip Time) wird reduziert [7, 8] und es wird eine deutlich messbare Beschleunigung erreicht [8, 14]. Durch die Reduktion der Anzahl separater Kommunikationsvorgänge kann der Energieverbrauch mobiler Geräte gesenkt werden [11].

Diese Arbeit soll einen Beitrag zum clientseitigen Prefetching von SQL Anfragen an einen Datenbankserver leisten. Dabei soll untersucht werden, ob der für das Prefetching von HTTP-Anfragen an einen Webserver durch Nanopoulos et al. entwickelte WM_o-Algorithmus [1] für die Verwendung im Datenbankbereich angepasst werden kann. Der Erfolg kann dabei in Metriken wie der Präzision des Prefetchers, der Beschleunigung einer Datenbankanwendung durch den Prefetcher oder der Zeit, die eine Anwendung durchschnittlich damit verbringt auf das Ergebnis einer Anfrage zu warten, gemessen werden. Die Vorteile, welche der WM_o-Algorithmus gegenüber anderen Prefetchingverfahren besitzt, sollen für Datenbankanwendungen zugänglich gemacht werden.

1.1 Prefetching

Prefetching wird in vielen Bereichen eingesetzt. Unter anderem ist der Einsatz in folgenden Gebieten bekannt: Datenbanken [4, 7, 13, 14, 15], World Wide Web [1, 5, 6, 10], Dateisysteme [3, 12, 16], CPU-Caches [17, 18], semantisches Prefetching in standortbezogenen Diensten (kontextsensitives Prefetching) [11] und Netzwerkdateisysteme [9]. Die Quelle [7] beschäftigt sich intensiv mit Strategien zum kombinierten Ausführen von SQL Anfragen, mit dem Ziel, die Zahl einzelner Kommunikationsvorgänge mit einem Datenbankserver zu reduzieren, sowie Datenfluss und Latenz zu optimieren.

Prefetchingverfahren können nach verschiedenen Kriterien gruppiert werden:

A. Bilgin gibt die folgenden Kategorien an [7]:

- Deterministische Vorhersagesysteme, die eine fixe Strategie verwenden, um Prefetchingentscheidungen zu treffen (vgl. z.B. [13]).
- Strategien, deren Vorhersagen auf Objektstrukturen basieren. Diese finden sich im Bereich objektorientierter Datenbanken (siehe z.B. [14]).
- Vorhersagen mittels statistischer Techniken basieren auf der Analyse vergangener Zugriffe. Es existiert eine Vielzahl an Arbeiten, die dieses Prinzip verwenden. Zu diesen gehören [3, 4, 9, 10, 15] und [18].

[2] `http://www.fim.uni-passau.de/fim/fakultaet/lehrstuehle/informationsmanagement/forschungsprojekte/iqcache.html` – zuletzt überprüft am 20.03.2014

Gerlhof und Kemper [13] unterscheiden zwischen strategiebasierten Systemen, die eine fest implementierte Strategie verwenden und damit den deterministischen Systemen von A. Bilgin entsprechen, (z.B. [14]) und trainingsbasierten Systemen, die Anfragemuster zur Laufzeit erlernen (z.B. [3, 4, 10, 15, 18]).

Die Quelle [3] unterteilt Prefetchingverfahren danach, in welchem Maß einer Anwendung das Vorhandensein des Prefetchers bewusst sein muss. Es werden drei Gruppen gebildet:

- **Black Box Ansatz**: Das Front End (Speichersystem-Front-End, Browser, Datenbankanwendung, etc.) muss nicht über das Vorhandensein des Prefetchers Bescheid wissen. Der Prefetcher trifft alle Entscheidungen selbstständig und ist für das Front End transparent. Beispiele sind [3, 4, 9, 10] und [15].
- **Gray Box Ansatz**: Der Prefetcher erlernt das Verhalten des Front-Ends durch Aufruf einiger Standardoperationen und Observieren der daraufhin gestellten Anfragen. Die Autoren verweisen auf [19].
- **White Box Ansatz**: Das Front End gibt semantische Informationen an den Prefetcher weiter, die Anfragemuster beschreiben. Alternativ können bei L1-Cache Prefetching auch durch einen Compiler Prefetchinghinweise in den Code eingefügt werden [18].

Ähnlich zum White Box Ansatz ist *informed prefetching* [1]: Eine Anwendung informiert den Prefetcher zuverlässig über ihre zukünftigen Anfragen und der Prefetcher ist dafür verantwortlich, die Daten rechtzeitig zu laden. Die Autoren verweisen auf [16]. Auch die Quelle [17] verwendet informiertes Prefetching.

[1] nennt als weitere Gruppe Systeme, die Vorhersagen auf Basis der Historie vergangener Anfragen machen. Diese nutzen räumliche Lokalität aus [1, 5, 10] und sind damit identisch mit der Gruppe der durch A. Bilgin erwähnten statistischen Techniken, sowie mit den trainingsbasierten Systemen bei Gerlhof und Kemper.

Systeme, die Prefetching auf Basis von Vorhersagen machen, werden durch [1] in solche unterteilt, die einen Abhängigkeitsgraph (*dependency graph* (DG)) verwenden und solche, die Schemata, die aus der Textkompression übernommen wurden, nutzen. Letztere werden als PPM (Prediction by Partial Match) Ansätze bezeichnet. Der WM_o-Algorithmus stellt eine Generalisierung beider Ansätze dar [1]. Das PPM Modell wird auch in [10] verwendet und beschrieben.

1.1.1 Herausforderungen bei Prefetching

Prefetching verursacht einen zusätzlichen Overhead. Bei starker Beanspruchung eines Servers oder Speichersystems können dadurch reguläre Anfragen so verlangsamt werden, dass der Vorteil, der durch die dank Prefetching verbesserte Cachehitrate erzielt wird, aufgebraucht wird [1, 3, 5, 8]. [3] passt daher die Aggressivität des Prefetchings an die Last eines Servers oder Speichersystems an. Inbesondere Prefetching falscher Vorhersagen verschwendet Ressourcen [1, 5, 7, 13]. Falsches Prefetching kann des Weiteren einen negativen Einfluss auf die Leistung des Caches, in welchen die vorzeitig abgerufenen Ergebnisse eingefügt werden, haben, falls nützliche Cacheinhalte durch fälschlicherweise abgerufene Prefetchingergebnisse ersetzt

werden [5]. Die Quelle bietet hierfür eine Lösung, die auch in Abschnitt 2.4 dieser Arbeit dargestellt wird.

Die Veränderung von Datenbeständen kann dazu führen, dass Cacheinhalte ungültig werden. Bei clientseitigem Prefetching muss zwischen Updates durch Transaktionen, die über die eigene Datenbankverbindung ausgeführt werden, und Updates durch andere Transaktionen unterschieden werden. In ersterem Fall gibt es praktikable Lösungsansätze. Im zweiten Fall sind Lösungen mit größerem Aufwand verbunden. Sofern Prefetchingergebnisse nicht über Transaktionsgrenzen hinweg verwendet werden, stellen Änderungen durch andere Transaktionen aufgrund der ACID-Eigenschaften von Datenbanktransaktionen jedoch kein Problem dar (alle [4]).

Schlussendlich sind sich über die Zeit verändernde Anfragemuster eine Herausforderung. Diesen kann im Falle statistischer Vorhersagemethoden durch wiederholtes [15] oder fortlaufendes [9, 10] Training begegnet werden.

1.2 Motivation

Wie erwähnt existieren bereits diverse Ansätze zum Prefetching von Datenbankanfragen ([4, 7, 13, 14, 15]). Insbesondere [4] konzentriert sich dabei ebenfalls auf das clientseitige Prefetching beliebiger SQL-Anfragen durch eine statistische Black-Box-Strategie. Der WM_o-Algorithmus für das Prefetching nicht parametrisierter (HTTP-)Anfragen unterscheidet sich von anderen statistischen Vorhersagetechniken durch seine Fähigkeit auf zufällig auftretende Anfragen reagieren zu können und mit einem abrupten Wechsel von einem Anfragemuster auf ein anderes umgehen zu können. Auch auf Basis einer Teilsequenz eines Anfragemusters können Vorhersagen gemacht werden. Pius Hübl hat erkannt, dass der WM_o-Algorithmus im Bereich des Prefetchings für Key-Value-Stores Vorteile gegenüber anderen Verfahren bietet [20]. Die vorliegende Arbeit möchte daher untersuchen, ob WM_o auch im Bereich relationaler Datenbanken sinnvoll eingesetzt werden kann.

1.3 Überblick

Die Arbeit ist in vier Teile untergliedert. Zunächst werden der WM_o-Algorithmus sowie weitere für die Arbeit verwendete Grundlagen beschrieben. Anschließend wird in Kapitel 3 der Arbeit eine Erweiterung von WM_o für parametrisierte SQL-Anfragen präsentiert. Das vorletzte Kapitel gibt einen Überblick über WMoCache, eine Middleware, die das neue Verfahren implementiert. Schlussendlich wird für die Evaluation von WMoCache eine Simulationsumgebung geschaffen, es werden geeignete Benchmarks ausgewählt und die Ergebnisse der Evaluation werden diskutiert.

Grundlagen

2 Grundlagen

Bevor in Kapitel 3 der erweiterte WM_o-Algorithmus für SQL-Anfragen präsentiert wird, sollen an dieser Stelle für die Arbeit verwendete Grundlagen beschrieben werden. So führt Abschnitt 2.1 anhand von [1, 20, 21] den WM_o-Algorithmus ein. Im darauf folgenden Abschnitt 2.2 wird das von Scalpel [2, 4, 22, 23] übernommene Konzept der Parameterkorrelationen erläutert. Ebenso wird die in Scalpel genutzte OUTER-UNION-Strategie zum kombinierten Ausführen von SQL Anfragen wiederverwendet. Siehe hierzu Abschnitt 2.3. Die Quelle [5] führt unter anderem ein Verfahren für Cachemanagement in Verbindung mit einem Prefetcher ein. Dessen Beschreibung erfolgt in Abschnitt 2.4. Schlussendlich wird das Projekt IQCache Query präsentiert, welches eine AST-Implementierung für SQL, sowie den zugehörigen Parser und einen Datenbankmetadaten-Cache zur Verfügung stellt (siehe Abschnitt 2.5). IQCache Query kommt bei der in Kapitel 4 der Arbeit vorgestellten Implementierung von WMoCache zum Einsatz.

2.1 WM_o-Algorithmus

Dieser Abschnitt beschreibt anhand der Originalquelle [1] und unter Rückgriff auf die dort referenzierte Quelle [21] den WM_o-Algorithmus für Web-Prefetching. Auch die Quelle [20] bietet eine vollständige Beschreibung des Verfahrens. Des Weiteren wird dort eine umfassende Evaluation des WM_o-Algorithmus, sowie ein Vergleich mit anderen Prefetchingverfahren durchgeführt. WM_o wird dort als gut geeigneter Prefetchingalgorithmus für Key-Value-Stores befunden.

Ein Web-Prefetcher sagt Anfragen eines Nutzers, der durch die Seiten einer Web-Site navigiert, vorher, um diese vorzeitig durchführen zu können. Abbildung 2.1 [3] zeigt, wie ein solcher Prefetcher gemäß [1] in die bestehende Webinfrastruktur eingebettet werden kann: Der Webserver ist um ein Vorhersagemodul erweitert, das für jede Benutzersitzung, auf der Basis der bisher darin aufgetretenen Anfragen, Vorhersagen über zukünftige Anfragen macht. Diese Vorhersagen können an reguläre Antworten des Servers an den Client angehängt werden (sog. Piggybacking). Ein dem Browser (Client) vorgeschalteter oder im Browser integrierter Prefetcher wertet die Vorhersagen aus und kann anschließend das Prefetching der vorhergesagten Anfragen durchführen.

Der WM_o-Algorithmus macht Prefetchingvorschläge aufgrund vorheriger Seitenzugriffe. Der Fokus liegt dabei rein auf Zugriffen auf statische Webseiten mit gleichbleibenden URLs, da der Algorithmus nicht in der Lage ist HTTP-Request-Parameter vorherzusagen. Er basiert auf der Annahme, dass Besucher einer Web-Site während

[3] Die Abbildung ist angelehnt an [1], Abbildung 1

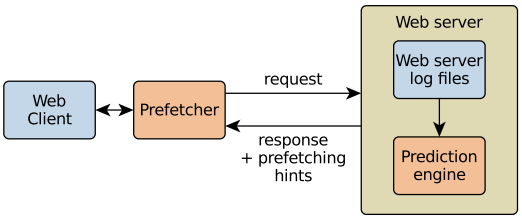


Abbildung 2.1: Einbettung eines Web-Prefetchers in die bestehende Webinfrastruktur

einer Sitzung bestimmten Navigationsmustern folgen. Jedoch kann es sein, dass bisweilen Anfragen auftreten, die das bisher befolgte Muster durchbrechen. Diese können entweder zu anderen Mustern oder zu gar keinem Muster gehören. Zudem kann ein Nutzer spontan zu einem anderen Navigationsmuster wechseln. Um dies zu berücksichtigen, kann das WM_o-Verfahren die Anzahl der vorherigen Anfragen, auf Basis derer ein zukünftiger Seitenzugriff vorhergesagt wird, bei Bedarf reduzieren. Die Navigationsmuster erlernt der Algorithmus aus einer Menge von Trainingssequenzen, die typische Benutzersitzungen widerspiegeln sollen und aus dem Webgraphen der jeweiligen Web-Site. Letzteren benötigt er, um die erlernten Navigationsmuster auf Pfade im Webgraphen zu beschränken. Der Algorithmus gliedert sich in zwei Phasen. Initial wird einmalig anhand der Trainingssequenzen und des Webgraphen eine Prefetchingstruktur generiert, mit Hilfe derer dann in der zweiten Phase das Prefetching durchgeführt werden kann.

Es folgen einige Definitionen. Anschließend wird der Algorithmus selbst beschrieben und anhand eines Beispiels illustriert.

2.1.1 Definitionen

Die Kenntnis der folgenden Definitionen ist für das Verständnis des WM_o-Algorithmus erforderlich.

Definition 2.1 (Gerichteter Graph). Ein gerichteter Graph $G = (V, E)$ ist definiert durch eine Menge von Knoten V und eine Menge von Kanten $E \subseteq V \times V$. Jede Kante $e = (u, v) \in E$ besitzt eine Richtung: Sie geht von ihrem Startknoten u zu ihrem Zielknoten v. □

Definition 2.2 (Pfad). Ein Weg der Länge n eines gerichteten Graphen $G = (V, E)$ ist eine Sequenz von Knoten $\langle v_1, ..., v_n \rangle, n \in \mathbb{N}, \forall i = 1, ..., n : v_i \in V$, sodass $\forall i = 1, ..., n-1 : (v_i, v_{i+1}) \in E$. Ein Weg, der jeden Knoten und jede Kanten seines Graphen höchstens einmal beinhaltet, wird Pfad genannt. □

Definition 2.3 (Subsequenz). Seien $P = \langle p_1, ..., p_n \rangle$ und $S = \langle s_1, ..., s_m \rangle$ Sequenzen von Knoten. Falls m natürliche Zahlen $i_1 < ... < i_m$ existieren, sodass $\forall j = 1, ..., m : s_j = p_{i_j}$, handelt es sich bei S um eine Subsequenz von P. Dies wird mit der Notation $S \preceq P$ zum Ausdruck gebracht. □

Definition 2.4 (Subpfad). Seien S und P Sequenzen von Knoten und es gelte $S \preceq P$. Falls S und P Pfade des selben Graphen G sind, ist S ein Subpfad von P. □

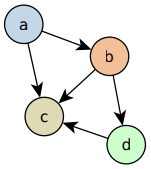


Abbildung 2.2: Ein gerichteter Graph

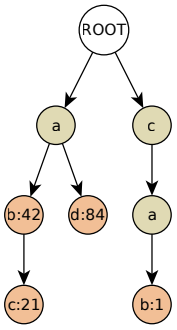


Abbildung 2.3: Beispiel für einen Trie zur Speicherung von Schlüssel-Wert-Paaren

Beispiel 2.1. Gegeben sei der in Abbildung 2.2 gezeigte gerichtete Graph G. Die Sequenz $P = \langle a, b, d, c \rangle$ ist ein Pfad des Graphen. $\langle b, c, a \rangle$ hingegen ist kein Pfad, da die Kante (c, a) in G nicht existiert. Für $S = \langle a, d, c \rangle$ und $T = \langle a, b, c \rangle$ gilt $S \preceq P$ sowie $T \preceq P$. Da S im Gegensatz zu T kein Pfad von G ist, handelt es sich nur bei T um einen Subpfad von P. □

Definition 2.5 (Support). Gegeben sei eine Menge D von Pfaden und eine Sequenz s. Der Support von s ist definiert als $support(s, D) = |\{p \mid p \in D, s \preceq p\}|$. Der Support von s entspricht also der Anzahl an Pfaden in D, die s als Subsequenz beinhalten. □

Definition 2.6 (Trie/Präfixbaum). Ein Trie, auch Präfixbaum genannt, ist eine baumartige Datenstruktur zur platzsparenden Speicherung von Listen und insbesondere von Zeichenketten. Dabei wird für das gemeinsame Präfix zweier Listen jeweils derselbe Pfad ab der Wurzel des Baumes verwendet. Ab dem Knoten, ab welchem die beiden Listen divergieren, wird für den Suffix jeder Liste ein eigener Pfad erstellt. Zur Speicherung des Präfixes p einer bereits im Trie gespeicherten Liste l, muss der Endknoten von p als Endknoten einer gespeicherten Liste markiert werden. Ein Trie kann zur Speicherung von Schlüssel-Wert-Paaren verwendet werden, indem die Werte in den Endknoten der Schlüsselpfade gespeichert werden. Beispiel 2.2 illustriert dies. □

Beispiel 2.2. Gegeben seien die Schlüssel-Wert-Paare (ab, 42), (abc, 21) (ad, 84), (cab, 1). Abbildung 2.3 zeigt deren Speicherung in einem Trie. □

2.1.2 Generierung der Prefetchingstruktur

Auf Basis der soeben gemachten Definitionen kann nun die Generierung der Prefetchingstruktur anhand eines Webgraphen G und einer Menge von Trainingssequenzen D beschrieben werden. Ziel ist es alle Pfade s in G zu finden, die die folgenden Eigenschaften erfüllen:

a) $support(s, D) \geq minSupport$ für einen gewählten Threshold $minSupport \in \mathbb{N}$

b) Jede Subsequenz von s der Länge ≥ 1 ist ein Pfad in G und genügt den hier geforderten Eigenschaften.

Die so gefundenen häufigen Subsequenzen der Trainingssequenzen sind die Navigationsmuster, mit Hilfe derer später Vorhersagen gemacht werden. Die geforderten Eigenschaften schränken die Menge der akzeptierten Subsequenzen derart ein, dass der Prefetchingalgorithmus nicht nur Vorhersagen auf der Basis von Navigationsmustern maximaler Länge machen kann, sondern auch auf der Basis des Auftretens einer Subsequenz eines Navigationsmusters. Algorithmus 2.1 nimmt einen Graphen $G = (V, E)$ (V sei die Menge der Knoten, $E \subseteq V \times V$ die Menge der Kanten) und eine Menge von Trainingssequenzen D, sowie einen Wert für $minSupport$ als Eingabe und implementiert die Suche nach häufigen Subsequenzen mit den oben genannten Eigenschaften in inkrementeller Art und Weise.

Zunächst werden alle Pfade von G der Länge eins generiert. Bei diesen handelt es sich gerade um die Knoten des Graphen. Die Pfade werden in der Menge C_1 (Kandidatenpfade der Länge eins) gespeichert. Ihr initialer Support sei jeweils implizit 0. Die Variable k, welche die Länge der aktuell betrachteten Kandidatenpfade angibt, wird mit eins initialisiert. Solange die Menge C_k, also die aktuell zu betrachtende Kandidatenmenge, nicht leer ist, wird die in Zeile fünf beginnende Schleife ausgeführt. Für jeden Kandidatenpfad in C_k wird dessen Support ermittelt. Anschließend wird die Menge L_k erstellt, welche alle häufigen Subsequenzen der Trainingssequenzen ($support \geq minSupport$) auf Level k, also mit der Länge k, enthält. Nun wird mit Hilfe der Funktion `genCandidates` (siehe Algorithmus 2.2) die Menge C_{k+1} generiert und der Support jedes darin enthaltenen Pfades mit 0 initialisiert. Abschließend wird k inkrementiert. Sofern das neue C_k nicht leer ist, wird die Schleife erneut durchlaufen. Das Ergebnis der Funktion ist die Vereinigung der Mengen $L_i, i = 1, ..., k-1$.

Algorithmus 2.2 wird durch die in Algorithmus 2.1 verwendete Funktion `genCandidates` implementiert.

Der Algorithmus nimmt als Eingabe die Menge L_k der bereits gefundenen häufigen Pfade der Länge k, sowie den Webgraphen $G = (V, E)$. Zu Beginn wird die Menge C_{k+1} der gefundenen Kandidatenpfade mit der Länge $k+1$ als leere Menge initialisiert. Die Pfade in L_k werden der Reihe nach abgearbeitet. Sei $L = \langle l_1, ..., l_k \rangle$ der aktuell betrachtete Pfad. Um L verlängern zu können, wird zunächst die Menge $N^+(l_k) \subseteq V$ aller Knoten generiert, die von l_k aus über eine direkte Kante erreichbar sind. Sofern ein Knoten v aus $N^+(l_k)$ noch nicht Teil von L ist, handelt es sich bei der Sequenz $C = \langle l_1, ..., l_k, v \rangle$ ebenfalls um einen Pfad in G. Die so gewonnenen verlängerten Pfade können also als Kandidaten für häufige Pfade der Länge $k+1$ verwendet werden. Um der Forderung nachzukommen, dass alle Subsequenzen eines häufigen Pfades ebenfalls häufige Pfade sein müssen, werden nur solche Kandidaten

Algorithmus 2.1: Bestimmung häufiger Pfade in einem Graphen G

input : Webgraph $G = (V, E)$,
eine Menge von Trainingssequenzen D,
Integer *minSupport*
output : Die Menge der häufigen Subsequenzen in D

```
1  begin
2      // Menge aller Pfade der Länge 1
3      C_1 ← V
4      k ← 1
5      while C_k ≠ ∅ do
6          for path p ∈ D do
7              S ← {s | s ∈ C_k, s ⪯ p}
8              foreach s ∈ S do
9                  ++s.support
10             end
11         end
12         // Pruning aller seltenen Kandidaten
           L_k ← {s | s ∈ C_k, s.support ≥ minSupport}
13         C_{k+1} ← genCandidates(L_k, G)
14         foreach c ∈ C_{k+1} do
15             c.support ← 0
16         end
17         ++k
18     end
19     return ⋃_{i=1}^{k-1} L_i
20 end
```

Algorithmus 2.2: Generierung von Kandidatenpfaden der Länge $k + 1$

input : Menge L_k der häufigen Pfade mit Länge k,
Webgraph $G = (V, E)$
output : Kandidatenpfade der Länge $k + 1$

```
1  begin
2      C_{k+1} = ∅
3      foreach L = ⟨l_1, ..., l_k⟩, L ∈ L_k do
4          N^+(l_k) ← {v | (l_k, v) ∈ E}
5          foreach v ∈ N^+(l_k) do
6              if v ∉ L and L' = ⟨l_2, ...., l_k, v⟩ ∈ L_k then
7                  C ← ⟨l_1, ..., l_k, v⟩
8                  if ∀S ⪯ C : ((S ≠ L' ∧ |S| = k) ⇒ S ∈ L_k) then
9                      C_{k+1} ← C_{k+1} ∪ {C}
10                 end
11             end
12         end
13     end
14     return C_{k+1}
15 end
```

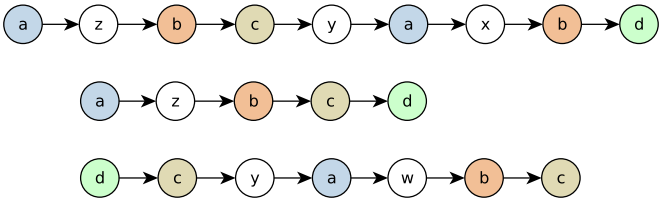


Abbildung 2.4: Trainingssequenzen für den WM_o-Algorithmus

in C_{k+1} eingefügt, die diese Eigenschaft erfüllen. Hierfür genügt es, alle Subsequenzen von C der Länge k auf deren Vorkommen in L_k zu überprüfen, da die Eigenschaft für alle kürzeren Subsequenzen implizit dadurch erfüllt ist, dass sie Subsequenzen der Subpfade der Länge k sind. Die zuletzt geschilderte Überprüfung findet in den Zeilen sechs und acht statt.

Der Leser kann sich überzeugen, dass das Ergebnis von Algorithmus 2.1 alle Pfade von G enthält, die die geforderten Eigenschaften a) und b) besitzen. Die speicher- und zugriffseffiziente Speicherung der gefundenen häufigen Subsequenzen erfolgt in einem Trie. Der jeweilige Endknoten einer häufigen Subsequenz speichert deren Support. Zur Veranschaulichung dient Beispiel 2.3.

Beispiel 2.3. Als Webgraph G diene der in Abbildung 2.2 gezeigte Graph. Zudem sei die Menge D der in Abbildung 2.4 gezeigten Trainingssequenzen gegeben. Es gelte $minSupport = 2$.

Es wird Algorithmus 2.1 mit den Parametern G, D und $minSupport$ aufgerufen. Die Menge der Kandidatenpfade mit Länge eins ist

$$C_1 = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle\}.$$

Da alle vier Kandidatenpfade in allen Trainingssequenzen vorkommen, hat jeder Kandidatenpfad Support drei und ist damit eine häufige Subsequenz der Trainingssequenzen. Somit gilt

$$L_1 = \{\langle a\rangle : 3, \langle b\rangle : 3, \langle c\rangle : 3, \langle d\rangle : 3\}.$$

Hinter dem Doppelpunkt ist jeweils der Support der häufigen Subsequenz angegeben. Es folgt der Aufruf von `genCandidates` mit den Parametern G und L_1. Die Pfade in L_1 können anhand von G folgendermaßen verlängert werden:

- $\langle a\rangle$ zu $\langle a, b\rangle$ und $\langle a, c\rangle$
- $\langle b\rangle$ zu $\langle b, c\rangle$ und $\langle b, d\rangle$
- $\langle c\rangle$ kann nicht verlängert werden
- $\langle d\rangle$ zu $\langle d, c\rangle$

Für alle fünf gefundenen Pfade der Länge zwei gilt, dass ihre Subsequenzen der Länge eins in L_1 enthalten sind. Somit ist

$$C_2 = \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle, \langle b, d\rangle, \langle d, c\rangle\}.$$

Ermittelt man den Support der Kandidatenpfade der Länge zwei, so erkennt man, dass der Support von $\langle d, c\rangle$ nur eins beträgt. Die Kandidatensequenz wird daher durch das Pruning in Algorithmus 2.1 gelöscht und es gilt

$$L_2 = \{\langle a, b\rangle : 3, \langle a, c\rangle : 3, \langle b, c\rangle : 3, \langle b, d\rangle : 2\}.$$

Der nächste Aufruf von `genCandidates` produziert Kandidatenpfade der Länge drei auf Basis von L_2. Die Pfade in L_2 werden folgendermaßen verlängert:

- $\langle a, b\rangle$ zu $\langle a, b, c\rangle$ und $\langle a, b, d\rangle$
- $\langle a, c\rangle$ kann nicht verlängert werden
- $\langle b, c\rangle$ kann nicht verlängert werden
- $\langle b, d\rangle$ zu $\langle b, d, c\rangle$

$\langle a, b, c\rangle$ wird zu C_3 hinzugefügt, da $\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle \in L_2$. $\langle a, b, d\rangle$ wird nicht zu C_3 hinzugefügt, da $\langle a, d\rangle \notin L_2$, d.h. $\langle a, d\rangle$ ist keine häufige Subsequenz. $\langle b, d, c\rangle \notin C_3$, da $\langle d, c\rangle$ bei der Generierung von L_2 durch das Pruning gelöscht wurde. Es resultiert

$$C_3 = \{\langle a, b, c\rangle\}.$$

Die äußere Schleife in Algorithmus 2.1 wird zum dritten Mal durchlaufen. Der Support von $\langle a, b, c\rangle$ ist drei. Damit gilt

$$L_3 = \{\langle a, b, c\rangle : 3\}.$$

Da es sich bei c um eine Senke von G handelt, werden keine Kandidatenpfade der Länge vier generiert. Damit ist $C_4 = \emptyset$ und die äußere Schleife in Algorithmus 2.1 wird nicht mehr durchlaufen. Die Menge aller gefundener häufiger Subsequenzen der Trainingssequenzen ist

$$\bigcup_{i=1}^{3} L_i = \{\langle a\rangle : 3, \langle b\rangle : 3, \langle c\rangle : 3, \langle d\rangle : 3, \langle a, b\rangle : 3, \langle a, c\rangle : 3, \langle b, c\rangle : 3, \langle b, d\rangle : 2, \langle a, b, c\rangle : 3\}.$$

Deren Speicherung erfolgt wie in Abbildung 2.5 dargestellt in einem Trie. □

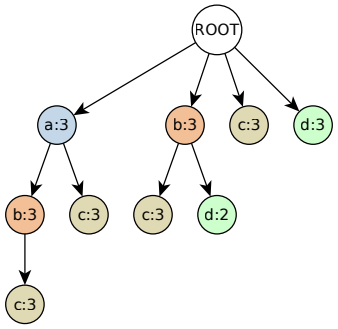


Abbildung 2.5: Trie mit den in Beispiel 2.3 gefundenen häufigen Subsequenzen

2.1.3 Prefetching

In Abschnitt 2.1.2 wurde die Generierung der Prefetchingstruktur in der ersten Phase des WM_o-Algorithmus beschrieben. Dieser Abschnitt erklärt, wie mit Hilfe des generierten Tries Vorhersagen gemacht werden können. Sei T der während der ersten Phase erstellte Trie. Sei d die Tiefe von T. Um mit Hilfe von T Vorhersagen über zukünftige Seitenzugriffe machen zu können, genügt es in einem Fenster der Länge $d-1$ die letzten $d-1$ Seitenaufrufe des Clients zu speichern. Sei $w = \langle r_{n-(d-2)}, r_{n-(d-3)}, ..., r_n \rangle$ dieses Fenster. r_n ist dabei die zuletzt gestellte Anfrage. Man testet nun, ob es einen Pfad ab der Wurzel von T gibt, dessen Präfix w ist. Ist dies nicht der Fall, wird die älteste Anfrage $r_{n-(d-2)}$ aus w entfernt und es wird versucht, auf der Basis der verbleibenden Anfragen eine Vorhersage zu treffen. Dies wird wiederholt, bis w leer ist oder ein entsprechender Pfad gefunden wurde. In ersterem Fall kann keine Vorhersage gemacht werden. Andernfalls sei P die Menge der Kindknoten des zu r_n korrespondierenden Knotens in T. Der Algorithmus wählt nun den Knoten aus P, der den höchsten Supportwert speichert, aus und schlägt die dazugehörige Webseite als Prefetchingkandidat vor.

Beispiel 2.4 illustriert die Vorhersage von Seitenzugriffen mit Hilfe des Tries.

Beispiel 2.4. Gegeben sei der in Abbildung 2.5 gezeigte Trie. Dessen maximale Tiefe ist drei. Folglich hat das Fenster w der letzten Zugriffe die Länge zwei. Sei $w = \langle d, b \rangle$. Offensichtlich ist im Trie kein Navigationsmuster mit Präfix $\langle d, b \rangle$ gespeichert. Folglich wird w auf $\langle b \rangle$ verkürzt. $\langle b \rangle$ hat im Trie die beiden Nachfolger c und d. Da der Support von $\langle b, c \rangle$ mit drei höher ist als jener von $\langle b, d \rangle$ mit zwei, wird der vorzeitige Abruf der Seite c vorgeschlagen. □

2.2 Parameterkorrelationen

Das System Scalpel [4] von Bowman et al. ist ein Prefetcher für SQL-Anfragen, der als Middlewarekomponente zwischen einer Clientanwendung und einem Datenbankserver verwendet werden kann. Analog zum WM_o-Algorithmus werden in einer anfänglichen Lernphase häufig auftretende Anfragesequenzen erlernt, um mit Hilfe dieser später Vorhersagen machen zu können. Da SQL-Anfragen parametrisiert sind und die Parameterwerte zweier Vorkommen einer Anfrage sich unterscheiden können, genügt es nicht, vorhersagen zu können, welche Anfrage als nächstes auftritt.

Vielmehr müssen auch deren Parameterwerte vorhergesagt werden können [23]. Beispielsweise besitzt die in Listing 2.1 gezeigte Anfrage zwei Parameter. Hat man nur die Vorhersage, dass als nächstes das gezeigte Prepared Statement verwendet wird, so ist aufgrund der fehlenden Information über die Parameterwerte kein Prefetching möglich.

```
1 SELECT v.*
2 FROM
3     vorlesungen v,
4     professoren p
5 WHERE
6     p.name = ?
7     AND p.persnr = v.gelesenvon
8     AND v.sws >= ?
```

Listing 2.1: Beispiel für eine parametrisierte SQL-Anfrage

Die Quelle [23] stellt nun fest, dass Parameter einer gegebenen Anfrage oft denselben Wert besitzen wie Parameter vorheriger Anfragen, oder gleich einem Ergebniswert einer vorherigen Anfrage sind. Als Ursache hierfür werden Datenabhängigkeiten im Anwendungscode genannt. Auch [7] beschreibt diese Datenabhängigkeiten und verweist ebenfalls auf Bowman et al.. Da Scalpel den Anwendungscode nicht zur Verfügung hat, können die Abhängigkeiten nicht durch eine Datenflussanalyse identifiziert werden. Stattdessen wird versucht während der Lernphase Korrelationen zwischen Ein- und Ausgabewerten der Anfragen innerhalb häufiger Anfragemuster zu erlernen (alle [23]). Neben Scalpel arbeitet auch der in der Quelle [7] beschriebene Prefetcher für SQL-Anfragen mit Parameterkorrelationen. Diese werden dort als *pattern correlations* bezeichnet.

Es werden drei Arten von Parameterkorrelationen verwendet [4]. Diese werden je nachdem, ob es sich bei dem aktuell betrachteten Anfragemuster um verschachtelte Anfragen, oder um Anfrage-Batches handelt in unterschiedlicher Weise interpretiert. Hier interessieren wir uns für die Interpretation im Kontext von Anfrage-Batches: [4]

- **Wertgleichheit mit einer Konstanten:** Der Parameter hat stets denselben Wert.
- **Eingabekorrelation:** Wertgleichheit mit einem Eingabeparameter einer vorhergehenden Anfrage.
- **Ausgabekorrelation:** Wertgleichheit mit einem Wert aus dem zuletzt durch die Anwendung betrachteten Tupel des Ergebnisses einer vorhergehenden Anfrage.

Während der Lernphase zeichnet Scalpel beim erstmaligen Auftreten einer bestimmten Sequenz von Anfragen jede aufgetretene Parameterkorrelation auf. Bei jedem weiteren Vorkommen derselben Anfragesequenz werden die zuvor zusammen mit der Sequenz gespeicherten Parameterkorrelationen verifiziert. Trifft eine Parameterkorrelation erstmalig nicht zu, so wird sie als insgesamt unzutreffend erachtet und verworfen [23]. Beispiel 2.5 illustriert wie Scalpel Parameterkorrelationen bestimmt.

#	Prepared Statement	Parameter	zuletzt abgerufenes Ergebnistupel
1	q_a	('abc', 42)	(42)
2	q_b	(42)	('xyz', 123)
3	q_x	(1, 'a', 3.1415)	(1, 2, 4, 5)
4	q_a	('asdf', 12)	(3)
5	q_b	(3)	('qwert', 21)

Tabelle 2.1: Eine Folge von Anfragen mit Parameterwerten und Ergebnissen

Beispiel 2.5. [4] Wir nehmen an, dass Scalpel zuletzt die in Tabelle 2.1 gezeigten Anfragen beobachtet habe.

In der Spalte *Parameter* sind die jeweiligen Parameterwerte angegeben. Es wird nun das erste Vorkommen der Anfragesequenz $\langle q_a, q_b \rangle$ in Zeile eins und zwei betrachtet. Für den Parameter von q_b kann zum einen angenommen werden, dass er konstant den Wert 42 besitzt. Darüber hinaus hat er denselben Wert wie der zweite Parameter der vorhergehenden Instanz von q_a. Dies resultiert in einer entsprechenden Eingabekorrelation. Schlussendlich wird eine Korrelation mit dem Ergebniswert von q_a festgestellt und die entsprechende Ausgabekorrelation vermerkt. Das nächste Vorkommen von $\langle q_a, q_b \rangle$ sind die Anfragen vier und fünf. Der Parameter von q_b hat dieses Mal den Wert drei. Die vermutete Korrelation mit dem Wert 42 ist somit nicht zutreffend. Der zweite Parameter von q_a hat hier den Wert zwölf. Somit wird auch die beim ersten Mal beobachtete Eingabekorrelation nicht weiter verwendet. Hingegen trifft die beim ersten Mal beobachtete Ausgabekorrelation mit dem Ergebniswert von q_a auch hier zu. □

Alle Korrelationen, die nach Abschluss der Lernphase für einen Parameter zur Verfügung stehen, sind gleichermaßen verwendbar. Eingabekorrelationen mit Anfragen, die aktuell selbst für Prefetching vorgeschlagen sind, werden nicht beachtet, da immer eine Korrelation mit der ursprünglichen Quelle des Wertes existiert. Scalpel wählt für einen Parameter stets jene Korrelation aus, deren Anwendung die geringsten Kosten verursacht. Bevorzugt werden Korrelationen mit konstanten Werten verwendet, da der konstante Wert immer sofort eingesetzt werden kann. Als nächstes werden Eingabe- und Ausgabekorrelationen mit vorhergehenden Anfragen, deren Parameterwerte und Ergebnisse bereits bekannt sind, ausgewählt. Als letzte Möglichkeit dienen Ausgabekorrelationen mit Anfragen, die aktuell selbst für Prefetching vorgeschlagen sind. In diesem Fall verbleibt eine Ausgabekorrelation, die nicht sofort durch Einsetzen des Parameterwertes aufgelöst werden kann, und damit anders verarbeitet werden muss [4].

In die zuletzt genannte Gruppe von Korrelationen können auch Ausgabekorrelationen mit der zuletzt durch die Datenbankanwendung gestellten Anfrage eingeordnet werden, sofern deren Ergebnis bisher unbekannt ist.

[4] Das Beispiel ist angelehnt an Abschnitt 3.2 von [23]

2.3 Lateral-Outer-Union-Strategie

In Abschnitt 2.2 wurde beschrieben, dass in Scalpel mitunter Ausgabekorrelationen auftreten, die auf das Ergebnis von Anfragen verweisen, die selbst gerade erst für Prefetching ausgewählt wurden und deren Ergebnis somit nicht bekannt ist. Ein weiterer Anwendungsfall ist denkbar, wenn eine Ausgabekorrelation mit der zuletzt durch eine Anwendung gestellten Anfrage besteht und nicht gewartet werden soll, bis deren Ergebnis verfügbar ist. Seien q_a und q_b zwei Anfragen, die demnächst ausgeführt werden sollen. Ein Parameter von q_b besitze eine Parameterkorrelation, die auf einen Wert im Ergebnis von q_a verweist. q_b kann also ohne die Kenntnis des Ergebnisses von q_a nicht ausgeführt werden. Eine Möglichkeit bestünde nun darin, zuerst q_a auszuführen und anschließend q_b. Alternativ können die beiden Anfragen derart zu einer Anfrage kombiniert werden, dass während der Auswertung von q_b auf das Ergebnis von q_a zugegriffen werden kann und aus dem kombinierten Ergebnis die Ergebnisse der einzelnen Anfragen rekonstruiert werden können. Scalpel verwendet hierzu mehrere Strategien, die in [2, 4] beschrieben werden. Eine dieser Strategien ist die Outer-Union-Strategie, die zur Kombination zweier beliebiger SQL-Anfragen geeignet ist. Da sie das in SQL/99 [24] eingeführte Schlüsselwort `LATERAL` verwendet, wird sie hier als Lateral-Outer-Union-Strategie bezeichnet. `LATERAL` ermöglicht es, Anfragen wie die in Listing 2.2 gezeigte Anfrage zu schreiben. Die Anfrage enthält zwei Unteranfragen, wobei die zweite Unteranfrage auf das Ergebnis der ersten zugreift. Die Outer-Union-Strategie wird auch in [7] verwendet.

```
1 SELECT b.*
2 FROM
3               ( SELECT p.persnr
4                 FROM professoren p
5                 WHERE p.name = 'Sokrates' ) AS a,
6 LATERAL (
7               SELECT *
8               FROM vorlesungen v
9               WHERE v.gelesenvon = a.persnr ) AS b
```

Listing 2.2: Beispiel für die Verwendung von `LATERAL`

Abbildung 2.6 illustriert das gemeinsame Anfrageergebnis zweier mit Hilfe der Lateral-Outer-Union-Strategie kombinierter Anfragen q_a und q_b, wie es bei Scalpel im Falle von Anfrage-Batches (siehe [4]) auftreten würde. Jeder Anfrage sind folglich ihre eigenen Spalten zugewiesen. Jedes Tupel gehört zu einer der beiden kombinierten Anfragen, wobei die Spalten, welche zur anderen Anfrage gehören, jeweils mit `NULL`-Werten aufgefüllt werden. Listing 2.3 zeigt den dazugehörigen SQL-Code.

