



Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Masterarbeit

Kohärentes Web-Caching von Datenbankobjekten im Cloud Computing

Felix Gessert

fgessert@informatik.uni-hamburg.de
Studiengang Informatik (MSc)
Matr.-Nr. 5945597
Fachsemester 4

Florian Bücklers

fbueckle@informatik.uni-hamburg.de
Studiengang Informatik (MSc)
Matr.-Nr. 5991327
Fachsemester 4

Erstgutachter: Professor N. Ritter
Zweitgutachter: Professor W. Lamersdorf

Kohärentes Web-Caching von Datenbankobjekten im Cloud Computing

Felix Gessert, Florian Bücklers

Abstract. Die Wissenschaft und Praxis von Datenbanksystemen befindet sich in einem der bisher radikalsten Umbrüche seiner Geschichte. Drei Paradigmenwechsel sind eindeutig identifizierbar: der Wechsel von dem dominierenden relationalen Datenmodell zu domänenspezifischen, anforderungsabhängigen Datenmodellen und Datenbanken („NoSQL“, „Polyglot Persistence“), die Ausrichtung vieler Datenbanken auf Technologien des Webs und seine Größenordnungen („Horizontal Scalability“, „Big Data“) und die zunehmende Verschiebung von Datenverarbeitung und -speicherung auf Cloud Computing Plattformen. Um diesem Paradigmenwechsel technisch angemessen zu begegnen, sind viele Neuerungen notwendig. In dieser Arbeit beschreiben wir, wie sich für Datenbanken auf Basis von Web-Techniken, Cloud Computing und etablierten Algorithmen horizontale Skalierbarkeit und niedrige Netzwerklatenzen durch Web Caching erzielen lassen. Die Herausforderung und Originalität unseres Ansatzes besteht dabei in der Anforderung, die Konsistenz und Transaktionalität des Datenbankzugriffs nicht einzuschränken. Die praktische und empirisch evaluierbare Umsetzung erfolgt in dem System ORESTES (Objects RESTfully Encapsulated in Standard Formats). Konkret zeigen wir, wie mit Hilfe von Bloomfiltern (einer probabilistischen Datenstruktur für Mengen) sichergestellt werden kann, dass Clients stets aktuelle Daten aus Web-Caches lesen. Wir belegen dabei statistisch die höhere Leistungsfähigkeit unserer Bloomfilter Implementierung in mehreren Varianten gegenüber existierenden Umsetzungen. Darüber hinaus untersuchen und implementieren wir die Nutzung von Content Delivery Networks und Reverse Proxy Caches als Caching Schicht für Datenbankobjekte. Aus der Kombination von ORESTES mit den neuen Techniken der Cache Kohärenz ergibt sich, wie wir zeigen werden, ein Datenbankzugriffsprotokoll, das die Skalierungsmechanismen des Webs (wie Load Balancing und Web Caching) in Einklang mit den Anforderungen an moderne Datenbanksysteme zu nutzen vermag.



Inhaltsverzeichnis

1	Einführung	1
1.1	ORESTES	9
1.1.1	Motivation für ein cloudfähiges Persistenzprotokoll	9
1.1.2	Konzept.....	11
1.1.2.1	Das ORESTES REST/HTTP Protokoll.....	11
1.1.2.2	Skalierungsmodell.....	13
1.1.2.3	Implementierung.....	22
1.2	Cloud Computing.....	24
1.2.1	Database-as-a-Service	27
1.2.2	Analyse des DBaaS Ökosystems	31
1.2.3	Benchmarkergebnisse für ORESTES in einem Cloud Szenario.....	35
1.2.4	Problematik der Cache Kohärenz	38
2	Related Work.....	40
3	Kohärentes Web-Caching.....	43
3.1	ACID Transaktionen ohne Cache Kohärenz	46
3.1.1	Serialization Graph Testing (SGT)	52
3.1.2	Timestamp Ordering.....	53
3.1.3	FOCC und BOCC.....	54
3.1.4	BOCC+ und Optimistic Locking.....	57
3.1.5	Zusammenfassung	60
3.2	Grundkonzept.....	61
3.2.1	Sortierte Listen	62
3.2.2	Balancierte Suchbäume.....	62
3.2.3	Hash-Tabellen	63
3.3	Bloomfilter und ihre mathematischen Eigenschaften	65
3.4	Anwendungen	74
3.4.1	Summary Cache.....	74
3.4.2	BigTable	76
3.5	Varianten	78
3.5.1	Counting Bloom Filter	78
3.5.2	Materialized Counting Bloomfilter	81

3.5.3	Bitwise Bloomfilter	82
3.5.4	Spectral Bloomfilter	82
3.5.5	Bloomier Filter	84
3.5.6	Stable Bloomfiler.....	86
3.5.7	A ² Bloomfilter	86
3.6	Existierende Implementierungen.....	87
3.6.1	Bloomfilter von Magnus Skjogstad	87
3.6.2	Bloomfilter von Ilya Grigorik	87
3.7	Implementierung.....	88
3.7.1	Die Redis Bloomfilter	91
3.7.1.1	Einfacher Redis Bloomfilter	93
3.7.1.2	Redis Bloomfilter mit Bit-Array Countern.....	95
3.7.1.3	Redis Bloomfilter mit expliziten Countern	98
3.7.1.4	Vergleich	100
3.7.2	Hashing.....	102
3.7.2.1	Kryptographische Hashfunktionen	107
3.7.2.2	Nicht-kryptographische Hashfunktionen.....	109
3.7.2.3	Zufallsgeneratoren	109
3.7.2.4	Universelle Hashfunktionen	110
3.7.2.5	Prüfsummen.....	111
3.7.2.6	Hash Qualität	111
3.7.2.7	Performance Aspekte.....	119
3.8	Bloomfilter als komprimiertes Write-Log Fenster	120
3.9	Evaluation.....	123
3.10	Zusammenfassung	125
4	Cache Invalidierungen.....	129
4.1	Reverse-Proxy-Cache: Varnish	129
4.2	Content-Delivery-Networks	134
4.2.1	CDN Aufbau	134
4.2.2	Server.....	135
4.2.3	Struktur	135
4.2.4	Interaktions Protokolle	135



4.2.5	Content.....	137
4.2.6	Verteilung.....	137
4.2.7	Auswahl des Contents.....	138
4.2.8	Aktualisierung Strategie.....	138
4.3	Netzwerklatenz verschiedener CDNs.....	139
4.4	Amazon CloudFront.....	140
4.4.1	Infrastruktur.....	140
4.4.2	Unterstütze Content-Typen.....	141
4.4.3	Cache-Verhalten und Aktualisierung.....	141
4.4.4	Invalidation-API.....	143
4.4.5	Evaluation.....	143
4.5	Der Orestes Invalidation-Service.....	144
4.6	Web-Caching und CDNs.....	145
4.7	Evaluation.....	149
4.8	Ausblick.....	152
5	Polyglot Persistence.....	156
5.1	Umsetzung von Polyglot Persistence: Redis.....	157
5.2	Umsetzung neuer Persistenz APIs: JSPA.....	161
5.2.1	Java Persistence API (JPA).....	162
5.2.2	Einführung in JSPA.....	163
5.2.3	Architektur.....	165
5.2.4	Synchronisierung der Operationen über die Queue.....	167
6	Future Work.....	170
7	Schluss.....	174
A	Abbildungsverzeichnis.....	176
B	Quellen und Referenzen.....	180
C	Arbeitsaufteilung.....	192
D	Erklärung.....	193

1 Einführung

Gleich mehrere Umbrüche vollziehen sich derzeit bei Datenbanksystemen und der gesamten Wissenschaft und Praxis, die sich mit ihnen beschäftigt. Über etwa drei Jahrzehnte dominierten relationale Datenbanken den Markt nahezu uneingeschränkt [1]. Doch mit der ebenso dominierenden Position des Webs und den exponentiell anwachsenden Datenmengen (cf. [2]) verschoben sich auch die Anforderungen zunehmend. Die Bandbreite zum Teil widerstreitender Anforderungen an ein modernes Datenbanksystem speist sich nach unserer Einschätzung aus vier Kategorien: NoSQL [3], Big Data [4], Cloud Computing [2] und klassischer Datenbanktheorie [5]. Wir wollen in dieser Arbeit zeigen, dass sich ein Bogen zwischen ihnen spannen lässt und die notwendigen technischen Voraussetzungen dafür schaffen. Um dieses ehrgeizige Ziel realisierbar zu machen, ist es unbedingt erforderlich, auf zahlreiche vorhandene und etablierte Implementierungen und Forschungsergebnisse zurückzugreifen.