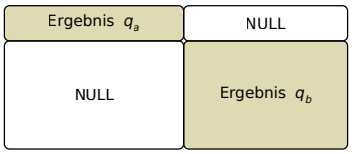


Abbildung 2.6: Schematische Darstellung des Ergebnis einer durch die Lateral-Outer-Union-Strategie erstellten kombinierten Anfrage

Die Funktion `Nulls` produziere `NULL`-Werte in der Anzahl der angegebenen Spalten. Über die Spalte `type` ist jedes Tupel eindeutig einer der beiden Anfragen zugeordnet, da diese für q_a stets den Wert 0 und für q_b stets den Wert eins hat. Dank des Schlüsselwortes `LATERAL` können aus `q_b.sql` heraus Joins mit dem durch den Alias `a` referenzierten Anfrageergebnis von q_a durchgeführt werden. Dieser Ansatz funktioniert nur, wenn das Ergebnis von q_a maximal ein Tupel enthält. Das `ORDER BY` stellt sicher, dass die originale Ordnung der Tupel aus dem Ergebnis von q_b erhalten bleibt.

In Abschnitt 3.7 wird unter anderem gezeigt, wie dieser Ansatz für beliebig große Ergebnisse von q_a erweitert werden kann.

Zur Veranschaulichung dient Beispiel 2.6.

```
SELECT c.*
FROM
        ( <q_a.sql> ) AS a,
LATERAL (
    VALUES (0 AS type, <a.columns>, <Nulls(b.columns)> )
    UNION ALL
    SELECT 1 AS type, <Nulls(a.columns)>, <b.columns>
    FROM ( <q_b.sql> ) AS b
) AS c
ORDER BY type, <q_b.orderby>
```

Listing 2.3: Schema zur Kombination zweier Anfragen q_a und q_b mit der Lateral-Outer-Union-Strategie

Beispiel 2.6.
Gegeben seien die beiden in Listings 2.4 (q_a) und 2.5 (q_b) gezeigten Anfragen.

```
SELECT p.persnr,
    p.vorname, p.name
FROM professoren p
WHERE
    p.name = 'Sokrates'
```

Listing 2.4: Eine SQL-Anfrage mit gesetztem Parameter

```
SELECT v.vorlnr, v.titel
FROM vorlesungen v
WHERE
    v.gelesenvon = ?
ORDER BY v.titel
```

Listing 2.5: Eine SQL-Anfrage mit nicht gesetztem Parameter

Es sei bekannt, dass das Ergebnis von q_a maximal ein Tupel enthalte. Es bestehe eine Ausgabekorrelation zwischen dem Parameter von q_b und dem Wert in der Spalte `persnr` des Ergebnisses von q_a. Listing 2.6 zeigt die Kombination der beiden Anfragen mit Hilfe der Lateral-Outer-Union-Strategie. Tabelle 2.2 zeigt das Ergebnis der kombinierten Anfrage. Das Ergebnis von q_a ist rot und jenes von q_b blau hinterlegt. □

type	c1	c2	c3	c4	c5
0	2125	Philosoph	Sokrates	NULL	NULL
1	NULL	NULL	NULL	5041	Ethik
1	NULL	NULL	NULL	4052	Logik
1	NULL	NULL	NULL	5049	Mäeutik

Tabelle 2.2: Ergebnis der in Listing 2.6 gezeigten Anfrage

```
1  SELECT c.*
2  FROM
3          ( SELECT p.persnr, p.vorname, p.name
4          FROM professoren p
5          WHERE
6              p.name = 'Sokrates' ) AS a,
7  LATERAL (
8          VALUES (0 AS type, a.persnr, a.vorname, a.name,
               NULL, NULL )
9          UNION ALL
10         SELECT 1 AS type, NULL, NULL, NULL, b.vorlnr, b.
               titel
11         FROM ( SELECT v.vorlnr, v.titel
12                 FROM vorlesungen v
13                 WHERE
14                     v.gelesenvon = a.persnr ) AS b
15 ) AS c (type, c1, c2, c3, c4, c5)
16 ORDER BY c.type, c.c5
```

Listing 2.6: Kombination der in Listings 2.4 und 2.5 gezeigten Anfragen mit Hilfe der Lateral-Outer-Union-Strategie

2.4 Cachemanagement in Verbindung mit Prefetching

Die Quelle [5] beschreibt einen um Prefetching erweiterten Web-Cache. Zur Cacheverwaltung wird ein auf dem in [25] beschriebenen W^2R-Algorithmus basierendes Verfahren namens *PECache* verwendet. W^2R steht für „Weighing/Waiting Room“ und spielt auf die logische Unterteilung des Caches in einen Anfragecache, genannt Weighing Room, und einen Prefetchingcache, genannt Waiting Room, an. Ziel der Trennung ist es, den Effekt falscher Vorhersagen des Prefetchers und den Effekt aggressiven Prefetchings zu mildern [5]. Ohne Unterteilung des Caches könnte es andernfalls passieren, dass durch den Prefetcher eingefügte Ergebnisse die Ergebnisse vergangener Anfragen verdrängen, bevor die entsprechenden Anfragen erneut auftreten. Hierdurch kann die bereits in Abschnitt 1.1.1 erwähnte Gefahr entstehen, dass durch den Einsatz des Prefetchers lediglich ein Mehraufwand ohne jeglichen Vorteil

erreicht wird. Der Weighing Room speichert Ergebnisse vergangener durch den Benutzer gestellter Anfragen. Er soll zeitliche Lokalität ausnutzen und verwendet daher Least Recently Used (LRU) (vgl. z.B. [26]) als Verdrängungsstrategie. Der Waiting Room speichert Ergebnisse von Prefetchinganfragen. Da er räumliche Lokalität ausnutzen soll, wird First in First Out (FIFO) (z.B. [27]) als Verdrängungsstrategie verwendet. Der Anteil des jeweiligen Caches am gesamten Cache sollte entsprechend der im Anfragestrom vorherrschenden Lokalität gewählt werden [5].

Algorithmus 2.3 zeigt die Caching Strategie *PECache*. Der Algorithmus bekommt als Eingabe ein Array R, das die kürzlich durch den Benutzer gestellten Anfragen enthält, und das zuletzt aufgerufene Dokument d.

Algorithmus 2.3: Caching Strategie *PECache*

```
Data : M ist die maximale Anzahl vorzeitig abzurufender Dokumente,
       maxSize beschränkt die Größe vorzeitig abzurufender Dokumente
input : Array R mit den kürzlich gestellten Anfragen,
        das zuletzt abgerufene Dokument d
1  begin
2      R ← R ∪ {d}
3      if not (d ∈ Weighing Room or d ∈ Waiting Room) then
4          Füge d am Kopf der LRU Liste des Weighing Room ein.
5          prefetchSeq ← Prefetch(R, M, maxSize)
6          foreach p ∈ prefetchSeq do
7              Füge p am Ende der FIFO Queue des Waiting Room an.
8          end
9      end
10     else if d ∈ Waiting Room then
11         Entferne d aus dem Waiting Room.
12         Füge d am Kopf der LRU Liste des Weighing Room ein.
13     end
14     else if d ∈ Weighing Room then
15         Verschiebe d an den Kopf der LRU Liste des Weighing Room.
16     end
17 end
```

Zunächst wird das zuletzt abgerufene Dokument d in das Array der kürzlich abgerufenen Dokumente eingefügt. Falls sich d noch nicht im Cache befindet, wird es in den Weighing Room eingefügt und es wird anhand der kürzlich abgerufenen Dokumente Prefetching durchgeführt. Die vorzeitig abgerufenen Dokumente werden in der Reihenfolge, in der sie vermutlich angefragt werden, in den Waiting Room eingefügt. Falls d sich bereits im Waiting Room befindet, wird es aus diesem entfernt und in den Weighing Room eingefügt, da es nun ein Ergebnis einer tatsächlich gestellten Anfrage ist. Falls sich d bereits im Weighing Room befindet, wird es wieder an den Anfang der LRU Liste verschoben, um seine Verweildauer im Cache zu verlängern.

2.5 IQCache Query

IQCache Query ist ein Teilprojekt des am Lehrstuhl für Informationsmanagement der Universität Passau entwickelten semantischen Datenbankcaches IQCache [5]. Das Projekt implementiert in der Programmiersprache Java [6] unter Verwendung des Projekts IQCache JSQLParser einen Parser für SQL Anfragen. Der Parser generiert abstrakte Syntaxbäume (AST) für SQL Anfragen. Die ASTs sind verknüpft mit einem Metadatenmodell, welches das Schema der Datenbank, an welche die Anfragen gerichtet sind, wiedergibt. Das Datenbankschema wird mit Hilfe der dafür durch `java.sql.Connection` [7] bereitgestellten Funktionalität geladen. IQCache JSQLParser ist eine Erweiterung von JSQLParser [8]. Falls sich das Datenbankschema nicht verändert, kann ein Cache zur Speicherung des Metadatenmodells verwendet werden.

IQCache Query implementiert lediglich einen Ausschnitt von SQL. Es existieren die Datentypen `BOOLEAN`, `INTEGER`, `DOUBLE` und `VARCHAR(n)`. Im Falle von `VARCHAR(n)` gibt n die Länge der Zeichenkette an. Andere Datentypen werden im Metadatenmodell als `VARCHAR(50)` wiedergegeben und durch den Parser dementsprechend als Strings interpretiert. Jede Tabelle des Datenbankschemas muss einen Primärschlüssel besitzen. Das Metadatenmodell kann Schemata nicht abbilden: Es wird über alle Schemata einer Datenbank hinweg angenommen, dass darin enthaltene Tabellen zum selben Schema gehören. Problematisch ist dies, wenn zwei Tabellen desselben Namens in verschiedenen Schemata existieren. Geparst werden können `INSERT`-, `UPDATE`-, `DELETE`- und `SELECT`-Statements. Es können weder temporäre Views noch Unteranfragen verwendet werden. Die Projektionsliste einer `SELECT`-Anfrage besteht ausschließlich aus Bezeichnern von Tabellenspalten. Innerhalb der `WHERE`-Klausel einer Anfrage können boolesche Verknüpfungen, Klammerung und arithmetische sowie lexikographische Vergleiche verwendet werden. Die rechte Seite einer Wertzuweisung (`SET`) innerhalb eines `UPDATE`-Statements muss ein Literal sein.

Abbildung 2.7 zeigt die wichtigsten Klassen des Metadatenmodells. Pro Klasse sind jeweils die wichtigsten Attribute darstellt. Die Klasse `Table` modelliert eine Datenbanktabelle. Eine Tabelle besitzt einen Namen und optional einen Alias. Ein Alias kann innerhalb der `FROM`-Klausel einer SQL-Anfrage für eine Tabelle vergeben werden. Tabellenspalten werden durch Instanzen der Klasse `Column` repräsentiert. Der Primärschlüssel einer Tabelle wird in `Table` durch eine Liste von `Columns` wiedergegeben. Eine weitere Liste speichert die nicht-primen Spalten der Tabelle. Eine Instanz der Klasse `Column` hat einen Namen und gehört zu einer Tabelle, die durch ihren Namen referenziert ist. Sofern vorhanden, kann auch der Alias der zugehörigen Tabelle gespeichert werden. Über boolesche Attribute wird angegeben ob eine Spalte Teil eines Primärschlüssels ist und ob ihr Wert auf `NULL` gesetzt werden darf. Der Datentyp der Spalte wird durch eine Referenz auf eine Instanz der Klasse `ColumnType` angezeigt. `ColumnType` ist der abstrakte Obertyp mehrerer Klassen, die die darstellbaren Datentypen repräsentieren.

[5] `http://www.fim.uni-passau.de/fim/fakultaet/lehrstuehle/informationsmanagement/forschungsprojekte/iqcache.html` – zuletzt überprüft am 28.02.2014

[6] `http://www.oracle.com/technetwork/java/index.html` – zuletzt überprüft am 28.02.2014

[7] `http://docs.oracle.com/javase/6/docs/api/java/sql/Connection.html` – zuletzt überprüft am 13.02.2014

[8] `http://jsqlparser.sourceforge.net/` – zuletzt überprüft am 13.02.2014

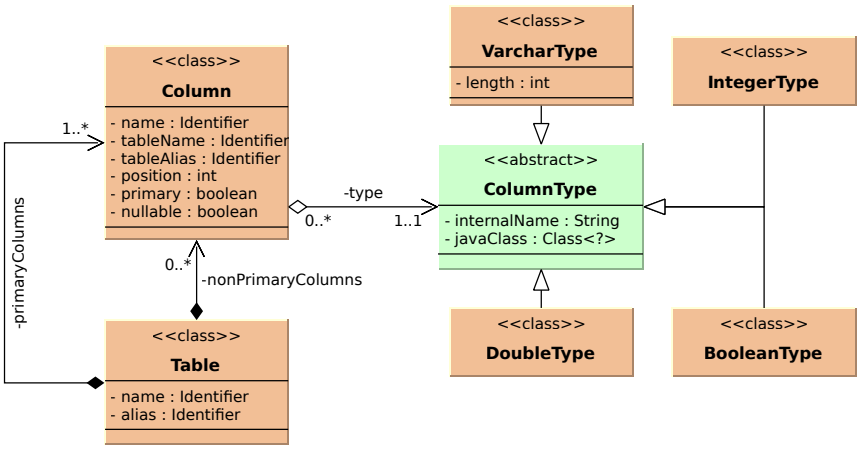


Abbildung 2.7: Darstellung der wichtigsten Klassen des Datenbankmetadatenmodells von IQCache Query

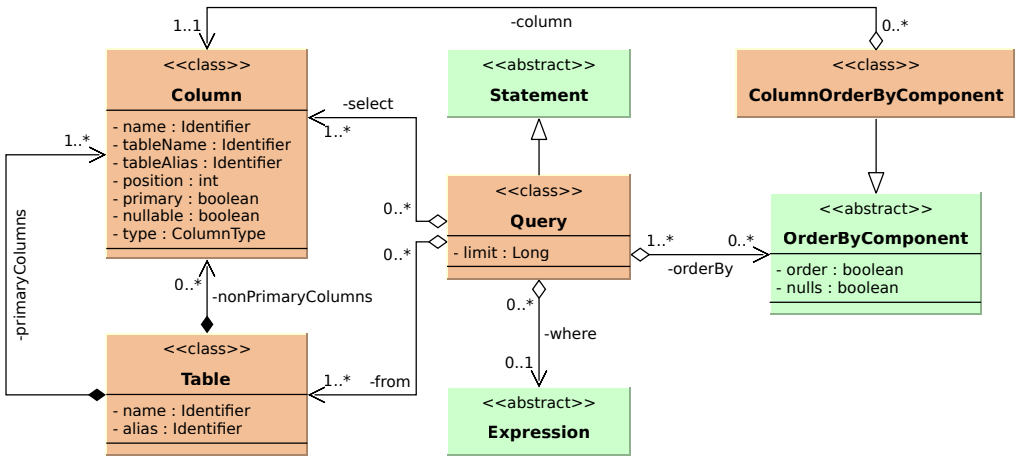


Abbildung 2.8: Zur Darstellung von SQL-Anfragen verwendete Typen

Abbildung 2.8 zeigt die Klasse `Query` zur Darstellung von SQL-Anfragen, sowie die Typen ihrer Attribute. Die Projektionsliste einer Anfrage wird durch eine Liste von Instanzen der Klasse `Column` gebildet. Die `FROM`-Klausel ist in Form einer Liste von `Table`-Instanzen vorhanden. Ein optionales `ORDER BY` wird durch eine Liste von Instanzen der abstrakten Klasse `OrderByComponent` geformt. Deren Attribut `order` gibt an, ob aufsteigend oder absteigend sortiert werden soll. Als ausschließliche Implementierung ist derzeit die Klasse `ColumnOrderByComponent` vorhanden. Das i-te `ColumnOrderByComponent`-Objekt in `orderBy` gibt an, dass in der i-ten Sortierrunde nach dessen referenzierter Spalte sortiert werden soll. Optional kann eine Datenbankanfrage mit einem `LIMIT` versehen werden. Hierfür besitzt die Klasse `Query` das Attribut `limit`. Ebenfalls optional ist die Angabe einer `WHERE`-Klausel. Diese wird in Form eines Objekts der Klasse `Expression` angegeben. `Expression` ist ein abstrakter Typ. Es existieren zahlreiche Untertypen, die innere Knoten und Blätter des Berechnungsbaumes der `WHERE`-Klausel repräsentieren. Die Kindknoten der inneren Knoten sind wiederum vom Typ `Expression`. Innere Knoten repräsentieren die booleschen Operatoren $\wedge$ und $\vee$ als n-äre Operatoren sowie $\neg$ als unären Operator. Blattknoten stehen für boolesche Literale oder binäre Vergleichsoperatoren wie $=, \leq$ und $<$. Letztere sind für die Datentypen `INTEGER`, `DOUBLE` und `VARCHAR` implementiert. Abbildung 2.9 zeigt einen Ausschnitt der Typhierarchie unterhalb der Klasse `Expression`.

Direkt von `Expression` erben die Typen `NodeExpression` und `LeafExpression`. Ersterer ist der Obertyp für innere Knoten, zweiterer für Blattknoten des Berechnungsbaumes. Eine `NodeExpression` besitzt eine Liste von Referenzen auf ihre Unterbäume. `LeafExpressions` unterteilen sich in `BooleanLeafExpressions` (Boolean-Literale) und `Comparisons` (Vergleichsoperatoren). Vergleichsoperatoren sind, wie bereits erwähnt, für unterschiedliche Datentypen implementiert. Beispielsweise stellt `EqualDoubleComparison` einen Vergleich der Form $x = y + c$ auf Gleitkommazahlen dar. x und y stehen für Tabellenspalten und c ist eine Konstante. x und y sind optional und können durch den Wert 0 ersetzt werden. Zu jeder `LeafExpression`, die konstante Werte beinhaltet, existiert zur Darstellung von Prepared Statements eine parametrisierte Variante. Zum Beispiel modelliert `ParamEqualDoubleComparison` einen Ausdruck der Form $x = y+?$. Das Fragezeichen repräsentiert einen Parameter dessen Wert jederzeit gesetzt und wieder gelöscht werden kann. Für Prepared Statements existiert die Klasse `PreparedQuery` als parametrisierte Variante von `Query`.

Nicht parametrisierte `Queries` sind immutable.

Instanzen von Untertypen der Klassen `Expression` und `ColumnType` werden meist zunächst nur über den jeweiligen Obertyp angesprochen. Um typspezifische Operationen auf Instanzen dieser Klassen ohne ausufernde Typüberprüfungen durchzuführen und das sichere Hinzufügen weiterer Untertypen zu gewährleisten, sollte für solche Operationen stets das Visitor Pattern [28] verwendet werden. Die benötigten Interfaces werden durch IQCache Query bereitgestellt.

Für die Generierung von SQL-Code stehen (datenbankspezifische) Pretty-Print-Methoden zur Verfügung.

Beispiel 2.7 illustriert das Ergebnis des Parsers anhand eines einfachen Datenbankschemas und einer dazu passenden SQL-Anfrage.

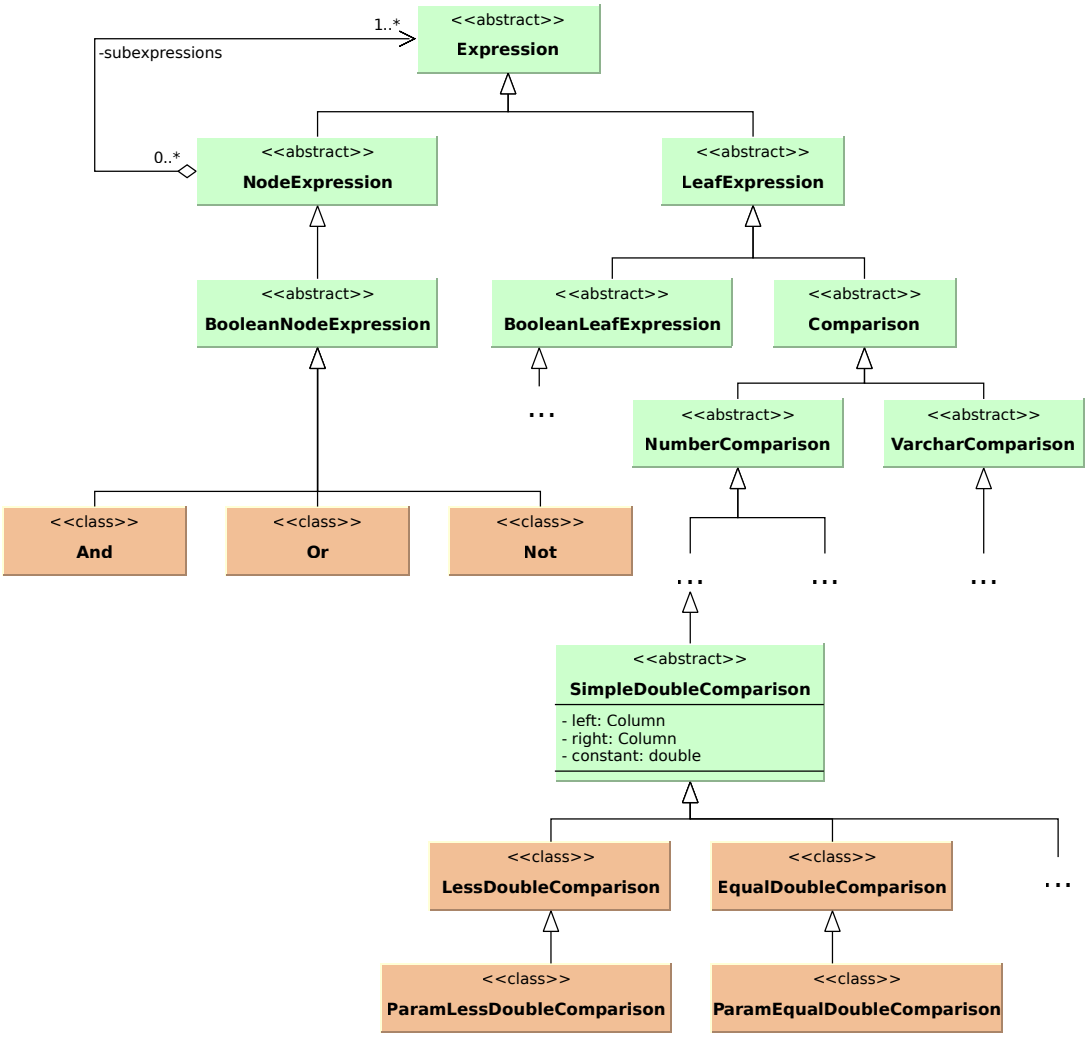


Abbildung 2.9: Ein Teil der zahlreichen Untertypen von *Expression*

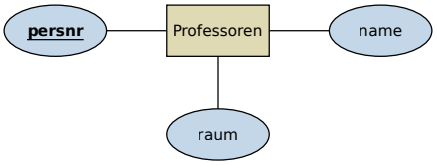


Abbildung 2.10: Ein einfaches Datenbankschema zur Speicherung von Informationen über Professoren

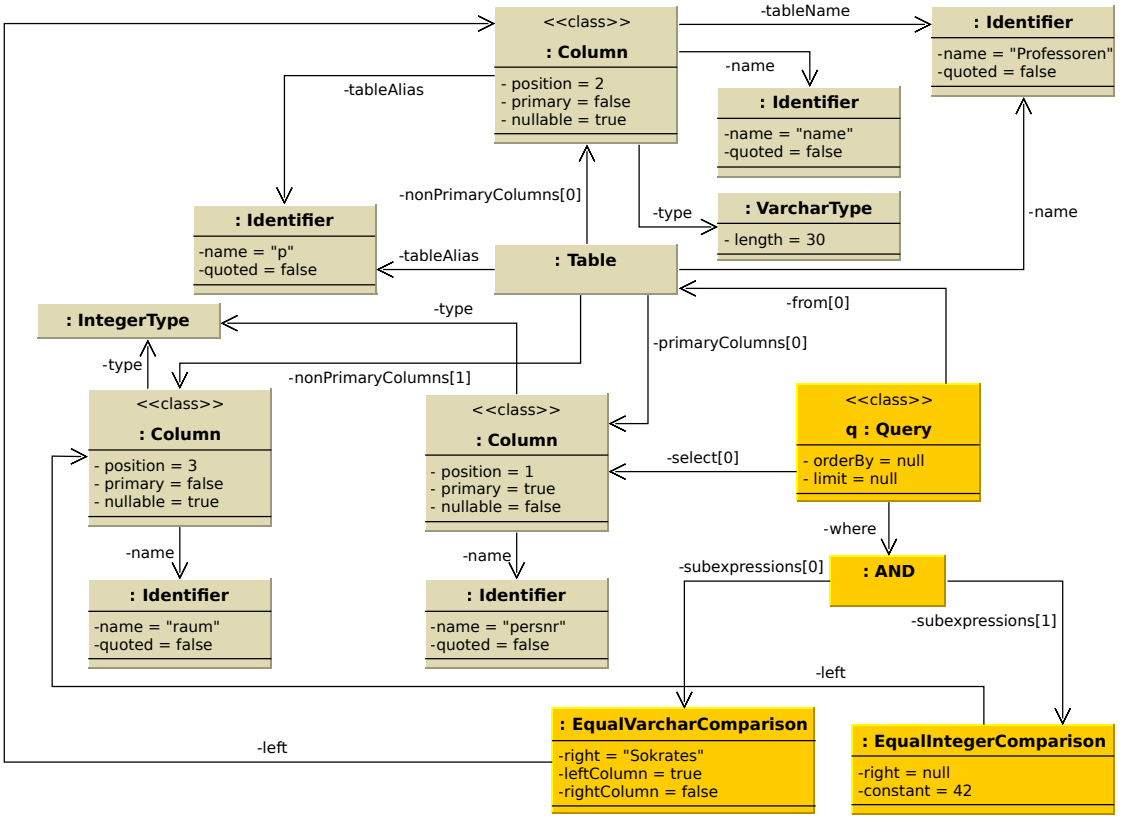


Abbildung 2.11: Ergebnis des Parsens der in Listing 2.7 dargestellten Anfrage q

Beispiel 2.7. Gegeben sei das in Abbildung 2.10 gezeigte Datenbankschema, sowie die in Listing 2.7 gezeigte Anfrage q.

```
1 SELECT p.persnr
2 FROM professoren p
3 WHERE
4     p.name = 'Sokrates' AND p.raum = 42
```

Listing 2.7: Eine Anfrage an die Tabelle Professoren aus Abbildung 2.10

Das Objektdiagramm in Abbildung 2.11 ist das Ergebnis des Parsens von q. Das Metadatenmodell ist braun dargestellt. Der AST der Anfrage q ist gelb.

WM_O für parametrisierte SQL-Anfragen

3 WM_o für parametrisierte SQL-Anfragen

In Abschnitt 2.1 von Kapitel 2 dieser Arbeit wurde der WM_o-Algorithmus für das Prefetching nicht-parametrisierter HTTP-Anfragen beschrieben. Dieses Kapitel beschreibt die in WMoCache implementierte Erweiterung des Algorithmus zu einem Verfahren für das Prefetching von SQL-Anfragen (`SELECT`-Anfragen). Die größte Herausforderungen dabei ist die Tatsache, dass SQL-Anfragen häufig parametrisiert sind. Viele SQL-Anfragen basieren auf einem Prepared Statement, welches in seiner `WHERE`-Klausel einen innerhalb des Wertbereichs seines Datentyps frei setzbaren Parameter beinhaltet. Ein Beispiel für ein parametrisiertes Prepared Statement zeigt das bereits bekannte Listing 2.1.

Das Kapitel gliedert sich wie folgt: Zunächst wird die grundlegende Systemarchitektur von WMoCache, innerhalb derer der Algorithmus verwendet werden soll, beschrieben. Anschließend wird schrittweise entlang des ursprünglichen WM_o-Algorithmus das neue Verfahren entwickelt. Mit Hilfe eines fortlaufenden Beispiels werden die einzelnen Schritte veranschaulicht. In Abschnitt 3.7 wird die bereits aus Abschnitt 2.3 bekannte Lateral-Outer-Union-Strategie an die Bedürfnisse des WMoCache angepasst. Das Kapitel endet mit der Integration des erweiterten WM_o-Algorithmus in die zu Beginn beschriebene Systemarchitektur.

In dem fortlaufenden Beispiel wird eine platzsparende Notation für SQL-Anfragen und deren Ergebnisse verwendet, die an dieser Stelle kurz erläutert sei. Abbildung 3.1 zeigt ein Beispiel dieser Notation. Die Anfrage verwende das Prepared Statement q_a. q_a besitze zwei Parameter. Der Wert des ersten Parameters sei an dieser Stelle $'Bob'$, der Wert des zweiten Parameters sei 42. Unterhalb der horizontalen Linie ist das Ergebnis der Anfrage als Liste von Tupeln dargestellt.

3.1 Systemarchitektur

In Abbildung 3.2 ist die für WMoCache gewählte grundlegende Systemarchitektur dargestellt. In der Literatur zu WM_o[1] wird sowohl die Möglichkeit einer client- als auch einer serverseitigen Lokalisierung des Vorhersagemoduls beschrieben. Hier

$$\frac{q_a('Bob', 42)}{\langle(42,' Alice')\rangle}$$

Abbildung 3.1: Beispiel für die verwendete Notation von Anfragen und Anfrageergebnissen. Das dargestellte Anfrageergebnis enthält nur ein Tupel.

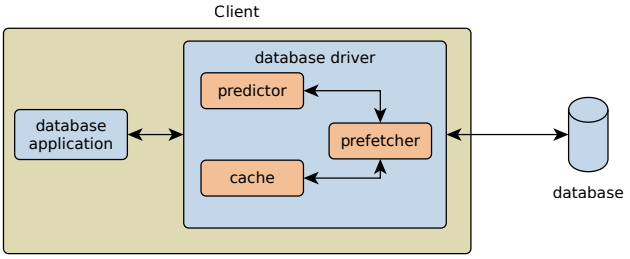


Abbildung 3.2: Systemarchitektur des WM_o-Algorithmus für parametrisierte SQL-Anfragen

wurde die clientseitige Unterbringung gewählt. Dementsprechend handelt es sich bei dem Server um ein unverändertes RDBMS, während der clientseitige Datenbanktreiber (dies ist beispielsweise ein JDBC- (Java Database Connectivity [9] [29]) oder ODBC- (Open Database Connectivity) [30]) Treiber [7]) die zusätzliche Caching-, Vorhersage- und Prefetchingfunktionalität beinhaltet. Falls die Anwendung mehrere Datenbankverbindungen verwendet, muss das Vorhersagemodul zwischen den einzelnen Verbindungen differenzieren, um sinnvolle Prefetchingvorschläge machen zu können.

Eine Implementierung kann somit erfolgen, indem ein clientseitiger Wrapper um einen RDBMS-spezifischen Datenbanktreiber geschaffen wird. Dieser Wrapper ist weitgehend unabhängig von der verwendeten Datenbank und kann somit generisch implementiert und mit verschiedenen Datenbanken verwendet werden. Ein weiterer Vorteil der clientseitigen Lokalisierung ist, dass in die Vorhersagen miteinbezogen werden kann, welches Tupel eines Anfrageergebnisses die Anwendung zuletzt ausgelesen hat. Das System Scalpel [4] macht von dieser Information Gebrauch.

Eine Integration des Vorhersagemoduls in den Server böte dagegen die Möglichkeit, erlerntes Wissen über das Verhalten eines Clients a für einen Client b wiederzuverwenden [1]. Der Nachteil der Integration in den Server ist die Festlegung auf ein bestimmtes RDBMS.

Eine dritte Möglichkeit wäre die Kommunikation der Clients mit einem generischen Proxy, welcher wiederum mit dem eigentlichen RDBMS kommuniziert. Der Proxy würde das Vorhersagemodul enthalten und ist, da alle Clients mit ihm kommunizieren, in der Lage Vorhersagen basierend auf den Anfragen aller Clients, bzw. Datenbankverbindungen zu machen. Clientseitig wären bei letzterem Modell nur der Prefetcher und der Cache lokalisiert. Diese wären in einem Modul implementiert, welches zwar das Interface eines Datenbanktreibers besäße, aber statt mit einer Datenbank mit dem Proxy kommunizieren würde.

[9] `http://www.oracle.com/technetwork/java/javase/jdbc/index.html` – zuletzt überprüft am 20.02.2014

$$\left\langle \frac{q_a('Bob', 42)}{\langle(42,'Alice')\rangle}, \frac{q_b(42)}{\langle(1),(2)\rangle}, \frac{q_h(3.141)}{\langle('Ted')\rangle}, \frac{q_c('Bob')}{\langle(8,9),(10,11)\rangle}, \frac{q_a('Carol', 1337)}{\langle(7,'Alice')\rangle}, \frac{q_b('7')}{\langle(12)\rangle}, \right.$$
$$\left. \frac{q_c('Carol')}{\langle(8,9),(10,11)\rangle} \frac{q_a('Eve', 31)}{\langle(19,'Walter')\rangle}, \frac{q_e(0)}{\langle('Peggy', 1)\rangle}, \frac{q_b(19)}{\langle(31)\rangle}, \frac{q_c('Eve')}{\langle(12,13)\rangle} \right\rangle$$

Abbildung 3.3: Beispiel für eine Folge von Anfragen

3.2 Ersatz für Dokumente

Der WM_o-Algorithmus verwendet als Knoten des Webgraphen, seiner Trainingssequenzen und dementsprechend auch der während der ersten Phase des Algorithmus identifizierten häufigen Subsequenzen der Trainingssequenzen die URLs einzelner Webseiten. Da es sich bei den Webseiten um statische Seiten mit nichtparametrisierten URLs handelt, ist die Zahl der URLs endlich und sie werden wiederholt abgerufen. Erlernt man somit, wie in Abschnitt 2.1 beschrieben, Navigationsmuster bestehend aus Abfolgen von URLs statischer Webseiten, so darf man davon ausgehen, dass diese Abfolgen wiederkehrend sind und für Vorhersagen genutzt werden können. In der Quelle [31] werden Experimente geschildert, die diese These unterstreichen. In Webauftritten, die Organisationen nutzen, um sich öffentlich zu präsentieren, sind statische Webseiten die Regel, daher ist diese Vorgehensweise legitim [10].

Nicht parametrisierte SQL-Anfragen sind bei typischen Datenbankanwendungen hingegen eher selten vertreten. Datenbankanwendungen verwenden zwar meist eine endliche Menge an Prepared Statements, aus welchen durch Setzen von Parameterwerten alle gestellten Anfragen erzeugt werden, allerdings kann es im Extremfall Prepared Statements geben, deren Anfragen jeweils nur ein einziges Mal auftreten. Dies könnte beispielsweise der Fall sein, wenn in den Anfragen Transaktionsidentifikationsnummern oder Zeitstempel als Parameter auftreten. Abbildung 3.3 zeigt eine Sequenz von Anfragen, deren Abfolge von Prepared Statements wiederkehrende Subsequenzen enthält, die aber keine sich wiederholenden Anfragen beinhaltet.

Es ist damit nicht sinnvoll häufige Sequenzen von Anfragen erlernen zu wollen. Stattdessen bietet es sich an, als Ersatz für die Dokumente zunächst die Prepared Statements der Anfragen, oder, mit anderen Worten, Äquivalenzklassen strukturell gleicher Anfragen gemäß Definition 3.1 zu verwenden. Beispiel 3.1 illustriert die Äquivalenzklassenbildung an einigen Paaren von SQL-Anfragen.

Definition 3.1 (**Strukturelle Gleichheit von SQL-Anfragen**). Zwei SQL-Anfragen sind strukturell äquivalent, wenn sie sich nur durch die in ihren `WHERE`-Klauseln vorkommenden Konstanten unterscheiden. Eine Äquivalenzklasse wird repräsentiert durch ein Prepared Statement ihrer dazugehörigen Anfragen, bei welchem alle in der `WHERE`-Klausel vorkommenden Konstanten durch nicht gesetzte Parameter ersetzt wurden. □

[10] vergleiche z.B. `http://www.uni-passau.de`– zuletzt überprüft am 18.02.2014, oder `http://www.fh-kaernten.at/`– zuletzt überprüft am 18.02.2014

```
1 SELECT v.name, v.sws
2 FROM vorlesungen v
3 WHERE
4     v.sws < 3
5 ORDER BY v.sws
6 LIMIT 10
```

```
1 SELECT v.name, v.sws
2 FROM vorlesungen v
3 WHERE
4     v.sws < 4
5 ORDER BY v.sws
6 LIMIT 10
```

Abbildung 3.4: Zwei strukturell äquivalente SQL-Anfragen

```
1 SELECT v.name
2 FROM
3     vorlesungen v,
4     professoren p
5 WHERE
6     p.name = 'Sokrates'
7     OR v.gelesenvon
8         = p.persnr
```

```
1 SELECT v.name
2 FROM
3     vorlesungen v,
4     professoren p
5 WHERE
6     p.name = 'Sokrates'
7     AND v.gelesenvon
8         = p.persnr
```

Abbildung 3.5: Zwei strukturell nicht äquivalente SQL-Anfragen

Beispiel 3.1. Die in Abbildung 3.4 gezeigten Anfragen unterscheiden sich nur durch den Wert des Parameters in Zeile vier. Sie gehören somit zur selben Äquivalenzklasse.

Das nächste Paar von Anfragen (Abbildung 3.5) ist offensichtlich nicht äquivalent, da sich die Anfragen durch den in der `WHERE`-Klausel verwendeten booleschen Operator unterscheiden.

Auch die in Abbildung 3.6 gezeigten Anfragen sind strukturell ungleich, da sie sich sowohl durch ihr `LIMIT`, als auch durch das `ORDER BY` unterscheiden. Beide Unterschiede sind hinreichende Gründe für die Ungleichheit. Andernfalls wäre es bei einer späteren Vorhersage des Prepared Statements der beiden Anfragen schwierig zu entscheiden, welcher Wert für das `LIMIT` und welches `ORDER BY` verwendet werden soll. □

```
1 SELECT *
2 FROM assistenten a
3 ORDER BY a.name ASC
4 LIMIT 42
```

```
1 SELECT *
2 FROM assistenten a
3 ORDER BY a.name DESC
4 LIMIT 21
```

Abbildung 3.6: Zwei strukturell nicht äquivalente SQL-Anfragen

3.3 Trainingssequenzen und Lernphase

Im vorherigen Abschnitt wurde die Entscheidung getroffen als Knoten der Trainingssequenzen Prepared Statements zu verwenden. Nun stellt sich die Frage, wie diese Trainingssequenzen gewonnen werden. Die Beschreibung des WM_o-Algorithmus aus Abschnitt 2.1 lässt diese Entscheidung offen und betrachtet die Trainingssequenzen schlicht als gegeben. Man könnte nun ebenfalls das Vorhandensein von Trainingssequenzen voraussetzen und die Bereitstellung dem Anwender von WMoCache überlassen. Dies würde einen relativ großen Aufwand beim Setup der Datenbankanwendung verursachen, da die Trainingssequenzen entweder anhand des Quellcodes des Programmes manuell erstellt werden müssten – was ohnehin nur möglich wäre, sofern dieser verfügbar ist – oder es müsste eine Testinstallation der Anwendung derart angepasst werden, dass die an die Datenbank gestellten Anfragen aufgezeichnet werden könnten. In beiden Fällen bestünde ein erhöhtes Risiko, dass die Sequenzen nur unzureichend den im produktiven Einsatz auftretenden Transaktionen entsprächen, da z.B. die Auftretenshäufigkeit der einzelnen Transaktionen nicht korrekt wiedergegeben sein könnte, oder andere Anwendungsdaten zugrundegelegt sein könnten.

Eine andere Möglichkeit ist die Erweiterung des Prefetchingtreibers um ein Lernmodul und initial nach dem (ersten) Start der Datenbankanwendung in einer Lernphase die Prepared Statements der auftretenden Anfragen durch das Lernmodul aufzeichnen zu lassen. Nachdem genügend Prepared Statements aufgezeichnet worden sind, werden aus diesen die Trainingssequenzen erstellt und die Prefetchingstruktur kann generiert werden. Danach wird in die Prefetchingphase übergegangen. Optional kann die Lernphase in gewissen zeitlichen Abständen wiederholt werden, um die Prefetchingstruktur beispielsweise an verändertes Nutzerverhalten anzupassen. Da diese Vorgehensweise eine große Erleichterung für den Anwender bedeutet, wurde die Lernphase als fester Bestandteil in den erweiterten WM_o-Algorithmus aufgenommen. Eine Lernphase wird unter Anderen auch in [3, 4, 15] und [31] gewählt. Die Quelle [7] nennt die Verwendung einer Lernphase als eine Möglichkeit zum Erlernen von Anfragemustern. Eine andere Vorgehensweise wählen der QuickMine-Algorithmus [9] und das in [10] beschriebene Modell. Diese lernen fortwährend und passen ihre Prefetchingstruktur ständig dem Nutzer- bzw. Anwendungsverhalten an. Der WM_o-Algorithmus ist für die fortwährende Anpassung der Prefetchingstruktur ungeeignet, da dies mit der erneuten Bestimmung häufiger Subsequenzen nach jeder aufgetretenen Anfrage verbunden wäre.

Ungeklärt ist bisher die Frage, wie der aufgezeichnete Strom von Anfragen in Trainingssequenzen unterteilt wird. Bei willkürlicher Unterteilung bestünde die Gefahr, dass potentiell interessante Anfragemuster durchtrennt würden und dadurch nicht mehr erkannt werden könnten.

Datenbankanwendungen verwenden häufig Transaktionen. Eine Transaktion ist eine Folge von SQL-Statements, die nur im Ganzen durchgeführt werden kann. Der erfolgreiche Abschluss einer Transaktion wird dem Datenbanksystem durch den Befehl `COMMIT` signalisiert. Die durch die Transaktion durchgeführten Änderungen können dann übernommen werden. Eine erfolglose Transaktion wird mittels `ABORT` abgebrochen, wodurch sämtliche während der Transaktion durchgeführten Änderungen rückgängig gemacht werden. Nimmt man an, dass häufige Subsequenzen der Trainingsse-

quenzen meist auch Subsequenzen von Transaktionen sind, so wäre eine Möglichkeit zur Vermeidung des ungewollten Durchtrennens von Anfragemustern die Teilung des Anfragestroms nur an Stellen, an welchen eine Transaktion abgeschlossen wurde. Da anzunehmen ist, dass Datenbankanwendungen existieren, welche ohne Transaktionen arbeiten, ist dieser Ansatz nicht allgemein verwendbar. Alternativ könnte die Datenbankanwendung WMoCache durch spezielle Kommandos oder Funktionsaufrufe den Beginn und das Ende von Transaktionen mitteilen. Dies käme dem in [3] beschriebenen *White Box* Ansatz für Prefetching nahe (siehe Abschnitt 1.1). WMoCache wäre für die Anwendung nicht mehr transparent und könnte nicht ohne Präparation einer Datenbankanwendung sinnvoll mit dieser kombiniert werden.

Die Quelle [3] argumentiert, dass bei einer ausreichenden Länge k der einzelnen Trainingssequenzen durch das Durchtrennen des Anfragestroms nur ein kleiner Anteil der auffindbaren Subsequenzen verworfen werde. Auf der Grundlage dieser Argumentation wird für den erweiterten WM_o-Algorithmus der Algorithmus 3.1 zum Erlernen von Trainingssequenzen gewählt.

Algorithmus 3.1: Erlernen von Trainingssequenzen

```
   Data : Liste D der bisher erlernten Trainingssequenzen,
   Parameter trainingSeqLength: maximale Länge einer Trainingssequenz,
   Anzahl n der bisher gelernten Anfragen
   input  : zuletzt aufgetretene Anfrage q
 1 begin
 2 |  i ← size(D)
 3 |  if i > 0 then
 4 |  |  lastTrainingSeq ← D[i]
 5 |  end
 6 |  if D = [] or size(lastTrainingSeq) ≥ trainingSeqLength then
 7 |  |  lastTrainingSeq ← ⟨⟩
 8 |  |  ++i
 9 |  end
10 |  lastTrainingSeq ← concat(lastTrainingSeq, ⟨q⟩)
11 |  D[i] ← lastTrainingSeq
12 |  ++ n
13 end
```

Der Konfigurationsparameter *trainingSeqLength* gibt die maximale Länge einer Trainingssequenz vor. Die bisher erlernten Trainingssequenzen werden in einer Liste D gespeichert. Als Eingabe bekommt der Algorithmus die zuletzt aufgetretene Anfrage q (bzw. bisher deren Prepared Statement). Falls D leer ist, oder die zuletzt eingefügte Trainingssequenz $lastTrainingSeq$ bereits die maximale Länge erreicht hat, wird eine neue Trainingssequenz $\langle q \rangle$ in D eingefügt. Andernfalls wird $lastTrainingSeq$ um q verlängert. Die Anzahl der bisher erlernten Anfragen wird in der Variablen n erfasst. Es wird ein weiterer Konfigurationsparameter benötigt, welcher angibt nach wie vielen erlernten Anfragen die Prefetchingstruktur generiert werden soll und in die Prefetchingphase gewechselt werden soll.

Bei mehreren offenen Datenbankverbindungen muss entschieden werden, ob für jede Verbindung eigene Trainingssequenzen erlernt werden und ob gegebenenfalls eine oder mehrere Prefetchingstrukturen generiert werden.

$$d_1 = \langle q_a, q_b, q_h, q_c, q_a, q_b, q_c \rangle$$
$$d_2 = \langle q_a, q_e, q_b, q_c \rangle$$

Abbildung 3.7: Zwei Trainingssequenzen

Beispiel 3.2. Wählt man $trainingSeqLength = 7$ und nimmt an, dass während der Lernphase die in Abbildung 3.3 gezeigten Anfragen aufgetreten sind, so erhält man die beiden Trainingssequenzen aus Abbildung 3.7. □

3.4 Ersatz für den Webgraphen

Für den WM_o-Algorithmus ist ein notwendiges Kriterium für die Verwendung einer häufigen Subsequenz s der Trainingssequenzen als Navigationsmuster, dass s ein Pfad in einem gegebenen Webgraphen G ist. Dies basiert auf der Annahme, dass ein Besucher einer Webseite p sein nächstes Navigationsziel primär unter jenen Seiten wählt, die von p aus verlinkt werden [1].

Es stellt sich nun die Frage, was im Kontext von Datenbankanfragen als äquivalenter Ersatz für den Webgraphen dienen kann. Im Folgenden werden drei mögliche Optionen beschrieben.

3.4.1 Verwendung eines vollständigen Graphen

Die Quelle [20] beschreibt eine Variante des WM_o-Algorithmus, die keinen Webgraphen, beziehungsweise einen vollständigen Graphen, der alle in den Trainingssequenzen vorkommenden Anfragen beeinhaltet und paarweise verbindet, verwendet. Es wird argumentiert, dass während des Generierens der Prefetchingstruktur zwar zunächst eine größere Anzahl an Kandidatenpfaden generiert werde, allerdings werde ein Großteil dieser Kandidaten durch das Pruning auf Basis des Subpfadkriteriums wieder verworfen. Die Umsetzung dieser Variante wäre mit niedrigen Kosten verbunden. Allerdings zeigt die Evaluation in [20], dass die Hitrate des Prefetchingcaches bei Verwendung dieser Variante des Algorithmus schlechter ist als bei der Variante mit Graphunterstützung. Zudem ist der Aufwand zur Generierung der Prefetchingstruktur größer, als bei Verwendung eines dünneren Graphen. Von dieser Option wurde daher abgesehen.

3.4.2 Ableitung aus dem Kontrollflussgraphen der Datenbankanwendung

Es ist leicht nachvollziehbar, dass anhand des Kontrollflußgraphen einer Datenbankanwendung entschieden werden kann, welche Datenbankanfragen potentiell auf eine gegebene Anfrage folgen. So setzt die Quelle [7] ein Lernmodul voraus, das Anfragemuster optional durch die Analyse des Programmcodes einer Datenbankanwendung gewinnt. Auch die in den Trainingssequenzen enthaltenen Anfragemuster entsprechen weitgehend den so gewinnbaren Anfragesequenzen. Es ist daher naheliegend,

den Graphen aus dem Kontrollflussgraphen der Datenbankanwendung zu gewinnen, für die WMoCache verwendet werden soll. Zunächst wird der Code der Datenbankanwendung geparst. Anschließend wird aus dem gewonnen AST der Kontrollflussgraph (CFG) des Programmes abgeleitet. Dieser wird nun vereinfacht: Wenn für ein Paar von Statements st_1 und st_2 zur Ausführung von Anfragen (`SELECT`) q_1 und q_2, im CFG ein Pfad einer Länge ≥ 1 von st_1 zu st_2 existiert, der nur über Knoten geht, die keine Ausführungen von Anfragen sind, dann werden die beiden Statements durch eine direkte Kante verbunden. Nun werden alle Kanten, die nicht zwei derartige Statements verbinden, gelöscht. Ebenso werden alle nun isolierten Knoten gelöscht. Die verbleibenden Knoten sind alle Statements zur Ausführung von SQL-Anfragen. Jeder der Knoten sei ab sofort dem Prepared Statement seiner zugehörigen Anfrage zugeordnet. Es existieren nun möglicherweise mehrere Knoten, die demselben Prepared Statement zugeordnet sind. Diese werden zu einem Knoten verschmolzen, der mit allen ein- und ausgehenden Kanten der ursprünglichen Knoten verbunden ist. Man kann sich überzeugen, dass der resultierende Graph alle im Ausgangsprogramm vorkommenden Transaktionen als Wege beinhaltet. Beispiel 3.3 illustriert die skizzierte Vorgehensweise an einem einfachen Programm.

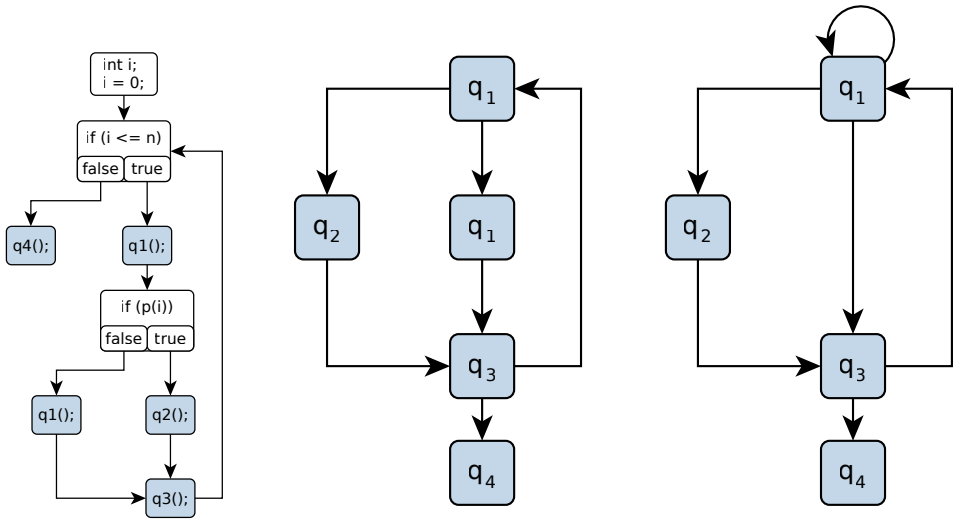
Beispiel 3.3.

```
1  int i;
2
3  for (i = 0; i <= n; ++i) {
4      q1();
5
6      if (p(i))
7          q2();
8      else
9          q1();
10     q3();
11 }
12 q4();
```

Listing 3.1: Ein kleines Programm in C-ähnlicher Syntax mit einigen Datenbankabfragen.

Gegeben sei das Programm aus Listing 3.1. Ein Aufruf der Form `qi()` stehe für das Ausführen einer Anfrage mit Prepared Statement q_i. Abbildung 3.8a zeigt den CFG des Programms. Abbildung 3.8b zeigt den CFG nach dem Eliminieren von Knoten, die keine Anfragen ausführen. Die Knoten sind bereits den Prepared Statements der Anfragen anstelle der ursprünglichen Anweisungen zugeordnet. q_1 sind zwei Knoten zugeordnet. Durch das Zusammenfassen dieser beiden Knoten erhält man den in Abbildung 3.8c dargestellten Anfragegraphen. □

Das Verfahren ist sehr akkurat. Die Analyse des CFG könnte unter Umständen weiterhin erlauben, Datenabhängigkeiten zwischen Schleifeniterationsvariablen und Anfrageparametern zu bestimmen. Das gewonnene Wissen könnte durch Annotationen im Anfragegraphen transportiert werden. Fraglich ist, wie zur Laufzeit die Werte von Kontextvariablen (z.B. Obergrenzen der Iterationsvariablen („n“)) der



(a) CFG des Programmes aus Listing 3.1

(b) Nach dem Eliminieren der Knoten, die keine Statements ausführen.

(c) Der fertige Anfragegraph

Abbildung 3.8: Stadien der Umwandlung eines CFG in einen Anfragegraphen

Schleifen bestimmt werden sollen, ohne Transformationen am Anwendungscode vorzunehmen. Aufgrund des zur Gewinnung des CFG erforderlichen Aufwandes und der notwendigen Verfügbarkeit des Quellcodes der Datenbankanwendung wird das Verfahren jedoch nicht weiter betrachtet.

3.4.3 Approximation des Graphen anhand der Trainingssequenzen

Letztendlich wurde ein Mittelweg zwischen der akkuraten Lösung anhand des CFG und einem vollständigen Graphen gewählt. Der im Folgenden beschriebene Algorithmus generiert aus der Sequenz der während der Lernphase erlernten Anfragen approximativ einen Graphen, der bestenfalls den im vorherigen Unterabschnitt beschriebenen Graphen als Subgraph enthält. In jedem Fall sollte der gewonnene Graph geeignet sein, die Zahl der Kandidatenpfade einzuschränken und so die Effizienz der Generierung der Prefetchingstruktur zu steigern. Aus diesem Grund ist es wichtig auf eine günstige Laufzeitkomplexität zu achten, wobei sinnvollerweise maximal lineare Laufzeit in der Anzahl der erlernten Anfragen anzustreben ist.

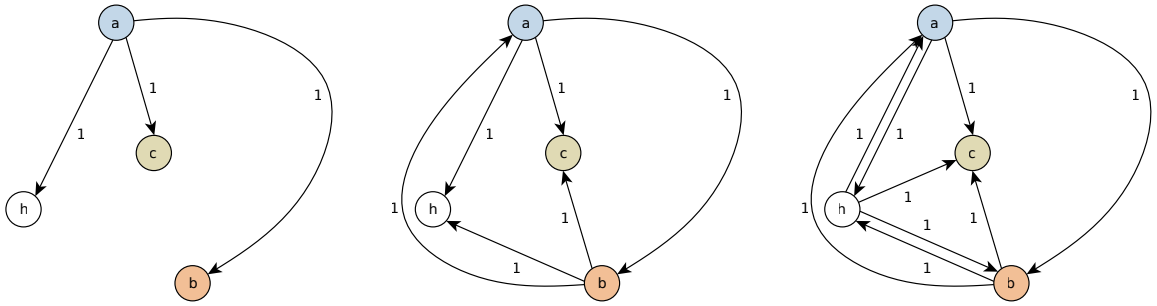
Algorithmus 3.2 verwendet als Eingabe die Liste Q der Prepared Statements der während der Lernphase erlernten Anfragen. Die Sortierung von Q muss der Reihenfolge entsprechen, in welcher die Anfragen aufgezeichnet wurden. Die Ausgabe des Algorithmus ist ein Graph $G = (V, E)$. V ist eine Teilmenge der Prepared Statements der erlernten Anfragen. $E \subseteq V \times V$ ist die Menge der Kanten des Graphen. Zwischen zwei Knoten q_i und q_j existiert eine Kante, wenn in Q mindestens $minFreq$ mal q_j mit einem maximalen Abstand $k-2$ auf q_i folgt.

Beginnend bei den ersten k erlernten Anfragen wird ein Fenster der Länge k über die Liste der erlernten Anfragen bewegt. Mit jeder Iteration der in Zeile sechs begin-

Algorithmus 3.2: Generierung des Anfragegraphen

Data : minimale Auftretenshäufigkeit minFreq eines Paares von Anfragen, um in E aufgenommen zu werden.,
Länge k des Fensters
input : Liste $Q = \langle q_1, q_2, ..., q_n \rangle$ der Prepared Statements der erlernten Anfragen
output : Templategraph $G = (V, E)$

```
1  begin
2      n ← size(Q)
3      // Bestimme Paare von Prepared Statements und deren Häufigkeit
4      // Menge der gefundenen Paare von Prepared Statements
5      P ← ∅
6      for wPos = 1, ..., (n − 1) do
7          // Positioniere das Fenster
8          W ← {q_i | wPos ≤ i ≤ min{wPos + k − 1, n}}
9          foreach q_j ∈ W, q_j ≠ q_wPos do
10             if (q_wPos, q_j) ∉ P then
11                 (q_wPos, q_j).count ← 1
12                 P ← P ∪ {(q_wPos, q_j)}
13             end
14             else
15                 ++(q_wPos, q_j).count
16             end
17         end
18     end
19     // Erstelle den Graphen
20     V ← ∅
21     E ← ∅
22     foreach (q_i, q_j) ∈ P do
23         if (q_i, q_j).count ≥ minFreq then
24             E ← E ∪ {(q_i, q_j)}
25             V ← V ∪ {q_i, q_j}
26         end
27     end
28     return (V, E)
29 end
```



(a) Der Anfragegraph nach Auswertung des Fensters an Position eins.

(b) Der Anfragegraph nach Auswertung des Fensters an Position zwei.

(c) Der Anfragegraph nach Auswertung des Fensters an Position drei.

Abbildung 3.9: Schrittweise Generierung des Anfragegraphen

nenden Schleife wird das Fenster um eine Position weiterbewegt. An jeder Position $wPos \in \{1, ..., |Q| - 1\}$ des Fensters werden Paare (q_{wPos}, q_j) bestehend aus der Anfrage q_{wPos} an Position eins im Fenster und allen im Fenster enthaltenen Anfragen q_j, die ungleich q_{wPos} sind, gebildet. Die Paare werden in der Menge P aufgesammelt. Ist ein Paar noch nicht in P enthalten, wird es eingefügt und seine Häufigkeit mit eins initialisiert. Andernfalls wird die Häufigkeit des Paares um eins erhöht. Nachdem das Fenster vollständig über Q iteriert ist, wird der Anfragegraph erstellt: Jedes Paar von Anfragen, das häufiger aufgetreten ist als eine gegebene Häufigkeit $minFreq$ wird als Kante zum Graphen hinzugefügt. Der resultierende Graph wird als Ergebnis zurückgegeben. Die Laufzeit des Algorithmus ist, sofern für dessen Implementierung Hashtabellen verwendet werden, asymptotisch linear in der Anzahl der erlernten Anfragen. Beispiel 3.4 illustriert den Algorithmus anhand der in Beispiel 3.2 gewonnenen Trainingssequenzen.

Es ist dringend zu beachten, dass k aufgrund des Subpfadkriteriums die maximale Länge häufiger Subsequenzen beschränkt. Ein sinnvoller Wert für den Parameter ist die maximale Anzahl paarweise verschiedener Prepared Statements die innerhalb einer Transaktion der Datenbankanwendung aufeinander folgen können.

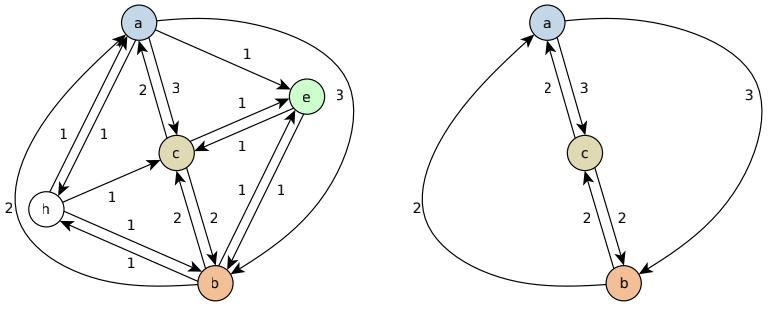
Beispiel 3.4. Gegeben seien die in Beispiel 3.2 aufgezeichneten Trainingssequenzen. Durch Konkatenation von d_1 und d_2 erhält man die folgende Liste von Prepared Statements:

$$Q = concat(d_1, d_2) = \langle q_a, q_b, q_h, q_c, q_a, q_b, q_c, q_a, q_e, q_b, q_c \rangle$$

Die Länge des Fensters sei vier. Es gelte $minFreq = 2$. Die initiale Positionierung des Fensters ist wie folgt:

$$\langle \boxed{q_a, q_b, q_h, q_c}, q_a, q_b, q_c, q_a, q_e, q_b, q_c \rangle$$

Es werden die Paare (q_a, q_b), (q_a, q_h) und (q_a, q_c) generiert. Abbildung 3.9a zeigt diese als Graph. Die Kanten sind mit der Häufigkeit der Paare annotiert.



(a) davor (b) danach

Abbildung 3.10: Der Anfragegraph vor und nach dem Pruning

Das Fenster wird um eine Position weiterbewegt:

$$\langle q_a, \boxed{q_b, q_h, q_c, q_a}, q_b, q_c, q_a, q_e, q_b, q_c \rangle$$

Der resultierende vorläufige Graph entspricht Abbildung 3.9b.

Wiederum wird das Fenster bewegt und befindet sich jetzt an Position drei:

$$\langle q_a, q_b, \boxed{q_h, q_c, q_a, q_b}, q_c, q_a, q_e, q_b, q_c \rangle$$

Der Graph ist in Abbildung 3.9c zu sehen. Das Fenster wird bis ans Ende von Q weitergeschoben. Man erhält abschließend die in Abbildung 3.10a veranschaulichten Paare von Prepared Statements mit ihren Häufigkeiten. Das Pruning seltener Kanten resultiert im in Abbildung 3.10b dargestellten endgültigen Anfragegraphen. □

3.5 Parameterwerte

Mit den bisher beschriebenen Änderungen ist der erweiterte WM_o-Algorithmus in der Lage, eine Vorhersage darüber zu machen, welcher Klasse strukturell äquivalenter Anfragen die nächste durch eine Datenbankanwendung gestellte Anfrage angehört. Das Verfahren ist jedoch noch nicht in der Lage zu entscheiden, auf welche Werte die Parameter des vorhergesagten Prepared Statements gesetzt werden sollen. Ohne diese Information kann offensichtlich kein Prefetching durchgeführt werden. Um Parameterwerte vorhersagen zu können, wurde das bereits aus Abschnitt 2.2 bekannte und von Scalpel (u.a. [4]) übernommene Konzept der Parameterkorrelationen an die Bedürfnisse von WMoCache angepasst. Einer der grundlegenden Unterschiede zu Scalpel ist, dass Ausgabekorrelationen nicht jeweils das zuletzt durch die Anwendung abgerufene Tupel eines Anfrageergebnisses referenzieren, sondern ein durch seinen Zeilenindex (row number) adressiertes Tupel aus dem gesamten Ergebnis der Anfrage. Dieser Abschnitt beschreibt das Erlernen von Parameterkorrelationen.

Zur Bestimmung von Parameterkorrelationen anhand der Trainingssequenzen ist es zunächst notwendig, dass die Trainingssequenzen nicht mehr nur die Prepared Statements der aufgetretenen Anfragen enthalten, sondern auch die Werte der Parameter und die Anfrageergebnisse. Aus Effizienzgründen wurde entschieden die Anzahl der Tupel innerhalb eines Anfrageergebnisses zu beschränken. Ergebnisse, die mehr Tupel enthalten, werden in den Trainingssequenzen durch eine leere Liste ersetzt. Folglich müssen auch im in der Prefetchingphase verwendeten Fenster, das die zuletzt aufgetretenen Anfragen puffert, Parameterwerte und Anfrageergebnisse mit enthalten sein. Typisch ist die Beschränkung der Größe von Anfrageergebnissen auf Anzahlen im einstelligen Bereich. Würden größere Anfrageergebnisse bei der Bestimmung von Ausgabekorrelationen mit einbezogen, wäre auch mit einer stark erhöhten Anzahl unzutreffender Korrelationen zu rechnen.

Um zu einer häufigen Subsequenz der Trainingssequenzen Parameterkorrelationen bestimmen zu können und diese verifizieren zu können, werden möglichst viele Instanzen bzw. Vorkommen dieser Subsequenz benötigt. Der erweiterte WM_o-Algorithmus testet daher für eine gegebene Kandidatensequenz nicht mehr nur, ob diese Subsequenz einer Trainingssequenz ist, sondern er bestimmt eine möglichst große Menge an Auftreten der Subsequenz innerhalb der Trainingssequenz. Insbesondere bei sehr langen Trainingssequenzen stellt man fest, dass es Vorkommen einer Subsequenz geben kann, bei welchen in der Subsequenz benachbarte Anfragen in der Trainingssequenz sehr weit voneinander entfernt liegen. Die Wahrscheinlichkeit, dass zwischen diesen Anfragen ein tatsächlicher semantischer Zusammenhang besteht, ist niedriger, als bei Instanzen der Subsequenz, deren Anfragen nahe beieinander liegen. Würde man zur Bestimmung und Verifikation von Parameterkorrelationen Subsequenzen mit großen Abständen heranziehen, bestünde einerseits ein erhöhtes Risiko, dass Korrelationen erkannt würden, die nicht mit tatsächlich vorhandenen Datenabhängigkeiten korrespondieren, und andererseits die Gefahr, dass tatsächlich zutreffende Korrelationen aufgrund dieser Subsequenzen verworfen würden.

Um der beschriebenen Problematik vorzubeugen, werden mehrere Mittel verwendet. Zunächst wird der Algorithmus zur Bestimmung der Auftreten einer Kandidatensequenz als Subsequenz innerhalb einer Trainingssequenz so gestaltet, dass er vorrangig Auftreten mit geringen Abständen findet. Des Weiteren wird ein Pruning von Subse-

quenzen mit zu großen Abständen eingeführt. Zur Auswahl von Pruningkandidaten sind verschiedene Vorgehensweisen denkbar. In WMoCache ist bisher das Pruning von Vorkommen implementiert, bei welchen der maximale Abstand benachbarter Anfragen über einem konfigurierbaren Schwellwert liegt, sowie das Pruning, falls der durchschnittliche Abstand zu groß ist. Als dritte Maßnahme wird eine Parameterkorrelation nicht sofort verworfen, sobald sie einmal unzutreffend war, sondern es wird zunächst der relative Anteil der betrachteten Subsequenzen ermittelt, in welchen die Korrelation nicht zutreffend war. Daraufhin werden Korrelationen gelöscht, die zu häufig unzutreffend waren. Die zulässige Fehlerrate ist ein weiterer Parameter des WM_o-Algorithmus für SQL.

Für Parameterkorrelationen wird von nun an die im Folgenden gezeigte Notation verwendet. Sie ist angelehnt an die in [4] verwendete Notation.

Alle Parameterkorrelationen besitzen zwei Zähler: h zählt die Anzahl der Subsequenzen, in welchen die Korrelation zutreffend war. Analog zählt f die Zahl der Subsequenzen, in denen die Korrelation nicht festgestellt werden konnte. Mit Hilfe dieser Zähler kann nach Abschluss der Verifikation einer Parameterkorrelation deren Fehlerrate bestimmt werden.

- $\langle i \mid C\ v\ \langle h\ f \rangle \rangle$ beschreibt die Korrelation des i-ten Parameters einer Anfrage mit dem konstanten Wert v. C steht für *constant correlation*.
- Bei $\langle i \mid I\ s_j\ k\ \langle h\ f\ \rangle \rangle$ handelt es sich um die Korrelation des i-ten Parameters einer Anfrage mit dem Wert des k-ten Parameters der j-ten Anfrage innerhalb der jeweiligen Subsequenz. I steht für *input correlation*, oder Eingabekorrelation.
- $\langle i \mid O\ s_j\ \langle r\ c \rangle\ \langle h\ f \rangle \rangle$ steht für die Korrelation des i-ten Parameters einer Anfrage mit dem Wert in der c-ten Spalte des Tupels in Zeile r des Ergebnisses der j-ten Anfrage innerhalb der jeweiligen Subsequenz. O steht für *output correlation*, oder Ausgabekorrelation.

Da für das Präfix jeder Kandidatensequenz einer Länge $k > 2$ aufgrund des Subpfadkriteriums bereits Parameterkorrelationen bekannt sein müssen, genügt es stets Parameterkorrelationen für die Parameter der zuletzt angefügten Anfrage zu suchen. Ist es nicht möglich für alle Parameter der letzten Anfrage zuverlässige Korrelationen zu finden, so ist die Kandidatensequenz für das Prefetching wertlos und wird unabhängig von ihrem Support nicht in L_k übernommen. Durch diese Vorgehensweise ergibt sich das in Beispiel 3.5 geschilderte Problem.

Beispiel 3.5. Im durch eine Datenbankanwendung generierten Anfragestrom trete häufig die Sequenz $\langle q_a, q_b, q_c \rangle$ auf. q_c besitze einen Parameter, der aus q_b, aber nicht aus q_a vorhergesagt werden könne. Somit existieren für die ebenfalls häufige Sequenz $\langle q_a, q_c \rangle$ nicht genügend Parameterkorrelationen. Diese wird somit nicht in L_2 übernommen. Damit kann jedoch aufgrund des Subpfadkriteriums $\langle q_a, q_b, q_c \rangle$ nicht in C_3 eingefügt werden. Insgesamt wird die eigentlich wertvolle häufige Subsequenz $\langle q_a, q_b, q_c \rangle$ nicht gefunden. □

Die Lösung des in Beispiel 3.5 dargestellten Problems besteht in der Einführung einer Menge L_k^*, die im Gegensatz zu L_k Elemente aus C_k aufnimmt, deren Support ausreichend hoch ist, für die aber nicht genügend Parmeterkorrelationen gefunden

werden konnten. Bei der Überprüfung des Subpfadkriteriums während der Generierung von Kandidatensequenzen wird nun auf Enthaltensein in $L_k \cup L_k^*$ geprüft. Verlängert werden aber nur Pfade aus L_k, um weiterhin sicherzustellen, dass für jedes Präfix im resultierenden Trie tatsächlich Prefetching durchgeführt werden kann. Durch diese Entscheidung wird die Eigenschaft des originalen WM_o-Algorithmus geopfert, dass jede Subsequenz einer häufigen Subsequenz der Trainingssequenzen, die für Vorhersagen verwendet wird, ebenfalls im Trie gespeichert ist. Die Methode lässt immer noch einige potentiell interessante Anfragemuster außer Acht. Im Ausblick der Arbeit (Abschnitt 6.1) wird daher eine Verallgemeinerung des Verfahrens präsentiert.

Algorithmus 3.3 beschreibt die neue Vorgehensweise zur Generierung der Prefetchingstruktur.

Als Eingabe erhält der Algorithmus eine Liste D, welche die erlernten Trainingssequenzen in der Reihenfolge ihres Auftretens enthält. Zu Beginn werden die Trainingssequenzen konkateniert und jede Anfrage wird durch ihr Prepared Statement ersetzt. Aus der so gewonnen Liste Q generiert der Aufruf der Funktion *buildGraph* in Zeile zehn den Anfragegraphen $G = (V, E)$. *buildGraph* implementiert den Algorithmus 3.2. Die Menge C_1 der Kandidatenpfade mit Länge eins wird mit der Menge V der Knoten von G initialisiert. Der Parameter k gibt die Länge der aktuell betrachteten Kandidatenpfade an und wird somit mit eins initialisiert. Der Trie, der die gefundenen häufigen Subsequenzen zusammen mit deren Support und Parameterkorrelationen speichern wird, wird initialisiert. Jetzt beginnt die eigentliche Suche nach häufigen Subsequenzen der Trainingssequenzen. Solange C_k nicht leer ist, wird die in Zeile 16 beginnende Schleife ausgeführt. Die Menge L_k zur Speicherung der gefundenen häufigen Subsequenzen mit ausreichend vielen Parameterkorrelationen wird als leere Menge initialisiert. Ebenso wird die zuvor beschriebene Menge L_k^* initialisiert. Es wird über die in C_k enthaltenen Kandidatensequenzen iteriert. Die Funktion $countSupportAndFindCorrs$ wird mit den Parametern D und der aktuell betrachteten Kandidatensequenz $c = \langle c_1, ..., c_k \rangle$ aufgerufen, um den Support von c, sowie Parameterkorrelationen für die Parameter des letzten in c enthaltenen Prepared Statements c_k zu bestimmen. Falls der Support von c ausreichend hoch ist, wird c als Schlüssel für ein Tupel bestehend aus Support und Parameterkorrelationen von c_k in den Trie eingefügt. Falls darüber hinaus zuverlässige Parameterkorrelationen für alle Parameter von c_k gefunden werden konnten oder c die Länge eins hat, wird c in L_k eingefügt. Ansonsten wird die Sequenz in L_k^* eingefügt. Anschließend werden auf Basis des Tries, der temporär auch die in L_k^* enthaltenen Sequenzen speichert, und L_k durch die Funktion *genCandidates* Kandidatenpfade der Länge $k + 1$ generiert und in C_{k+1} gespeichert. Der Trie wird in *genCandidates* zur Überprüfung des Subpfadkriteriums verwendet (vgl. Algorithmus 3.9). Abschließend werden die Elemente aus L_k^* wieder aus dem Trie gelöscht. Diese Vorgehensweise ist korrekt, da es, wie bereits in Abschnitt 2.1.2 begründet, für die Überprüfung des Subpfadkriteriums genügt, das Enthaltensein von Subsequenzen der Länge k in L_k, bzw. eben im Trie zu testen. k wird inkrementiert und es werden in der nächsten Iteration der Schleife die Kandidatenpfade der Länge $k + 1$ verarbeitet. Sobald *genCandidates* keine weiteren Kandidaten mehr generieren kann, terminiert die Schleife. Der resultierende Trie, der nun ausschließlich alle gefundenen häufigen Subsequenzen der Trainingssequenzen, die Korrelationen für alle Parameter aller darin auftretender

Algorithmus 3.3: Generierung der Prefetchingstruktur

```
Data : Mindestsupport minSupport häufiger Subsequenzen
input  : Liste D der Trainingssequenzen
output : Trie mit den gefundenen häufigen Subsequenzen
1  begin
2      // Liste der Prepared Statements der erlernten Anfragen
3      Q ← ⟨⟩
4      foreach d = ⟨d_1, d_2, ...⟩ ∈ D do
5          foreach i = 1, ..., size(d) do
6              Q ← concat(Q, ⟨d_i.template⟩)
7          end
8      end
9      // Generiere Templategraph G = (V, E)
10     G ← buildGraph(Q) // Algorithmus 3.2
11     // Kandidatenpfade der Länge eins
12     C_1 ← V
13     k ← 1
14     // Trie zur Speicherung häufiger Subsequenzen mit Support und
       Parameterkorrelationen
15     t ← emptyTrie()
16     while C_k ≠ ∅ do
17         L_k ← ∅
18         // Sequenzen mit hohem Support aber zu wenigen
           Parameterkorrelationen
19         L*_k ← ∅
20         foreach c = ⟨c_1, ..., c_k⟩ ∈ C_k do
21             (support, corrs) ← countSupportAndFindCorrs(D, c) // Algorithmen
               3.4 und 3.5
22             if support ≥ minSupport then
23                 t ← insert(t, c, (support, corrs))
24                 if corrs = ∅ and size(c_k.params) > 0 and size(c) > 1 then
25                     L*_k ← L*_k ∪ {c}
26                 end
27                 else
28                     L_k ← L_k ∪ {c}
29                 end
30             end
31         end
32         C_{k+1} ← genCandidates(t, G, L_k) // Algorithmus 3.9
33         t ← removeAll(t, L*_k)
34         ++ k
35     end
36     return t
37 end
```

Prepared Statements mit Ausnahme derer an Position eins besitzen, speichert, wird zurückgegeben.

Algorithmus 3.4 zählt den Support von Kandidatensequenzen der Länge eins.

Algorithmus 3.4: Ermittelung von Support und Parameterkorrelationen für Kandidatensequenzen der Länge eins

input : Menge D der erlernten Trainingssequenzen,
Kandidatensequenz $\mathsf{cand} = \langle q_1 \rangle$
output : Tupel aus Support und Parameterkorrelationen für die letzte Anfrage in cand

```
1  begin
2      // Initialisiere den Support von cand
3      support ← 0
4
5      // Zähle den Support
6      foreach d ∈ D do
7          if cand ⪯ d then
8              ++support
9          end
10     end
11     return (support, ∅)
12 end
```

Als Eingabe erhält der Algorithmus die Menge D der Trainingssequenzen und eine Kandidatensequenz *cand* mit Länge eins. Da *cand* aus nur einem Prepared Statement besteht, werden keine Parameterkorrelationen gesucht. Am Anfang wird der Support von *cand* mit 0 initialisiert. Der Algorithmus iteriert über die Trainingssequenzen und erhöht für jede Trainingssequenz, die *cand* als Subsequenz enthält, den Support der Kandidatensequenz um eins. Als Ergebnis wird ein Tupel bestehend aus dem ermittelten Support und der leeren Menge zurückgegeben.

Der Algorithmus 3.5 zählt den Support einer Kandidatensequenz *cand* der Länge $k > 1$ und ermittelt dabei Parameterkorrelationen für das letzte in *cand* enthaltene Prepared Statement.

Der Algorithmus wird durch zwei Parameter konfiguriert. *minSupport* gibt an, ab welchem Support eine Parameterkorrelation als häufig angesehen wird. *maxCorrFailRatio* gibt die maximal zulässige Fehlerrate für Parameterkorrelationen an. Als Eingabe erhält das Verfahren eine Kandidatensequenz $cand = \langle q_1, ..., q_k \rangle$ und die Menge D der Trainingssequenzen. Initial wird der Support von *cand* mit 0 und die Menge $corrs$ der Parameterkorrelationen für q_k mit der leeren Menge initialisiert. Danach werden aus den Trainingssequenzen Subsequenzen extrahiert, die *cand* entsprechen. Für jede Trainingssequenz, die *cand* mindestens einmal als Subsequenz enthält, wird der Support von *cand* inkrementiert. Alle gefundenen passenden Subsequenzen werden in der Liste *matchingSubseqs* gespeichert. Falls der Support unter dem Wert von *minSupport* liegt, ist der Algorithmus fertig und es wird ein Tupel bestehend aus dem ermittelten Support und einer leeren Menge zurückgegeben. Andernfalls werden Parameterkorrelationen für die Parameter von q_k ermittelt. Mit Hilfe der Methode $findCorrelations$ werden potentielle Parameterkorre-

Algorithmus 3.5: Zähle den Support und ermittle die Parameterkorrelationen der letzten Anfrage einer Kandidatensequenz der Länge $k > 1$

Data : Mindestsupport minSupport häufiger Subsequenzen,
Fehlertoleranz maxCorrFailRatio für Parameterkorrelationen
input : Menge D der erlernten Trainingssequenzen,
Kandidatensequenz cand $= \langle q_1, ..., q_k \rangle$
output : Tupel aus Support und Parameterkorrelationen für die letzte Anfrage in cand

```
1  begin
2      // Initialisiere den Support von cand
3      support ← 0
4      // Menge der Parameterkorrelationen für die letzte Anfrage in cand
5      corrs ← ∅
6      // Bestimme möglichst viele passende Subsequenzen
7      matchingSubseqs ← ∅
8      foreach d ∈ D do
9          S ← findMatchingSubsequences(d, cand) // Algorithmus 3.6
10         if S ≠ ∅ then
11             ++ support
12         end
13         matchingSubseqs ← matchingSubseqs ∪ S
14     end
15     if support ≥ minSupport then
16         sf ← εx(x ∈ matchingSubseqs)
17         corrs ← findCorrelations(sf) // Algorithmus 3.7
18         // Entferne die erste Subsequenz
19         matchingSubseqs ← matchingSubseqs \ {sf}
20         foreach s ∈ matchingSubseqs do
21             verifyParameterCorrelations(s, corrs) // Algorithmus 3.8
22         end
23         // Lösche unwahrscheinliche Parameterkorrelationen
24         foreach c ∈ corrs do
25             r ← c.f / (c.h + c.f)
26             if r > maxCorrFailRatio then
27                 corrs ← corrs \ {c}
28             end
29         end
30         if not ∀p ∈ q_k.params ∃c ∈ corrs : c.param = p then
31             corrs ← ∅
32         end
33     end
34     return (support, corrs)
35 end
```

lationen anhand einer der Sequenzen in *matchingSubseqs* bestimmt. Diese werden in *corrs* gespeichert. Anschließend werden die gefundenen Parameterkorrelationen durch die Methode $verifyParameterCorrelations$ anhand der übrigen Sequenzen in *matchingSubseqs* verifiziert. Korrelationen, die nur in wenigen der gefundenen Subsequenzen zutreffend waren, werden gelöscht. Dazu wird für jede Korrelation überprüft, ob der relative Anteil der Subsequenzen, in welchen die Korrelation nicht zutreffend war, unter dem Wert von $maxCorrFailRatio$ liegt. Abschließend wird getestet, ob für jeden Parameter von q_k mindestens eine Korrelation gefunden wurde. Ist dies nicht der Fall, so wird *corrs* durch die leere Menge ersetzt. Das Ergebnis des Algorithmus ist ein Tupel, bestehend aus dem Support von *cand* und der Menge *corrs*.