Wir bauen in dieser Arbeit auf einem System auf, das wir erstmals in unserer Bachelorarbeit [6] vorgestellt haben: ORESTES (Objects RESTfully Encapsulated in Standard Formats) [7]. Es handelt sich dabei um ein REST/HTTP Protokoll und Framework für transaktionalen, objektorientierten Zugriff auf Datenbanken. Eine entscheidende Fähigkeit des Protokolls ist die Nutzung von Web-Infrastruktur, d.h. Web-Caches und Load-Balancern, als Skalierungsmechanismus für das Datenbanksystem. Ermöglicht wird dies durch den Mechanismus der sogenannten optimistischen Nebenläufigkeitskontrolle [8]: zum Zeitpunkt eines Transaktionscommits wird validiert, dass der Client keinerlei veraltete Objektkopien aus Web-Caches gelesen hat. Trotz der Performancevorteile durch die Nutzung von Web-Caches entstehen zwei Nachteile, die in dieser Arbeit gelöst werden sollen. Dies sind einerseits Transaktionsabbrüche als Ergebnis von veralteten Leseergebnissen aus Web-Caches und andererseits die fehlende Nutzbarkeit serverseitig kontrollierbarer Cache Verbunde (CDNs und Reverse Proxy Caches). Außerdem soll gezeigt werden, dass die erarbeiteten Techniken nicht auf das objektorientierte Datenmodell und entsprechende Datenbanken beschränkt sind, sondern auf andere Datenmodelle und Datenbanksysteme übertragen werden können. Durch die Synthese von ORESTES mit den neuen Techniken zur Cache-Kohärenz wollen wir den gewandelten Anforderungen an moderne Datenbanksysteme begegnen, die wir nun darlegen werden.

Die NoSQL Bewegung hat rückblickend zwei primäre Beweggründe [3], [9], [10]:

- Die Erkenntnis, dass einfach nutzbare Datenmodelle und Schnittstellen die Produktivität der Anwendungsentwicklung erheblich verbessern können
- Die Notwendigkeit, Datenbanken auf Clustern auszuführen, um Last und Datenvolumen horizontal zu skalieren

Aus dem Erfolg der NoSQL Bewegung leiten wir deshalb zwei notwendige Forderungen ab: die Ermöglichung horizontaler Skalierbarkeit (I) und eine transparente und intuitive Daten-



bankschnittstelle (II). Allen Vertretern aus dem breiten Spektrum der NoSQL Lösungen ist gemeinsam, dass sie keinen Alleinvertretungsanspruch als allgemeingültige Datenlösung erheben. Sie verweisen vielmehr auf einen Paradigmenwechsel zur *Polyglot Persistence* [9], d.h. der Koexistenz von Datenbanksystemen, in der jedes System eine spezialisierte Aufgabe übernimmt. Daraus ergibt sich die Anforderung, dass ein modernes Datenbanksystem das Paradigma der Polyglot Persistence annehmen und unterstützen sollte (III). Ein weiteres häufig anzutreffendes Attribut von NoSQL Lösungen ist ihre Ausrichtung auf das Web [11]. Dies äußert sich einerseits in der starken Verbreitung von JSON (JavaScript Object Notation) [12] als schemaloses Darstellungsformat und andererseits in der Umsetzung von Datenbankschnittstellen als REST API [3], [13]. Beides erlaubt die reibungsarme Nutzung des Datenbanksystems in Web Kontexten. Eine dritte Anforderung ist deshalb: Nutzung von Web Standards (IV). Web Standards bieten eine weitere vorteilhafte Anwendung für Datenbanken: die webbasierte Administration. Keine Anwendungsplattform erreicht ähnlich viele Nutzer wie der Stack aus HTML5, CSS und JavaScript. Ein Datenbanksystem sollte diese Plattform nutzen, um Administration, Monitoring und Deployment zu ermöglichen (V). Dies hat einerseits den Vorteil, dass keine proprietäre Verwaltungssoftware installiert und konfiguriert werden muss und ermöglicht andererseits eine leicht erlernbare Bedienung mit vielen Freiheitsgraden in der Gestaltung. Des Weiteren wird dadurch die zunehmende Verbreitung von Smart Clients und Single Page Apps unterstützt, die nun Persistenz als Dienst nutzen können, ohne einen Application Server zu benötigen. Das Web wurde darüber hinaus beispielhaft für die Problematik exponentiell anwachsender Datenbestände und hoher Nutzungsparallelität. Aus dem Wunsch, die gewaltigen nutzer- und maschinengenerierten Informationen für analytische Zwecke zu analysieren und integrieren, entstand die Big Data Bewegung [4]. Sie machte es einerseits notwendig, neue, clusterkompatible Datenverarbeitungsmodelle wie MapReduce [14], Dryad [15], Impala [16], Spanner [17], Pregel [18] und Dremel [19] einzusetzen und andererseits Datenbanksysteme mit der Fähigkeit zur horizontalen Skalierung auf große Lasten und Datenmengen zu konzipieren [10]. Hieraus schlussfolgern wir erneut, dass ein Datenbanksystem für Webkontexte sowohl für analytische als auch operative Workloads die Fähigkeit zur horizontalen Skalierung besitzen muss (I).

Ein weiterer Motor für neue Anforderungen ist das Cloud Computing. Es eröffnete ein bisher nicht dagewesenes Nutzungsszenario für Datenbanken: die Nutzung eines Datenbanksystems als Service (Database-as-a-Service, DBaaS). Anstatt das Datenbanksystem wie die Anwendung aufzusetzen und zu konfigurieren, erlaubt Cloud Computing ein Pay-as-you-go Geschäftsmodell und den weitgehenden Verzicht auf klassische Datenbankadministration. Das Database-as-a-Service Problem ist nach unserer Einschätzung jedoch bisher nicht gelöst. Greift ein Client über ein WAN auf eine in der Cloud als Service deployte Datenbank zu, ist die Netzwerklatenz unweigerlich so hoch, dass dies zu einer starken Performancebeschränkung führt. Eine niedrige Latenz ist jedoch kritisch für einen hohen Durchsatz und Effizienz von Datenbankabfragen. Wir fordern deshalb, dass ein Datenbanksystem eine Lösung für die Latenzproblematik in der Cloud bereitstellen muss (VI). Aus der beispiellosen Erfolgsgeschichte klassischer Datenbanksysteme lassen sich zahllose weitere Anforderungen ableiten.

Eine jedoch erscheint uns dabei von besonders weitreichender Bedeutung: Transaktionen [20]. Transaktionen sind das bisher erfolgreichste Verfahren, das in der Lage ist, die Komplexität nebenläufiger Prozesse durch ein einfaches Konzept kontrollierbar zu machen, ohne die Nebenläufigkeit empfindlich einzuschränken. Die Komplexität der Nebenläufigkeitsbeherrschung wird dabei aus der Applikation in das Datenbanksystem verlagert. Moderne NoSQL Datenbanken verzichten vornehmlich auf Transaktionen zu Gunsten der Skalierbarkeit und vereinfachten Implementierung. Wir sind jedoch der Meinung, dass ein Datenbanksystem Transaktionen als ein optionales Konzept anbieten sollte (VII). Dies erlaubt bei Verzicht bessere Skalierbarkeit und sonst eine höhere Produktivität und reduzierte Fehleranfälligkeit der Anwendungsentwicklung. Diese Einschätzung wird von der sogenannten NewSQL Bewegung [21] und der Softwareindustrie geteilt, wie folgende vielzitierte Aussage belegt: „We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions“ (aus Google Spanner, 2012 [17]).



Abbildung 1 Anforderungen. Diese sieben Forderungen an ein modernes Datenbanksystem sollen durch die Ergebnisse dieser Arbeit erfüllt werden.

Die sieben ermittelten Anforderungen sind in Abbildung 1 zusammengefasst. An ihnen orientierte sich bereits die bisherige Entwicklung von ORESTES. Mit dieser Arbeit sollen die Anforderungen durch die neuen Mechanismen zur Cache Kohärenz weitergehend umgesetzt werden. Die Sicherstellung der Cache Kohärenz teilt sich auf zwei neue Mechanismen auf: Verhinderung von veralteten Leseergebnissen aus Caches und proaktives Invalidieren serverkontrollierter Caching Verbunde. Wir stellen nun beide kurz vor und erläutern ihre Implikationen auf die ermittelten Anforderungen.



Das Lesen veralteter Cache-Inhalte kann immer dann auftreten, wenn eine Schreiboperation zuvor durch andere Caches weitergeleitet wurde. Da die Lebensdauer der gecachten Objekte durch eine statisch spezifizierte Zeitspanne gekennzeichnet ist, führen diese ad-hoc Änderungen unweigerlich zu invaliden Cache-Einträgen. In dieser Arbeit entwickeln wir jedoch einen Mechanismus, den wir *bloomfilterbasierte Cache Kohärenz* nennen. Dabei übernimmt der Client die Verantwortung dafür, ggf. veraltete Cache Inhalte mit einer Revalidierungsaufforderung anzufragen, so dass betroffene Caches die aktuellste Objektversion aus der Datenbank beziehen. Um festzustellen, welche Anfragen mit einer Revalidierungsaufforderung versehen werden müssen, benötigt der Client eine Darstellung aller geänderten Objekte innerhalb des Zeitfensters, das der Caching-Dauer der Objekte entspricht. Aus zahlreichen Gründen ist die optimale Datenstruktur für die Speicherung dieser Änderungen ein Bloomfilter. Er erlaubt es, in $O(1)$ zu überprüfen, ob ein Objekt geändert wurde. Außerdem ist seine Größe so gering, dass er ohne nennenswerten Overhead bei jedem Transaktionsbeginn oder auf Anfrage übertragen werden kann. Der Server nimmt also jede ändernde Operation in den Bloomfilter auf und Clients fragen diesen Bloomfilter ab, um *Stale Reads* zu verhindern. Die bloomfilterbasierte Cache Kohärenz erlaubt es so, aus nicht-kontrollierten Caches, die nach dem Prinzip *Eventual Consistency* arbeiten, optional ein konsistentes System zu schaffen.

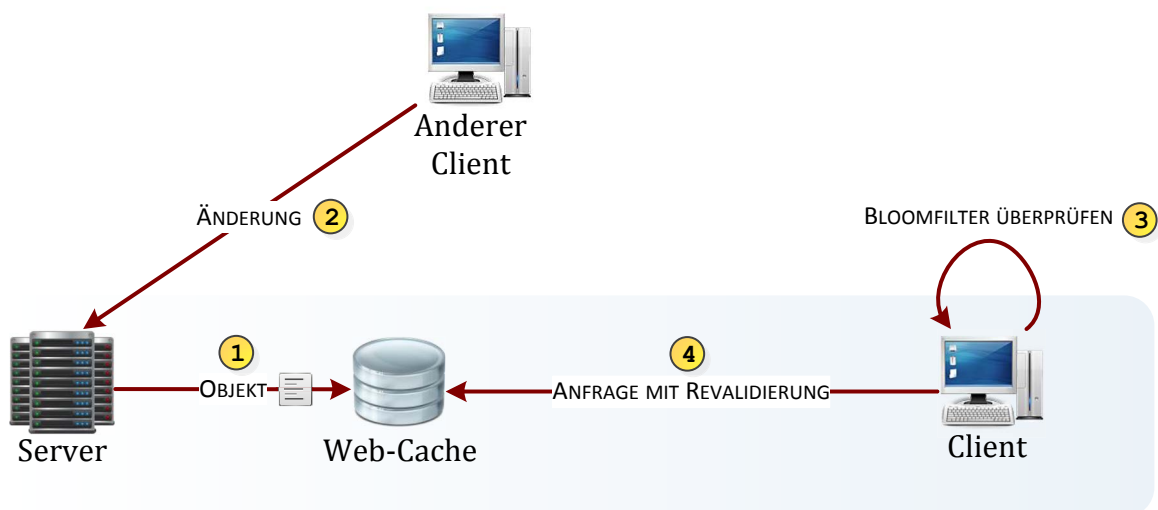


Abbildung 2 Bloomfilterbasierte Cache Kohärenz.

Die Arbeitsweise der bloomfilterbasierten Cache Kohärenz ist in Abbildung 2 dargestellt:

1. Wird über einen Web-Cache ein Objekt angefordert, antwortet der Server mit dem angefragten Objekt, das der Web-Cache fortan speichert.
2. Erreicht den Server eine Änderung des ausgelieferten Objekts, übernimmt er die Änderungen auch in den Bloomfilter. Zu diesem Zeitpunkt ist die Objektkopie, die im Web-Cache gehalten wird, veraltet (*stale*).
3. Bevor ein Client ein Objekt anfragt, überprüft er den Bloomfilter, der z.B. bei jedem Transaktionsbeginn bezogen wird.
4. Da die Änderung vom Server im Bloomfilter vermerkt wurde, stellt der Client fest, dass der Objektanfrage eine Revalidierungsaufforderung beigefügt werden muss.

Die bloomfilterbasierte Cache Kohärenz hat Auswirkungen auf nahezu alle Punkte des Anforderungskataloges. Die horizontale Skalierbarkeit wird durch die uneingeschränkte Nutzbarkeit von Web-Caches auch bei der Randbedingung starker Konsistenz möglich (I). Die transparente Datenbankschnittstelle wird aufrechterhalten, indem der Bloomfilter durch einen einfachen REST-API Aufruf abgefragt werden kann (II). Dabei werden selbstverständlich die Web Standards eingehalten und der Bloomfilter als sprachunabhängig verarbeitbares JSON Objekte übertragen (III). Auf den Zugriff mit geringer Netzwerklatenz wird ebenfalls Einfluss genommen. Eine höhere Netzwerklatenz entsteht bei bloomfilterbasierter Cache Kohärenz bei einer Revalidierung. Dies mag auf den ersten Blick wie ein Nachteil erscheinen, ist jedoch in Wirklichkeit ein bemerkenswerter Vorteil: die signifikant höheren Kosten eines andernfalls notwendigen Transaktionsabbruches werden durch die einmalige Inkaufnahme einer höheren Netzwerklatenz verhindert (V). Die Folge ist starke Konsistenz gleichermaßen für Transaktionen und nichttransaktionale Clients, die in beliebigen Abständen aktualisierte Bloomfilter anfragen können (VI).

Der zweite Pfeiler der angestrebten Cache Kohärenz sind die Verbunde aus serverkontrollierten Caches. Zu ihnen zählen Content Delivery Networks (CDNs) [22] und Reverse-Proxy Caches [23]. CDNs sind die etablierte Infrastruktur mit der sehr große Webseiten Skalierbarkeit erzielen. CDNs (z.B. Akamai, LimeLight, Cloudfront) unterhalten weltweite Netzwerke an Caching Servern an Positionen, die nahe an potentiellen Clients liegen. Dadurch können gecachte Inhalte unabhängig von der geographischen Position des Clients mit geringer Latenz ausgeliefert werden. Reverse-Proxy Caches sind Web Caches, die im gleichen Netzwerk wie der Server aufgestellt werden und als sein Stellvertreter HTTP-Anfragen beantworten und dabei Objekte cachen.