Die Algorithmen 3.4 und 3.5 werden gemeinsam durch die in Zeile 21 von Algorithmus 3.3 aufgerufene Funktion $countSupportAndFindCorrs$ implementiert.

Algorithmus 3.6 findet innerhalb einer Trainingssequenz Subsequenzen, die zu einer gegebenen Kandidatensequenz passen. Dabei wird, wie zuvor beschrieben, ein Pruning von Subsequenzen durchgeführt, bei welchen der maximale oder durchschnittliche Abstand benachbarter Anfragen zu groß ist.

Der Algorithmus erhält als Eingabe eine Trainingssequenz $d = \langle d_1, ..., d_n \rangle$ und eine Kandidatensequenz $c = \langle c_1, ..., c_m \rangle$. Er iteriert $maxNumIterations$-mal über d und identifiziert dabei Subsequenzen von d, deren Abfolge von Prepared Statements c entspricht. j gibt den Index des Elements der Trainingssequenz an, das aktuell betrachtet wird. Die in Zeile sieben beginnende Schleife wird ausgeführt, solange j kleiner oder gleich der Länge der Trainingssequenz ist. Damit entspricht die Schleife einer Iteration über d. Innerhalb dieser Schleife wird über die Kandidatensequenz c iteriert. Für das jeweils aktuell betrachtete Element c_k wird solange weiter forwärts über d gelaufen, bis eine Anfrage d_j gefunden wurde, die das aktuell betrachtete Element aus c als Prepared Statement verwendet, oder bis das Ende von d erreicht wurde. Die gefundenen Anfragen werden in der Liste s eingesammelt. Eine Anfrage aus d, die bereits als Startpunkt einer potentiell passenden Subsequenz verwendet wurde, wird mit dem Flag $usedAsStP$ (besteht nur während des aktuellen Durchlaufs des Algorithmus) markiert und wird daher nicht erneut als solcher verwendet. Andernfalls würde bei jeder Iteration über die Trainingssequenz immer wieder ab dieser Anfrage gesucht. Dies hätte zur Folge, dass einerseits passende Subsequenzen möglicherweise nicht gefunden würden und andererseits Subsequenzen mehrfach gefunden würden. Beispiel 3.6 illustriert dies. Falls s nach einer Iteration über c eine vollständige zu c passende Subsequenz enthält, prüft die Funktion *decide* die ursprünglichen Abstände in s benachbarter Anfragen und gegebenenfalls wird die Subsequenz in die Ergebnismenge S eingefügt.

Beispiel 3.6. Gegeben seien die Sequenzen $c = \langle a, b, c \rangle$ und $d = \langle a, a, b, c \rangle$. Aufgabe sei es, in d analog zu Algorithmus 3.6 Subsequenzen zu finden, die c entsprechen. Es gelte $maxNumIterations = 2$. In der ersten Iteration wird die an Position eins beginnende Subsequenz gefunden. Würde das erste Element von d daraufhin nicht als verwendet markiert, würde bei der zweiten Iteration über d wiederum die selben Subsequenzen gefunden und die an Position zwei beginnende Subsequenz bliebe unentdeckt. □

Der Algorithmus wird durch die in Zeile neun von Algorithmus 3.5 verwendete Funktion $findMatchingSubsequences$ implementiert. Er findet in Abhängigkeit vom

Algorithmus 3.6: Finde Subsequenzen zu einer Kandidatensequenz in einer Trainingssequenz

Data : Maximale Anzahl maxNumIterations an Iterationen über d
input : Trainigssequenz $d = \langle d_1, ..., d_n \rangle$,
Kandidatensequenz $c = \langle c_1, ..., c_m \rangle$
output : Menge aus Subsequenzen von d, die zu c passen

```
1  begin
2      // Gefundene Subsequenzen
3      S ← ∅
4      for i = 1, ..., maxNumIterations do
5          // Iteriere über d
6          j ← 1
7          while j ≤ n do
8              // Gefundene Subsequenz
9              s ← ⟨⟩
10             // Iteriere über c
11             for k = 1, ..., m do
12                 // Suche nächste passende Anfrage in d
13                 found ← false
14                 while (not found) and j ≤ n do
15                     if d_j.template = c_k then
16                         if not(s = ⟨⟩ and d_j.usedAsStP) then
17                             if s = ⟨⟩ then
18                                 d_j.usedAsStP ← true
19                             end
20                             s ← concat(s, ⟨d_j⟩)
21                             found ← true
22                         end
23                     end
24                     ++ j
25                 end
26             end
27             if size(s) = m and decide(s) then
28                 S ← S ∪ {s}
29             end
30         end
31     end
32     return S
33 end
```

Wert, welcher für $maxNumIterations$ gewählt wurde, nicht alle in einer Trainingssequenz vorkommenden passenden Subsequenzen.

Als Nächstes wird die in Zeile 17 von Algorithmus 3.5 aufgerufene Funktion *findCorrelations* betrachtet. Sie implementiert Algorithmus 3.7.

Algorithmus 3.7: Finden möglicher Parameterkorrelationen

```
   Data : Größe corrsWindowSize des Fensters, innerhalb dessen Korrelationen
          gesucht werden
   input  : Subsequenz s = ⟨s_1, ..., s_n⟩
   output : Menge möglicher Parameterkorrelationen für s_n
 1 begin
 2     // Gefundene Parameterkorrelationen
 3     corrs ← ∅
 4     foreach p_i ∈ s_n.params do
 5         v ← p_i.value
 6         // Korrelation mit konstantem Wert
 7         corrs ← corrs ∪ {⟨i | C v ⟨1 0⟩⟩}
 8         for j = (max{1, n − corrsWindowSize}), ..., (n − 1) do
 9             // Eingabekorrelationen
10             foreach p_k ∈ s_j.params, p_k.value.type = v.type and p_k.value = v do
11                 corrs ← corrs ∪ {⟨i | I s_j k ⟨1 0⟩⟩}
12             end
13             // Ausgabekorrelationen
14             foreach v_{k,l} ∈ s_j.result, v_{k,l}.type = v.type and v_{k,l}.value = v do
15                 corrs ← corrs ∪ {⟨i | O s_j ⟨k l⟩ ⟨1 0⟩⟩}
16             end
17         end
18     end
19     return corrs
20 end
```

Der Algorithmus findet in einer gegebenen Subsequenz $s = \langle s_1, ..., s_n \rangle, n > 1$ einer Trainingssequenz Parameterkorrelationen für die Parameter von s_n. Zu diesem Zweck werden die Werte der Parameter von s_n mit den Eingabeparametern und Ergebniswerten von maximal $corrsWindowSize$ vielen Vorgängern von s_n innerhalb von s verglichen. Zu Beginn ist die Menge $corrs$ der gefundenen Parameterkorrelationen leer. Die in Zeile vier beginnende Schleife iteriert über die Parameter von s_n. Für jeden Parameter p_i kann eine konstante Korrelation mit dessen aktuellem Wert erstellt werden. Anschließend wird über maximal $corrsWindowSize$ viele Vorgänger von s_n innerhalb von s iteriert. Für jeden Parameter eines der Vorgänger, der denselben Typ und Wert wie p_i hat, wird eine Eingabekorrelation mit p_i erzeugt. Für jeden Ergebniswert eines Vorgängers, der gleich dem Wert von p_i ist, wird eine Ausgabekorrelation mit p_i erzeugt. Bei jeder Korrelation wird der Zähler für Subsequenzen, in welchen die Korrelation zutreffend war, mit eins initialisiert. Der Zähler für Subsequenzen, in denen die Korrelation nicht zutreffend war, wird mit 0 initialisiert.

Algorithmus 3.8 beschreibt die in Zeile 21 von Algorithmus 3.5 aufgerufene Funktion $verifyParameterCorrelations$.

Algorithmus 3.8: Verifikation von Parameterkorrelationen

```
   input  : Menge corrs von Parameterkorrelationen,
            Subsequenz s = ⟨s_1, ..., s_n⟩
 1 begin
 2   foreach c ∈ corrs do
 3     holds ← false
 4     // Korrelationen mit konstantem Wert
 5     if c = ⟨i | C v ⟨h f⟩⟩ then
 6       holds ← (s_n.params[i].value = v)
 7     end
 8     // Eingabekorrelation
 9     else if c = ⟨i | I s_j k ⟨h f ⟩⟩ then
10       holds ← (s_n.params[i].value = s_j.params[k].value)
11     end
12     // Ausgabekorrelation
13     else if c = ⟨i | O s_j ⟨r c⟩ ⟨h f⟩⟩ then
14       holds ← (size(s_j.result) ≥ r and s_n.params[i].value = s_j.result[r][c])
15     end
16     if holds then
17       ++c.h
18     end
19     else
20       ++c.f
21     end
22   end
23 end
```

Der Algorithmus erhält als Eingabe eine Menge $corrs$ von Parameterkorrelationen und eine Subsequenz $s = \langle s_1, ..., s_n \rangle$ einer Trainingssequenz. Um die Korrelationen sinnvoll anhand von s verifizieren zu können, sollte s dieselbe Abfolge von Prepared Statements besitzen, wie die Subsequenz, anhand derer $corrs$ erstellt wurde. Der Algorithmus iteriert über die Elemente von $corrs$. Für jede Korrelation wird geprüft, ob sie auch für s_n (ggf. mit den entsprechenden Vorgängern aus s) zutreffend ist. Im Falle einer Ausgabekorrelation muss bedacht werden, dass der Index eines adressierten Tupels im Ergebnis der adressierten Anfrage s_j aus s möglicherweise nicht vorkommt, falls das Anfrageergebnis, anhand dessen die Korrelation erstellt wurde, mehr Tupel als das Ergebnis von s_j enthielt. In diesem Fall wird die Korrelation als unzutreffend erachtet. Ist der adressierte Ergebniswert in s `NULL`, so wird die Korrelation ebenfalls als unzutreffend betrachtet. Trifft eine Korrelation auch in s zu, so wird deren Zähler für Subsequenzen, in denen sie zutreffend ist, inkrementiert. Andernfalls wird der Zähler für Subsequenzen, in denen die Korrelation nicht festgestellt werden konnte, um eins erhöht.

Abschließend beschreibt Algorithmus 3.9 die Generierung von Kandidatenpfaden der Länge $k+1$ auf Basis von L_k, des bisher generierten Tries, der temporär auch häufige Subsequenzen der Länge k, für die nicht genügend zuverlässige Parameter-

korrelationen gefunden werden konnten, enthält, und des Anfragegraphen G. Die in Algorithmus 3.3 aufgerufene Funktion *genCandidates* implementiert diesen Algorithmus.

Algorithmus 3.9: Generierung von Kandidatenpfaden der Länge $k+1$

input : Trie t,
Graph $G = (V, E)$,
Menge L_k der häufigen Subsequenzen mit Länge k
output : Kandidatensequenzen der Länge $k+1$

```
1  begin
2      C_{k+1} = ∅
3      foreach l = ⟨l_1, ..., l_k⟩, l ∈ L_k do
4          N^+(l_k) ← {v | (l_k, v) ∈ E}
5          foreach v ∈ N^+(l_k) do
6              l' ← ⟨l_2, ...., l_k, v⟩
7              // Überprüfung des Subpfadkriteriums
8              if v ∉ L and contains(t, l') then
9                  c ← ⟨l_1, ..., l_k, v⟩
10                 if ∀s ⪯ c : (s ≠ l' ∧ |s| = k ⇒ contains(t, s)) then
11                     C_{k+1} ← C_{k+1} ∪ {c}
12                 end
13             end
14         end
15     end
16     return C_{k+1}
17 end
```

Der Algorithmus entspricht der Vorgehensweise des WM_o-Algorithmus für Web-Prefetching zur Verlängerung häufiger Subsequenzen der Länge k zu Kandidatenpfaden der Länge $k+1$ (vgl. Algorithmus 2.2, bzw. Abschnitt 2.1) mit dem Unterschied, dass zur Überprüfung des Subpfadkriteriums nicht L_k, sondern der bisher generierte Trie herangezogen wird.

Zur Veranschaulichung der Generierung der Prefetchingstruktur des WM_o-Algorithmus für SQL-Anfragen dient Beispiel 3.7. Das Beispiel wendet den Algorithmus 3.3 auf die Trainingssequenzen des fortlaufenden Beispiels an.

Beispiel 3.7. Abbildung 3.11 zeigt die in Beispiel 3.2 gewonnenen Trainingssequenzen. Diesmal sind zu jedem Prepared Statement auch die Parameterwerte und die Anfrageergebnisse angegeben.

Die Generierung des Anfragegraphen wurde bereits in Beispiel 3.4 durchgeführt. Abbildung 3.12 zeigt noch einmal den resultierenden Graphen.

Es gelte $minSupport = 2$. Die maximal zulässige Fehlerrate für Parameterkorrelationen sei 0. Die Länge $corrsWindowSize$ des Fenster, innerhalb dessen nach Parameterkorrelationen gesucht wird, sei drei. Zwischen benachbarten Anfragen innerhalb einer Subsequenz dürfen maximal zwei Anfragen liegen.

$$d_1 = \left\langle \frac{q_a('Bob', 42)}{\langle(42,'Alice')\rangle}, \frac{q_b(42)}{\langle(1),(2)\rangle}, \frac{q_h(3.141)}{\langle('Ted')\rangle}, \frac{q_c('Bob')}{\langle(8,9),(10,11)\rangle}, \frac{q_a('Carol', 1337)}{\langle(7,'Alice')\rangle}, \frac{q_b('7')}{\langle(12)\rangle}, \frac{q_c('Carol')}{\langle(8,9),(10,11)\rangle} \right\rangle$$

$$d_2 = \left\langle \frac{q_a('Eve', 31)}{\langle(19,'Walter')\rangle}, \frac{q_e(0)}{\langle('Peggy', 1)\rangle}, \frac{q_b(19)}{\langle(31)\rangle}, \frac{q_c('Eve')}{\langle(12,13)\rangle} \right\rangle$$

Abbildung 3.11: Die Trainingssequenzen aus Abbildung 3.7 inklusive Parameterwerten und Anfrageergebnissen

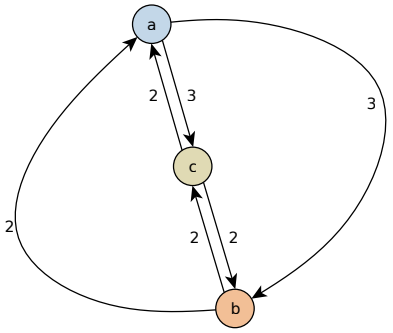


Abbildung 3.12: Aus den Trainingssequenzen in Abbildung 3.11 gewonnener Anfragegraph (Fenstergröße: 4, $minFreq = 2$)

Die Menge C_1 der Kandidatenpfade mit Länge $k = 1$ entspricht den Knoten des Anfragegraphen. Somit gilt

$$C_1 = \{\langle q_a\rangle, \langle q_b\rangle, \langle q_c\rangle\}.$$

Runde 1

Verarbeitung von Kandidatenpfaden der Länge eins.

Zunächst wird der Support der Kandidatensequenzen bestimmt:

- $\langle q_a\rangle : 2$
- $\langle q_b\rangle : 2$
- $\langle q_c\rangle : 2$

Alle drei Sequenzen sind häufig. Somit gilt

$$L_1 = \{\langle q_a\rangle, \langle q_b\rangle, \langle q_c\rangle\}.$$

Abbildung 3.13 zeigt den Trie zur Speicherung der häufigen Subsequenzen nach Runde eins.

Durch Verlängerung der Sequenzen in L_1 anhand des Anfragegraphen, erhält man die folgenden Sequenzen:

$$\{\langle q_a, q_b\rangle, \langle q_a, q_c\rangle, \langle q_b, q_a\rangle, \langle q_b, q_c\rangle, \langle q_c, q_a\rangle, \langle q_c, q_b\rangle\}$$

Das Subpfadkriterium wird durch jede der verlängerten Sequenzen erfüllt. Somit gilt

$$C_2 = \{\langle q_a, q_b\rangle, \langle q_a, q_c\rangle, \langle q_b, q_a\rangle, \langle q_b, q_c\rangle, \langle q_c, q_a\rangle, \langle q_c, q_b\rangle\}.$$

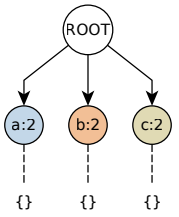


Abbildung 3.13: Trie mit häufigen Subsequenzen der Länge eins. Die Knoten speichern den Support der jeweiligen Subsequenzen und deren Parameterkorrelationen.

Runde 2

In Runde zwei des Algorithmus wird der Support der Kandidatenpfade in C_2 anhand der Trainingssequenzen ermittelt:

- $\langle q_a, q_b \rangle : 2$
- $\langle q_b, q_a \rangle : 1$
- $\langle q_c, q_a \rangle : 1$
- $\langle q_a, q_c \rangle : 2$
- $\langle q_b, q_c \rangle : 2$
- $\langle q_c, q_b \rangle : 1$

Nach dem Pruning seltener Pfade verbleiben $\langle q_a, q_b \rangle : 2$, $\langle q_a, q_c \rangle : 2$ und $\langle q_b, q_c \rangle : 2$. Für jede dieser Sequenzen werden im Folgenden Parameterkorrelationen bestimmt.

Bestimmung von Parameterkorrelationen für $\langle q_a, q_b \rangle$

Aus den Trainingssequenzen werden zu $\langle q_a, q_b \rangle$ passende Subsequenzen extrahiert. In der Trainingssequenz d_1 findet man

$$S_1 = \left[\left\langle \frac{q_a('Bob', 42)}{\langle (42,' Alice') \rangle}, \frac{q_b(42)}{\langle (1), (2) \rangle} \right\rangle, \left\langle \frac{q_a('Carol', 1337)}{\langle (7,' Alice') \rangle}, \frac{q_b('7')}{\langle (12) \rangle} \right\rangle \right].$$

In d_2 werden die in der Liste [11] S_2 gespeicherten Vorkommen gefunden:

$$S_2 = \left[\left\langle \frac{q_a('Eve', 31)}{\langle (19,' Walter') \rangle}, \frac{q_b(19)}{\langle (31) \rangle} \right\rangle \right]$$

Durch Konkatenation erhält man

$$S = concat(S_1, S_2) = \left[\left\langle \frac{q_a('Bob', 42)}{\langle (42,' Alice') \rangle}, \frac{q_b(42)}{\langle (1), (2) \rangle} \right\rangle, \left\langle \frac{q_a('Carol', 1337)}{\langle (7,' Alice') \rangle}, \frac{q_b('7')}{\langle (12) \rangle} \right\rangle, \left\langle \frac{q_a('Eve', 31)}{\langle (19,' Walter') \rangle}, \frac{q_b(19)}{\langle (31) \rangle} \right\rangle \right].$$

[11] Anmerkung: In Algorithmus 3.5 werden an dieser Stelle Mengen verwendet. Zur Vereinfachung der Beschreibung werden hier Listen benutzt.

Anhand der ersten Sequenz in S werden die folgenden Paramterkorrelationen für q_b gefunden:

$$C_{a\to b} = \{\langle 1 \mid C\ 42\ \langle 1\ 0\rangle\rangle, \langle 1 \mid I\ q_a\ 2\ \langle 1\ 0\rangle\rangle, \langle 1 \mid O\ q_a\ \langle 1\ 1\rangle\ \langle 1\ 0\rangle\rangle\}$$

q_b besitzt einen Parameter. Dieser hat in der betrachteten Instanz der Subsequenz den Wert 42. Somit wird eine Korrelation mit der Konstanten 42 vermutet. Außerdem tritt der Wert 42 im zweiten Parameter der vorhergehenden Anfrage mit Prepared Statement q_a auf. Entsprechend wird die passende Eingabekorrelation erzeugt. Schlussendlich ist der Wert 42 auch im Ergebnis von q_a vertreten. Dies resultiert in der entsprechenden Ausgabekorrelation.

Die Korrelationen werden anhand der beiden anderen Vorkommen der Subsequenz $\langle q_a, q_b\rangle$ verifiziert.

Zunächst wird die zweite Instanz der Subsequenz betrachtet:

$$\left\langle \frac{q_a('Carol', 1337)}{\langle(7,'Alice')\rangle}, \frac{q_b('7')}{\langle(12)\rangle} \right\rangle$$

Diesmal hat der Parameter von q_b den Wert sieben. Somit ist die zuvor gefundene Korrelation mit dem konstanten Wert 42 unzutreffend. Der Zähler f (Anzahl der Subsequenzen, in welchen eine Korrelation nicht zutreffend ist) wird inkrementiert. Da der zweite Parameter von q_a hier den Wert 1337 hat, ist auch die zuvor gefundene Eingabekorrelation nicht zutreffend. Auch bei dieser Korrelation wird damit f um eins erhöht. Die Ausgabekorrelation mit dem Wert an Position $\langle 1\ 1\rangle$ des Ergebnisses von q_a ist erneut zutreffend. Damit hat der Zähler h (Anzahl der Subsequenzen, in welchen eine Korrelation zutreffend ist) den Wert zwei. Man erhält

$$C_{a\to b} = \{\langle 1 \mid C\ 42\ \langle 1\ 1\rangle\rangle, \langle 1 \mid I\ q_a\ 2\ \langle 1\ 1\rangle\rangle, \langle 1 \mid O\ q_a\ \langle 1\ 1\rangle\ \langle 2\ 0\rangle\rangle\}.$$

Die dritte Instanz der Subsequenz wird betrachtet:

$$\left\langle \frac{q_a('Eve', 31)}{\langle(19,'Walter')\rangle}, \frac{q_b(19)}{\langle(31)\rangle} \right\rangle$$

Auch hier ist nur die Ausgabekorrelation zutreffend. Damit resultiert

$$C_{a\to b} = \{\langle 1 \mid C\ 42\ \langle 1\ 2\rangle\rangle, \langle 1 \mid I\ q_a\ 2\ \langle 1\ 2\rangle\rangle, \langle 1 \mid O\ q_a\ \langle 1\ 1\rangle\ \langle 3\ 0\rangle\rangle\}.$$

Die konstante Korrelation und die Eingabekorrelation waren nur in einer der drei Instanzen der Subsequenz $\langle q_a, q_b\rangle$ zutreffend. Damit liegt deren Fehlerrate über der maximal zulässigen Fehlerrate 0. Deshalb gilt abschließend

$$C_{a\to b} = \{\langle 1 \mid O\ q_a\ \langle 1\ 1\rangle\ \langle 3\ 0\rangle\rangle\}.$$

Bestimmung von Parameterkorrelationen für $\langle q_a, q_c \rangle$

In den Trainingssequenzen werden die folgenden Instanzen der Subsequenz $\langle q_a, q_c \rangle$ gefunden:

$$S = \left[\left\langle \frac{q_a('Bob', 42)}{\langle (42,' Alice') \rangle}, \frac{q_c('Bob')}{\langle (8,9),(10,11) \rangle} \right\rangle, \left\langle \frac{q_a('Carol', 1337)}{\langle (7,' Alice') \rangle}, \frac{q_c('Carol')}{\langle (8,9),(10,11) \rangle} \right\rangle, \right.$$
$$\left. \left\langle \frac{q_a('Eve', 31)}{\langle (19,' Walter') \rangle}, \frac{q_c('Eve')}{\langle (12,13) \rangle} \right\rangle \right]$$

Anhand der ersten Sequenz in S werden die folgenden potentiellen Parameterkorrelationen für den Parameter von q_c gefunden:

$$C_{a \to c} = \{ \langle 1 \mid C\ 'Bob', \langle 1\ 0 \rangle \rangle, \langle 1 \mid I\ q_a\ 1\ \langle 1\ 0 \rangle \rangle \}$$

Die Verifikation der Parameterkorrelationen in $C_{a \to c}$ anhand der zweiten und dritten Instanz der Subsequenz $\langle q_a, q_c \rangle$ ergibt, dass nur die Eingabekorrelation in allen Instanzen der Subsequenz zutreffend war. Die Korrelation mit der Konstanten $'Bob'$ wird entfernt.

$$\Rightarrow C_{a \to c} = \{ \langle 1 \mid I\ q_a\ 1\ \langle 3\ 0 \rangle \rangle \}$$

Bestimmung von Parameterkorrelationen für $\langle q_b, q_c \rangle$

In den Trainingssequenzen werden die folgenden Instanzen der Subsequenz $\langle q_b, q_c \rangle$ gefunden:

$$S = \left[\left\langle \frac{q_b(42)}{\langle (1),(2) \rangle}, \frac{q_c('Bob')}{\langle (8,9),(10,11) \rangle} \right\rangle, \left\langle \frac{q_b('7')}{\langle (12) \rangle}, \frac{q_c('Carol')}{\langle (8,9),(10,11) \rangle} \right\rangle, \right.$$
$$\left. \left\langle \frac{q_b(19)}{\langle (31) \rangle}, \frac{q_c('Eve')}{\langle (12,13) \rangle} \right\rangle \right]$$

Anhand der ersten Instanz wird lediglich eine potentielle Korrelation mit der Konstanten $'Bob'$ für den Parameter von q_c gefunden. Da dieser Parameter in den beiden anderen Instanzen der Subsequenz andere Werte hat, existiert für den Parameter keine zuverlässige Parameterkorrelation. Damit wird $\langle q_b, q_c \rangle$ nicht in L_2 eingefügt und es gilt

$$L_2 = \{ \langle q_a, q_b \rangle, \langle q_a, q_c \rangle \}.$$

Da der Support von $\langle q_b, q_c \rangle$ über dem Mindestsupport liegt, wird $\langle q_b, q_c \rangle$ in L_2^* aufgenommen. Alle drei häufigen Subsequenzen der Länge zwei werden zunächst in den Trie eingefügt (siehe Abbildung 3.14).

Anhand des Anfragegraphen können zwei der in L_2 enthaltenen Sequenzen verlängert werden. Man erhält die Pfade $\langle q_a, q_b, q_c \rangle$ und $\langle q_a, q_c, q_b \rangle$. Da $\langle q_c, q_b \rangle$ nicht im Trie als Schlüssel vorhanden ist, erfüllt $\langle q_a, q_c, q_b \rangle$ das Subpfadkriterium nicht. Somit gilt

$$C_3 = \{ \langle q_a, q_b, q_c \rangle \}.$$

Die temporär in den Trie aufgenommene Sequenz aus L_2^* wird aus diesem entfernt. Damit resultiert nach Runde zwei der in Abbildung 3.15 gezeigte Trie.

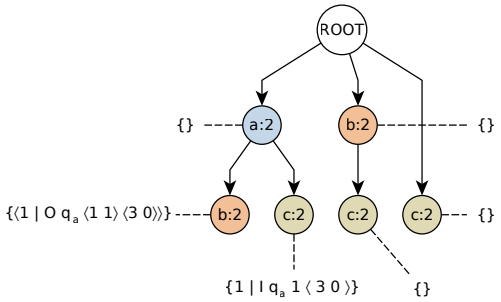


Abbildung 3.14: Trie nach dem Einfügen der häufigen Subsequenzen mit Länge zwei

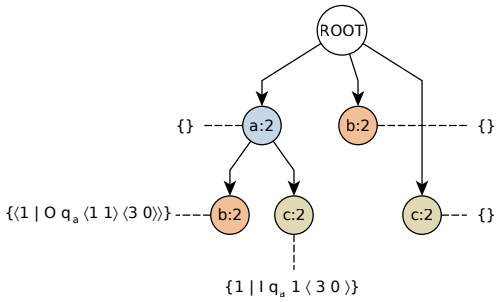


Abbildung 3.15: Trie aus Runde zwei von Beispiel 3.7 nach dem Löschen von Sequenzen aus L_2^*

Runde 3

Es werden Kandidatenpfade der Länge drei aus C_3 betrachtet.

Der Support von $\langle q_a, q_b, q_c \rangle$ ist drei. Analog zur Verarbeitung der Subsequenzen der Länge zwei werden anhand der Trainingssequenzen Parameterkorrelationen für q_c bestimmt. Man findet eine verlässliche Eingabekorrelation des Parameters von q_c mit dem ersten Parameter von q_a. Damit gilt

$$C_{a \to b \to c} = \{\langle 1 \mid I \ q_a \ 1 \ \langle 3 \ 0 \rangle \rangle\}$$

sowie

$$L_3 = \{\langle q_a, q_b, q_c \rangle\}.$$

Der resultierende Trie nach Runde drei ist in Abbildung 3.16 zu sehen.

Da der Templategraph nur drei Knoten besitzt, können keine Kandidaten der Länge vier generiert werden. Somit ist das Verfahren abgeschlossen und Abbildung 3.16 zeigt die endgültige Prefetchingstruktur des fortlaufenden Anwendungsbeispiels. □

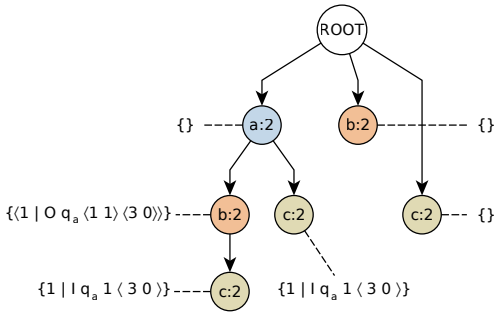


Abbildung 3.16: Prefetchingstruktur des fortlaufenden Anwendungsbeispiels für den WM_o-Algorithmus für SQL-Anfragen

3.6 Vorhersage von SQL-Anfragen

Bisher wurde in diesem Kapitel beschrieben, welche Änderungen am WM_o-Algorithmus vonnöten sind, um Anfragemuster bestehend aus parametrisierten SQL-Anfragen anstelle nicht-parametrisierter HTTP-Anfragen zu erlernen und dabei alle notwendigen Informationen zu sammeln, um anschließend SQL-Anfragen inklusive ihrer Parameterwerte vorhersagen zu können. Dieser Abschnitt beschäftigt sich mit der Vorhersage von SQL-Anfragen auf Basis bisher aufgetretener Anfragen und einer Prefetchingstruktur, wie sie im vorherigen Abschnitt 3.5 gewonnen wurde.

Gegeben sei ein Trie t, der mit Hilfe des im vorherigen Abschnitt beschriebenen Verfahrens aus einer Menge von Trainingssequenzen gewonnen wurde. Um mit Hilfe dieses Tries die nächste SQL-Anfrage der zugehörigen Datenbankanwendung zu prognostizieren, wird ein Fenster w benötigt, dass die zuletzt durch die Anwendung gestellten Anfragen in der Reihenfolge ihres Auftretens enthält. Zu jeder Anfrage muss auch deren Ergebnis bekannt sein, wobei analog zum Erlernen der Trainingssequenzen jedes Ergebnis, dessen Größe in Anzahl Tupel über einem konfigurierbaren Schwellwert liegt, durch eine leere Liste ersetzt wird. Dabei wird sinnvollerweise derselbe Schwellwert wie während der Lernphase gewählt. Eine Ausnahme stellt das Ergebnis der zuletzt gestellten Anfrage dar. Dieses ist, sofern es sich nicht im Cache befindet, zum Zeitpunkt der Vorhersage unbekannt. Sobald das Ergebnis bekannt ist, muss es in w hinzugefügt werden. Dies muss erfolgen, bevor die nächste Vorhersage gemacht wird. Die Länge des Fensters sollte der maximalen Tiefe des Tries minus eins entsprechen. Eine größere Länge bewirkt lediglich einen größeren Aufwand zur Gewinnung einer Vorhersage. Eine Verkürzung des Fensters unter den empfohlenen Wert senkt den Aufwand zur Vorhersage von Anfragen, dies sollte allerdings nur erfolgen, falls bekannt ist, dass sich hieraus keine negativen Auswirkungen auf den Nutzen von WMoCache ergeben.

Neben dem Setzen von Parameterwerten vorhergesagter Prepared Statements anhand von Parameterkorrelationen und der zuletzt aufgetretenen Anfragen und Anfrageergebnisse, sind weitere Anpassungen am in Abschnitt 2.1.3 beschriebenen Verfahren zum Prefetching mit Hilfe des durch den WM_o-Algorithmus generierten Tries erforderlich: Bereits in Abschnitt 3.5 wurde erkannt, dass Ausgabekorrelationen mitunter Tupel aus Anfrageergebnissen über deren Zeilennummer referenzieren, die im Ergebnis des letzten Vorkommens des referenzierten Prepared Statements nicht vorkommen. Wird nun das aktuelle Fenster w, bzw. die Liste w_p der Prepared State-

ments der darin enthaltenen Anfragen, (evtl. nach Verkürzung) erfolgreich in t als Präfix eines Pfades ab dem Wurzelknoten gefunden, so ist es möglich, dass ein Prefetching der Folgeanfrage mit dem höchsten Support aufgrund einer aktuell nicht anwendbaren Ausgabekorrelation unmöglich ist. Einen Sonderfall stellt eine Ausgabekorrelation mit dem Ergebnis der zuletzt gestellten Anfrage q_{recent} dar, sofern deren Ergebnis bisher unbekannt ist. In diesem Fall können beide Anfragen mit Hilfe der Lateral-Outer-Union-Strategie (siehe Abschnitt 2.3 bzw. 3.7 zur konkreten Anpassung für WMoCache) so in einer Anfrage kombiniert werden, dass bei der Auswertung der vorhergesagten Anfrage auf das Ergebnis von q_{recent} zugegriffen werden kann und so bisher unbekannte Parameterwerte verfügbar werden. Anstatt andernfalls keine Vorhersage zu machen, wird die Folgeanfrage mit dem nächsthöheren Support als Kandidat für Prefetching in Betracht gezogen. Ist keines der im Trie auf das Präfix w_p direkt folgenden Prepared Statements für Prefching geeignet, so könnte w_p (weiter) verkürzt werden und versucht werden, ob ein Prefetching auf Basis der nun erhaltenen Prepared Statements und Parameterkorrelationen möglich ist. Diese Vorhersage würde jedoch auf geringerem Wissen als die vorherige beruhen, womit die Wahrscheinlichkeit, dass diese unzutreffend ist, höher wird. Man riskiert dadurch eine niedrigere Präzision des Prefetchers und einen unnötig erhöhten Ressourcenverbrauch. Aus diesem Grund wird an dieser Stelle keine Vorhersage gemacht.

Stehen für die Bestimmung eines Parameterwertes mehrere Parameterkorrelationen zur Wahl, so wird die Korrelation bevorzugt, welche den geringsten Aufwand verursacht und die geringste Fehlerrate besitzt. Der Aufwand von Korrelationen wird in Anlehnung an die Anwendung von Parameterkorrelationen in [4] (vgl. Abschnitt 2.2) eingeschätzt. Korrelationen mit konstanten Werten werden gegenüber Eingabekorrelationen bevorzugt. Letztere stehen wiederum über Ausgabekorrelationen, die sofort anwendbar sind, da das referenzierte Anfrageergebnis verfügbar ist. Als letzter Ausweg dienen Ausgabekorrelationen mit der zuletzt durch die Anwendung gestellten Anfrage, wenn deren Ergebnis unbekannt ist. In diesem Fall muss eine kombinierte Anfrage verwendet werden, wodurch asynchrones Prefetching (siehe Abschnitt 3.8) unmöglich wird.

Algorithmus 3.10 setzt das beschriebene Vorgehen um.

Der Algorithmus erhält als Eingabe die zuletzt durch die Anwendung gestellte Anfrage q. Sofern bekannt, wird mit q auch deren Ergebnis übergeben. q wird in das Fenster w der zuletzt gestellten Anfragen eingefügt. Falls das Fenster voll ist, wird hierdurch die älteste darin gespeicherte Anfrage verdrängt. Anschließend wird aus w die Liste w_p der Prepared Statements der darin enthaltenen Anfragen erstellt. Solange es sich bei w_p nicht um ein echtes Präfix eines Pfades ab der Wurzel des Tries t handelt, wird das älteste Statement aus w_p entfernt. Kommt w_p als Präfix in t vor, so werden alle Kindknoten des durch w_p adressierten Knotens in t in die Menge $prefCandidates$ möglicher Prefetchingkandidaten übernommen. $prefCandidates$ kann als eine Menge von Tripeln bestehend aus je einem Prepared Statement ps, dem Support der in dem Knoten endenden häufigen Subsequenz und der Menge $corrs$ der dem Prepared Statement an dieser Stelle zugeordneten Parameterkorrelationen gesehen werden. Von dieser Menge werden jene Tripel abgesondert, bei welchen für jeden Parameter von ps mindestens eine anwendbare Parameterkorrelation in $corrs$ vorhanden ist. Diese Tripel werden in der Menge $feasiblePrefCandidates$ gespei-

Algorithmus 3.10: Vorhersage einer SQL-Anfrage

Data : Trie t,
Queue w der festen Länge `maxDepth`(t) mit den zuletzt gestellten Anfragen und deren Ergebnissen
input : Die zuletzt gestellte Anfrage q. Über q.result kann – sofern dieses bekannt ist – auf das Ergebnis von q zugegriffen werden.
output : Ein Tupel bestehend aus einer vorhergesagten Anfrage q und einer Menge von Ausgabekorrelationen, die nicht direkt angewandt werden konnten.

```
1  begin
2      // Füge die neue Anfrage in das Fenster ein
3      w ← offer(w, q)
4      Wandle w in eine Liste von Prepared Statements um
5      w_p ← mapPrepare(w)
6      // Suche Prefetchingkandidaten im Trie
7      repeat
8          // Menge von Tripeln (Prepared Statement, Support, corrs)
9          prefCandidates ← getChildren(t, w_p)
10         // Lösche die älteste Anfrage
11         w_p ← tail(w_p)
12     until prefCandidates ≠ ∅ or w_p = ⟨⟩
13     feasiblePrefCandidates ← {(ps, support, corrs) ∈ prefCandidates | ∀p ∈
       ps.params : (∃c ∈ corrs : (c ist anwendbar ∧ c.param = p))}
14     if feasiblePrefCandidates = ∅ then
15         // Kein Prefetching möglich
16         return null
17     end
18     else
19         (ps_max, support_max, corrs_max) ← Element aus feasiblePrefCandidates mit
           max. Support
20         C ← {c ∈ corrs_max | c ist anwendbar}
21         (q_result, leftOverCorrs) ← applyCorrs(ps_max, C) // Algorithmus 3.11
22         return (q_result, leftOverCorrs)
23     end
24 end
```

chert. Falls $feasiblePrefCandidates$ leer ist, findet kein Prefetching statt. Andernfalls wird das Tripel $(ps_{max}, support_{max}, corrs_{max})$ mit dem höchsten Support aus $feasiblePrefCandidates$ ausgewählt. Mit Hilfe der Funktion *applyCorrs* werden die Parameter von ps_{max} anhand der anwendbaren Korrelationen in $corrs_{max}$ gesetzt. Falls das Ergebnis von q noch unbekannt ist, werden Ausgabekorrelationen mit diesem, die zum Setzen von Parametern ausgewertet werden müssten, in der Menge $leftOverCorrs$ gesammelt. Diese wird zusammen mit der aus dem Setzen der Parameter resultierenden Anfrage q_{result} zurückgegeben.

Die Funktion *applyCorrs* implementiert Algorithmus 3.11.