Der Punkt, der Reverse-Proxy Caches und CDNs von allen übrigen Web Caches unterscheidet, ist ihre Fähigkeit, Invalidierungen für gecachte Objekte zu empfangen [24]. Der Server ist in der Lage, bei Änderungen von Objekten das CDN und die Reverse-Proxy Caches anzuweisen, ihre gespeicherte Kopie freizugeben und stattdessen bei Bedarf eine neue anzufordern. Dies bietet den enormen Vorteil, dass bei einem Server, der Änderungen erkennt und entsprechende Invalidierungen propagiert, das Zeitfenster potentiell veralteter Reads für diese Caches nahezu verschwindet. Selbst in Verbindung mit Bloomfiltern hat dieser Ansatz Vorteile. So können z.B. Reverse-Proxy Caches konfiguriert werden, Revalidierungen zu ignorieren und stattdessen gecachte Objekte auszuliefern. Dies schützt den Server vor der potentiellen Last, die er erfährt, sobald ein Objekt durch Aufnahme in den Bloomfilter für einen Zeitraum stets direkt vom Server revalidiert wird. Die Konsistenz wird dabei nicht herabgesetzt, da die Reverse-Proxy Caches unmittelbar über Invalidierungen informiert werden.

Der Ablauf einer Anfrage im Zusammenhang mit Invalidierungen ist in Abbildung 3 gezeigt:



1. Wenn ein Objekt über einen Reverse-Proxy Cache oder das CDN angefragt wird, ist das Objekt dort anschließend gecacht.
2. Erreicht den Server eine Änderung von einem Client, wird diese Änderung i.d.R. auch einen Cache aus dem CDN und einen Reverse-Proxy Cache passieren. Diese Caches können bereits jetzt ihren Eintrag invalidieren. Die nicht-passierten Caches behalten jedoch den veralteten Eintrag.
3. Da der Server von der Änderung weiß, informiert das Invalidierungssystem alle registrierten Reverse-Proxy Caches und das CDN mit einer Invalidierung.
4. Ein Client, der nun ein Objekt anfragt, erhält die neueste Version des Objektes.

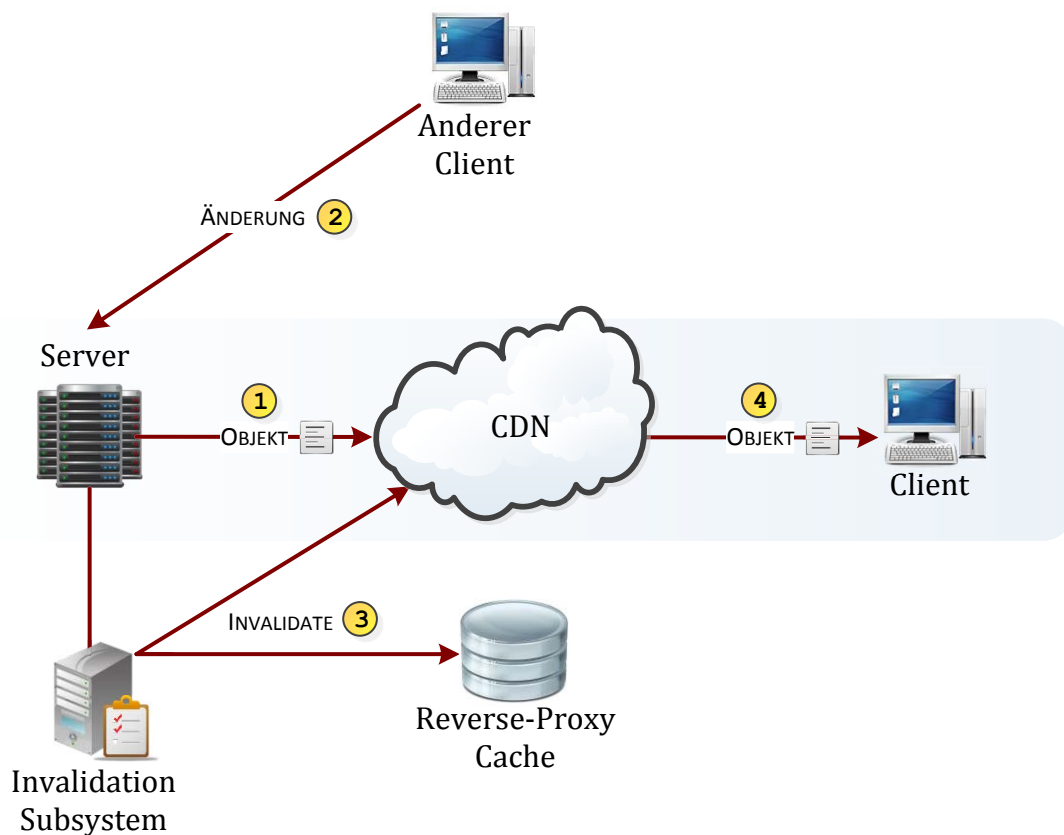


Abbildung 3 Nutzung serverkontrollierter Cache Verbunde.

Die instantane Invalidierung serverkontrollierter Cache Verbunde hat ebenfalls Einfluss auf die Anforderungen. Die horizontale Skalierbarkeit wird dadurch erhöht, dass CDNs und Reverse-Proxy Caches uneingeschränkt als skalierbare Caching Schicht des Servers arbeiten können (I). Besondere Auswirkung haben die Invalidierungen auch auf die Netzwerklatenz – selbst gerade geänderte Objekte werden nach wie vor aus Caches bedient. Bei einem globalen CDN beispielsweise heißt das, dass die Latenz zum Client stets sehr gering bleibt (V). Die Konsistenz von CDNs und Reverse-Proxy Caches wird dadurch automatisch und ohne Hilfe des Clients aufrechterhalten (VI).

Eine Einschränkung des bisherigen Ansatzes von ORESTES ist die Festlegung auf das objektorientierte Modell. Die Techniken zur Skalierung durch Web Caches und Load Balancing sowie die neuen Mechanismen der Cache Kohärenz sind jedoch nicht auf dieses Modell beschränkt. Wir erweitern ORESTES deshalb um die notwendigen Aspekte zur Unterstützung von Polyglot Persistence (III). Wir demonstrieren dies beispielhaft durch eine Implementierung zur Anbindung eines Key-Value Stores. Die ORESTES REST/HTTP Schnittstelle erhält dazu die neuen Abstraktionen für Operationen eines Key-Value Stores. Darüber hinaus zeigen wir die Vorteile der losen Kopplung von Persistenz API und Datenbank durch das ORESTES Protokoll: der Key-Value Store kann auch die Schnittstelle der objektorientierten Persistenz übernehmen - unter Weiterverwendung existierender Client APIs. Das Resultat ist die Möglichkeit zur hochperformanten Speicherung von Objekten auf Kosten von Querys. Da navigierender (queryloser) Zugriff bei objektorientierter Persistenz eines der häufigsten Szenarien darstellt, ist die Objektpersistenz durch einen Key-Value Store für viele Use-Cases eine interessante Option.

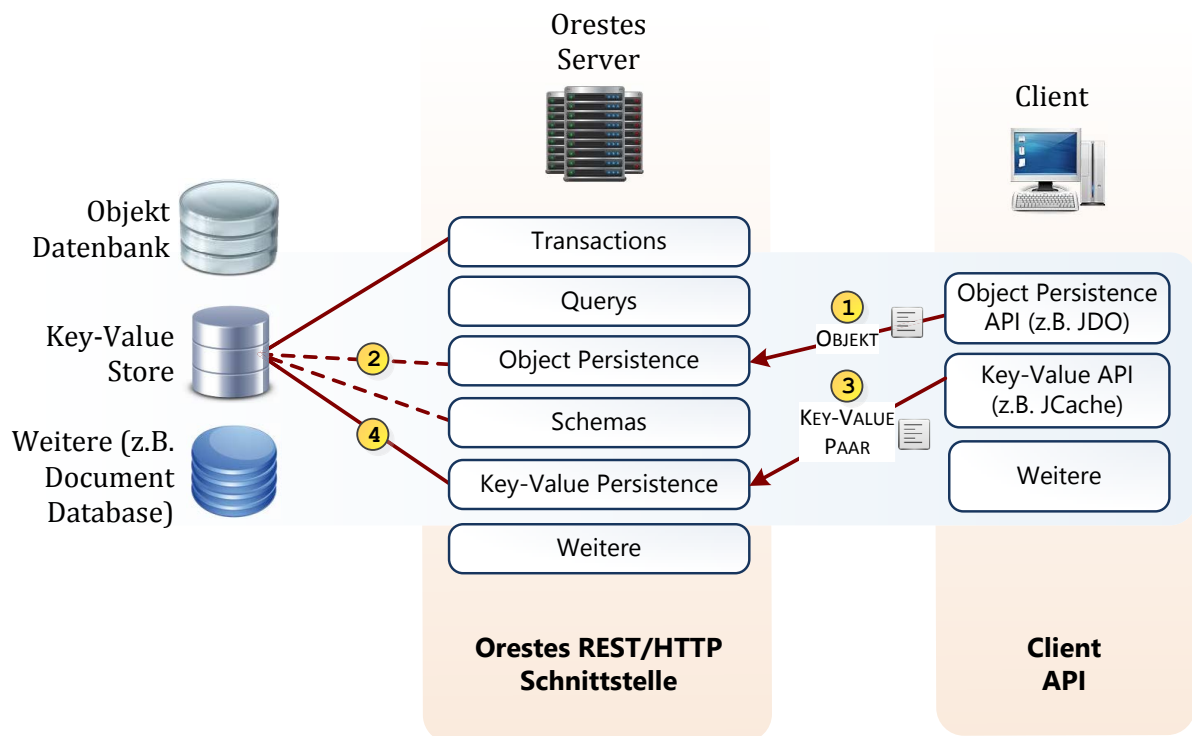


Abbildung 4 Polyglot Persistence. Das ORESTES Protokoll erhält eine neue Abstraktion zur Speicherung von Key-Value Paaren. Die Implementierung erhält eine Anbindung an einen Key-Value Store, der einerseits Objektpersistenz (ohne Querys) übernehmen kann und andererseits Key-Value Operationen.

Die Neuerungen zur Polyglot Persistence sind in Abbildung 4 gezeigt. Die Fähigkeiten der ORESTES REST/HTTP Schnittstelle gliedern sich auch in verschiedene Kategorien, die nach Bedarf für unterschiedliche Datenbanksysteme implementiert werden können. Die Anbindung des Key-Value Stores kann z.B. über reine Key-Value Persistenz hinausgehen und zusätzlich objektorientierte Persistenz und Schemaverwaltung anbieten (gestrichelte Linie). Ein Client hat damit mehrere Optionen:



1. Mithilfe einer objektorientierten Persistenz-API, die auf dem ORESTES-Protokoll aufbaut, kann der Client Objekte speichern, laden, löschen, etc.
2. Mehrere Datenbanksysteme können die Funktionalität auf Serverseite anbieten, z.B. eine Objektdatenbank (hohe Querymächtigkeit, Indizierung, etc.) oder aber auch ein Key-Value Store (schneller In-Memory Zugriff, hohe Verfügbarkeit, etc.). Durch die Entkopplung von Client und Server durch das ORESTES Protokoll ist die Persistenz-API auf Clientseite agnostisch gegenüber dem zugrundeliegenden Datenbanksystem und bedarf keiner Neuimplementierung.
3. Der Client kann außerdem die native Key-Value Schnittstelle benutzen, die ebenfalls über das ORESTES Protokoll angeboten wird. Dabei werden die gleichen Techniken zur Transaktionsverwaltung, Web Caching, etc. benutzt, die bereits erforscht und umgesetzt sind.
4. Der Aufruf wird durch eine serverseitige Anbindung an den Key-Value Store weitergereicht. Diese Anbindung entscheidet, welche Funktionalitäten bereitgestellt werden sollen, z.B. ob Transaktionen unterstützt werden.

Wir erachten Polyglot Persistence als eine wichtige Entwicklung. Wir zeigen in dieser Arbeit deshalb, dass die Mechanismen von ORESTES für Polyglot Persistence geeignet sind und verschiedene Datenbanken gleichermaßen von den in dieser Masterarbeit untersuchten Methoden zur Cache-Kohärenz profitieren können.

Diese Masterarbeit leistet damit fünf Beiträge:

- Wir zeigen, wie durch die Lösung der Latenzproblematik Database-as-a-Service Modelle praktisch nutzbar werden.
- Das Konzept der bloomfilterbasierten Cache Kohärenz liefert ein generisches Prinzip, mit dem nicht-invalidierbare Caches ohne Verlust der Kohärenz betrieben werden können.
- Wir beweisen, dass serverkontrollierte Cache Verbunde aus Reverse Proxy Caches und CDNs eine bereits existierende Infrastruktur darstellen, die zur Skalierung von Datenbanksystemen genutzt werden kann.
- Wir veröffentlichen eine Bloomfilter Implementierung, die anderen Implementierungen überlegen ist, da sie sowohl Verteilung als auch besonders umfassende Auswahlmöglichkeiten für Hashfunktionen bietet.
- Wir zeigen, dass Polyglot Persistence auf Basis eines generischen Protokolls (ORESTES) möglich ist und dabei erhebliche Synergien entstehen, wie z.B. gemeinsame Web Caching Techniken.

Der Rest dieser Arbeit ist wie folgt strukturiert: die übrigen Teile der Einleitung beschreiben ORESTES, den Kontext für die Entwicklung der neuen Cache-Kohärenz-Mechanismen. Kapitel 2 gibt einen Überblick über verwandte Forschung. In Kapitel 3 wird der Mechanismus der bloomfilterbasierten Cache Kohärenz diskutiert, sowie die Konzepte und Techniken zur Realisierung eines verteilten, hochperformanten Bloomfilters mit hoher statistischer Güte. Kapi-

tel 4 erörtert die Anbindung von serverkontrollierten Cache Verbunden und die Auswirkungen auf die Skalierbarkeit. Kapitel 5 zeigt die Umsetzung von ORESTES als Polyglot Persistence Protokoll und die Nutzbarkeit aus JavaScript. In Kapitel 6 werden die offenen Fragen und potentiellen Forschungsrichtungen dargelegt. Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen.

1.1 ORESTES

Da zahlreiche Aspekte von ORESTES bereits in unser Bachelorarbeit [6] diskutiert wurden und außerdem ein erläuterndes Poster [25] und eine Website (<http://ORESTES.info>) zur Verfügung stehen, beschränken wir uns in dieser Einführung auf die wesentlichen und für das Verständnis dieser Masterarbeit entscheidenden Aspekte.