Algorithmus 3.11: Anwendung von Parameterkorrelationen

Data : Queue w der festen Länge `maxDepth`(t) mit den zuletzt gestellten Anfragen und deren Ergebnissen
input : Prepared Statement ps, Menge von Parameterkorrelationen corrs
output : Tupel aus Anfrage q_{result} auf Basis von ps mit (teilweise) gesetzten Parametern und einer Menge von Ausgabekorrelationen für die nicht gesetzten Parameter

```
1  begin
2      // alternativlose nicht direkt anwendbare Ausgabekorrelationen
3      leftOverCorrs ← ∅
4      // resultierende Anfrage
5      q_result ← ps
6      foreach p_j ∈ ps.params do
7          // Parameterkorrelationen für p_j
8          C_j ← [c ∈ corrs | c.param = p_j]
9          Sortiere C_j aufsteigend nach Aufwand und aufsteigend nach Fehlerrate
10         if C_j enthält nur Ausgabekorrelationen mit w[size(w)] and
           w[size(w)].result = null then
11             leftOverCorrs ← leftOverCorrs ∪ {head(C_j)}
12         end
13         else
14             q_result ← apply(q_result, p_j, head(C_j), w)
15         end
16     end
17     return (q, leftOverCorrs)
18 end
```

Der Algorithmus erhält als Eingabe ein Prepared Statement ps und eine Menge dazupassender Parameterkorrelationen $corrs$. Er benötigt Zugriff auf das Fenster w zuletzt gestellter Anfragen. Zu Beginn wird die Menge $leftOverCorrs$ alternativloser nicht direkt anwendbarer Ausgabekorrelationen mit der zuletzt gestellten Anfrage als leere Menge initialisiert. Nun wird über die Parameter von ps iteriert. Aus $corrs$ werden die zum aktuell betrachteten Parameter p_j gehörigen Korrelationen separiert und in die Liste C_j eingefügt. C_j wird zuerst aufsteigend nach Aufwand und anschließend aufsteigend nach Fehlerrate $(\frac{f}{f+h})$ sortiert. Falls in C_j nur Ausgabekorrelationen mit der zuletzt in w eingefügten Anfrage vorkommen und deren Ergebnis

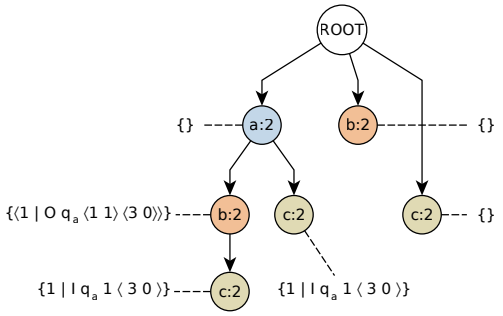


Abbildung 3.17: Prefetchingstruktur des fortlaufenden Anwendungsbeispiels für den WM_o-Algorithmus für SQL-Anfragen

unbekannt ist, wird die erste in C_j enthaltene Korrelation in $leftOverCorrs$ eingefügt. Diese Wahl ist beliebig. Andernfalls wird p_j entsprechend der Korrelation an vorderster Position von C_j gesetzt. Das Ergebnis des Parametersetzens ist die Anfrage q_{result}. Als Resultat gibt der Algorithmus das Tupel $(q_{result}, leftOverCorrs)$ zurück.

Beispiel 3.8 illustriert die Vorhersage einer SQL-Anfrage auf Basis der in Beispiel 3.7 generierten Prefetchingstruktur.

Beispiel 3.8. Gegeben sei der in Beispiel 3.7 generierte Trie (siehe Abbildung 3.17). Der Trie besitzt die maximale Tiefe drei. Somit wird das Fenster w der zuletzt gestellten Anfragen auf die Länge zwei dimensioniert. Es enthalte aktuell die in Abbildung 3.18 gezeigten Anfragen.

$$w = \left\langle \frac{q_b(42)}{\langle (6), (7) \rangle}, \frac{q_c('Bob')}{\langle (8,9), (0,4), (2,3) \rangle} \right\rangle$$

Abbildung 3.18: Ausgangszustand des Fensters w der zuletzt gestellten Anfragen

Als nächstes werde durch die Anwendung die Anfrage $q_a('Alice', 21)$ gestellt. Deren Ergebnis liege nicht im Cache. Somit wird die Anfrage zunächst ohne Ergebnis in w eingefügt. Dies führt zur Verdrängung der ältesten Anfrage $q_b(42)$ aus dem Fenster. Abbildung 3.19 zeigt den neuen Zustand des Fensters.

$$w = \left\langle \frac{q_c('Bob')}{\langle (8,9), (0,4), (2,3) \rangle}, \frac{q_a('Alice', 21)}{\langle -- \rangle} \right\rangle$$

Abbildung 3.19: Zustand des Fensters w nach dem Einfügen von $q_b(42)$. Das Ergebnis der Anfrage ist bisher unbekannt.

Zur Vorhersage der nächsten Anfrage wird im Trie ein Pfad ab der Wurzel mit Präfix $w_p = \langle q_c, q_a \rangle$ gesucht. Da ein solcher Pfad nicht existiert, wird das älteste Prepared Statement aus w_p entfernt. Das Präfix $\langle q_a \rangle$ existiert im Trie. Mögliche Folgeanfragen sind q_b und q_c. Beide sind für Prefetching geeignet, da ihre Parameterkorrelationen anwendbar sind. Im Falle von q_b wird der Wert des Parameters über eine Ausgabekorrelation mit dem Ergebnis des letzten Auftretens von q_a bestimmt. Dieses Ergebnis ist, da die Anfrage $q_a('Alice', 21)$ noch nicht ausgeführt wurde, bisher unbekannt. Um nicht bis zum Eintreffen des Ergebnisses von q_a warten zu müssen,

müssten $q_a('Alice', 21)$ und q_b mit Hilfe der Lateral-Outer-Union-Strategie kombiniert ausgeführt werden. q_c besitzt ebenfalls einen Parameter. Dessen Wert wird über eine Eingabekorrelation mit dem ersten Parameter von q_a bestimmt. Somit ist der vermutliche Wert des Parameters bereits bekannt. Da der Support der häufigen Subsequenzen $\langle q_a, q_b \rangle$ und $\langle q_a, q_c \rangle$ gleich ist, sei willkürlich die Entscheidung für das Prefetching von q_c getroffen worden. Anhand der Eingabekorrelation wird erkannt, dass der Parameter von q_c auf den Wert $'Alice'$ gesetzt werden muss. Da alle Parameter von q_c sofort gesetzt werden konnten, ist die Menge der nicht direkt anwendbaren Ausgabekorrelationen leer. Als Prefetchingvorschlag wird damit das Tupel $(q_c('Alice'), \emptyset)$ ausgegeben. □

3.7 Lateral-Outer-Union

Im vorherigen Abschnitt wurde erkannt, dass WMoCache in der Lage sein sollte, zwei beliebige SQL-Anfragen q_a und q_b derart miteinander zu kombinieren, dass bei der Auswertung von q_b auf das Ergebnis von q_a zugegriffen werden kann, um Ausgabekorrelationen zwischen q_a und q_b auflösen zu können, ohne dass zuvor das Ergebnis von q_a bekannt sein muss. Hierfür bietet sich prinzipiell die in Abschnitt 2.3 beschriebene Lateral-Outer-Union-Strategie an. Diese bringt drei Nachteile mit sich. Der größte Nachteil besteht in der Beschränkung der Größe des Ergebnisses von q_a auf ein Tupel. Des Weiteren wird die Sortierung der Tupel in den ursprünglichen Ergebnissen von q_a und q_b nicht explizit in Form einer zusätzlichen Spalte `row_num` des kombinierten Anfrageergebnisses wiedergegeben. Dies ist jedoch unerlässlich für die Umsetzung von Ausgabekorrelationen, die ein beliebiges Tupel eines Anfrageergebnisses anhand von dessen Zeilennummer referenzieren können. Schlussendlich ist die geringe Verbreitung des Schlüsselwortes `LATERAL` ein Problem [2]. Bowman et al. erkannten im Jahre 2005, dass lediglich eines von drei kommerziellen Datenbanksystemen, die sie in ihrer Studie verwendeten, das `LATERAL` Konstrukt direkt unterstützt. Eine aktuelle eigene Untersuchung ergibt, dass IBM DB2 [12] in Version 9.7 in der Lage ist, die in Listing 2.2 gezeigte Anfrage auszuwerten, jedoch nicht jene in Listing 2.6. PostgreSQL [13] unterstützt seit der zum Zeitpunkt der Erstellung dieser Arbeit neuesten Version 9.3 das Schlüsselwort `LATERAL` [14] und ist auch fähig die zweite Anfrage mit geringfügigen Anpassungen auszuwerten. HSQLDB in Version 2.3 unterstützt `LATERAL` Joins ebenfalls [15]. Microsoft SQL Server [16] kennt anstelle von `LATERAL` das Schlüsselwort `APPLY` [17]. Hiermit kann Anfrage 2.2 leicht umschrieben werden. Um eine größere Bandbreite an Datenbanksystemen und insbesondere auch ältere Versionen von Datenbanksystemen zu unterstützen, nutzt WMoCache eine durch Bowman et al. in [2] skizzierte durch [32] motivierte Umschreibung von `LATERAL` mittels temporärer Views.

[12] `http://www-01.ibm.com/software/data/db2/` – zuletzt überprüft am 25.02.2014
[13] `http://www.postgresql.org/` – zuletzt überprüft am 25.02.2014
[14] `https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.3#LATERAL_JOIN` – zuletzt überprüft am 25.02.2014
[15] `http://hsqldb.org/doc/guide/guide.pdf` – zuletzt überprüft am 25.02.2014
[16] `http://www.microsoft.com/en-us/sqlserver/` – zuletzt überprüft am 27.02.2014
[17] `http://technet.microsoft.com/en-us/library/ms175156%28v=sql.105%29.aspx` – zuletzt überprüft am 27.02.2014

Listing 3.2 zeigt das Schema zur Kombination zweier Anfragen q_a und q_b mit Hilfe der verallgemeinerten Lateral-Outer-Union-Strategie.

```
1  WITH q_a (query_id, row_num, c0, c1, ...) AS (
2      SELECT 1 AS query_id, ROW_NUMBER() OVER (<q_a.
           order_by>) AS row_num, <q_a.select>
3      <q_a.from>
4      <q_a.where>
5      <q_a.limit>
6  ), q_b (query_id, row_num, c0, c1, ...) AS (
7      SELECT 2 AS query_id, ROW_NUMBER() OVER (<q_b.
           order_by>) AS row_num, <q_b.select>
8      <q_b.from>
9      <q_b.where>
10     <q_b.limit>
11 )
12 SELECT *
13 FROM
14     (
15         SELECT query_id, row_num, <q_a.select>,
16             <Nulls(q_b.columns)>
17         FROM q_a
18         UNION
19         SELECT query_id, row_num, <Nulls(q_a.columns)>,
20             <q_b.select>
21         FROM q_b
22      )
23 ORDER BY query_id, row_num
```

Listing 3.2: Schema zur Kombination zweier Anfragen mit der verallgemeinerten Lateral-Outer-Union-Strategie

Über den Alias `q_a` kann aus `<q_b.sql>` auf das Ergebnis von q_a zugegriffen werden. Der Aufruf der Funktion `ROW_NUMBER() OVER(<order_by>)` gibt jeweils die ursprüngliche Position im Ergebnis der zugehörigen Anfrage q_a oder q_b zurück. Auf diese Weise können die Tupel bei der Wiederherstellung der einzelnen Anfrageergebnisse zur Rekonstruktion der ursprünglichen Ordnung der Tupel nach der Spalte `row_num` sortiert werden. Die Spalte `query_id` ordnet ein Tupel entweder dem Ergebnis von q_a oder jenem von q_b zu. Die Sortierung der Tupel nach `query_id` und `row_num` erleichtert das Auslesen des kombinierten Ergebnisses.

Sei `op` ein Vergleichsoperator. Zur Umsetzung einer Ausgabekorrelation $\langle i \mid O\ q_a\ \langle r\ c\rangle\ \langle h\ f\rangle\rangle$ in einem Vergleich `<left.sql> op ?` wird dieser folgendermaßen umschrieben: `<left.sql> op (SELECT cc FROM q_a WHERE row_num = r)`. Vorkommen von Parametern an anderen Stellen werden analog ersetzt.

Auch die verallgemeinerte Umschreibung wird nicht durch alle Datenbanksysteme unterstützt. IBM DB2 9.7 und PostgreSQL 8.4.4 können die umschriebene Anfrage direkt ausführen. Der Microsoft SQL Server fordert bei jeder Verwendung von

`ROW_NUMBER` die Angabe eines `ORDER BY` als Parameter der Fensterfunktion (`OVER`) [18]. Dies ist problematisch, falls eine der Anfragen kein `ORDER BY` besitzt. Ein möglicher Ausweg ist die Angabe `ORDER BY (SELECT NULL)`. Hierdurch wird eine zufällige Sortierung erreicht. Weitere Anpassungen der Anfrage sind notwendig. HSQLDB in Version 2.3.2 kennt die Funktion `ROW_NUMBER()`, allerdings werden Fensterfunktionen nicht unterstützt [19]. Dasselbe gilt für H2 [20] [21]. Darüber hinaus unterstützt H2 keine temporären Views. MySQL [22] besitzt keine Funktion zur Bestimmung der Zeilennummer eines Tupels. Eine Umschreibung mittels benutzerdefinierter Variablen [23] ist möglich.

In Beispiel 3.9 werden zwei Anfragen mit Hilfe der verallgemeinerten Later-Outer-Union-Strategie kombiniert.

Beispiel 3.9. Gegeben seien die in Listing 3.3 und 3.4 gezeigten Anfragen q_a und q_b.

```
1 SELECT p.persnr,
2     p.vorname, p.name
3 FROM professoren p
4 WHERE
5     p.name = 'Sokrates'
6     OR p.name = 'Curie'
```

Listing 3.3: Eine SQL-Anfrage q_a mit gesetztem Parameter

```
1 SELECT v.vorlnr, v.titel
2 FROM
3     vorlesungen v
4 WHERE
5     v.gelesenvon = ?
6 ORDER BY v.titel
```

Listing 3.4: Eine SQL-Anfrage q_b mit nicht gesetzten Parametern

Zur Vorhersage des Wertes des Parameters von q_b sei die Ausgabekorrelation $\langle 1 \mid O\ q_a\ \langle 1\ 0\rangle\ \langle h\ f\rangle\rangle$ gegeben. Listing 3.5 zeigt die aus q_a, q_b und der Ausgabekorrelation konstruierte kombinierte Anfrage. Tabelle 3.1 enthält das Ergebnis der kombinierten Anfrage. □

[18] `http://technet.microsoft.com/en-us/library/ms186734.aspx` – zuletzt überprüft am 27.02.2014

[19] `http://hsqldb.org/doc/guide/builtinfunctions-chapt.html` – zuletzt überprüft am 27.02.2014

[20] `http://www.h2database.com/html/main.html` – zuletzt überprüft am 27.02.2014

[21] `http://www.h2database.com/html/functions.html#rownum` – zuletzt überprüft am 27.02.2014

[22] `https://www.mysql.com` – zuletzt überprüft am 05.03.2014

[23] `https://dev.mysql.com/doc/refman/5.6/en/user-variables.html` – zuletzt überprüft am 05.03.2014

```
WITH q_a(QUERY_ID, ROW_NUM, c0, c1, c2) AS (
        SELECT 1 AS QUERY_ID,
            ROW_NUMBER() OVER () AS ROW_NUM,
            P.PERSNR, P.VORNAME, P.NAME
        FROM PROFESSOREN P
        WHERE P.NAME = 'Sokrates'
            OR P.NAME = 'Curie'
    ), q_b(QUERY_ID, ROW_NUM, c0, c1) AS (
        SELECT 2 AS QUERY_ID, ROW_NUMBER()
            OVER (ORDER BY V.TITEL) AS ROW_NUM,
            V.VORLNR, V.TITEL
        FROM VORLESUNGEN V
        WHERE V.GELESENVON
            = (SELECT c0 FROM q_a WHERE ROW_NUM = 1)
        ORDER BY V.TITEL
    )
SELECT *
FROM (
    SELECT QUERY_ID, ROW_NUM, c0, c1, c2, NULL, NULL
    FROM q_a

    UNION

    SELECT QUERY_ID, ROW_NUM, NULL, NULL, NULL, c0, c1
    FROM q_b
) as unified
ORDER BY QUERY_ID, ROW_NUM
```

Listing 3.5: Kombination der Anfragen q_a und q_b aus Listings 3.3 und 3.4

query_id	row_num	c0	c1	c2	6	7
1	1	2125	Philosoph	Sokrates	NULL	NULL
1	2	2136	Marie	Curie	NULL	NULL
2	1	NULL	NULL	NULL	5041	Ethik
2	2	NULL	NULL	NULL	4052	Logik
2	3	NULL	NULL	NULL	5049	Mäeutik

Tabelle 3.1: Ergebnis der Anfrage aus Listing 3.5. Das Ergebnis von q_a ist rot hinterlegt. Jenes von q_b ist blau markiert.

3.8 Integration in das Systemmodell

Dieser Abschnitt beschreibt, wie der WM_o-Algorithmus für SQL in das zu Beginn des Kapitels beschriebene Systemmodell integriert werden kann. In Anlehnung an das in Abschnitt 2.4 beschriebene Verfahren PECache wird der Cache, bzw. im Falle von WMoCache die lokale Cachedatenbank, logisch in einen Prefetchingcache (*Waiting Room*) und einen Anfragecache (*Weighing Room*) unterteilt. Die Größe der Caches wird jeweils durch die Anzahl der Anfrageergebnisse beschränkt, welche darin gespeichert werden können. Unter Umständen kann es sinnvoll sein, zusätzlich die Anzahl der im Cache befindlichen Tupel zu beschränken. Der Waiting Room verwendet als Verdrängungsstrategie FIFO. Der Weighing Room verwendet LRU. Insbesondere die Entscheidung, wann Prefechting durchzuführen ist, weicht von PECache ab. PECache macht dies nur dann, wenn sich das Ergebnis der zuletzt gestellten Anfrage in keinem der beiden Caches befindet. WMoCache führt immer Prefetching durch. Falls eine vorhergesagte Anfrage q_{pred} und die zuletzt gestellte Anfrage q nicht kombiniert ausgeführt werden müssen, können das Prefetching und das anschließende Einfügen des vorzeitig abgerufenen Ergebnisses in den Waiting Room nebenläufig erfolgen. Hierzu öffnet WMoCache eine zweite Datenbankverbindung, die für nebenläufiges Prefetching reserviert ist. Das Prefetching kann dann parallel zum Abrufen von q und der Verarbeitung des Ergebnisses der Anfrage durch die Datenbankanwendung geschehen. Ebenso können Aufgaben, wie das Einfügen einer Anfrage in den Cache, das Hinzufügen einer Anfrage zu den Trainingssequenzen, das Generieren der Prefetchingstruktur und das nachträgliche Informieren des Vorhersagemoduls über ein Anfrageergebnis, nebenläufig durchgeführt werden. Zur sicheren Nutzung der Parallelität ist das Setzen von Sperren auf den jeweils benötigten Komponenten erforderlich.

Algorithmus 3.12 beschreibt den Ablauf der Beantwortung einer Anfrage q während der Lernphase.

Zunächst wird getestet, ob sich das Ergebnis von q in einem der Caches befindet. Während der Lernphase kommt hierfür nur der Weighing Room in Frage. In diesem Fall wird das Ergebnis der Anfrage aus dem Cache geholt. Andernfalls muss es von der Datenbank abgerufen werden und anschließend in den Weighing Room eingefügt werden. Danach wird q zusammen mit ihrem Ergebnis zu den Trainingssequenzen hinzugefügt. Hierfür wird Algorithmus 3.1 verwendet. Es wird gezählt, wie viele Anfragen bereits erlernt wurden. Der Schwellwert $learningDuration$ beschränkt die Zahl der zu erlernenden Anfragen. Sobald genügend Anfragen erlernt wurden, wird die Funktion $switchToPrefetchingPhase$ aufgerufen. Sie generiert die Prefetchingstruktur mit Hilfe des Algorithmus 3.3 und schaltet WMoCache anschließend in den Prefetchingmodus um.

In der Prefetchingphase wird Algorithmus 3.13 zur Beantwortung von Anfragen verwendet.

Analog zur Lernphase wird für eine durch die Datenbankanwendung gestellte Anfrage q zunächst getestet, ob deren Ergebnis in einem der Caches enthalten ist. Dies geschieht mit Hilfe der Funktion *checkCaches*. Falls q sich im Waiting Room befindet, wird die Anfrage aus diesem entfernt und in den Weighing Room eingefügt. Anschließend wird entsprechend Algorithmus 3.10 eine Vorhersage p über den Nachfolger von q gemacht. Sofern bekannt, wird in diese auch das Ergebnis von q

Algorithmus 3.12: Beantwortung einer Anfrage während der Lernphase

```
Data : learningDuration: Anzahl während der Lernphase zu erlernender
       Anfragen,
  numQueriesLearned: Anzahl bisher erlernter Anfragen
  input  : Anfrage q
  output : Ergebnis der Anfrage q
1 begin
2   cacheHit ← false
3   (result, cacheHit) ← checkCaches(q)
4   if not cacheHit then
5     result ← fetch(q)
6   end
7   learn (q, result) // Algorithmus 3.1
8   ++ numQueriesLearned
9   if numQueriesLearned ≥ learningDuration then
10    switchToPrefetchingPhase () // Algorithmus 3.3
11  end
12  if not cacheHit then
13    weighingRoom ← add(weighingRoom, q, result)
14  end
15  return result
16 end
```

einbezogen. Konnte eine Vorhersage gemacht werden und das Ergebnis von q ist bereits bekannt, so muss lediglich die vorhergesagte Anfrage abgerufen und deren Ergebnis in den Waiting Room eingefügt werden. Dies kann nebenläufig erfolgen. Davor wird mit der Funktion *checkLocal* getestet, ob die vorhergesagte Anfrage $p.q$ bereits in einem der beiden Caches enthalten ist. In jedem Fall wird sie neu in den Waiting Room eingefügt. Falls eine Vorhersage gemacht werden, aber in diese das Ergebnis von q nicht einbezogen werden konnte, da es nach wie vor unbekannt ist, muss getestet werden, ob Ausgabekorrelationen zwischen q und $p.q$ bestehen. Ist dies nicht der Fall, wird das Ergebnis von q abgerufen und parallel dazu, falls $p.q$ noch nicht lokal beantwortet werden kann, das Prefetching von $p.q$ durchgeführt. Andernfalls müssen die beiden Anfragen mittels der Lateral-Outer-Union-Strategie gemeinsam ausgeführt werden. Dabei kann das Ergebnis der vorhergesagten Anfrage nur in den Cache eingefügt werden, wenn die Ausgabekorrelationen tatsächlich in Verbindung mit dem Ergebnis von q anwendbar gewesen sind. Falls keine Vorhersage möglich gewesen ist, muss q noch auf der Datenbank ausgeführt werden, sofern deren Ergebnis nicht lokal verfügbar ist. Falls das Ergebnis von q initial nicht im Cache vorhanden war, muss das Vorhersagemodul über dieses informiert werden und das Ergebnis muss in den Cache eingefügt werden. Bei asynchronem Prefetching wird stets zuerst getestet, ob das Ergebnis von $p.q$ bereits lokal vorhanden ist. In diesem Fall wird es an die Spitze der LRU-Liste des Waiting Rooms gesetzt.

Update-Statements wie `UPDATE`, `DELETE` oder `INSERT` werden direkt an die Datenbank weitergeleitet.

Algorithmus 3.13: Beantwortung von Anfragen während der Prefetchingphase

```
   input  : Anfrage q
   output : Ergebnis der Anfrage q
 1 begin
 2     cacheHit ← false
 3     (result, cacheHit) ← checkCaches(q)
 4     // result kann null sein.
 5     p ← predict(q, result) // Algorithmus 3.10
 6     if p ≠ null then
 7         if cacheHit then
 8             if not checkLocal(p.q) then
 9                 prefetchAsynch(p.q)
10             end
11         end
12         else
13             if p.leftOverCorrs = ∅ then
14                 if not checkLocal(p.q) then
15                     prefetchAsynch(p.q)
16                 end
17                 result ← fetch(q)
18             end
19             else
20                 (result, predResult) ← fetchCombined(q, p)
21                 if verify(result, p.leftOverCorrs) then
22                     applyCorrs(p.q, p.leftOverCorrs, result)
23                     waitingRoom ← add(waitingRoom, p.q, predResult)
24                 end
25             end
26         end
27     end
28     else
29         if not cacheHit then
30             result ← fetch(q)
31         end
32     end
33     if not cacheHit then
34         informPredictor(q, result)
35         weighingRoom ← add(weighingRoom, q, result)
36     end
37     return result
38 end
```

3.9 Fazit

Der erweiterte WM_o-Algorithmus für das Prefetching von SQL-Anfragen fügt dem in Abschnitt 2.1 wiedergegebenen Verfahren von Nanopoulos et al. eine initiale Lernphase hinzu. Diese dient der Gewinnung von Trainingssequenzen und des durch den WM_o-Algorithmus benötigten Graphen. Als passendes Äquivalent zu Dokumenten wurden Äquivalenzklassen strukturell gleicher SQL-Anfragen bzw. die Prepared Statements der auftretenden Anfragen gewählt. Da SQL-Anfragen parametrisiert sind, ist es neben der Vorhersage des Prepared Statements der nächsten Anfrage notwendig, Werte für dessen Parameter vorauszusagen. Hierfür wurde das von Bowman et al. übernommene Konzept des Erlernens von Parameterkorrelationen an WM_o angepasst. Die Verwendung von Parameterkorrelationen bringt die Aufgabe der Eigenschaft von WM_o mit sich, dass jede Subsequenz einer im Trie gespeicherten häufigen Subsequenz der Trainingssequenzen ebenfalls im Trie vorhanden ist und für Prefetching genutzt werden kann. Würde man diese Eigenschaft weiterhin durchgehend fordern, würde dies den Anteil der für Vorhersagen nutzbaren Anfragemuster stark verringern. Eine weitere Modifikation des Lernalgorithmus würde das Erlernen noch allgemeinerer Anfragemuster erlauben. Diese wird im Ausblick der Arbeit beschrieben (siehe Abschnitt 6.1). Es wurde eine Verallgemeinerung der Lateral-Outer-Union-Strategie präsentiert, die durch geschicktes Kombinieren von Anfragen auch dann Prefetching erlaubt, wenn zum Zeitpunkt der Vorhersage einer Anfrage noch nicht alle Informationen zur Bestimmung der Werte ihrer Parameter verfügbar sind. Analog zum Verfahren von Nanopoulos et al. handelt es sich bei dem in diesem Kapitel präsentierten Verfahren um einen Black-Box-Ansatz. Dies bedeutet, dass eine Datenbankanwendung mit einer Implementierung des Verfahrens wie mit einem regulären Datenbanktreiber interagieren kann und Prefetching und Caching für die Anwendung transparent sind.

Implementierung

4 Implementierung

Dieses Kapitel beschreibt die Implementierung von WMoCache auf Basis des theoretischen Ansatzes aus Kapitel 3. Das Kapitel gibt zunächst einen Überblick über WMoCache. Dieser beinhaltet die Verwendung der Implementierung und deren Architektur. Im weiteren Verlauf werden interessante Aspekte der Umsetzung im Detail betrachtet. Näher eingegangen wird unter anderem auf Aspekte wie Nebenläufigkeit, die Repräsentation von Anfrageergebnissen, Caching, die effiziente Überprüfung des Subpfadkriteriums bei der Gewinnung von Kandidatensequenzen und die Implementierung der Lateral-Outer-Union-Strategie.

4.1 Überblick

WMoCache ist eine in der Programmiersprache Java [24] geschriebene Middlewarekomponente entsprechend der in Abschnitt 3.1 beschriebenen Architektur. Sie implementiert die in Kapitel 3 beschriebenen Algorithmen. Neben weiteren Bibliotheken wird unter anderem die Bibliothek IQCache Commons genutzt, die Teil des Projektes IQCache ist und eine Reihe (datenbankspezifischer) Funktionalitäten bietet. WMoCache besitzt ein Interface, das ähnlich zu `java.sql.Connection` [25] ist und dient als Wrapper um eine Instanz von `iqcache.common.sql.DatabaseConnector`, einer Abstraktion von `java.sql.Connection`. Statt direkt über eine `Connection` mit einer Datenbank zu kommunizieren, kommuniziert eine Datenbankanwendung stattdessen mit einer Instanz von WMoCache, um Prefetching und Caching nutzen zu können. Als Wrapper kann WMoCache prinzipiell in Verbindung mit jedem RDBMS verwendet werden, für welches ein JDBC-Treiber verfügbar ist. Einschränkungen entstehen durch die Verwendung temporärer Views und Fensterfunktionen in mittels der Lateral-Outer-Union-Strategie kombinierten Anfragen (vgl. Abschnitt 3.7). Ziel der Implementierung war es, einen Prototypen zu schaffen, dessen Funktionsumfang für die Evaluation des erweiterten WM_o-Algorithmus für SQL-Anfragen hinreichend ist.

4.2 Verwendung

Die Kommunikation einer Datenbankanwendung mit dem Prefetcher erfolgt vollständig über eine Instanz der Klasse `de.uni_passau.ganser.wmocache.WMoCache`. Deren Aufbau ist modular. Der Konstruktor von `WMoCache` benötigt Referenzen auf

[24] `http://www.oracle.com/technetwork/java/index.html` – zuletzt überprüft am 28.02.2014

[25] `http://docs.oracle.com/javase/6/docs/api/java/sql/Connection.html` – zuletzt überprüft am 28.02.2014

die zu verwendenden Module und auf zwei Instanzen von `DatabaseConnector`. Wie bereits in Abschnitt 3.8 beschrieben, dient eine der beiden Datenbankverbindungen dem Ausführen durch die Datenbankanwendung gestellter Anfragen, während die zweite Verbindung für asynchrones Prefetching vorhergesagter Anfragen benutzt wird.

WMoCache kann durch eine große Anzahl an Konfigurationsparametern optimal an eine Datenbankanwendung angepasst werden. Diese werden an die Konstruktoren der jeweiligen Komponenten übergeben. Im Folgenden seien die Konfigurationsparameter von WMoCache zusammen mit ihrer Beschreibung aufgelistet. Neben den aufgezählten Parametern existieren weitere zur Konfiguration der Verbindungen zur Anwendungsdatenbank und zur Cachedatenbank.

- `waitingRoomSize`: Beschränkt die Größe des Prefetchingcaches (Waiting Room). Dessen Fassungsvermögen wird in der Anzahl der enthaltenen Anfrageergebnisse bemessen.
- `weighingRoomSize`: Beschränkt analog zu `waitingRoomSize` die Größe des Anfragecaches (Weighing Room).
- `learningDuration`: Nachdem `learningDuration` viele Anfragen erlernt wurden, erfolgt der Wechsel von der Lernphase in die Prefetchingphase.
- `lmTrainingSeqLength`: Maximale Länge einer Trainingssequenz.
- `lmResultSetMaxSize`: Maximale Größe eines Anfrageergebnisses in Anzahl Tupeln, um in die Trainingssequenzen aufgenommen zu werden.
- `freqPairsGraphBuilderMaxPairDist`: Größe des Fensters, das zur Generierung des Anfragegraphen aus den Trainingssequenzen verwendet wird.
- `freqPairsGraphBuilderMinFreq`: Mindesthäufigkeit eines Paares von Anfragen innerhalb der erlernten Anfragen, um als Kante in den Anfragegraphen aufgenommen zu werden. Der Abstand der beiden Anfragen darf dabei jeweils maximal dem Wert von `freqPairsGraphBuilderMaxPairDist` entsprechen.
- `lmMinSupport`: Minimaler Support einer häufigen Subsequenz.
- `findSubsequencesDistThresholdMaxNumIterations`: Beschränkt die Anzahl der Iterationen von Algorithmus 3.6 über eine Trainingssequenz.
- `maxSubsequencePruningStrategyThreshold`: Bei der Bestimmung von Subsequenzen einer Trainingssequenz, welche einem Kandidatenpfad entsprechen, sind nur Subsequenzen von Interesse, bei welchen in der Subsequenz benachbarte Anfragen in der Trainingssequenz einen maximalen Abstand `maxSubsequencePruningStrategyThreshold` besitzen.
- `lmCorrelationsWindowSize`: Bei der Bestimmung von Parameterkorrelationen für eine Anfrage wird innerhalb einer Subsequenz maximal `lmCorrelationsWindowSize` weit zurückgegangen.
- `lmMaxCorrFailRatio`: Maximal zulässige Fehlerrate einer zuverlässigen Parameterkorrelation.

- `pfWindowSize`: Die Größe des Fensters, in welchem während der Prefetchingphase die zuletzt gestellten Anfragen gespeichert werden. Der Parameter beschränkt damit die Anzahl der Anfragen, welche sich WMoCache merkt und auf welchen Vorhersagen beruhen.
- `pfResultSetMaxSize`: Beschränkt analog zu `lmResultSetMaxSize` die maximale Größe eines Anfrageergebnisses, um während der Prefetchingphase in das Fenster der zuletzt gestellten Anfragen aufgenommen zu werden.

Zwischen einigen der Parameter bestehen starke Abhängigkeiten. So sollte beispielsweise der Wert von `lmMinSupport` auf das Verhältnis zwischen `learningDuration` und `lmTrainingSeqLength` abgestimmt sein, da es die Anzahl der Trainingssequenzen festlegt. Die Länge `lmTrainingSeqLength` der Trainingssequenzen sollte nicht zu kurz gewählt sein, da andernfalls möglicherweise Transaktionen in keiner Trainingssequenz vollständig auftreten. Wünschenswert wäre, dass häufige Transaktionen in allen Trainingssequenzen auftreten und seltene nur in manchen. Wie bereits zu Algorithmus 3.2 beschrieben, beschränkt der Parameter `freqPairsGraphBuilderMaxPairDist` aufgrund des Subpfadkriteriums direkt die maximale Länge auffindbarer häufiger Subsequenzen in den Trainingssequenzen. Es ist daher sinnvoll für `pfWindowSize` den um eins verringerten Wert dieses Parameters zu verwenden. Auch `lmCorrelationsWindowSize` richtet sich nach diesem Wert. Der Wert von `freqPairsGraphBuilderMaxPairDist` selbst sollte der maximalen Anzahl paarweise verschiedener Prepared Statements, die in einer Transaktion der Datenbankanwendung aufeinander folgen können, entsprechen. Auch `lmResultSetMaxSize` und `pfResultSetMaxSize` sollten denselben Wert besitzen.

Die Benutzung von WMoCache beschränkt sich im Wesentlichen auf sechs Methoden der Klasse `WMoCache`:

- `executeQuery(String)` führt eine gegebene Anfrage aus und gibt deren Ergebnis als Instanz der Klasse `QueryResult` zurück.
- `executeUpdate(String)` schickt ein Update-Statement (`UPDATE`, `INSERT`, `DELETE`, etc.) an die Datenbank.
- `setAutoCommit(boolean)` (de-)aktiviert automatische Commits nach jeder Änderung einer Datenbank.
- `commit()` schließt eine Transaktion ab und hat die Übernahme aller Änderungen durch die Datenbank zur Folge.
- `rollback()` bricht eine Transaktion ab, ohne dass deren vorläufige Änderungen Auswirkungen auf die Datenbank haben.
- `close()` terminiert alle internen Threads von WMoCache und leert die Cachedatenbank. Verwendete Ressourcen werden geschlossen. Dies betrifft *nicht* die von außen übergebenen Datenbankverbindungen.

4.3 Repräsentation von Anfragen

Als interne Repräsentation für SQL-Anfragen wurde die Darstellung durch ASTs gewählt. Diese abstrahiert von Formatierungsvarianten des SQL-Codes der Eingabe und ermöglicht eine unkomplizierte Analyse der Struktur einer Anfrage. Bezeichner von Tabellen und Spalten sowie Aliase sind hier in eine kanonische Form überführt. Auch das Erstellen und die Modifikation von Anfragen ist bedeutend leichter als anhand des Quellcodes. Der AST einer Anfrage ist mit einem Metadatenmodell verknüpft, das in Form eines Objektgraphen das Schema der Anwendungsdatenbank wiedergibt. Dadurch kann auch dieses leicht in die Analyse von Anfragen einbezogen werden.

Als AST-Implementierung inklusive des Parsers und des Metadatenmodells verwendet WMoCache die bereits in Abschnitt 2.5 beschriebene Bibliothek IQCache Query.

4.4 Architektur

Der folgende Abschnitt beschreibt die Architektur von WMoCache. Es werden dabei die wichtigsten Klassen und Interfaces, sowie deren Zusammenspiel betrachtet. Die gezeigten UML-Klassendiagramme sind teilweise vereinfacht und konzentrieren sich auf wesentliche Eigenschaften der dargestellten Klassen.

Wie bereits in Abschnitt 4.2 erwähnt, besitzt WMoCache einen modularen Aufbau. Die Hauptklasse `WMoCache` kommuniziert mit den einzelnen Komponenten nur über definierte Interfaces entsprechend des Strategy-Patterns [28]. Auf diese Weise kann `WMoCache` selbst prinzipiell mit jedem Prefetchingalgorithmus für SQL-Anfragen verwendet werden, der sich analog zum WM_o-Algorithmus in die Phasen Lern- und Prefetchingphase gliedert und das Konzept der Parameterkorrelationen entsprechend der Definition zu Beginn von Abschnitt 3.5 verwendet. Auch das Verfahren zur kombinierten Ausführung von SQL-Anfragen ist austauschbar. Abbildung 4.1 zeigt den Aufbau der Klasse `WMoCache`.

Das Interface von `WMoCache` wurde schon im Abschnitt 4.2 beschrieben. Die Methode `executeQuery(String)` implementiert die Algorithmen 3.12 und 3.13.

Für das Parsen von Anfragen verwendet `WMoCache` eine Instanz der Klasse `iqcache.query.parser.QueryParser`, bzw. deren Methode `parseQuery(String)`.

Die `DatabaseConnector`-Instanz `dbc` dient dem Ausführen von der Datenbankanwendung erhaltener Anfragen und Update-Statements sowie kombinierter Anfragen. Die zweite Verbindung `prefetchingCon` wird für asynchrones Prefetching verwendet.

4.4.1 Paket `de.uni_passau.ganser.wmocache.cache`

Die abstrakte Klasse `Cache` beschreibt einen Cache für SQL-Anfragen. Es werden lediglich vollständige Anfrageergebnisse vorgehalten. `Cache` implementiert keine spezifischen Verdrängungsstrategien. Diese können entsprechend des Template-Method-Patterns [28] in Unterklassen von `Cache` implementiert werden. Bisher sind

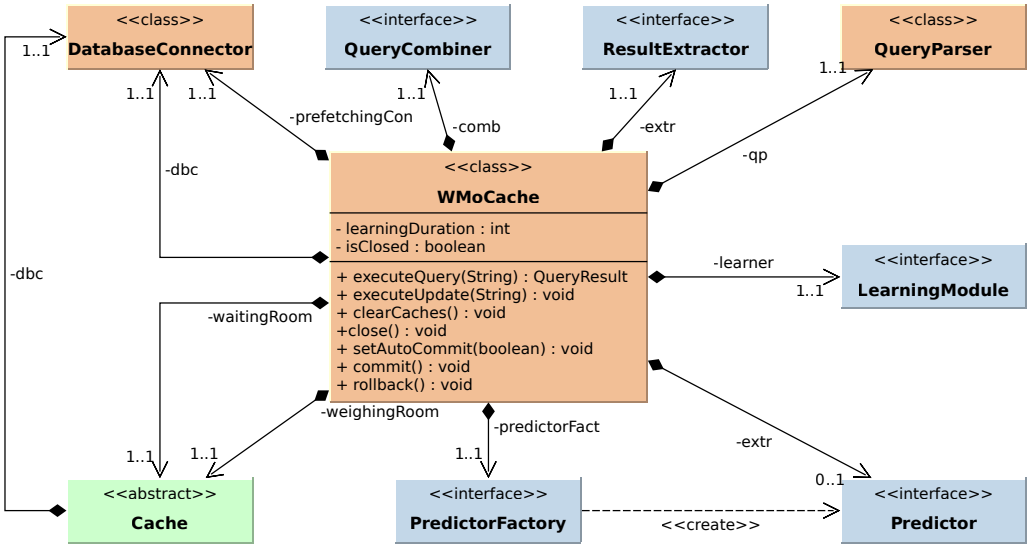


Abbildung 4.1: Komponenten der Klasse `WMoCache`

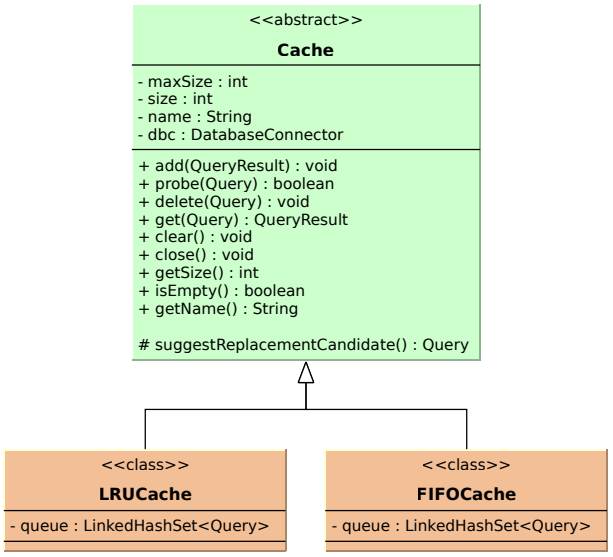


Abbildung 4.2: Die Klasse `Cache` und ihre Unterklassen

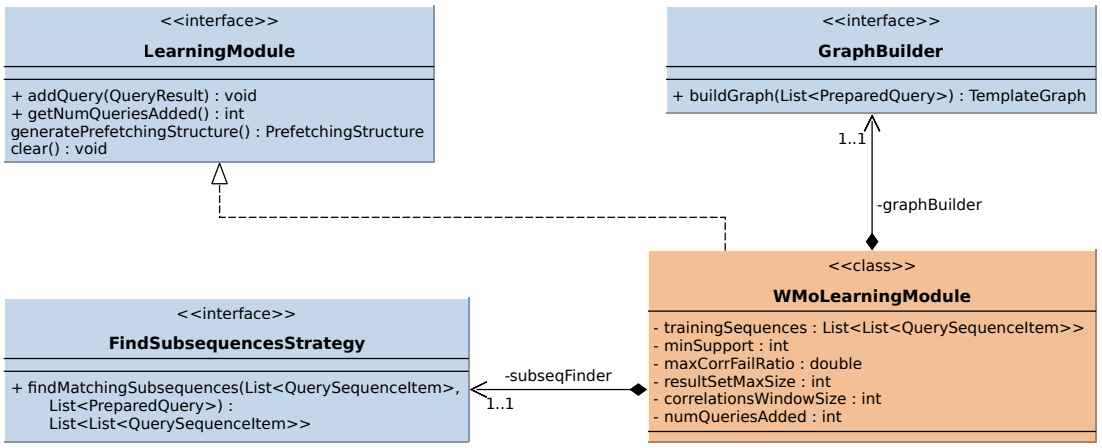


Abbildung 4.3: Das Interface `learning.LearningModule` zusammen mit seiner Implementierung `WMoLearningModule` und den Interfaces der Komponenten dieser Klasse

die beiden Verdrängungsstrategien LRU und FIFO in den Klassen `LRUCache` und `FIFOCache` umgesetzt. Abbildung 4.2 zeigt die Klasse `Cache` zusammen mit ihren derzeitigen Unterklassen. Falls beim Hinzufügen eines Anfrageergebnisses zu einem Cache ein altes Ergebnis aus dem Cache verdrängt werden muss, wird die Methode `suggestReplacementCandidate() : Query` aufgerufen. Diese wird durch die Unterklassen entsprechend der jeweiligen Verdrängungsstrategie implementiert und gibt eine Anfrage zurück, deren Ergebnis aus dem Cache entfernt werden kann. Abschnitt 4.7 betrachtet die am Caching beteiligten Klassen detailliert. Ein Anfrageergebnis wird durch eine Instanz der Klasse `wmocache.query_result.QueryResult` [26] repräsentiert. Dieses besitzt eine Referenz auf die zugehörige `Query`.