1.1.1 Motivation für ein cloudfähiges Persistenzprotokoll

Der Siegeszug von Cloud Computing und NoSQL zeigt den deutlichen Bedarf an skalierbaren Datenbanksystemen mit cloudfähigen, webbasierten Schnittstellen [1]. Um den in der Einleitung angeführten Anforderungen gerecht zu werden, müssen die Eigenschaften *horizontale Skalierbarkeit von Leseanfragen, niedrige Latenz, HTTP basierter Zugriff mit Standardprotokollen* und die *Unabhängigkeit von Client APIs und Datenbanken* von einem Persistenzprotokoll erfüllt werden. Diese Eigenschaften erlauben neue Datenbank- und Anwendungsarchitekturen. Datenbanken, die als Service angeboten werden, benötigen die Möglichkeit, Leseanfragen zu skalieren und die potentielle Verteilung von Datenbank und Applikation durch Zugriff mit geringer Latenz zu ermöglichen [26]. Geringe Latenz ist die unabdingbare Voraussetzung, um hohen Durchsatz und die Reaktivität eines lokalen Datenbank-Deployments zu erreichen. Das Verbergen der unausweichlichen Latenz von Cloud Services ist damit ein eminent wichtiger Aspekt eines Persistenzprotokolls. Um die Elastizität und on-demand Ressourcen von Cloud Umgebungen zu nutzen, muss das Protokoll außerdem einen Mechanismus zur horizontalen Skalierbarkeit anbieten. Dies gestattet es dem Datenbanksystem, leseintensive Workloads ohne Performanceverluste zu bewältigen. Als Folge kann eine große Anzahl von Clients die Datenbank parallel und effizient nutzen und dabei von der Flexibilität des Cloud Computings profitieren.

Viele Anwendungen in Cloud Umgebungen müssen mit ihren Services (zu denen Persistenz zählt) durch statusloses HTTP interagieren [2]. HTTP erlaubt es, die Datenbank direkt über einen Webbrowser zu verwalten, monitoren und bereitzustellen. Dies macht HTTP zu einer naheliegenden Wahl für ein Persistenzprotokoll. Aufgrund der zunehmenden Heterogenität existierender Datenbanklösungen ist für ein Persistenzprotokoll die Gernerizität und Erweiterbarkeit in Bezug auf Datenmodelle, Querys, Transaktionen und Konsistenzstrategien essentiell. Das Einführen einer losen Kopplung zwischen Persistenz API und Datenbank auf der Protokollebene ist eine Grundvoraussetzung, um die freie Kombinierbarkeit und Koexistenz verschiedener Datenbanktypen zu fördern (*Polyglot Persistence* [1]). Die Umsetzungsidee dieser Anforderungen ist in Tabelle 1 gezeigt.



Eigenschaft	Mechanismus
<i>Elastische Read Skalierbarkeit</i>	Web-Caching von Datenbankobjekten, Workload-abhängiges Starten neuer Web Caches
<i>Geringe Latenz</i>	Caches mit geographischer Nähe zum Client, d.h. CDNs und Forward-Proxy Caches
<i>Lose Kopplung zwischen Persistenz API und Datenbank</i>	Universelles HTTP Protokoll mit erweiterbarer Ressourcenstruktur für Transaktionen, Objekte, Schemaverwaltung und Querys
<i>Web Standards</i>	Erweiterbare HTTP Content Negotiation und vordefinierte JSON Formate

Tabelle 1 Anforderungen an ein Persistenzprotokoll und die Umsetzung in ORESTES.

In Abbildung 5 sind die Anforderungen visualisiert: Die Kommunikation zwischen der Anwendung und Datenbank muss durch ein Protokoll erfolgen, das einen Mechanismus besitzt, um horizontale Read Skalierbarkeit und geringe Latenz zu ermöglichen. Darüber hinaus muss es so erweiterbar sein, dass neue Persistenz APIs und Datenbanken problemlos von dem Protokoll Gebrauch machen können.

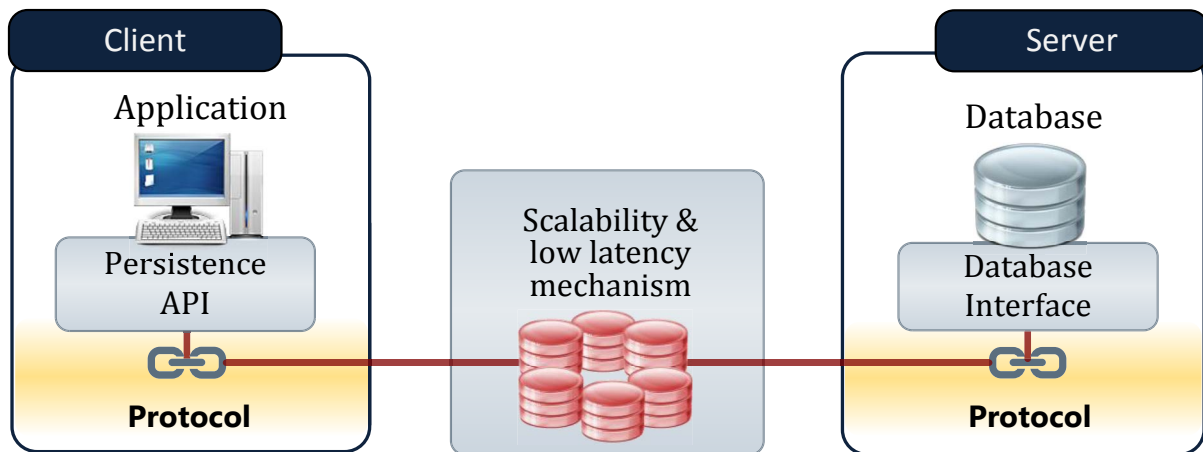


Abbildung 5 Die Anforderungen an ein Persistenzprotokoll.

Unsere Lösung (ORESTES) ist darauf ausgelegt, die obigen Eigenschaften zu erfüllen. In seiner jetzigen Implementierung unterstützt ORESTES objektorientierte Persistenz bereits vollständig. Es kann deshalb durch Persistenz APIs wie JDO (Java Data Objects) [27] oder JPA (Java Persistence API) [28] auf der Clientseite und objektorientierte Datenbanken [29] oder relationale Datenbank mit objektrelationalem Mapping auf Serverseite eingesetzt werden. Das Protokoll von ORESTES Protokoll ist jedoch so generisch, dass mit den gleichen Prinzipien und der gleichen Implementierung auch andere Datenbanksysteme wie Key-Value Stores, Dokumentendatenbanken, Wide-Column Stores, Graphdatenbanken und sogar relationale Datenbanken unterstützt werden können (cf. [30]).

Das zentrale Problem, das durch ORESTES erstmals gelöst wird, ist die scheinbare Unverträglichkeit des HTTP Caching Modells und dynamischen Datenbankobjekten. In HTTP ist die Caching Lebensdauer von Objekten ein statischer Parameter, ad-hoc Invalidierungen sind für alle Caches, die nicht direkt dem Server unterstehen (Reverse-Proxys und CDNs), un-

möglich (cf. [31]). ORESTES zeigt, dass starke Konsistenz und ACID Transaktionen dennoch mit dem HTTP Caching Modell vereinbar sind. Dies geschieht ohne die neuen Mechanismen dieser Masterarbeit jedoch unter der Gefahr von Transaktionsabbrüchen:

1. Wenn ein Client ein Objekt lädt, wird ein HTTP Request versendet.
2. Das Objekt kann von einem Web Cache ausgeliefert werden. Die Objektmetadaten enthalten eine Versionsnummer.
3. Wenn der Client die laufende Transaktion committet, überträgt er die Versionsnummern aller gelesenen Objekte (das sogenannte *Read-Set*) an den Server. Auf Basis optimistischer Nebenläufigkeitskontrolle entscheidet der Server, ob Stale Reads oder Konflikte nebenläufiger Transaktionen aufgetreten sind und rollt die Transaktion ggf. zurück. Dies ist die Schwachstelle des bisherigen Ansatzes – jeder Stale Read zieht einen Transaktionsabbruch nach sich. Die bloomfilterbasierte Cache Kohärenz kann dies verhindern.