`WMoCache` referenziert entsprechend Abschnitt 3.8 zwei `Cache`-Instanzen.

4.4.2 Paket `wmocache.prefetching.learning`

`LearningModule` ist das Interface für jene Komponente, die für das Aufzeichnen von Anfragen während der Lernphase und die Generierung einer Prefetchingstruktur während des Übergangs zur Prefetchingphase verantwortlich ist. Abbildung 4.3 zeigt das Interface zusammen mit seiner Implementierung `wmocache.prefetching.wmo.learning.WMoLearningModule` sowie mit den Interfaces der Komponenten von `WMoLearningModule`. Über die Methode `addQuery(QueryResult) : void` wird das Lernmodul darüber informiert, dass zuletzt die Anfrage mit dem gegebenen Anfrageergebnis aufgetreten ist. Mit der Methode `getNumQueriesAdded() : int` kann abgefragt werden, wie viele Anfragen bereits erlernt wurden. Die Methode `generatePrefetchingStructure() : PrefetchingStructure` generiert aus den erlernten Anfragen die Prefetchingstruktur.

[26] Anmerkung: Das Präfix `de.uni_passau.ganser.wmocache` von Paketnamen wird ab sofort auf das Präfix `wmocache` verkürzt.

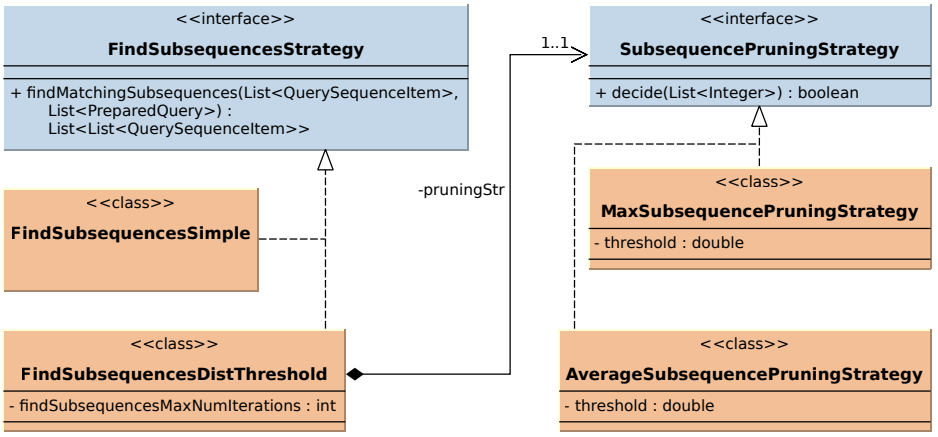


Abbildung 4.4: Klassen zum Auffinden passender Subsequenzen innerhalb von Trainingssequenzen

4.4.3 Paket `wmocache.prefetching.wmo.learning`

`WMoLearningModule` implementiert in der Methode `addQuery` Algorithmus 3.1. Die Methode `generatePrefetchingStructure` setzt Algorithmus 3.3 um. Um die Implementierung flexibel zu gestalten, werden einige Subroutinen mit Hilfe des Strategy-Patterns ausgelagert. So ist das Generieren des Anfragegraphen nach Algorithmus 3.2 in die Klasse `FreqPairsGraphBuilder` ausgelagert, die das Interface `GraphBuilder` implementiert. Auch das Auffinden zu einem Kandidatenpfad passender Subsequenzen innerhalb einer Trainingssequenz ist variabel gehalten. Implementierungen müssen das Interface `find_subseqs.FindSubsequencesStrategy` implementieren. Abbildung 4.4 zeigt die Typhierarchie von `FindSubsequencesStrategy`, sowie von deren Implementierungen benötigte Typen. Es kann aus zwei Implementierungen von `FindSubsequencesStrategy` gewählt werden. `find_subseqs.FindSubsequencesSimple` führt einen linearen Scan durch eine Trainingssequenz aus und sammelt paarweise nicht-überlappenden Subsequenzen auf, die so gefunden werden können. Die Klasse `find_subseqs.FindSubsequencesDistThreshold` implementiert Algorithmus 3.6. Die Pruning-Bedingung für Subsequenzen mit zu großen Abständen benachbarter Anfragen ist in die Strategy `SubsequencePruningStrategy` ausgelagert und somit ebenfalls austauschbar. Derzeit ist ein Pruning wahlweise aufgrund eines zu großen maximalen Abstandes oder eines zu großen durchschnittlichen Abstandes möglich.

4.4.4 Paket `wmocache.prefetching.predictor`

Das Vorhersagemodul von `WMoCache` ist stets vom Typ `Predictor`. Da `Predictor` ein Interface ist und das Vorhersagemodul erst nach der Generierung der Prefetchingstruktur erstellt werden kann, kann einer Instanz von `WMoCache` zum Zeitpunkt des Programmstarts keine spezifische Implementierung von `Predictor` zugewiesen werden. Zur eleganten Instantiierung des Vorhersagemoduls wird das Abstract-Factory-Pattern [33] verwendet. Auf diese Weise muss `WMoCache` weder der Laufzeittyp der Prefetchingstruktur noch jener des Vorhersagemoduls bekannt sein. Factories für `Predictor`-Instanzen implementieren das Interface `PredictorFactory`.

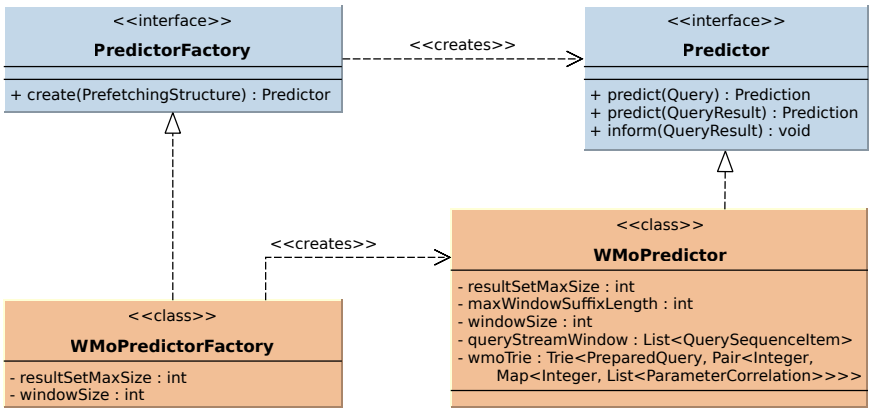


Abbildung 4.5: UML-Klassendiagramm der am Vorhersagemodul von WMoCache beteiligten Klassen und Interfaces

Abbildung 4.5 zeigt die am Vorhersagemodul von WMoCache beteiligten Interfaces, sowie deren Implementierungen. Die Methode `Predictor.predict(QueryResult) : Prediction` dient der Vorhersage einer Anfrage auf Basis einer gegebenen Anfrage und deren Ergebnis. Da das Ergebnis der zuletzt durch die Datenbankanwendung gestellten Anfrage nicht immer bereits lokal verfügbar ist, existiert zusätzlich die Methode `predict(Query) : Prediction`, die allein auf Basis einer Anfrage q eine Vorhersage macht. Wie bereits in Abschnitt 3.6 beschrieben, muss, nachdem letztere Methode verwendet wurde, vor dem nächsten Aufruf einer `predict`-Methode der `Predictor` durch Aufruf der Methode `inform(QueryResult) : void` über das Ergebnis von q informiert werden.

4.4.5 Paket `wmocache.prefetching.wmo.predictor`

Die Klasse `WMoPredictor` implementiert Algorithmus 3.10 zur Vorhersage von SQL-Anfragen entsprechend des erweiterten WM_o-Algorithmus aus Kapitel 3 dieser Arbeit. `WmoPredictorFactory` erzeugt Instanzen von `WMoPredictor` aus Instanzen von `wmocache.prefetching.wmo.pref_structure.WMoPrefetchingStructure`.

4.4.6 Paket `wmocache.query_combine`

Dieses Paket enthält Klassen zum kombinierten Ausführen von Anfragen. Abbildung 4.6 zeigt das UML-Klassendiagramm der darin enthaltenen Typen.

Ein `QueryCombiner` kombiniert zwei Anfragen q_a und q_b miteinander. Dabei können Ausgabekorrelationen von q_a nach q_b aufgelöst werden. Aufgrund der eingeschränkten Ausdrucksfähigkeit des AST gibt die Methode `combine(Query, Query, List<LocalOutputParameterCorrelation>) : String` die kombinierte Anfrage als `String` zurück. Zu jeder Implementierung von `QueryCombiner` muss es eine passende Implementierung von `ResultExtractor` geben, die aus dem Ergebnis einer kombinierten Anfrage die Ergebnisse der einzelnen Anfragen rekonstruiert.

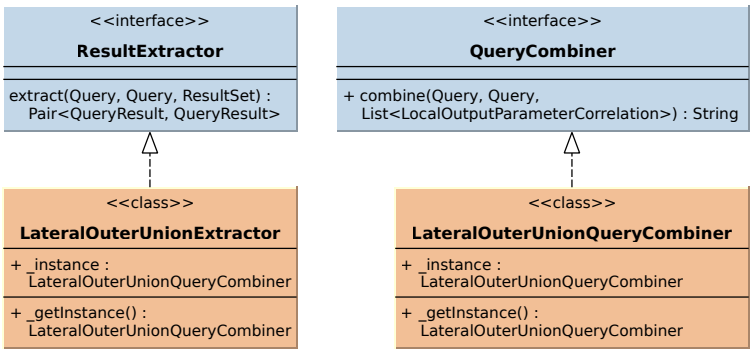


Abbildung 4.6: Klassendiagramm zum Paket `wmocache.query_combine`

4.4.7 Paket `wmocache.query_combine.lateral_outer_union`

Die in diesem Paket enthaltenen Klassen `LateralOuterUnionQueryCombiner` und `LateralOuterUnionExtractor` implementieren die in Abschnitt 3.7 beschriebene verallgemeinerte Lateral-Outer-Union-Strategie. Da Instanzen beider Klassen zustandslos sind, verwenden sie das Singleton-Pattern [33]. Eine genaue Betrachtung der Implementierung erfolgt in Abschnitt 4.12.

4.5 Effizienter Umgang mit Mengen von Anfragen

In WMoCache werden an diversen Stellen Mengen von SQL-Anfragen verwendet. Um effizient testen zu können, ob eine Anfrage in einer Menge enthalten ist, empfiehlt sich die Verwendung von `java.util.HashSet` [27] zur Darstellung von Mengen. Dies erfordert eine Implementierung der Methoden `hashCode()` und `equals(Object)` durch `iqcache.query.Query`, bzw. `iqcache.query.PreparedQuery`, deren Ergebnis jeweils ausschließlich von der Struktur einer Anfrage abhängt. Leider weisen beide Klassen diese nicht auf. Zur Lösung der Problematik wurden die Typen zu `wmocache.util.helper.query.QueryWithHashFunction` sowie `wmocache.util.helper.query.PreparedQueryWithHashFunction` abgeleitet, die diese Eigenschaft besitzen. Vergleiche werden anhand der SQL-Strings der Anfragen durchgeführt. Ebenso werden aus diesen die Hashwerte errechnet. Nahezu alle in WMoCache vorkommenden Instanzen von `(Prepared)Query` sind Instanzen der abgeleiteten Typen. Sie werden jedoch stets über ihre Obertypen aus `iqcache.query` angesprochen, um eine Rückpropagation der Änderungen in IQCache zu erleichtern.

4.6 Anfrageergebnisse

Anfrageergebnisse werden in WMoCache durch die Klasse `QueryResult` modelliert. Diese bietet wahlfreien Zugriff auf alle in einem Ergebnis enthaltenen Werte. Der Umgang mit Anfrageergebnissen stellt die größte Einschränkung des Pro-

[27] `http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html` – zuletzt überprüft am 04.03.2014

totypen WMoCache dar: Bevor ein aus dem Cache oder von der Anwendungsdatenbank eingetroffenes Anfrageergebnis weiterverwendet werden kann, wird es durch WMoCache vollständig ausgelesen und die Werte werden in eine Instanz von `wmocache.query_result.QueryResult` übertragen. Eine effiziente Implementierung würde ein Anfrageergebnis sofort der Datenbankanwendung zur Verfügung stellen und, während diese die Tupel des Ergebnisses abruft, die Werte zeitgleich zur internen Verwendung abgreifen. Nachdem die Anwendung ein Anfrageergebnis geschlossen hätte, würden die bisher nicht abgerufenen Tupel ausgelesen und in den lokalen Cache eingefügt. Ein Nachteil der hier gewählten einfachen Implementierung ist die Unmöglichkeit realistisch die relative Beschleunigung, die eine Datenbankanwendung durch die Verwendung von WMoCache erfährt, zu bestimmen.

4.7 Caching

`wmocache.cache.Cache` verwendet zur Speicherung von Anfrageergebnissen eine (lokale) relationale Datenbank. Die Größe eines Caches ist durch die Anzahl der Anfrageergebnisse, welche darin gespeichert werden können, beschränkt. Die im Cache enthaltenen Anfragen werden nach Prepared Statement gruppiert (vgl. Definition 3.1). Die Anfrageergebnisse einer solchen Gruppe werden jeweils zusammen in einer Tabelle gespeichert. Tabellen für Prepared Statements werden bei Bedarf erzeugt. Durch die Verwendung von `HashMaps` [28] kann mit konstantem Zeitaufwand das Enthaltensein einer Anfrage im Cache überprüft werden oder die Bestimmung der Tabelle, die das Ergebnis einer Anfrage enthält, erfolgen. Durch die Verwendung von Namenspräfixen bei Tabellennamen können sich mehrere benannte Caches dieselbe Cachedatenbank teilen.

Bei der Implementierung der Verdrängungsstrategien wurden anstelle von Listen `java.util.LinkedHashSets` [29] verwendet. Diese kombinieren die Laufzeiteigenschaften von `HashSets` mit der festen Reihenfolge von Elementen in Listen.

4.8 Anfragesequenzen

Zur Darstellung der Trainingssequenzen, von Subsequenzen der Trainingssequenzen oder des Fensters zuletzt gestellter Anfragen werden Listen von `wmocache.prefetching.wmo.learning.QuerySequenceItems` verwendet. Abbildung 4.7 zeigt das Interface dieser Klasse. Ein `QuerySequenceItem` macht alle über eine Anfrage und deren Ergebnis benötigten Informationen verfügbar. Einmal beschaffte Informationen werden intern gecacht.

[28] `http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html` – zuletzt überprüft am 04.03.2014

[29] `http://docs.oracle.com/javase/6/docs/api/java/util/LinkedHashSet.html` – zuletzt überprüft am 04.03.2014

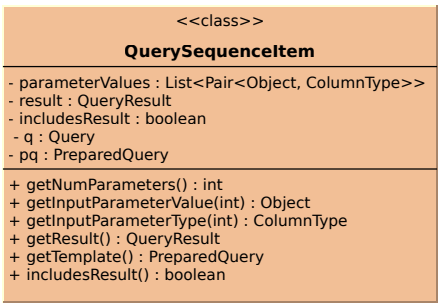


Abbildung 4.7: `wmocache.prefetching.wmo.learning.QuerySequenceItem`

4.9 Parameterkorrelationen

Die Repräsentation von Parameterkorrelationen in WMoCache folgt weitgehend der in Abschnitt 3.5 gewählten Modellierung, mit dem Unterschied, dass aus implementierungstechnischen Gründen durch Eingabe- und Ausgabekorrelationen referenzierte Anfragen nicht über ihr Prepared Statement, sondern über ihren Abstand zur Ausgangsanfrage der Korrelation innerhalb der jeweiligen häufigen Subsequenz adressiert werden. Beispiel 4.1 illustriert dies.

Beispiel 4.1. Gegeben sei die häufige Subsequenz $\langle q_a, q_b, q_c, q_d \rangle$. Es existiere eine Eingabekorrelation zwischen dem ersten Parameter von q_d und dem dritten Parameter von q_b. In Abschnitt 4.1 wäre dies folgendermaßen notiert worden:

$$\langle 1 \mid I \ \mathbf{q_b} \ 3 \ \langle h \ f \rangle \rangle$$

Die Implementierung verwendet die äquivalente Notation

$$\langle 1 \mid I \ \mathbf{1} \ 3 \ \langle h \ f \rangle \rangle.$$

□

Abbildung 4.8 zeigt das UML-Klassendiagramm des Pakets `wmocache.prefetching.wmo.param_corrs`. Zur Implementierung von Operationen auf Parameterkorrelationen wird das Visitor-Pattern verwendet.

Zur Unterscheidung wird für bei der Vorhersage einer Anfrage übergebliebene Ausgabekorrelationen mit der zuletzt gestellten Anfrage, die durch kombiniertes Ausführen der Anfragen aufgelöst werden müssen, ein eigener Typ verwendet.

4.10 Parallelität in WMoCache

Durch den Einsatz von WMoCache sollen Datenbankanwendungen beschleunigt werden. Zur Erfüllung dieses Ziels ist es notwendig, dass alle Operationen von WMoCache möglichst geringen Zeitaufwand verursachen. Der dennoch entstehende Aufwand kann auf Multiprozessor/-core Systemen teilweise dadurch kaschiert werden, dass Operationen nebenläufig ausgeführt werden. In Abschnitt 3.8 wurde erklärt,

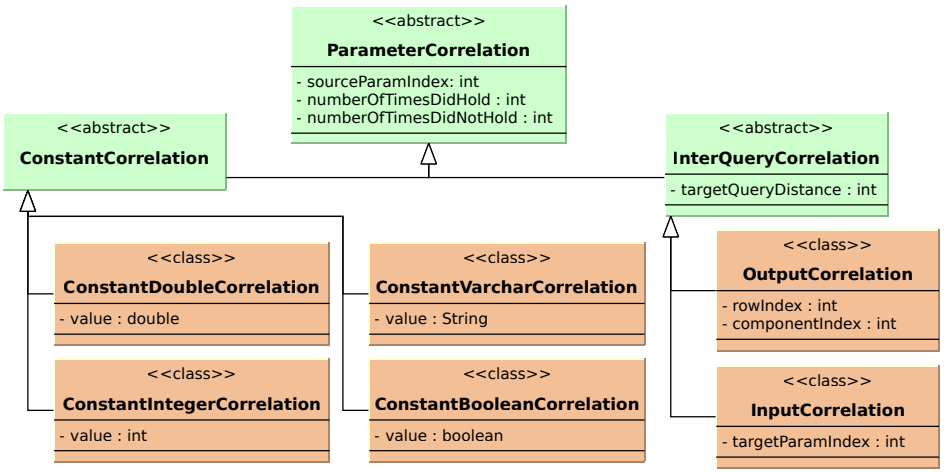


Abbildung 4.8: Klassen zur Speicherung von Parameterkorrelationen

dass das Prefetching vorhergesagter Anfragen häufig nebenläufig geschehen kann. Darüber hinaus können weitere Operationen asynchron durchgeführt werden. Dieser Abschnitt beschreibt die Umsetzung von Parallelität in WMoCache.

Jeder der in Abbildung 4.1 dargestellten Komponenten von `WMoCache` ist ein Thread des Typs `wmocache.WMoCache.MaintenanceThread` zugeordnet. Die Threads werden durch den Konstruktor von `WMoCache` gestartet und erst beim Schließen von WMoCache gestoppt. Einem Thread t können Aufgaben in Form von `Runnables` [30] zugewiesen werden. Beim Zuweisen einer Aufgabe wird der aufrufende Thread solange blockiert, bis t seine zuletzt erhaltene Aufgabe abgeschlossen hat. Die Rückgabe von Ergebniswerten einer Operation erfolgt auf ähnlichem Wege: Das Ergebnis einer Aufgabe ist erst verfügbar, wenn das Ergebnis der vorherigen Aufgabe durch den Aufrufer abgeholt wurde. Ist ein Ergebnis zu erwarten, wird der aufrufende Thread blockiert, bis dieses verfügbar ist. Diese Mechanismen können effektiv genutzt werden, um Locks auf Komponenten von WMoCache zu realisieren. Wichtig ist hierzu, dass alle Operationen, welche auf eine Komponente von WMoCache zugreifen, durch den jeweiligen Thread durchgeführt werden, oder diesen zumindest analog zu Beispiel 4.2 für andere Aufgaben blockieren. Zudem ist die Verwendung aller öffentlichen Methoden von `WMoCache` auf exklusiven Zugriff beschränkt.

Das Beispiel 4.2 demonstriert die Verwendung von `MaintenanceThreads`.

Beispiel 4.2. Gegeben seien zwei Kompontenen c_1 und c_2. Für Zugriffe auf c_1 sei `thread1` zuständig, für Zugriffe auf c_2 `thread2`. Es sei eine Operation durchzuführen, die zunächst beide Komponenten einbezieht. Anschließend wird anhand von c_2 ein Ergebnis produziert und zurückgegeben. Listing 4.1 zeigt den Code, der den Threads die Aufgabe zuweist.

In Zeile eins wird `thread1` eine Aufgabe zugewiesen. Diese wiederum weist `thread2` ein `Runnable` zu, welches den ersten Teil der Aufgabe, der c_1 und c_2 einbezieht, durchführt. Da der zweite Teil der Operation erst ausgeführt werden kann, nachdem der erste Teil abgeschloßen wurde, und zudem der exklusive Zugriff auf c_1 aufrecht

[30] `http://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html` – zuletzt überprüft am 05.03.2014

erhalten werden muss, wird `thread1` blockiert, indem er in Zeile 11 auf ein Resultat von `thread2` wartet, das, da die Operation in Zeile sieben ergebnislos ist, aus einem beliebigen Objekt besteht. Sobald `thread2` seine Aufgabe beendet hat, führt `thread1` den zweiten Teil der Aufgabe durch und produziert in Zeile 13 ein Ergebnis, das in Zeile 16 durch den Hauptthread erwartet wird. □

```
1  thread1.todos.put(new Runnable() {
2
3      public void run() {
4          thread2.todos.put(new Runnable() {
5
6              public void run() {
7                  // Operation auf c_1 und c_2
8                  thread2.results.put(new Object());
9              }
10         });
11         thread2.results.take();
12         // Operation auf c_1
13         thread1.results.put(result);
14     }
15 });
16 result = thread1.results.take();
```

Listing 4.1: Beispiel für die Verwendung von `MaintenanceThread`

Abbildung 4.9 illustriert den Nutzen von Parallelität in WMoCache. Gezeigt ist das UML-Sequenzdiagramm der Beantwortung einer Anfrage q. Zunächst werden die Caches abgefragt (2. und 3.). Keiner der Caches enthält das Ergebnis von q. Das Vorhersagemodul wird mit Parameter q aufgerufen und ist in der Lage eine Vorhersage zu machen (4.). Da zwischen q und der vorhergesagten Anfrage q_{pred} keine Ausgabekorrelationen bestehen, kann das Prefetching von q_{pred} asynchron erfolgen (8.), während gleichzeitig die Anfrage q an die Datenbank gestellt wird (9.). Deren Ergebnis kann bereits an die Datenbankanwendung zurückgegeben werden, während das Vorhersagemodul noch über das Ergebnis von q informiert wird (10.) und die Anfrageergebnisse in die Caches eingefügt werden (8.1. und 11.).

4.11 Effiziente Überprüfung des Subpfadkriteriums

Bei der Überprüfung des Subpfadkriteriums in Algorithmus 3.9 wird getestet, ob jede um eins kürzere Subsequenz eines gegebenen Pfades im bisher generierten Trie gespeichert ist. Bei Verwendung eines beliebigen Tries müsste jede dieser Subsequenzen zunächst erstellt werden. Die in WMoCache verwendete generische Trie-Implementierung besitzt die Methode `contains(List<K>, int) : boolean`. Diese testet ob eine Liste von Elementen vom Typ `K` als Schlüssel im Trie vorkommt, wenn das i-te Element der Liste nicht berücksichtigt wird. Dies ermöglicht die in Listing 4.2 gezeigte effiziente Überprüfung des Subpfadkriteriums.

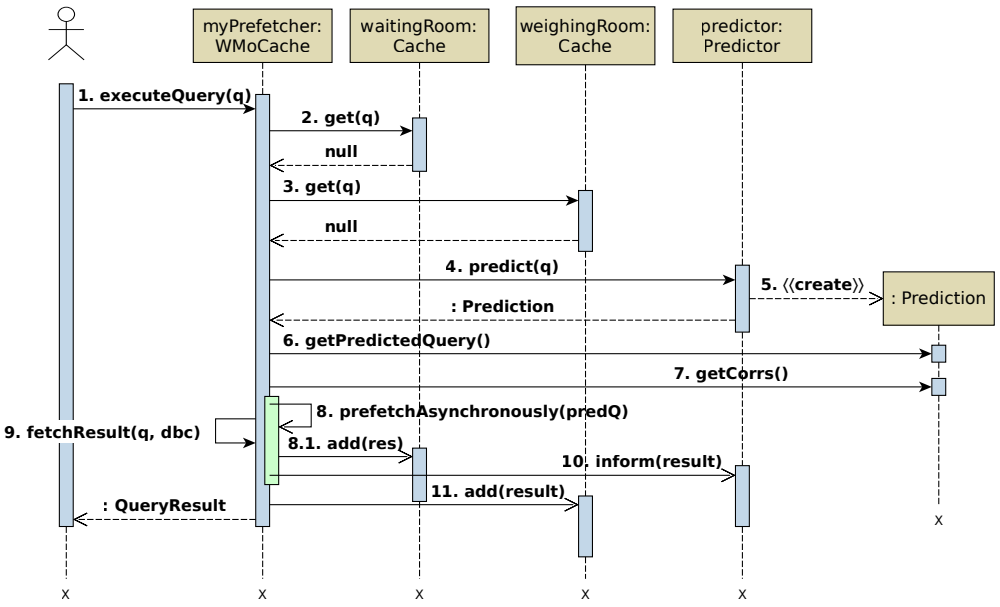


Abbildung 4.9: UML-Sequenzdiagramm der Beantwortung einer Anfrage durch WMoCache

```
1 boolean foundAllSubsequences = true;
2
3 for (int i = 0; i < cand.size(); ++i) {
4
5     if (!trie.contains(cand, i)) {
6         foundAllSubsequences = false;
7     }
8 }
```

Listing 4.2: Effiziente Überprüfung des Subpfadkriteriums in WMoCache

4.12 Implementierung der Lateral-Outer-Union-Strategie

Die Kombination zweier Anfragen q_a und q_b anhand des in Abschnitt 3.7 beschriebenen Verfahrens ist im AST nicht möglich, da IQCache Query weder temporäre Views, noch die Vereinigung von Anfragen mit `UNION` oder Unteranfragen darstellen kann. Die Implementierung in `LateralOuterUnionQueryCombiner` verfolgt daher eine kombinierte Strategie. Die Klasse `wmocache.util.helper.query.StringExpression` dient dem Einfügen von SQL-Code an einer beliebigen Stelle eines ASTs. Im ersten Schritt werden im AST der abhängigen Anfrage alle Blattknoten, in welchen aufgrund von Ausgabekorrelationen Parameterwerte durch Unteranfragen ersetzt werden müssen, durch `StringExpressions` ersetzt, die dem ursprünglichen Teilausdruck nach Ersetzung des Parameters durch die Unteranfrage entsprechen. Anschließend werden beide zu kombinierenden Anfragen in SQL-Strings umgewan-

delt. Durch Modifikation der beiden Zeichenketten und direkter Generierung von SQL-Code wird schließlich die kombinierte Anfrage erstellt.

4.13 Fazit

In diesem Kapitel wurde WMoCache, eine Implementierung des in Kapitel 3 beschriebenen erweiterten WM_o-Verfahrens, vorgestellt. WMoCache ist eine in der Programmiersprache Java geschriebene Middlewarekomponente, die als Wrapper um eine JDBC-Verbindung zu einer relationalen Datenbank Prefetching und Caching für Datenbankanwendungen verfügbar macht. Die Realisierung als Wrapper erlaubt die Verwendung von WMoCache in Kombination mit beliebigen SQL-Datenbanken, für welche ein JDBC-Treiber existiert und die die verwendete SQL-Syntax unterstützen. Durch die Verwendung von IQCache Query ist die Menge der SQL-Anfragen, welche durch WMoCache verarbeitet werden können, eingeschränkt. Die Implementierung besitzt keine besondere Strategie zum Umgang mit durch Update-Statements ungültig gewordenen Cacheinhalten.

Evaluation

5 Evaluation

Anhand der in Kapitel 4 der Arbeit beschriebenen Implementierung wurde der WM_o-Algorithmus für das Prefetching von SQL-Anfragen evaluiert. Ziel der Evaluation war einerseits zu untersuchen ob Datenbankanwendungen existieren, mit welchen WMoCache gewinnbringend eingesetzt werden kann, und andererseits den Einfluss einzelner Konfigurationsparameter auf die Effektivität und Performanz von WMoCache herauszufinden. Zu diesem Zweck wurden ausgewählte Datenbankbenchmarks verwendet.

Das Kapitel beschreibt zunächst die Vorgehensweise bei der Vorbereitung und Durchführung der Evaluation. Anschließend wird eine Reihe von Metriken vorgestellt, die in WMoCache gemessen werden können. Die verwendeten Benchmarks werden beschrieben. Schließlich werden die durchgeführten Tests erklärt und deren Ergebnisse präsentiert und analysiert.

5.1 Durchführung der Evaluation

In WMoCache ist die Erfassung einer Reihe von Metriken integriert. (Zwischen-)ergebnisse können über entsprechende Methoden von `wmocache.WMoCache` abgefragt werden.

Datenbankbenchmarks liegen üblicherweise in Form von Programmen vor, die Anfragen an ein Datenbanksystem senden und dabei das Verhalten einer bestimmten Datenbankanwendung oder einer Klasse von Datenbankanwendungen (z.B. Echtzeit-Transaktionsverarbeitung (OLTP), oder Online Analytical Processing (OLAP)) simulieren. Oft werden mit Hilfe von Multi-Threading und parallelen Datenbankverbindungen mehrere Clients oder Terminals simuliert (z.B. in BenchmarkSQL [31] oder OLTP-Bench [32] [34, 35]). Die Ausführung der Benchmarks gliedert sich im Falle der hier verwendeten Benchmarks in drei Phasen:

1. Vorbereitung der Testdatenbank
2. Durchführung des Benchmarks
3. Zurücksetzen der Datenbank

Die Benchmarkimplementierungen wurden nicht direkt mit WMoCache verbunden, sondern es wurden die während einer Ausführung durch ein Terminal [33] an die Testdatenbank gesendeten SQL-Statements aufgezeichnet, um sie später wiederholt abspielen zu können.

[31] `http://benchmarksql.sourceforge.net/` – zuletzt überprüft am 07.03.2014
[32] `http://oltpbenchmark.com/` – zuletzt überprüft am 07.03.2014
[33] entspricht einer Datenbankverbindung

Mit `WMoPrefetchingSimulation` steht ein Programm zur Verfügung, das ein SQL-Skript inklusive darin enthaltener `COMMIT`- und `ROLLBACK`-Anweisungen einlesen kann und WMoCache verwendet, um die Statements der Reihe nach an eine Testdatenbank zu senden. Dabei werden (kontinuierlich) Messwerte gesammelt. `WMoPrefetchingSimulation` nimmt zwei weitere SQL-Skripte entgegen, die der Vorbereitung der Testdatenbank und dem Zurücksetzen der Datenbank nach Abschluss eines Messvorgangs dienen.

Zunächst musste ein SQL-Skript gewonnen werden, das den Zustand der Testdatenbank nach der Vorbereitung durch den jeweiligen Benchmark herstellt. Dies beinhaltet das Erstellen des verwendeten Datenbankschemas und das initale Befüllen der Tabellen mit Daten. Im Falle von BenchmarkSQL (siehe Abschnitt 5.4.2) konnten die Statements zum Befüllen der Tabellen aus CSV-Dateien gewonnen werden, die der Benchmark optional pro Tabelle generiert. Im Falle von AuctionMark bzw. OLTP-Bench (siehe Abschnitt 5.4.3) wurde das Werkzeug `pg_dump` [34] verwendet, um den Zustand der Datenbank nach der ersten Phase des Benchmarks in einem SQL-Skript zu sichern.

Zum Festhalten der während der Ausführung eines Benchmarks gestellten Anfragen wurden die Benchmarkimplementierungen derart erweitert, dass die durch ein Terminal generierten SQL-Statements in einer Datei aufgezeichnet werden konnten. Dabei ist es zur Sicherstellung der späteren Ausführbarkeit des gewonnenen SQL-Skripts wichtig, dass nur erfolgreich ausgeführte Statements aufgezeichnet werden. Falls ein Benchmark Transaktionen verwendet, müssen auch Commit- und Rollback-Anweisungen vermerkt werden. Einige in den Benchmarks auftretende Anfragen können durch IQCache Query nicht geparst bzw. im AST dargestellt werden. In diesen Fällen wurden die Anfragen umschrieben, wobei darauf geachtet wurde, dass `WHERE`-Klausel und Ergebnis der Umschreibung dieselbe Struktur wie im Falle der ursprünglichen Anfragen besitzen. Die im Folgenden beschriebenen Fälle sind aufgetreten:

- Das Vorkommen anderer Elemente als der Bezeichner von Tabellenspalten in der Projektionsliste einer Anfrage kann, sofern das Ergebnis der ursprünglichen Anfrage nur einen Wert enthält, folgendermaßen umschrieben werden: Gegeben sei die in Listing 5.1 gezeigte Anfrage.

```
1 SELECT SUM(ol_amount) AS ol_total
2 FROM order_line
3 WHERE ol_o_id = ?
4     AND ol_d_id = ?
5     AND ol_w_id = ?
```

Listing 5.1: Eine SQL Anfrage, die die Aggregationsfunktion `SUM` verwendet

  Sie verwendet die Aggregationsfunktion `SUM`. Zunächst wird die Anfrage in der modifizierten Benchmarkanwendung normal ausgeführt. Nun wird an das zu generierende SQL-Skript ein Statement angefügt, das den Wert s der Summe in eine einspaltige und bisher leere Tabelle `helper_tbl` einfügt. Die Anfrage kann jetzt wie in Listing 5.2 dargestellt umschrieben werden.

[34] `http://www.postgresql.org/docs/9.1/static/app-pgdump.html` – zuletzt überprüft am 07.03.2014

```
1 SELECT helper_tbl.* AS ol_total
2 FROM order_line,
3      helper_tbl
4 WHERE ol_o_id = ?
5     AND ol_d_id = ?
6     AND ol_w_id = ?
7 LIMIT 1
```

Listing 5.2: Umschreibung der SQL Anfrage aus Listing 5.1

Abschließend wird dem Skript ein Statement hinzugefügt, das die Hilfstabelle wieder leert. Mit ähnlichen Fällen wurde analog verfahren.

- Bisweilen besitzen Anfragen den Zusatz `FOR UPDATE`, um einen Write-Lock auf die selektierten Tupel für eine darauffolgende Änderung zu erhalten. Da bei der Evaluation von WMoCache aus Sicht der Benchmarks keine parallelen Datenbankverbindungen genutzt werden, kann der Zusatz entfernt werden.

- Tabellennamen vorangestellte Bezeichner von Schemata wurden entfernt.

- `(a BETWEEN x AND y)` ist äquivalent zu `(x <= a AND a <= y)`.

- Bei einem der Benchmarks kommt regelmäßig in `VARCHAR`-Konstanten die Zeichenfolge `'&'` vor. `WMoPrefetchingSimulation` interpretiert Strichpunkte als Begrenzungszeichen von SQL-Statements. Daher wurde die Zeichenfolge stets durch `'&amp'` ersetzt.

Beide zur Evaluation herangezogenen Benchmarks verwenden SQL-Datentypen, die in IQCache Query nicht implementiert sind. Da, wie in Abschnitt 2.5 beschrieben, IQCache Query alle unbekannten Datentypen durch `VARCHAR(50)` ersetzt und WMoCache bei Wertvergleichen ausschließlich auf Gleichheit testet, verarbeitet WMoCache die Anfragen dennoch korrekt.

Zur Bestimmung für die Evaluation interessanter Konfigurationsparameter wurden Experimente mit einer Reihe einfacher Transaktionen auf Basis des von Kemper und Eickler in [36] verwendeten Universitätsdatenbankschemas und künstlich generierter Daten gemacht [35]. Dabei konnten Parameter identifiziert werden, die besonders starken Einfluss auf das Verhalten des Prefetchingverfahrens haben.

Vor der Durchführung der Messungen mit einem Benchmark wurde jeweils die Ausgangskonfiguration von WMoCache für die aufgezeichneten Anfragen optimiert. Dazu wurden auch, wie in Abschnitt 4.2 empfohlen, die einzelnen Transaktionen der Benchmarks analysiert.

Jede Zeitmessung wurde zur Minimierung des Einflusses von Messungenauigkeiten fünffach durchgeführt. Anschließend wurde jeweils der Median der gemessenen Werte gebildet.

[35] Zur Generierung der Daten wurde IQCache RQBench verwendet

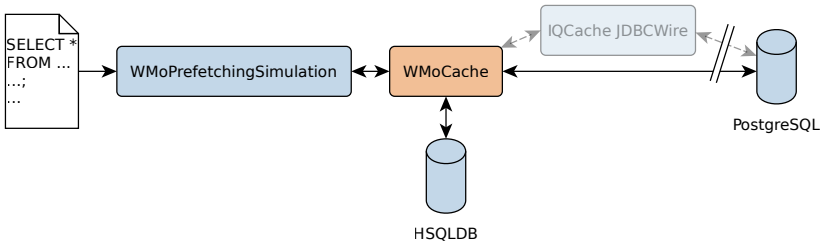


Abbildung 5.1: Versuchsaufbau zur Evaluation von WMoCache

5.2 Versuchsaufbau

Evaluiert wurde eine Variante von WMoCache mit `LRUCache` als Anfragecache und `FIFOCache` als Prefetchingcache. Als Implementierung von `FindSubsequencesStrategy` diente `FindSubsequencesDistThreshold` mit `MaxSubsequencePruningStrategy` als Pruningsstrategie. Als lokale Cachedatenbank wurde HSQLDB [36] in Version 2.3.1 in der Konfiguration als In-Memory-Datenbank verwendet. Als Client für die Zeitmessungen diente ein Dell Studio XPS 7100 Rechner (AMD Phenom^{TM}II X6 1045T, 8GB RAM) mit Ubuntu [37] 12.10 (Linux 3.5.0-generic x86 64) als Betriebssystem. Die CPU-Frequenz war auf konstant 2,7 GHz eingestellt. Es wurde OpenJDK 7, 64 Bit als Java-Laufzeitumgebung verwendet. Als Datenbankserver diente PostgreSQL [38] in Version 8.4.4, 32 Bit. Client- und Serverhost befanden sich im selben lokalen Netzwerk. Die durchschnittliche RTT bei 100 Pings betrug $0.644ms$.

Zur Erzeugung einer künstlichen Latenz vorgegebener Dauer vor dem Absenden von SQL Statements an den Datenbankserver wurde WMoCache bei Bedarf der JDBC-Treiber IQCache JDBC Wire vorgeschaltet.

Abbildung 5.1 zeigt den vollständigen Versuchsaufbau.

5.3 Verwendete Metriken

Im Folgenden sind die in der Evaluation verwendeten Metriken aufgezählt. Es wird jeweils beschrieben was und wie gemessen wird und wie die gemessenen Werte zu interpretieren sind. Die Metriken sind größtenteils durch andere Arbeiten zum Thema Prefetching inspiriert. Die entsprechenden Quellen sind genannt.

5.3.1 Hitrate des Caches

Die Hitrate des Caches gibt den Anteil durch die Anwendung gestellter Anfragen an, die durch den Cache beantwortet werden konnten. Im Falle von WMoCache wird die Metrik sowohl für den gesamten Cache als auch differenziert für den Prefetching- oder den Anfragecache erfasst. Letztere Messungen geben Anhaltspunkte über den

[36] `http://hsqldb.org/` – zuletzt überprüft am 07.03.2014
[37] `http://www.ubuntu.com/` – zuletzt überprüft am 07.03.2014
[38] `http://www.postgresql.org/` – zuletzt überprüft am 07.03.2014

Anteil von Prefetching und regulärem Caching an einer möglicherweise durch den Einsatz von WMoCache erreichten Beschleunigung einer Datenbankanwendung. Zur Messung werden die Treffer des Caches gezählt und anschließend durch die Anzahl gestellter Anfragen dividiert. Der Wertebereich liegt zwischen 0 und eins. Ein möglichst hoher Wert ist erstrebenswert. Im Falle des Prefetchingcaches erfolgt die Messung stets nur während der Prefetchingphase. Die Hitrate des Caches wird auch in [1, 5, 9, 10, 11] gemessen. Die Quellen [3, 15] und [18] messen stattdessen die Missrate.

5.3.2 Präzision des Prefetchers

Die Präzision gibt den Anteil zutreffender Vorhersagen an der Gesamtzahl an Vorhersagen an. Dabei werden nur Vorhersagen gezählt, bei welchen Prefetching durchgeführt werden konnte und die vorhergesagte Anfrage aufgetreten ist bevor das nächste Mal Prefetching erfolgt ist. Die Metrik wird gemessen, indem die Größe des Prefetchingcaches auf eins verringert wird. Nun werden die Zahl n_h der Treffer des Prefetchingcaches während der Prefetchingphase und die Anzahl n_p der Vorhersagen gezählt und deren Verhältnis $\frac{n_h}{n_p}$ wird gebildet. Der Wert der Metrik liegt im Intervall $[0, 1]$. Ein möglichst hoher Wert wird angestrebt.

Die Metrik wird in ähnlicher Form auch in [1, 9, 10, 15, 18] verwendet.

5.3.3 Zeitlicher Aufwand zur Generierung der Prefetchingstruktur

Mit Hilfe der Metrik kann untersucht werden, welchen Einfluss die Anzahl der während der Lernphase erlernten Anfragen und weitere Parameter wie die Länge der Trainingssequenzen auf den Aufwand zur Generierung der Prefetchingstruktur haben. Darüber hinaus erhält man einen Anhaltspunkt für die Verzögerung, die beim Übergang von der Lern- zur Prefetchingphase besteht. Bei der Evaluation von WMoCache hat sich gezeigt, dass eine exakte Messung der Metrik nur in größtmöglicher Isolation möglich ist. Daher wurde zur Erfassung lediglich die Lernphase durchgeführt mit der Generierung der Prefetchingstruktur als abschließendem Vorgang. Im Falle der Messung des Zeitaufwandes in Abhängigkeit von der Zahl erlernter Anfragen wurde zudem eine spezialisierte Variante von WMoCache verwendet, die vor der Messung wartet, bis andere Aktivitäten, wie Cacheverwaltung, beendet sind. Diese Metrik wird auch in [37] verwendet.

5.3.4 Gesamtdauer der Prefetchingphase

Diese Metrik gibt Aufschluss über den Nutzen von WMoCache für eine Datenbankanwendung. Sie zeigt, ob WMoCache in der Lage ist Latenzen, die beim Abrufen von Anfrageergebnissen entstehen, zu verbergen und ob dieser Vorteil nicht durch den entstehenden Mehraufwand aufgezehrt wird. Es wird lediglich die Dauer der Prefetchingphase betrachtet, da die Annahme besteht, dass die Lernphase im Vergleich zur Prefetchingphase bedeutend kürzer ist und eine notwendige Vorbedingung zum

Einsatz des Prefetchers darstellt, zumal lediglich aufgetretene Anfragen aufgezeichnet werden. Bei Einsatz des Prefetchers und des Caches sollte die Dauer kürzer sein, als ohne deren Verwendung.

Die Quelle [2] verwendet eine ähnliche Metrik.

5.3.5 Durchschnittlicher Aufwand zur Beantwortung einer Anfrage

Die Metrik zeigt, ob WMoCache in der Lage ist, die Zeit zu verkürzen, die eine Anwendung damit verbringt auf Ergebnisse von Datenbankanfragen zu warten. Die Messung erfolgt, indem für jede Anfrage die Zeitspanne zwischen Stellen der Anfrage und Eintreffen des Ergebnisses gemessen wird. Die Summe der Zeiten wird durch die Anzahl gestellter Anfragen dividiert. Wie auch bei der Gesamtdauer der Prefetchingphase sollte der Wert mit aktivem Prefetcher und Cache niedriger sein, als sonst.

Die Messung ist inspiriert durch die Quellen [6] und [9].

5.3.6 Maximale Länge gefundener häufiger Subsequenzen der Trainingssequenzen

Die Messung gibt Aufschluss darüber, in wie weit der erweiterte WM_o-Algorithmus in der Lage ist für Prefetching verwendbare Anfragemuster zu identifizieren. Falls keine Anfragemuster gefunden werden können, spricht dies entweder für eine unzureichende Konfiguration von WMoCache oder die Tatsache, dass WMoCache für eine gegebene Datenbankbankanwendung oder einen Benchmark nicht geeignet ist. Die Metrik wird auch in [37] zur Evaluation verwendet.

5.3.7 Anzahl gefundener häufiger Subsequenzen der Trainingssequenzen

Ähnlich zur maximalen Länge der identifizierten häufigen Subsequenzen gibt die Metrik Aufschluss darüber, wie gut WMoCache in der jeweiligen Konfiguration, bzw. überhaupt in der Lage ist Anfragemuster im Anfragestrom einer Datenbankanwendung zu finden. Ähnliche Metriken werden auch in [37] verwendet.

5.4 Benchmarks

Es wurde mit zwei Benchmarks evaluiert. Dieser Abschnitt erläutert zunächst die für die Wahl der Benchmarks relevanten Kriterien. Danach wird jeder der ausgewählten Benchmarks kurz beschrieben.

5.4.1 Auswahlkriterien

Es wurden insgesamt vier Kriterien verwendet. Zwei davon betreffen die Art der Datenbankanwendungen, die durch einen Benchmark simuliert werden: Damit WMoCache sinnvoll eingesetzt werden kann, muss eine Datenbankanwendung mindestens eine regelmäßig auftretende Transaktion besitzen, sodass WMoCache in der Lage ist, deren Folge von Prepared Statements als wiederkehrendes Anfragemuster in den Trainingssequenzen zu identifizieren. Die Parameter von Anfragen sollten entweder konstant sein, oder es muss aufgrund von Datenabhängigkeiten möglich sein, von den Parameterwerten und Ergebnissen bisheriger Anfragen auf die Parameterwerte zukünftiger Anfragen zu schließen. Andernfalls kann WMoCache die Werte von Parametern vorhergesagter Anfragen nicht bestimmen.

Diese Kriterien schließen beispielsweise den TPC-H [38] [39] Benchmark aus, da dieser Ad-Hoc-Anfragen verwendet. Damit ist auch der Star Schema Benchmark [39, 40] als Derivat von TPC-H ausgeschlossen. Auch der an der Universität Passau entwickelte Benchmark IQCache RQBench ist gänzlich ungeeignet, da er zufällige, nicht notwendigerweise wiederkehrende Anfragen ohne semantischem Zusammenhang generiert.

Um die Reproduzierbarkeit und Vergleichbarkeit der Messergebnisse zu gewährleisten, ist es zudem sinnvoll, nur Benchmarks zu verwenden, bei welchen auf eine frei verfügbare Implementierung zurückgegriffen werden kann, die bestenfalls auch schon in anderen Veröffentlichungen zur Evaluation angewandt wurde.

Anhand der genannten Kriterien wurden die beiden OLTP-Benchmarks TPC-C [41] [40] und AuctionMark [42, 43] ausgewählt. Sie werden im Folgenden beschrieben.

5.4.2 TPC-C

Der TPC-C Benchmark wird durch das Transaction Processing Performance Council (TPC) [41] standardisiert. Beschrieben wird der Benchmark unter anderem in [44] und [45]. Die aktuelle Version des Standards ist [41]. TPC-C ist eine Simulation der Lagerverwaltung und Auftragsabwicklung eines Versandunternehmens. Dabei werden die typischen Transaktionen, die Mitarbeiter eines solchen Unternehmens ausführen, simuliert. Insgesamt existieren fünf Transaktionen:

- die Eintragung einer Bestellung
- die Abwicklung einer Zahlung eines Kunden

[39] `http://www.tpc.org/tpch/default.asp` – zuletzt überprüft am 10.03.2014
[40] `http://www.tpc.org/tpcc/` – zuletzt überprüft am 10.03.2014
[41] `www.tpc.org` – zuletzt überprüft am 10.02.2014

- die Abfrage des Bearbeitungsstatus einer Bestellung
- die Auslieferung einer Bestellung
- die Überprüfung des Lagerbestandes

Ein Mitarbeiter kann zu jeder Zeit eine beliebige Transaktion auswählen und ausführen. Die Häufigkeiten der Transaktionen entsprechen dabei realistischen Szenarien.

Die Datenbank besteht aus einer Reihe von Warenlagern. Jedes Warenlager ist für zehn Vertriebsregionen zuständig, in welchen jeweils 3'000 Kunden beheimatet sind. Jedes Warenlager hält 100'000 Artikel in einer bekannten Stückzahl vor. Knapp zehn Prozent der Bestellungen können nicht durch das eigentlich zuständige Warenlager abgefertigt werden und müssen an ein anderes delegiert werden. Die Unternehmensstruktur, die TPC-C zugrundegelegt ist, wird in [41] und [44] graphisch dargestellt.

Die einzige Transaktion, deren Häufigkeit nicht limitiert ist, ist die Aufnahme neuer Bestellungen. TPC-C misst die Leistung eines Systems anhand der Zahl *tpm-C* der neu aufgenommenen Bestellungen pro Minute.

Die Implementierung des Benchmarks ist jeder Organisation, die ihn nutzen möchte, selbst überlassen. Durch eine rigorose Verpflichtung zur Offenlegung wird sichergestellt, dass der Standard eingehalten wird. Daneben existiert auch eine Reihe frei nutzbarer Implementierungen (teilweise [46]): BenchmarkSQL [42] [46, 47], jTPCC [43] und OLTP-Bench implementieren den Benchmark in der Programmiersprache Java. Implementierungen in C stehen mit OSDL-DBT2 [44] und TPCC-UVa [48, 49] [45] zur Verfügung.

In dem Interview [50] argumentieren Repräsentanten des TPC anhand der Verbreitung, der Allgemeingültigkeit und der exakten Spezifikation für die hohe Bedeutung des TPC-C Benchmarks.

Zur Evaluation von WMoCache wird BenchmarkSQL verwendet. Diese Implementierung kommt außerdem in [47, 51, 52] und [53] zum Einsatz.

Tabelle 5.1 zeigt die optimierte Basiskonfiguration von WMoCache für die Evaluation mit BenchmarkSQL.

5.4.3 AuctionMark

AuctionMark ist ein auf TPC-C basierender OLTP-Benchmark [43], der die Aktivitäten in einer „wohlbekannten" Online-Auktionsplattform simuliert [42]. Er verwendet 13 Tabellen [46] und 14 verschiedene Transaktionen, welche die Basistransaktionen der Auktionsplattform widerspiegeln. Die Transaktionen gliedern sich in zwei Gruppen: Es gibt Transaktionen, die durch Benutzer initiiert werden und solche, die der

[42] `http://benchmarksql.sourceforge.net/` – zuletzt überprüft am 07.03.2014
[43] `http://jtpcc.sourceforge.net/` – zuletzt überprüft am 10.03.2014
[44] `http://sourceforge.net/projects/osdldbt/` – zuletzt überprüft am 10.03.2014
[45] `http://www.infor.uva.es/~diego/tpcc-uva.html` – zuletzt überprüft am 10.03.2014
[46] Unter `https://github.com/apavlo/h-store/blob/master/src/benchmarks/edu/brown/benchmark/auctionmark/docs/db_diagram.pdf` – zuletzt überprüft am 11.03.2014 – ist ein Diagramm des Datenbankschemas hinterlegt.

Parameter	Wert
lmMinSupport	6
lmMaxCorrFailRatio	0
lmTrainingSeqLength	72
lmResultSetMaxSize	1
lmCorrelationsWindowSize	1
freqPairsGraphBuilderMaxPairDist	4
freqPairsGraphBuilderMinFreq	15
findSubsequencesDistThresholdMaxNumIterations	1
maxSubsequencePruningStrategyThreshold	3
pfResultSetMaxSize	1
pfWindowSize	4
waitingRoomSize	80
weighingRoomSize	120
learningDuration	720
autoCommit	false

Tabelle 5.1: Basiskonfiguration von WMoCache für die Evaluation mit BenchmarkSQL

Buchführung dienen und regelmäßig durch das System ausgeführt werden. Die beiden primären Benutzergruppen des Systems sind Käufer und Verkäufer. Verkäufer bieten unter anderem Waren zum Verkauf an oder aktualisieren Informationen zu Angeboten. Käufer fügen Angebote zu ihrer Beobachtungsliste hinzu, rufen Informationen zu Angeboten ab und machen Gebote oder kaufen Gegenstände [42]. Das Verhältnis zwischen Benutzern und Angeboten folgt einer Zipfverteilung (siehe z.B. [54]), d.h. ein kleiner Anteil der Benutzer ist Anbieter eines Großteils der Waren [55]. Je näher das Auktionsende eines Angebots rückt, desto stärker wird es frequentiert [55]. Eine ausführliche Beschreibung des Datenbankschemas und der Transaktionen erfolgt in [42] und [43].

Implementiert wurde AuctionMark im Zuge des Projektes H-Store [47] [56, 57, 58, 59] und in der Benchmark-Suite OLTP-Bench. Zur Evaluation wird er unter anderem in [55, 59, 60] und [61] angewandt.

Zur Evaluation von WMoCache wurde die Implementierung in OLTP-Bench gewählt. OLTP-Bench kommt auch in [55, 59, 60, 62, 63, 64] und [65] zum Einsatz.

In Tabelle 5.2 ist die optimierte Grundkonfiguration von WMoCache für die Evaluation mit AuctionMark dargestellt.

[47] `http://hstore.cs.brown.edu/` – zuletzt überprüft am 11.03.2014

Parameter	Wert
lmMinSupport	10
lmMaxCorrFailRatio	0
lmTrainingSeqLength	100
lmResultSetMaxSize	1
lmCorrelationsWindowSize	1
freqPairsGraphBuilderMaxPairDist	1
freqPairsGraphBuilderMinFreq	21
findSubsequencesDistThresholdMaxNumIterations	1
maxSubsequencePruningStrategyThreshold	1
pfResultSetMaxSize	1
pfWindowSize	1
waitingRoomSize	6
weighingRoomSize	194
learningDuration	1100
autoCommit	false

Tabelle 5.2: Basiskonfiguration von WMoCache für die Evaluation mit AuctionMark

5.5 Aufzeichnen der Anfragen

Zur Evaluation mit TPC-C wurden 600 Transaktionen eines Terminals aufgezeichnet. Das Skript umfasst ingesamt 15732 SQL-Statements, wovon 7202 Anfragen sind. Für Messungen des Zeitaufwandes zur Generierung der Prefetchingstruktur wurden aufgrund der mangelnden Aussagekraft der ursprünglichen Messung mehr Transaktionen durchgeführt. Das Skript umfasst 2700 Transaktionen mit insgesamt 77632 SQL-Statements. Unter diesen sind 35652 Anfragen.

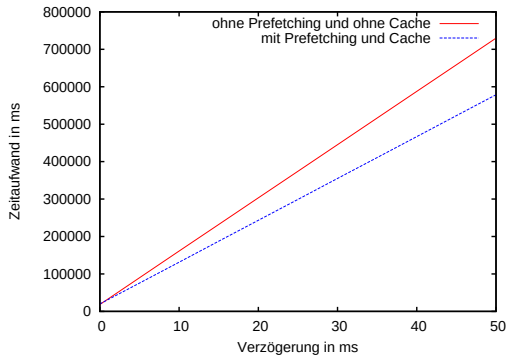
Mit AuctionMark wurden analog ein kürzeres und ein längeres SQL-Skript aufgezeichnet. Diese umfassen 29555 SQL-Statements mit 12298 Anfragen, sowie 82245 Statements mit 34255 Anfragen.

5.6 Messungen

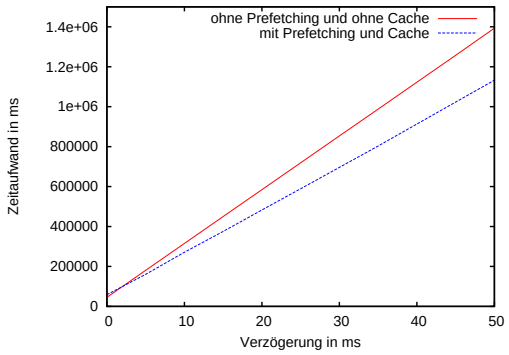
Im Folgenden werden die durchgeführten Messungen zusammen mit ihren Ergebnissen vorgestellt. Begonnen wird mit Messungen, welche die Nützlichkeit von WMoCache untersuchen. Darauf folgt einer Untersuchung des Einflusses einiger Konfigurationsparameter auf das Verhalten von WMoCache.

5.6.1 Leistungsverhalten von WMoCache

Dieser Abschnitt betrachtet das Leistungsverhalten von WMoCache. Dabei wird sowohl auf die Prefetchingphase als auch die Generierung der Prefetchingstruktur eingegangen. Besondere Aufmerksamkeit wird auf den Beitrag von Prefetching zur durch WMoCache erreichten Beschleunigung der Benchmarks gerichtet.

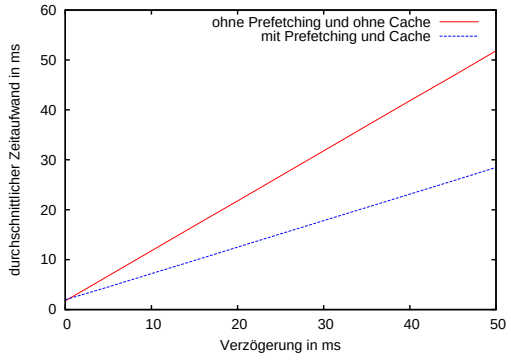


(a) TPC-C

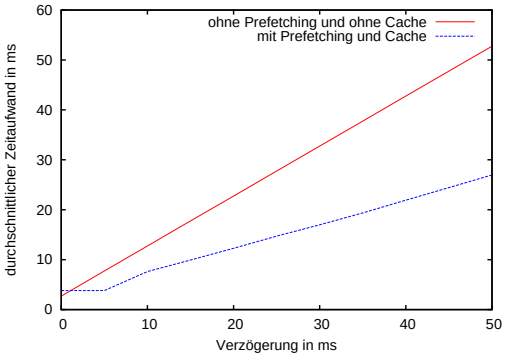


(b) AuctionMark

Abbildung 5.2: Dauer der Prefetchingphase in Abhängigkeit von der Anfragelatenz



(a) TPC-C



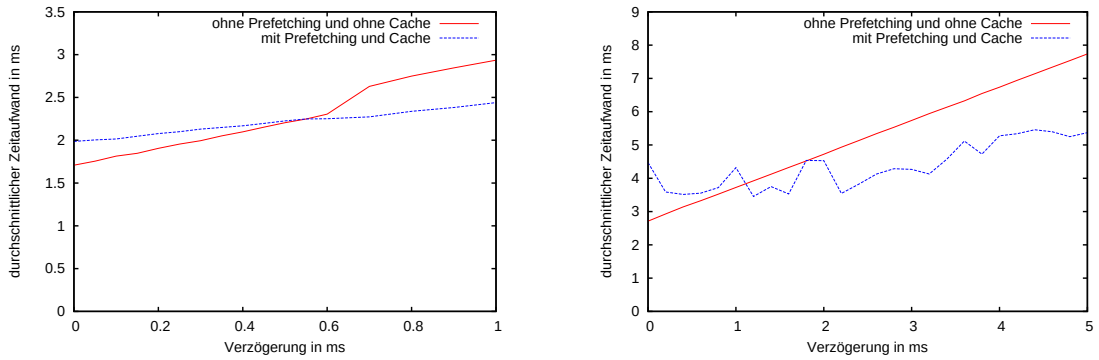
(b) AuctionMark

Abbildung 5.3: Durchschnittliche Wartezeit des Clients auf ein Anfrageergebnis in Abhängigkeit von der Anfragelatenz

5.6.1.1 Dauer der Prefetchingphase

Abbildung 5.2a zeigt für TPC-C die Dauer der Prefetchingphase in Millisekunden (ms) in Abhängigkeiten von der Dauer einer künstlichen Latenz vor dem Absenden jedes SQL-Statements. Dabei wurde sowohl mit aktivem Prefetcher und Cache (blau, gestrichelt), als auch ohne (rot) gemessen. Abbildung 5.2b zeigt das Ergebnis derselben Messung für AuctionMark. Man erkennt, dass in beiden Fällen bereits bei niedrigen Latenzen die Variante mit aktivem Prefetcher und Cache schneller ist. Bei TPC-C lohnt sich der Einsatz von WMoCache bereits bei etwas geringeren Latenzen als bei AuctionMark.

An dieser Stelle ist die Beschleunigung, die jede einzelne Anfrage im Durchschnitt erfährt, interessant. Abbildung 5.3 zeigt die durchschnittliche Wartezeit des Clients auf das Ergebnis einer Anfrage für große Latenzen. Bei der Messung zu Abbildung 5.4 wurde hingegen mit sehr kleinen Verzögerungen gearbeitet. Beide Messungen beziehen sich ausschließlich auf die Prefetchingphase. Für TPC-C wird durch den



(a) TPC-C (b) AuctionMark

Abbildung 5.4: Durchschnittliche Wartezeit des Clients auf ein Anfrageergebnis bei sehr kleinen Latenzen

Einsatz von Prefetching und Caching bei einer Latenz von knapp über $50ms$ eine Beschleunigung jeder Anfrage um durchschnittlich ca. $23ms$ erreicht. Bei AuctionMark werden über $25ms$ eingespart. Abbildung 5.4 zeigt, dass bei TPC-C Anfragen ab einer zusätzlichen Verzögerung von knapp $0,6ms$ durch den Einsatz von WMoCache beschleunigt werden. Bei AuctionMark ist dies erst ab etwa $2ms$ der Fall. Darunter überwiegt der durch WMoCache verursachte zusätzliche Aufwand.

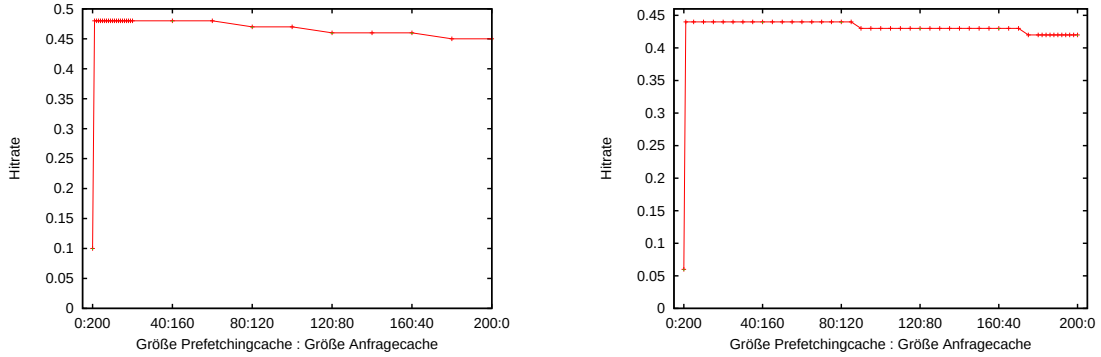
Es muss untersucht werden, ob die Beschleunigung primär durch den Anfragecache erreicht wurde, oder ob Prefetching dazu einen nennenswerten Beitrag geleistet hat. Die nächste Messung gibt hierzu einen Anhaltspunkt.

5.6.1.2 Auswirkung des Cacheverhältnisses auf die Hitrate

Entsprechend der Grundkonfiguration von WMoCache (Tabellen 5.1 und 5.2) fasst der lokale Cache insgesamt 200 Anfrageergebnisse. Der Anteil des Anfragecaches sowie des Prefetchingcaches am verfügbaren Speicherplatz ist konfigurierbar. Um herauszufinden, ob Treffer des Caches maßgeblich durch Caching oder aber durch Prefetching ermöglicht werden, wurde gemessen, wie sich unterschiedliche Größenverhältnisse von Anfrage- und Prefetchingcache auf die Hitrate des Caches während der Prefetchingphase auswirken. Abbildung 5.5 zeigt die Ergebnisse.

Falls nur der Anfragecache verwendet wird (Größenverhältnis $0 : 200$), kann der Cache im Falle des TPC-C Benchmarks 10 % der Anfragen beantworten. Erhöht man die Größe des Prefetchingcaches um ein Anfrageergebnis, steigt die Hitrate auf 48 %. In Folge sinkt die Trefferrate durch weitere Verkleinerung des Anfragecaches stetig. Der Prefetchingcache ist trotz zunehmenden Volumens nicht in der Lage dies auszugleichen. Bei ausschließlicher Verwendung von Prefetching wird eine Hitrate von 45 % erreicht. Somit ist maßgeblich Prefetching für die Cachetreffer verantwortlich.

Bei AuctionMark wird, falls ausschließlich Caching mit einem Anfragecache der Größe 200 verwendet wird, eine Hitrate von 6 % erreicht. Hinzunahme des Prefetchingcaches lässt die Trefferrate sofort auf 44 % aller Cacheanfragen anwachsen. Analog



(a) TPC-C (b) AuctionMark

Abbildung 5.5: Hitrate des Caches in Abhängigkeit vom Größenverhältnis von Prefetching- und Anfragecache

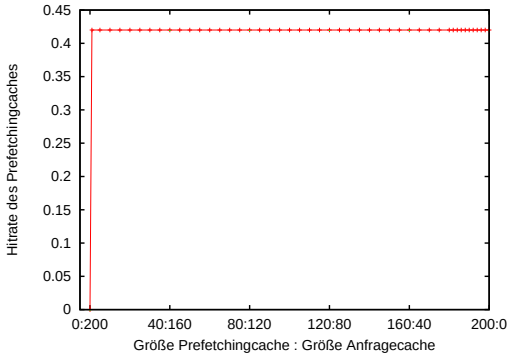
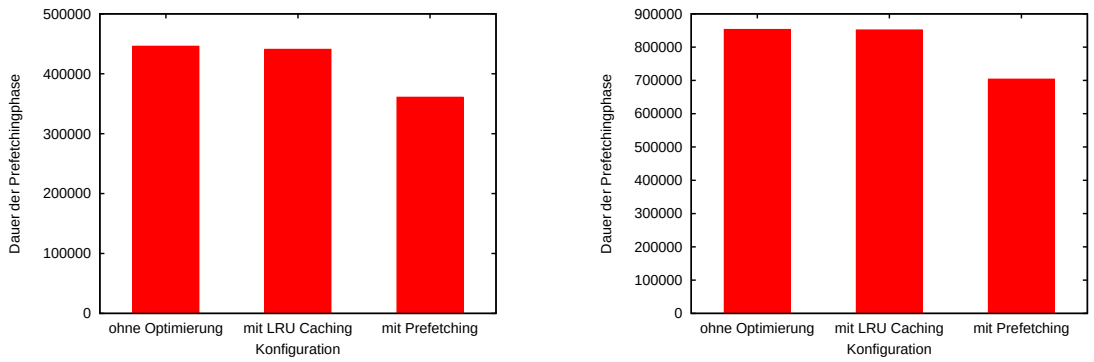


Abbildung 5.6: AuctionMark: Hitrate des Prefetchingcaches in Abhängigkeit vom Größenverhältnis von Prefetching- und Anfragecache

zum TPC-C Benchmark nimmt durch weitere Verkleinerung des Anfragecaches zugunsten des Prefetchingcaches die Hitrate stetig ab. Mit Prefetching alleine können 42 % der Anfragen durch den Cache beantwortet werden. Insgesamt werden wiederum primär durch Prefetching Cachetreffer erzielt. Interessanterweise erzielt der Prefetchingcache ab seiner Aktivierung über den gesamten Verlauf der Messung eine Hitrate von 42 % (siehe Abbildung 5.6). Somit genügt bei diesem Benchmark für Prefetching in Kombination mit einem FIFO-Cache im Gegensatz zu LRU-Caching bereits eine sehr geringe Cachegröße, um eine gute Trefferrate zu erziehen.

5.6.1.3 Dauer der Prefetchingphase nach verwendeter Optimierung

Bisher wurde gezeigt, dass durch den Einsatz von WMoCache bereits bei niedriger Latenz eine Beschleunigung der verwendeten Benchmarks erzielt werden kann. Diese geht jeweils mit einer hohen Trefferrate des Caches, die hauptsächlich durch Prefetching erreicht wurde, einher. Dies legt den Schluss nahe, dass für die Beschleunigung primär Prefetching verantwortlich ist. Eine weitere Messung soll dies verifizieren. Ge-



(a) TPC-C (b) AuctionMark

Abbildung 5.7: Dauer der Prefetchingphase mit einzelnen Optimierungen und ohne Optimierung

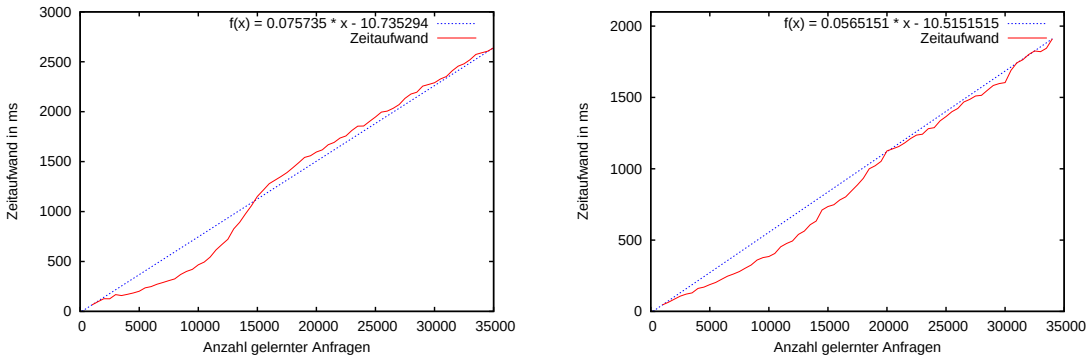
messen wurde die Dauer der Prefetchingphase ohne Caching und Prefetching sowie mit jeweils nur einer der Optimierungen. In den letzten beiden Fällen wurden 20 Cacheplätze für den jeweiligen Cache zur Verfügung gestellt. Auf Basis der in Abschnitt 5.6.1.1 präsentierten Ergebnisse wurde entschieden jedes Statement um $30ms$ zu verzögern. Abbildung 5.7 zeigt die Ergebnisse.

Bei ausschließlicher Verwendung von LRU Caching wird sowohl bei TPC-C als auch bei AuctionMark eine sehr geringe Beschleunigung gegenüber der Variante ohne Optimierung erreicht. Prefetching erreicht indes eine deutliche Verbesserung. Somit ist der Vorteil von Prefetching bestätigt.

5.6.1.4 Aufwand zur Generierung der Prefetchingstruktur

Da die Generierung der Prefetchingstruktur nach dem Ende der Lernphase die Ausführung nachfolgender Anfragen blockiert, ist es wichtig, den dafür benötigten Zeitaufwand zu minimieren. Die in Abbildung 5.8 gezeigten Graphen geben den Zeitaufwand in Abhängigkeit von der Anzahl während der Lernphase aufgezeichneter Anfragen an. Die Länge der Trainingssequenzen ist dabei konstant. Die rote durchgezogene Linie zeigt jeweils den Zeitaufwand an, während die blau gepunktete Gerade der Orientierung beim Betrachten des Graphen dient. Man erkennt, dass im Falle der in den Tabellen 5.1 und 5.2 gezeigten Basiskonfigurationen für TPC-C und AuctionMark lediglich wenige Millisekunden benötigt werden, um aus den 700 bzw. 1100 aufgezeichneten Anfragen die Prefetchingstruktur zu generieren. Es handelt sich somit um einen zu vernachlässigenden Aufwand.

Weiterhin erkennt man, dass bei TPC-C bis ca. 15'000 und bei AuctionMark bis ungefähr 20'000 gelernter Anfragen der Aufwand stärker als linear wächst, um dann in ein lineares Wachstum überzugehen. Anhand des TPC-C Benchmarks wurden die Gründe für dieses Verhalten untersucht. Eine initiale Betrachtung der Anzahl erzeugter Kandidatenpfade (vgl. Algorithmus 3.9) zeigt, dass diese bis 15'000 gelernter Anfragen stark linear zunimmt und anschließend nur noch schwach wächst (siehe Abbildung 5.9). Dies allein genügt jedoch nicht als Erklärung für das mehr



(a) TPC-C (b) AuctionMark

Abbildung 5.8: Zeitaufwand zur Generierung der Prefetchingstruktur in Abhängigkeit von der Anzahl erlernter Anfragen

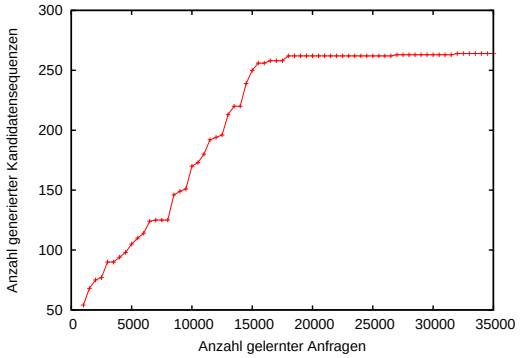


Abbildung 5.9: TPC-C: Anzahl an Kandidatenpfaden in Abhängigkeit von der Zahl erlernter Anfragen

als lineare Wachstum des Zeitaufwands zur Generierung der Prefetchingstruktur, da dieser linear in der Anzahl der Kandidatensequenzen ist.

Konzentriert man sich weiter auf die Generierung von Kandidatenpfaden der Länge $k+1$ auf Basis der Menge $L_k \cup L_k^*$ (Algorithmus 3.9), stellt man fest, dass es sich bei dieser um eine Schleife über alle häufigen Subsequenzen der Länge k aus L_k und darin verschachtelt eine weitere Schleife über alle Nachbarn (Menge $N^+(l_k)$) des letzten Knotens der aktuell betrachteten Sequenz aus L_k im Anfragegraphen handelt. Sei $n = |L_k|$ und m die Zahl der Kanten des Anfragegraphen. Dann hat die beschriebene mehrfach verschachtelte Schleife eine Laufzeit in $O(n \cdot m \cdot k)$. Wie in den Abbildungen 5.10 und 5.11 gezeigt, nimmt bis zur Grenze von 15'000 Anfragen sowohl die Zahl häufiger Subsequenzen als auch die Zahl der Kanten des Anfragegraphen stark zu, woraus das beobachtete Wachstum der Laufzeit resultiert. Danach weisen beide Größen kein Wachstum mehr auf.

Jenseits von 15'000 gelernten Anfragen wächst nur mehr die Zahl der erlernten Anfragen selbst. Der Aufwand zur Generierung der Prefetchingstruktur ist lineare in dieser Größe. Dies spiegelt sich erwartungsgemäß in den gemessenen Werten wider.

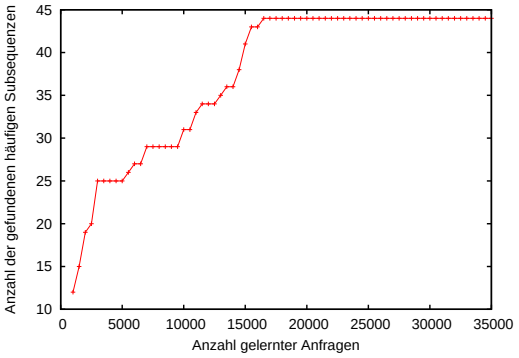


Abbildung 5.10: TPC-C: Anzahl gefundener häufiger Subsequenzen in Abhängigkeit von der Anzahl gelernter Anfragen

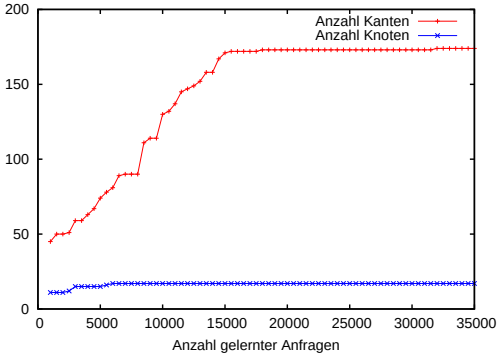
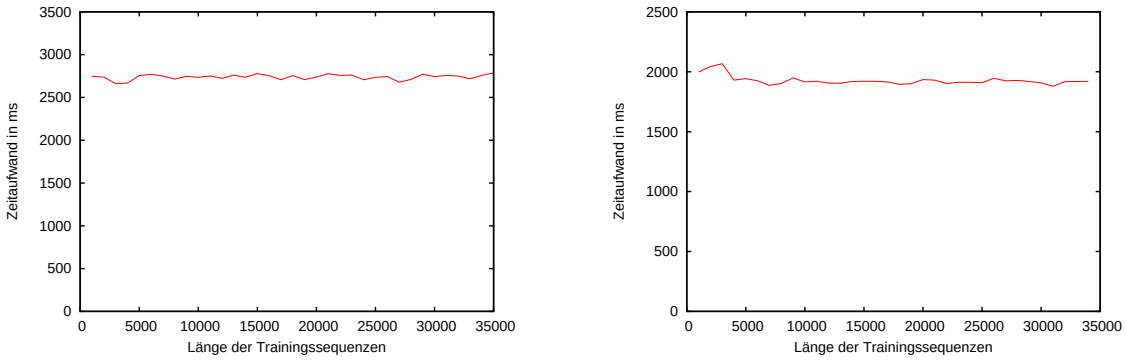


Abbildung 5.11: Anzahl der Knoten und Kanten des Anfragegraphen in Abhängigkeit von der Anzahl gelernter Anfragen



(a) TPC-C (b) AuctionMark

Abbildung 5.12: Aufwand zur Generierung der Prefetchingstruktur in Abhängigkeit von der Länge der Trainingssequenzen

Man kann weiterhin den Schluss ziehen, dass bei ansonsten gleichbleibender Konfiguration eine Verlängerung der Lernphase über 15'000 Anfragen nicht sinnvoll ist, da der Mehraufwand nicht zum Auffinden zusätzlicher Anfragemuster führt.

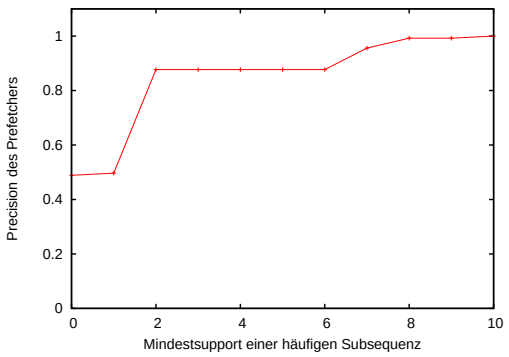
Neben der Messung des Aufwandes in Abhängigkeit von der Zahl erlernter Anfragen wurde auch der Einfluss der Länge und damit der Anzahl der Trainingssequenzen bei gleichbleibender Gesamtanzahl der aufzuzeichnenden Anfragen untersucht. Abbildung 5.12 zeigt die Ergebnisse der Messungen. Aus diesen geht hervor, dass Länge und Anzahl der Trainingssequenzen bei konstanter Gesamtanzahl der erlernten Anfragen allenfalls sehr geringen Einfluss auf den Aufwand besitzen.