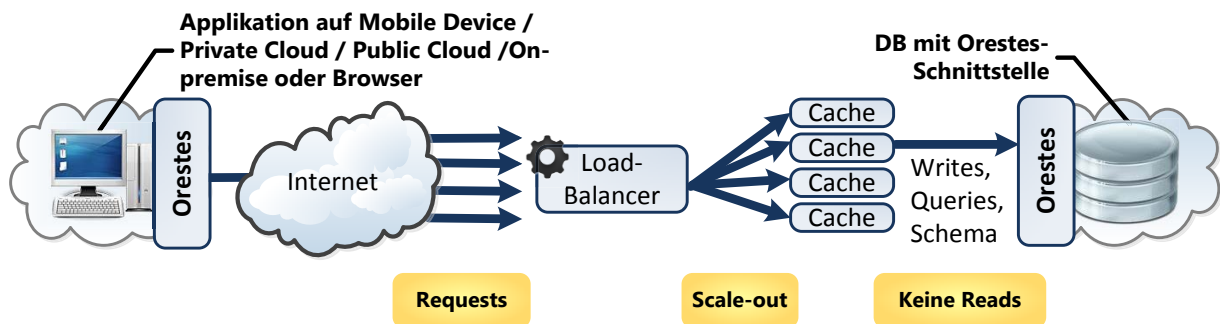


Abbildung 6 Das Skalierungsmodell von ORESTES. Der Scale-out basiert auf dem Fan-out von HTTP Requests durch Load Balancer und dem Abfangen von Leseanfragen durch Web-Caches.

In Abbildung 6 ist das Skalierungsmodell von ORESTES zusammengefasst. Wir werden den Skalierungsaspekt im Folgenden genauer beschreiben, da er den Grundpfeiler von ORESTES bildet und die Originalität des ganzen Ansatzes ausmacht. Außerdem ist er die primäre Motivation für die Cache-Kohärenz Mechanismen dieser Masterarbeit. Unsere Implementierung von ORESTES unterstützt derzeit zwei Persistenz APIs (Java/JDO und JavaScript/JPA) und zwei Datenbanksysteme (Versant Object Database und db4o [32], [33]). In ausführlichen Benchmarks, die wir in dieser Arbeit nur kurz ansprechen, konnten wir nachweisen, dass mit ORESTES ein Speedup von 220-800% für massiv parallele, leseintensive Workloads gegenüber den nativen binären Protokollen erzielt werden kann.

1.1.2 Konzept

1.1.2.1 Das ORESTES REST/HTTP Protokoll

REST (*Representational State Transfer*) ist ein Architekturstil, der eine a-posteriori Erklärung für den Erfolg des Webs gibt [34]. REST ist eine Sammlung von Constraints, die zu wünschenswerten Systemeigenschaften wie Skalierbarkeit und Einfachheit führen, wenn sie auf



das Design eines Protokolls angewendet werden [35]. Viele Dienste im Cloud Computing werden technisch als REST/HTTP Services angeboten, da sie dadurch leicht verständlich und in jeder Programmiersprache nutzbar sind [36]. HTTP ist ein Anwendungsprotokoll auf Basis des Transmission Control Protocol (TCP) und bildet die Grundlage des Webs [35]. Die meisten Platform-as-a-Service Provider gestatten Anwendungen die HTTP Kommunikation, wohingegen native TCP Verbindungen verhindert werden [2]. Da außerdem nahezu alle Netzwerkknoten - wie Firewalls und Proxys - HTTP erlauben und unterstützen, ist HTTP ein ideales Protokoll für einen hohen Grad an Interoperabilität und Zugänglichkeit. Da eine REST-API die Nutzung aus jeder modernen Programmiersprache erlaubt, bieten viele NoSQL Datenbanken REST Schnittstellen an. Mit ORESTES zeigen wir jedoch, dass eine geeignete HTTP Datenbankschnittstelle noch erheblich stärkere Gründe für die Nutzung von HTTP und REST bietet als Interoperabilität und leichte Nutzbarkeit.

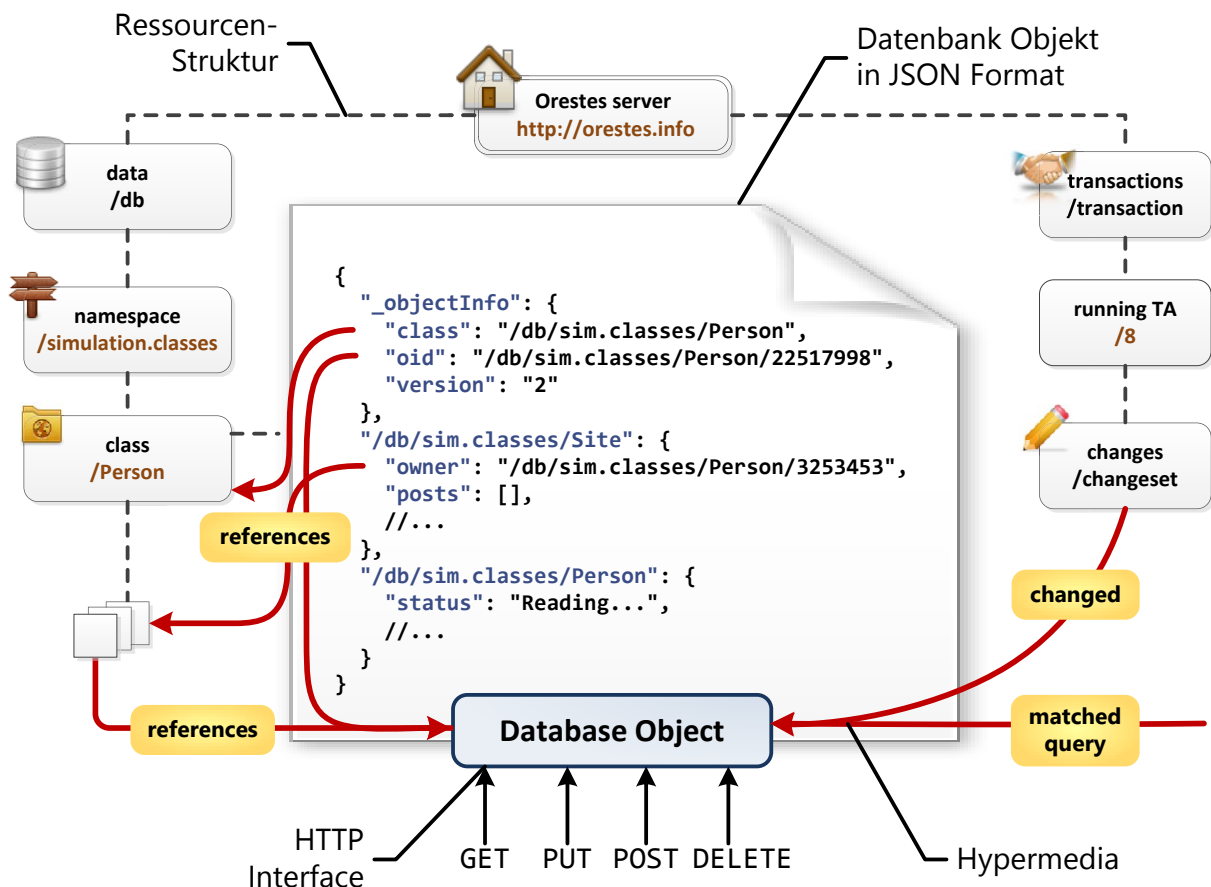


Abbildung 7 Die REST/HTTP Schnittstelle von ORESTES. Die Abbildung zeigt einen Ausschnitt der Ressourcenstruktur `root > db > namespace > class > object`. Die Abhängigkeiten von Ressourcen (Hypermedia) sind in rot markiert.