5.6.2 Einfluss verschiedener Konfigurationsparameter auf das Verhalten von WMoCache

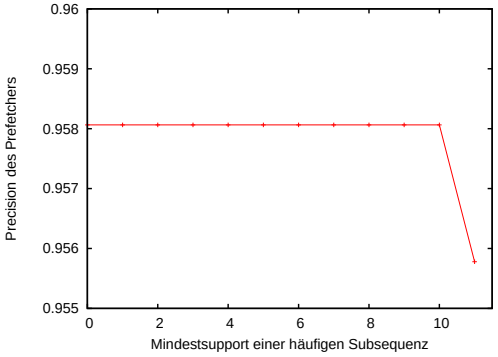
Im bisherigen Verlauf dieses Abschnitts der Arbeit lag der Fokus auf der Leistung von WMoCache und insbesondere des in Kapitel 3 entwickelten Prefetchingalgorithmus bei guter Konfiguration des Systems. Um die Einstellungen für eine bestimmte Datenbankanwendung optimieren zu können, ist es wichtig deren Einfluss auf das Verhalten von WMoCache zu kennen. Es soll daher nun die Auswirkung einiger Konfigurationsparameter auf ausgewählte Metriken analysiert werden. Begonnen wird mit dem zentralen Konfigurationsparameter des WM_o-Algorithmus, dem Mindestsupport häufiger Subsequenzen (vgl. Abschnitte 2.1 und 3.5).

5.6.2.1 Einfluss des Mindestsupports häufiger Subsequenzen

Der Parameter gibt an, in wievielen Trainingssequenzen ein Pfad des Anfragegraphen mindestens als Subsequenz vorkommen muss, um als Anfragemuster für Vorhersagen genutzt zu werden (vgl. Abschnitt 4.2). Die grundsätzliche Erwartung ist, dass ein niedriger Wert zu einer hohen Anzahl erlernter Anfragemuster führt, aber auch zu einer erhöhten Anzahl falscher Vorhersagen und damit die Präzision des Prefetchers sowie die Hitrate des Prefetchingcaches senkt. Im Gegenzug sollte ein hoher

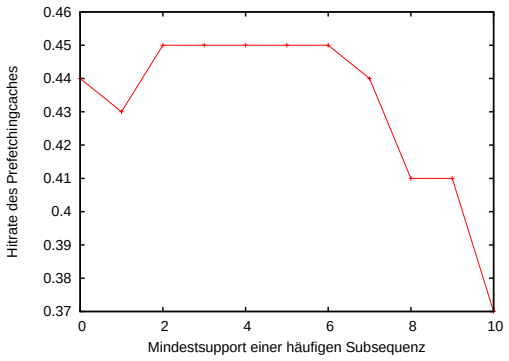


(a) TPC-C

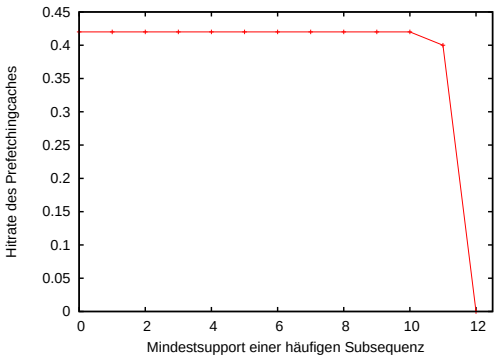


(b) AuctionMark

Abbildung 5.13: Präzision des Prefetchers in Abhängigkeit vom Mindestsupport einer häufigen Subsequenz



(a) TPC-C



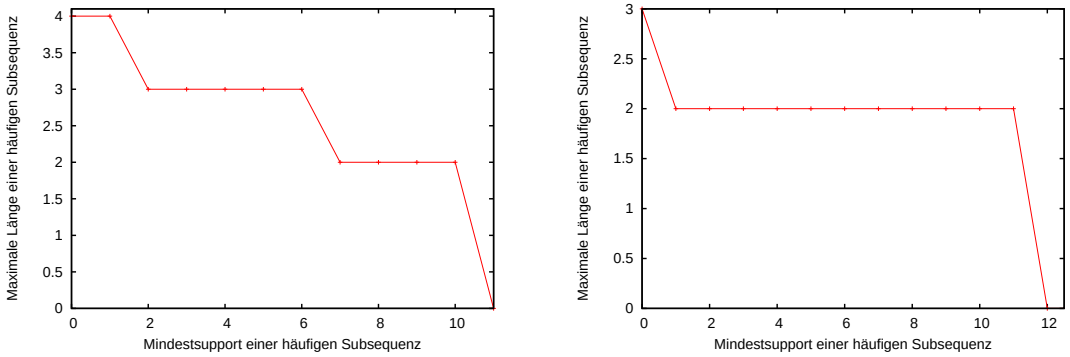
(b) AuctionMark

Abbildung 5.14: Auswirkung des Mindestsupports häufiger Subsequenzen auf die Hitrate des Prefetchingcaches

Mindestsupport das Erlernen nützlicher Anfragemuster verhindern und damit die Trefferrate des Prefetchingcaches, bzw. den Recall des Vorhersagemoduls wiederum reduzieren, aber positive Auswirkung auf die Präzision des Prefetchers haben. Es wird insgesamt erwartet, dass die Hitrate des Prefetchingcaches zunächst steigt, um anschließend wieder zu fallen.

Abbildung 5.13 zeigt die Präzision des Prefetchers in Abhängigkeit vom Mindestsupports häufiger Subsequenzen. Für den TPC-C Benchmark ist der erwartete Effekt eingetreten: Je höher der Mindestsupport ist, desto stärker wächst die Präzision des Prefetchers. Bei AuctionMark nimmt diese hingegen ab. Dies kann damit zusammenhängen, dass für gemachte Vorhersagen kein Prefetching möglich ist.

Auch die Erwartung an die Auswirkung auf die Hitrate des Prefetchingcaches ist im Falle von TPC-C eingetreten (siehe Abbildung 5.14a). Da, wie soeben gesehen, bei niedrigem Mindestsupport der Anteil falscher Vorhersagen groß ist, ist die Hitrate zunächst klein, um mit steigendem Mindestsupport zu wachsen und schließlich wie-



(a) TPC-C (b) AuctionMark

Abbildung 5.15: Länge der gefundenen Anfragemuster in Abhängigkeit von `lmMinSupport`

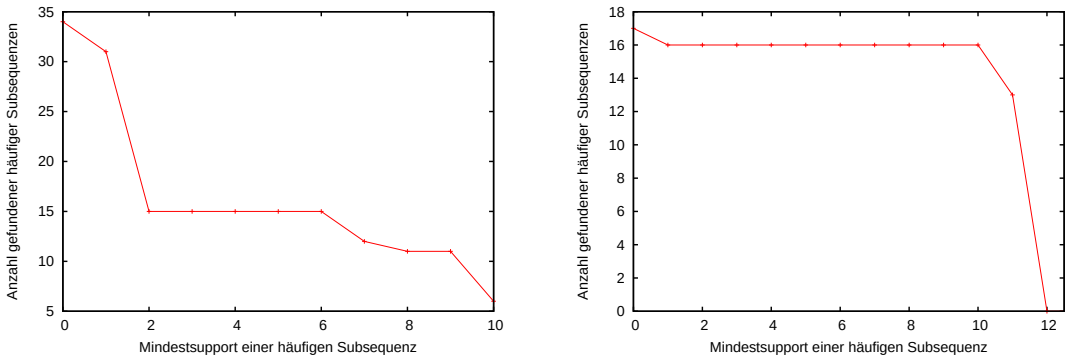
der abzunehmen. Bei AuctionMark (Abbildung 5.14b) kann zumindest beobachtet werden, dass ein zu hoher geforderter Mindestsupport Prefetching verhindert und sich damit negativ auf die Hitrate des Prefetchingcaches auswirkt.

Es empfiehlt sich also einen Mittelweg zwischen einem zu niedrigen Support, der eine große Anzahl falscher Vorhersagen begünstigt und damit zu einer ineffizienten Ressourcennutzung führt und einem zu hohen Support, der Prefetching verhindert, zu wählen.

Abschließend betrachten wir die direkte Auswirkung des Mindestsupports auf die gefundenen Anfragemuster. Der Graph 5.15 stellt die Entwicklung der Länge der gefundenen Anfragemuster dar. Je höher der Mindestsupport gewählt ist, desto kürzer sind die Sequenzen, was zu unspezifischeren Vorhersagen führt und ein weiterer Grund für die zunächst unerwartete Entwicklung der Präzision bei AuctionMark sein kann. In Abbildung 5.16 ist die Anzahl der gefundenen Anfragemuster dargestellt. Erwartungsgemäß nimmt diese mit steigendem Mindestsupport ab.

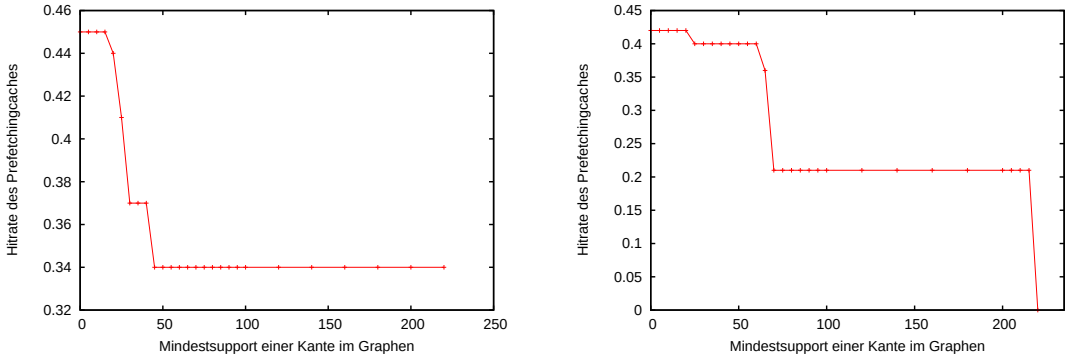
5.6.2.2 Einfluss der Mindesthäufigkeit von Kanten im Anfragegraphen

Pius Hübl [20] beobachtete, dass sich beim Einsatz des WM_o-Algorithmus für Prefetching in Key-Value-Stores die Verwendung eines vollständigen Graphen negativ auf die Hitrate des Caches auswirkt. Die Dichte des durch Algorithmus 3.2 generierten Anfragegraphen hängt maßgeblich vom Wert des Konfigurationsparameters `freqPairsGraphBuilderMinFreq` ab. Der Parameter gibt an, wie häufig ein Paar von Prepared Statements mit einem konfigurierbaren Maximalabstand im während der Lernphase aufgezeichneten Anfragestrom auftreten muss, um im Anfragegraphen durch eine Kante verbunden zu sein. Der Einfluss des Parameters auf die Hitrate des Prefetchingcaches soll untersucht werden. Man erwartet, dass mit steigendem Wert und damit abnehmender Dichte des Anfragegraphen, entsprechend der in [20] gemachten Erfahrung, die Hitrate zunächst zunimmt und für hohe Werte wieder sinkt, da die Zahl der verwendeten Anfragemuster aufgrund der nun geringen Dichte des Anfragegraphen klein ist.



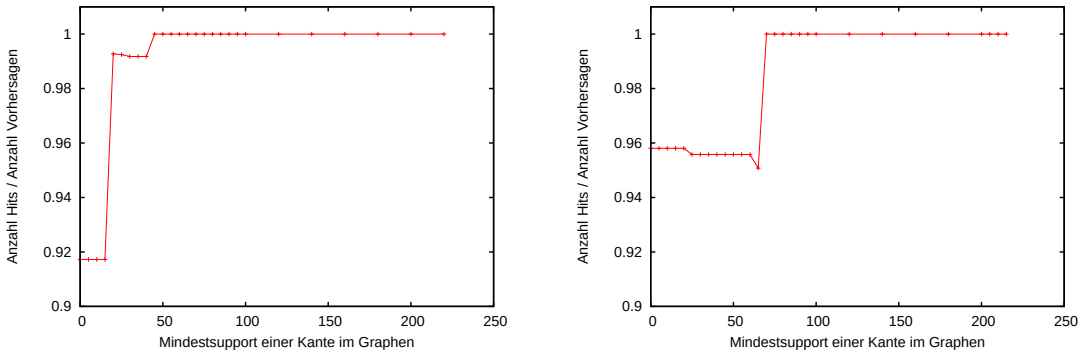
(a) TPC-C (b) AuctionMark

Abbildung 5.16: Entwicklung der Anzahl gefundener Anfragemuster in Abhängigkeit von `lmMinSupport`



(a) TPC-C (b) AuctionMark

Abbildung 5.17: Hitrate des Prefetchingcaches in Abhängigkeit von der Mindesthäufigkeit von Kanten im Anfragegraphen



(a) TPC-C (b) AuctionMark

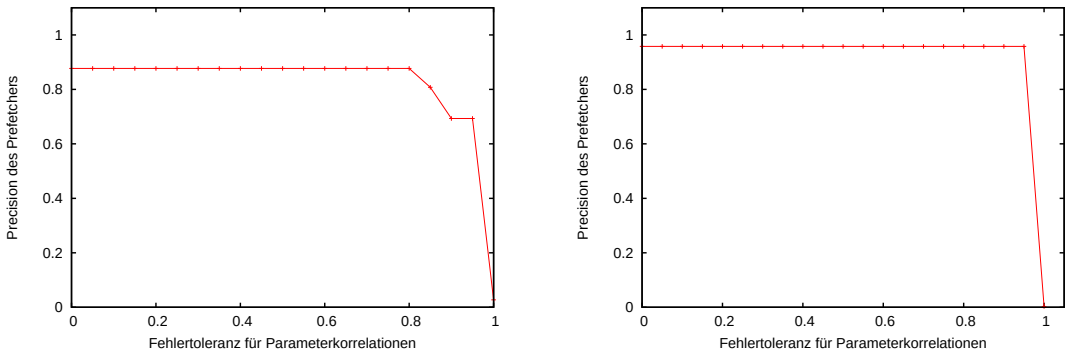
Abbildung 5.18: Verhältnis der Hitrate des Prefetchingcaches und der Anzahl gemachter Vorhersagen in Abhängigkeit von der Mindesthäufigkeit von Kanten im Anfragegraphen

Abbildung 5.17 zeigt die gemessenen Werte. Ein negativer Einfluss eines dichten Graphen auf die Hitrate kann nicht festgestellt werden. Allerdings sinkt diese wie vermutet mit zunehmendem Wert des Parameters. Ein möglichst niedriger Wert des Parameters ist jedoch nur bedingt hilfreich, da dies sich negativ auf die Präzision des Prefetchers auswirkt. Abbildung 5.18 zeigt hierzu für beide Benchmarks das Verhältnis zwischen der Anzahl der Vorhersagen und der Anzahl der Hits des Prefetchingcaches. Dieses Verhältnis steigt letztendlich mit zunehmendem Wert von `freqPairsGraphBuilderMinFreq`, oder mit anderen Worten: Die Zahl falscher Vorhersagen nimmt ab. Insgesamt muss somit auch bei diesem Parameter ein Kompromiss zwischen Hitrate und Präzision des Prefetchers gefunden werden.

5.6.2.3 Auswirkung der Fehlertoleranz für Parameterkorrelationen auf die Präzision des Prefetchers

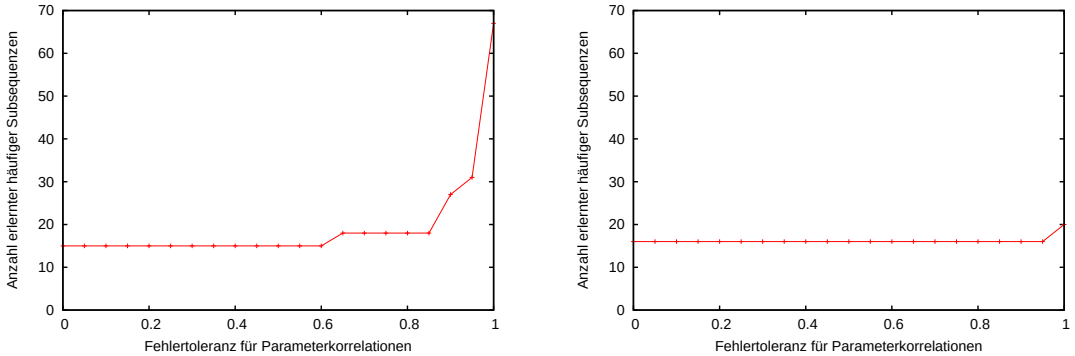
Schlussendlich wird die Auswirkung der Fehlertoleranz für Parameterkorrelationen auf die Präzision des Prefetchers untersucht. Die Erwartung ist, dass sich ein hoher Wert negativ auf die Präzision auswirkt, da vermehrt Anfragemuster ohne Parameterkorrelationen mit niedriger Fehlerrate erlernt werden.

Bei beiden Benchmarks können die erwarteten Effekte beobachtet werden (siehe Abbildungen 5.19 und 5.20). Auffällig ist, dass sich erst bei sehr hohen Werten die Präzision verschlechtert und die Zahl der erlernten häufigen Subsequenzen wächst. Dies kann mit der restriktiven Wahl der Parameter `lmCorrelationsWindowSize` und `freqPairsGraphBuilderMaxPairDist` zusammenhängen (siehe Tabellen 5.1 und 5.2). Sie bewirkt, dass nur in der näheren Umgebung von Anfragen nach Parameterkorrelationen gesucht wird. Zieht man noch die Hitrate des Prefetchingcaches hinzu (siehe Abbildung 5.21), so stellt man fest, dass die optimale Fehlertoleranz 0 ist, d.h. man sollte im Falle von TPC-C und AuctionMark fordern, dass Parameterkorrelationen in den Trainingssequenzen stets gültig sein müssen, um als zutreffend eingestuft zu werden.



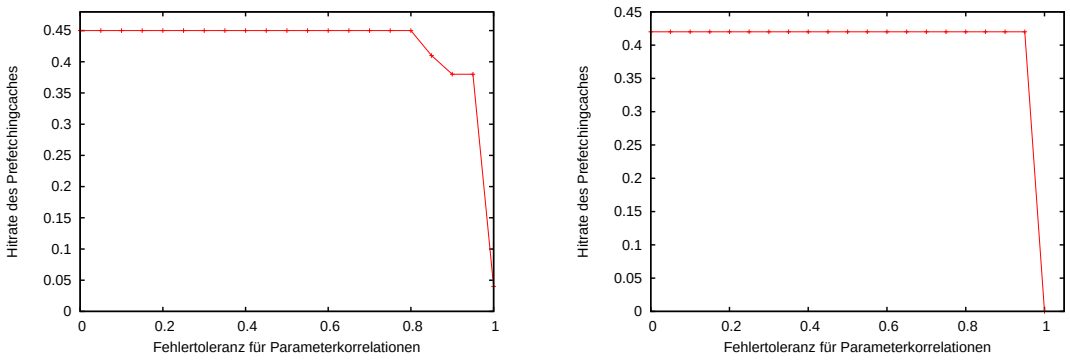
(a) TPC-C (b) AuctionMark

Abbildung 5.19: Präzision des Vorhersagemoduls in Abhängigkeit von der Fehlertoleranz für Parameterkorrelationen



(a) TPC-C (b) AuctionMark

Abbildung 5.20: Anzahl gefundener Anfragemuster in Abhängigkeit von der Fehlertoleranz für Parameterkorrelationen



(a) TPC-C

(b) AuctionMark

Abbildung 5.21: Hitrate des Prefetchingcaches in Abhängigkeit von der Fehlertoleranz für Parameterkorrelationen

5.7 Fazit

Es konnte gezeigt werden, dass die OLTP-Benchmarks TPC-C und AuctionMark durch WMoCache signifikant beschleunigt werden können. Eine Evaluation mit produktiv eingesetzten Datenbankanwendungen wurde bisher nicht durchgeführt. Die in Abschnitt 5.6.2 untersuchten Konfigurationsparameter von WMOCache haben einen starken Einfluss auf die Effektivität des Verfahrens. Vor einem Einsatz sollte daher stets eine anwendungsspezifische Optimierung der Einstellungen erfolgen.

Schluss

6 Schluss

Der WM_o-Algorithmus konnte erfolgreich für das Prefetching von SQL-Anfragen angepasst werden. Dem Verfahren wurde eine initiale Lernphase zur Erfassung von Trainingssequenzen hinzugefügt. Aus diesen wird durch ein approximatives Verfahren ein Ersatz für den im WM_o-Algorithmus verwendeten Webgraphen generiert. Analog zum Basisalgorithmus handelt es sich bei dem neuen Verfahren weiterhin um einen Black-Box-Ansatz. Die zu Beginn der Arbeit beschriebene Eigenschaft von WM_o, dass auch auf Basis jeder Subsequenz eines erlernten Anfragemusters Vorhersagen gemacht werden können, musste im Zuge der Anpassung an SQL aufgegeben werden.

Der WM_o-Algorithmus für das Prefetching von SQL-Anfragen wurde in Form der Middlewarekomponente WMoCache implementiert. Durch diese kann eine Datenbankanwendung mit einer Datenbank interagieren. Dabei kommen transparent für die Anwendung Prefetching und Caching zum Einsatz.

Die Evaluation konnte zeigen, dass WMoCache in der Lage ist die Ausführung der OLTP-Benchmarks TPC-C und AuctionMark signifikant zu beschleunigen. Prefetching ist dabei die primäre Ursache der erreichten Zeiteinsparung. WMoCache besitzt eine größere Anzahl am Konfigurationsparametern. Für eine Reihe von Parametern wurde erkannt, dass sie das Leistungsverhalten von WMoCache stark beeinflussen. Somit sollte die Konfiguration vor jedem Einsatz von WMoCache an die jeweilige Datenbankanwendung angepasst werden. Abschnitt 4.2 gibt Anhaltspunkte zur Erstellung einer passenden Ausgangskonfiguration.

Insgesamt wird festgestellt, dass WM_o neben den Einsatzbereichen Web [1] und Key-Value-Stores [20] mit Anpassungen auch im Bereich relationaler Datenbanken gewinnbringend angewandt werden kann.

6.1 Ausblick

WMoCache wurde bisher nicht an ähnlichen Ansätzen wie beispielsweise Scalpel [4] gemessen.

Auch Update-Statements sind parametrisiert und können für Vorhersagen genutzt werden. Sie müssten hierfür in den Trie aufgenommen werden, dürften jedoch selbst nicht vorhergesagt werden. Durch eine modifizierte Breitensuche in den jeweiligen Unterbäumen könnte bestimmt werden welche Anfragen im Falle einer Vorhersage stattdessen auszuführen wären.

Wie bereits in Abschnitt 3.5 angedeutet, ist WMoCache derzeit nicht in der Lage eine Reihe potentiell interessanter Anfragemuster zu erlernen und für Prefetching zu

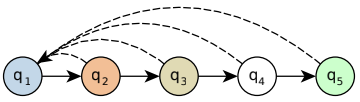


Abbildung 6.1: Beispiel für ein Anfragemuster, das WMoCache nicht finden kann. Bei jeder der Anfragen q_2, q_3, q_4, q_5 existiere jeweils mindestens ein Parameter, dessen Wert ausschließlich aus q_1 bestimmt werden kann.

nutzen. Die Ursache liegt im Umgang der Algorithmen 3.3 (Generierung der Prefetchingstruktur) und 3.9 (Generierung von Kandidatenpfaden der Länge $k+1$) mit häufigen Subsequenzen der Trainingssequenzen, für welche nicht genügend zuverlässige Parameterkorrelationen gefunden werden konnte. Ein Beispiel einer betroffenen Anfragesequenz ist in Abbildung 6.1 dargestellt. Die Sequenz besteht aus fünf Anfragen. Bei jeder der Anfragen q_2, q_3, q_4, q_5 existiert jeweils mindestens ein Parameter, dessen Wert ausschließlich aus q_1 bestimmt werden kann. Veranschaulicht man sich das Verhalten der Algorithmen 3.3 und 3.9 in dieser Situation, so erkennt man, dass die Sequenz $\langle q_1, q_2, ..., q_5 \rangle$ durch Algorithmus 3.9 nicht als Kandidat für C_5 generiert wird, da $\langle q_2, q_3, q_4, q_5 \rangle$ nicht in $L_4 \cup L_4^*$ enthalten sein kann. Dies resultiert analog aus der Tatsache, dass u.a. $\langle q_2, q_3, q_4 \rangle$ nicht in L_3 enthalten ist. Würde man entgegen der Vorgehensweise in Algorithmus 3.9 auch Sequenzen aus L_k^* anhand des Anfragegraphen verlängern und die resultierenden Kandidatenpfade, sofern sie das Subpfadkriterium erfüllen und ausreichend Support besitzen, in L_{k+1}^* einfügen, so wäre das Problem gelöst. Dies würde jedoch eine weitere Abschwächung der ursprünglichen Eigenschaft des WM_o-Algorithmus, dass jede Subsequenz eines erlernten Anfragemusters ebenfalls als Anfragemuster für Prefetching genutzt werden kann, bedeuten. Folglich wurde auf das Erlernen derartiger Anfragemuster vorerst verzichtet. Weitergehende Untersuchungen sollten die Nützlichkeit für die verwendeten Benchmarks überprüfen.

Derzeit wird pro durch eine Anwendung gestellter Anfrage maximal eine vorhergesagte Anfrage vorzeitig abgerufen. Es ist denkbar, dass insbesondere bei großer Bandbreite und eher geringer Auslastung des Datenbankservers ein größerer Speed-Up erreicht werden könnte, wenn jedes Mal eine größere Anzahl an Prefetchingkandidaten ausgewählt würde. Insbesondere, da im Falle von TPC-C und AuctionMark mit der aktuellen Prefetchingstrategie nur ein sehr kleiner Prefetchingcache benötigt wird, erscheint aggressiveres Prefetching möglich. In der Quelle [7] wird untersucht wie unter Verwendung verschiedener Strategien zum kombinierten Ausführen mehrerer vorhergesagter SQL-Anfragen die Zahl einzelner Kommunikationsvorgänge mit dem Datenbankserver reduziert werden kann und gleichzeitig Latenz und Datenfluss optimiert werden können. Es müsste untersucht werden, ob die Kombination dieser Strategien mit dem durch WMoCache erstellten Präfixbaum einen Gewinn gegenüber der bisherigen Verwendung asynchronen Prefetchings und der Lateral-Outer-Union-Strategie ermöglicht. Auch im Falle der bisherigen Strategie ist offen, ob die gewählte Vorgehensweise mit asynchronem Prefetching und der Verwendung von Lateral-Outer-Union bei Bedarf optimal ist. Weitere Strategien zum kombinierten Ausführen (siehe z.B. [4] und [7]) von Anfragen sollten getestet werden. Zudem kann häufigeres kombiniertes Ausführen gewinnbringend sein. Auch der Einfluss der

Umschreibung von `LATERAL` durch temporäre Views (vgl. Abschnitt 3.7) ist ungeklärt.

Bei den Messungen für die Evaluation wurde stets LRU als Verdrängungsstrategie für den Anfragecache und FIFO für den Prefetchingcache verwendet. Diese Wahl ist typisch. Dennoch wäre interessant, welchen Effekt beispielsweise LFU (Least Frequently Used) anstelle von LRU bewirkt.

Die Arbeit hat bereits gezeigt, dass die Leistung eines Anfragecaches mit LRU als Verdrängungsstrategie durch den vorgestellten WM_o-Algorithmus für das Prefetching von SQL-Anfragen deutlich gesteigert werden kann. Interessant wäre, ob auch die Leistung eines semantischen Caches wie IQCache verbessert werden könnte. Man würde eine vorhergesagte Anfrage, statt deren Enthaltensein im Cache zu überprüfen und gegebenenfalls anschließend Prefetching durchzuführen, zunächst an den Cache übergeben und ein Rewriting der Anfrage durchführen lassen, sodass diese nur noch Tupel selektiert, die noch nicht im lokalen Cache enthalten sind. Anschließend würde die neue Anfrage ausgeführt.

Anhang

Literaturverzeichnis

[1] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos, "A Data Mining Algorithm for Generalized Web Prefetching," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 5, pp. 1155–1169, 2003.

[2] I. T. Bowman and K. Salem, "Optimization of Query Streams Using Semantic Prefetching," *ACM Trans. Database Syst.*, vol. 30, no. 4, pp. 1056–1101, 2005.

[3] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," in *FAST*, pp. 173–186, USENIX, 2004.

[4] I. Bowman, *Scalpel: Optimizing Query Streams Using Semantic Prefetching*. Thesis or dissertation, University of Waterloo, School of Computer Science, 2005.

[5] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos, "Exploiting Web Log Mining for Web Cache Enhancement," in *WEBKDD* (R. Kohavi, B. M. Masand, M. Spiliopoulou, and J. Srivastava, eds.), vol. 2356 of *Lecture Notes in Computer Science*, pp. 68–87, Springer, 2001.

[6] V. N. Padmanabhan and J. C. Mogul, "Using Predictive Prefetching to Improve World Wide Web Latency," *SIGCOMM Comput. Commun. Rev.*, vol. 26, pp. 22–36, July 1996.

[7] A. S. Bilgin, *Deriving Efficient SQL Sequences Via Prefetching*. Phd thesis, North Carolina State University, Raleigh, North Carolina, 2007.

[8] A. Bilgin, R. Chirkova, T. Salo, and M. Singh, "Deriving Efficient SQL Sequences via Read-Aheads," in *Data Warehousing and Knowledge Discovery* (Y. Kambayashi, M. Mohania, and W. Wöß, eds.), vol. 3181 of *Lecture Notes in Computer Science*, pp. 299–308, Springer Berlin Heidelberg, 2004.

[9] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-Aware Prefetching at the Storage Server ," in *USENIX Annual Technical Conference* (R. Isaacs and Y. Zhou, eds.), pp. 377–390, USENIX Association, 2008.

[10] Z. Ban, Z. Gu, and Y. Jin, "An Online PPM Prediction Model for Web Prefetching," in *Proceedings of the 9th annual ACM international workshop on Web information and data management*, WIDM '07, (New York, NY, USA), pp. 89–96, ACM, 2007.

[11] S.-W. Kang, J. Kim, S. Im, H. Jung, and C.-S. Hwang, "Cache Strategies for Semantic Prefetching Data," in *Web-Age Information Management Workshops, 2006. WAIM '06. Seventh International Conference on*, pp. 7–7, 2006.

[12] J. Griffioen and R. Appleton, "Reducing File System Latency using a Predictive Approach," in *USENIX Summer*, pp. 197–207, 1994.

[13] C. A. Gerlhof and A. Kemper, "A Multi-Threaded Architecture for Prefetching in Object Bases," in *EDBT* (M. Jarke, J. A. B. Jr., and K. G. Jeffery, eds.), vol. 779 of *Lecture Notes in Computer Science*, pp. 351–364, Springer, 1994.

[14] P. A. Bernstein, S. Pal, and D. Shutt, "Context-Based Prefetch for Implementing Objects on Relations," in *VLDB* (M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, eds.), pp. 327–338, Morgan Kaufmann, 1999.

[15] M. Palmer and S. B. Zdonik, "Fido: A Cache That Learns to Fetch," in *VLDB* (G. M. Lohman, A. Sernadas, and R. Camps, eds.), pp. 255–264, Morgan Kaufmann, 1991.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *SOSP*, pp. 79–95, 1995.

[17] D. M. Huizinga and S. Desai, "Implementation of Informed Prefetching and Caching in Linux," *Information Technology: Coding and Computing, International Conference on*, vol. 0, p. 443, 2000.

[18] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 252–263, ACM, 1997.

[19] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and Control in Gray-box Systems," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, (New York, NY, USA), pp. 43–56, ACM, 2001.

[20] P. Huebl, "Prefetching für Key-Value-Stores," master's thesis, University of Passau, Jan. 2013.

[21] A. Nanopoulos and Y. Manolopoulos, "Mining Patterns from Graph Traversals," *Data Knowl. Eng.*, vol. 37, pp. 243–266, Aug. 2001.

[22] K. S. Ivan T. Bowman, "Semantic Prefetching of Correlated Query Sequences," Technical Report CS-2006-43, David R. Cheriton School of Computer Science, University of Waterloo, Nov. 2006.

[23] I. Bowman and K. Salem, "Semantic Prefetching of Correlated Query Sequences," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pp. 1284–1288, 2007.

[24] "Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)," 1999.

[25] H. S. Jeon and S. H. Noh, "A Database Disk Buffer Management Algorithm Based on Prefetching," in *CIKM* (G. Gardarin, J. C. French, N. Pissinou, K. Makki, and L. Bouganim, eds.), pp. 167–174, ACM, 1998.

[26] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," in Buneman and Jajodia [66], pp. 297–306.

[27] D. F. Faria and C. S. Lytle, "Programmable Logic Array Integrated Circuit Incorporating a First-In First-Out Memory," Oct. 1996.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, ch. 5: Behavioral Patterns, pp. 331–344. Addison-Wesley Publishing Company, 1995.

[29] C. R. David Jordan, *Java Data Objects*. O'Reilly Media, Inc., Apr. 2003.

[30] "Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI)," 2003.

[31] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas, "PocketWeb: instant web browsing for mobile devices," *SIGARCH Comput. Archit. News*, vol. 40, pp. 1–12, Mar. 2012.

[32] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald, "Efficiently publishing relational data as XML documents," *VLDB J.*, vol. 10, no. 2-3, pp. 133–154, 2001.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, ch. 3: Creational Patterns, pp. 81–136. Addison-Wesley Publishing Company, 1995.

[34] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, "Benchmarking OLTP/Web Databases in the Cloud: The OLTP-bench Framework," in *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, (New York, NY, USA), pp. 17–20, ACM, 2012.

[35] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013.

[36] A. Kemper and A. Eickler, *Datenbanksysteme*. Oldenbourg Verlag, 9 ed., 2013.

[37] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Databases," in *SDM* (D. Barbará and C. Kamath, eds.), SIAM, 2003.

[38] J. Stephens, "TPC BENCHMARK ™H," 2013.

[39] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak, "The Star Schema Benchmark and Augmented Fact Table Indexing," in *TPCTC* (R. O. Nambiar and M. Poess, eds.), vol. 5895 of *Lecture Notes in Computer Science*, pp. 237–252, Springer, 2009.

[40] P. O'Neil, E. O'Neil, and X. Chen, "Star Schema Benchmark," tech. rep., University of Massachusetts at Boston, 2009.

[41] F. Raab, "TPC BENCHMARK ™C," Feb. 2010.

[42] V. Angkanawaraphan and A. Pavlo, "AuctionMark: An OLTP Benchmark for Shared-Nothing Database Management Systems." webpage. http://hstore.cs.brown.edu/projects/auctionmark/.

[43] V. Angkanawaraphan, A. Pavlo, R. Shoup, and S. Zdonik, "AuctionMark: A Stored Procedured-based OLTP Auction Site Benchmark." https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/auctionmark/docs, 2012.

[44] S. T. Leutenegger and D. M. Dias, "A Modeling Study of the TPC-C Benchmark," in Buneman and Jajodia [66], pp. 22–31.

[45] F. Raab, W. Kohler, and A. Shah, "Overview of the TPC Benchmark C: The Order-Entry Benchmark," tech. rep., Transaction Processing Performance Council, San Francisco, ?

[46] M. R. de Lima, M. S. Sunyé, E. C. de Almeida, and A. I. Direne, "Distributed Benchmarking of Relational Database Systems," in *APWeb/WAIM* (Q. Li, L. Feng, J. Pei, X. S. Wang, X. Zhou, and Q.-M. Zhu, eds.), vol. 5446 of *Lecture Notes in Computer Science*, pp. 544–549, Springer, 2009.

[47] A. Macdonald, *The Architecture of an Autonomic, Resource-Aware, Workstation-Based Distributed Database System.* PhD thesis, University of St Andrews, 2012.

[48] D. R. Llanos, "TPCC-UVa: An Open-source TPC-C Implementation for Global Performance Measurement of Computer Systems," *SIGMOD Rec.*, vol. 35, pp. 6–15, Dec. 2006.

[49] D. Llanos and B. Palop, "TPCC-UVa: An Open-Source TPC-C Implementation for Parallel and Distributed Systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 8 pp.–, April 2006.

[50] E. Whalen, D. Simons, D. Lindauer, and C. Levine, "Why You Should Look at TPC-C First. An Interview between TPC-C Subcommittee members and Kim Shanley, TPC Chief Operating Officer," Oct. 1996.

[51] S.-K. Cheong, C. Lim, and B.-C. Cho, "Database processing performance and energy efficiency evaluation of DDR-SSD and HDD storage system based on the TPC-C," in *Cloud Computing and Social Networking (ICCCSN), 2012 International Conference on*, pp. 1–3, April 2012.

[52] H. Jung, H. Han, S.-g. Kim, and H. Y. Yeom, "A Practical Evaluation of Large-memory Data Processing on a Reliable Remote Memory System," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, (New York, NY, USA), pp. 343–344, ACM, 2009.

[53] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom, "Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, (San Jose, CA), USENIX, 2013.

[54] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *INFOCOM*, pp. 126–134, 1999.

[55] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 61–72, ACM, 2012.

[56] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an Architectural Era: (It's Time for a Complete Rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pp. 1150–1160, VLDB Endowment, 2007.

[57] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a High-Performance, Distributed Main Memory Transaction Processing System," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, 2008.

[58] E. P. Jones, D. J. Abadi, and S. Madden, "Low Overhead Concurrency Control for Partitioned Main Memory Databases," in *SIGMOD '10: Proceedings of the 2010 International Conference on Management of Data*, (New York, NY, USA), pp. 603–614, ACM, 2010.

[59] A. Pavlo, E. P. C. Jones, and S. Zdonik, "On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems," *Proc. VLDB Endow.*, vol. 5, pp. 85–96, Oct. 2011.

[60] F. R. C. Sousa and J. C. Machado, "Towards Elastic Multi-Tenant Database Replication with Quality of Service," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, (Washington, DC, USA), pp. 168–175, IEEE Computer Society, 2012.

[61] K. M. Kavi, S. Pianelli, G. Pisano, G. Regina, and M. Ignatowski, "3D DRAM and PCMs in Processor Memory Hierarchy," in *ARCS* (E. Maehle, K. Römer, W. Karl, and E. Tovar, eds.), vol. 8350 of *Lecture Notes in Computer Science*, pp. 183–195, Springer, 2014.

[62] A. Tatarowicz, C. Curino, E. Jones, and S. Madden, "Lookup Tables: Fine-Grained Partitioning for Distributed Databases," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pp. 102–113, 2012.

[63] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational Cloud: A Database-as-a-Service for the Cloud," in *5th Biennial Conference on Innovative Data Systems Research,*, 2011.

[64] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A Workload-driven Approach to Database Replication and Partitioning," *Proc. VLDB Endow.*, vol. 3, pp. 48–57, Sept. 2010.

[65] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 85–100, ACM, 2011.

[66] P. Buneman and S. Jajodia, eds., *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, ACM Press, 1993.

Tabellenverzeichnis

Abbildungsverzeichnis

Liste der Algorithmen

Listings

Abkürzungsverzeichnis

$\preceq$ S ist eine Subsequenz von P., Seite 10

C_k Durch den WM_o-Algorithmus auf Level k betrachtete Menge an Kandidatenpfaden, Seite 12

f Einer Parameterkorrelation zugeordneter Zähler, der angibt in wievielen Subsequenzen die Korrelation unzutreffend war., Seite 44

h Einer Parameterkorrelation zugeordneter Zähler, der angibt in wievielen Subsequenzen die Korrelation zutreffend war., Seite 44

L_k Im WM_o-Algorithmus die Menge der häufigen Subsequenzen mit Länge k, Seite 12

$N^+(l_k)$ Im WM_o-Algorithmus die Menge der im Webgraphen vom Knoten l_k direkt über eine Kante erreichbaren Knoten., Seite 12

AST abstrakter Syntaxbaum, Seite 23

CFG Control Flow Graph (deutsch: Kontrollflussgraph), Seite 38

FIFO First In First Out Verdrängungsstrategie, Seite 22

LFU Least Frequently Used Verdrängungsstrategie, Seite 121

LRU Least Recently Used Verdrängungsstrategie, Seite 22

OLAP *Online Analytical Processing*, Seite 93

OLTP Echtzeit-Transaktionsverarbeitung, oder auch *Online Transaction Processing*, Seite 93

RTT Round Trip Time, Seite 4

TPC Transaction Processing Performance Council, Seite 99

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, den 16. April 2014

Stefan Ganser