Im ORESTES REST Protokoll sind die Schlüsselabstraktionen durch Ressourcen wie `schema`, `transaction`, `query`, etc. abgebildet. Diese Ressourcen werden durch URLs eindeutig identifiziert. Durch das *Uniform Interface* von HTTP mit seinen Methoden GET, PUT, POST und DELETE können Operationen auf diesen Ressourcen durchgeführt werden, z.B. das Anlegen eines neuen Objekts. Repräsentationen (z.B. Objekte) können jeden beliebigen Medientyp

annehmen. Wir definieren Defaultformate in JSON (JavaScript Object Notation) [12], erlauben aber beliebige andere Medientypen (z.B. XML), die dynamisch über HTTP Content Negotiation ausgehandelt werden können. Durch die Erweiterbarkeit können die Datenbankinhalte potentiell stets in einer Weise repräsentiert werden, die dem jeweiligen Use Case angemessen sind. Die Ressourcenstruktur ist ebenfalls frei erweiterbar. Um z.B. das Anlegen von Map-Reduce Views zu unterstützen, würde auf Protokollebene die Definition einer neuen Ressourcen `mr_view` und einem zugehörigen Medientyp genügen, um das Protokoll zu erweitern.

1.1.2.2 Skalierungsmodell

HTTP erlaubt es, Ressourcen als cachebar zu kennzeichnen. In diesem Fall werden sie für eine statische und pro Objekt spezifizierte Zeitspanne als frisch erachtet und anschließend verworfen. Ein Client, der ein Objekt anfragt, kann dieses Objekt immer dann aus einem Web-Cache erhalten, wenn dieser das entsprechende Objekt gespeichert hat und es noch immer als frisch gilt. Anhand einer Versionsnummer (*ETag*) oder eines Änderungsdatums können Clients ein Objekt beim Origin Server revalidieren, d.h. potentielle Änderungen erfragen. So entfällt der Aufwand der Übertragung und Neuberechnung der Repräsentation, wenn sich ein Objekt nicht geändert hat.

In ORESTES werden alle Objekte mit einer konfigurierbaren Cache Lebensdauer (z.B. 1 Stunde) ausgeliefert. Ein solches Objekt kann dann direkt vom Cache zum Client übertragen werden, ohne dass auf dem Server Last entsteht. Gleichzeitig sinkt die Netzwerklatenz, da einerseits Web-Caches auf das schnelle Ausliefern von Objekten spezialisiert sind und andererseits dichter an Clients positioniert sind als der Datenbankserver. Anhand ihrer Position unterscheiden wir fünf Arten von Web-Caches (siehe Abbildung 8) [31], [35], [37].

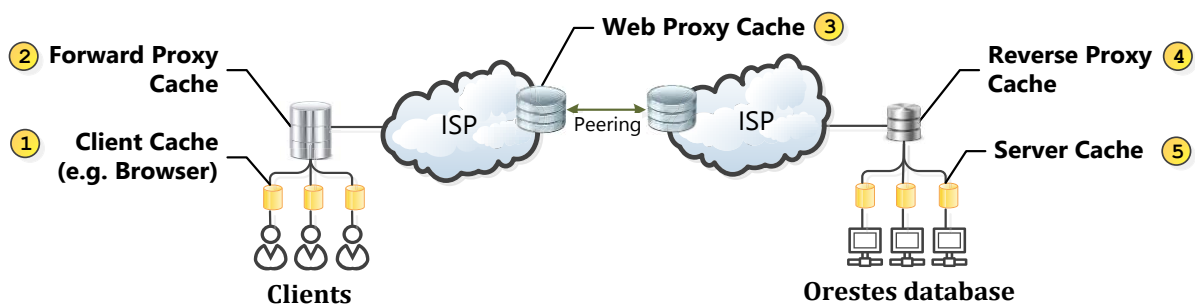


Abbildung 8 Die fünf verschiedenen Arten von Web-Caches. Man kann zwischen eingebetteten (1, 5) und serverbasierten (2,3,4) Web Caches unterscheiden.

Web-Caches können direkt in den Client eingebettet werden (*Client Cache*). Das populärste Beispiel eines solchen Caches ist der Browser Cache. Da Client Caches direkt vom Client verwaltet werden, können sie zwar einerseits Objekte mit einer besonders geringen Latenz ausliefern, enthalten aber nicht die von mehreren Clients genutzten Objekte und sind meistens klein. Analog kann ein Cache direkt in den Server eingebettet sein (z.B. der Page Buffer eines DBMS). Fast alle Datenbanksysteme betreiben diese Art von Caching, um besonders



teure Operationen abzufedern, z.B. einen Festplattenzugriff [38]. Die leistungsfähigsten Arten von Web-Caches sind eigenständige Server. Sie werden entweder explizit im Client konfiguriert, automatisiert über Konfigurationsprotokolle abgerufen oder übernehmen als Interception Proxys transparent die Verbindung zwischen Client und Server. *Forward Proxy Caches* (z.B. Squid oder Microsoft TMG [39], [40]) können im Netzwerk der Clients als ausgehender Server deployt werden. Durch ihre geographische Nähe zum Client kann die Netzwerklatenz sehr gering gehalten werden, typischerweise im Bereich von 1 ms oder weniger. Viele Internet Service Provider (ISPs) setzen *Web Proxy Caches* ein (häufig Hardware Appliances, z.B. Bluecoat Cacheflow), um Traffic und damit Transitgebühren einzusparen oder die Dienstgüte für Kunden zu erhöhen. Der von vielen Websites eingesetzte Typ von Web-Cache ist der sogenannte *Reverse-Proxy Cache* (z.B. Varnish oder Apache Traffic Server [23]). Er agiert bei eingehenden Requests als Stellvertreter des Servers und nimmt an seiner statt Anfragen entgegen. Reverse-Proxy Caches unterscheiden sich von den übrigen Web-Cache Arten vor allem dadurch, dass sie Invalidierungen des Origin Servers empfangen können. Die in CDNs eingesetzten Caches sind konzeptionell nahezu identisch mit Reverse-Proxy Caches.

Während Web-Caching den Umstand nutzt, dass Objekte öfter gelesen als geschrieben werden, nutzt Web Load-Balancing die inhärente Request-Parallelität des HTTP Protokolls. Wie Web-Caching kann Load-Balancing sowohl in Hardware als auch Software implementiert werden. Voraussetzung für Load-Balancing ist die Statuslosigkeit des HTTP Protokolls. Eingehende Requests können dadurch auf beliebige Reverse-Proxy Caches oder Server verteilt werden, ohne dass im Load-Balancer oder Server ein client-spezifischer Zustand gehalten werden muss. Load-Balancing kann auf verschiedenen Netzwerkebenen implementiert werden. Sehr verbreitet ist z.B. Layer-2 Forwarding (*Direct Routing*), bei dem Load-Balancer und alle Origin Server sich eine (virtuelle) IP Adresse teilen. Der Load-Balancer leitet nach einer vorgegebenen Policy (z.B. Round Robin) und einem TCP SYN Paket eingehende Layer-2 Frames an die Origin Server weiter und ersetzt dabei die Ziel-MAC-Adresse des eingehenden Frames durch diejenige des ausgewählten Servers [41]. So wird völlig transparent eine Ende-zu-Ende TCP-Verbindung etabliert, während alle TCP Verbindungen gleichmäßig auf alle Server aufgeteilt werden (*Layer-4 Server Load Balancing*). Eine weitere sehr einfache Umsetzung von Load-Balancing ist *DNS Round-Robin*. Dabei existieren für einen Domainnamen (z.B. ORESTES.info) mehrere Resource Records. Der Domainname wird dadurch von verschiedenen Clients in unterschiedliche IP Adressen aufgelöst, die den einzelnen Backend Servern entsprechen. Da die Netzwerkebene von Web Load-Balancing für Client und Server vollständig transparent ist, sei für die Details der einzelnen Verfahren hier nur auf Gilly et al. [41] verwiesen. Um Load-Balancing für ORESTES zu nutzen, kann im Cloud Umfeld z.B. auf Amazons Elastic Load Balancing Service zurückgegriffen werden [42]. Auch das Aufsetzen eines eigenen Load-Balancers ist leicht möglich mit Open-Source Lösungen wie HAProxy oder Nginx [23].

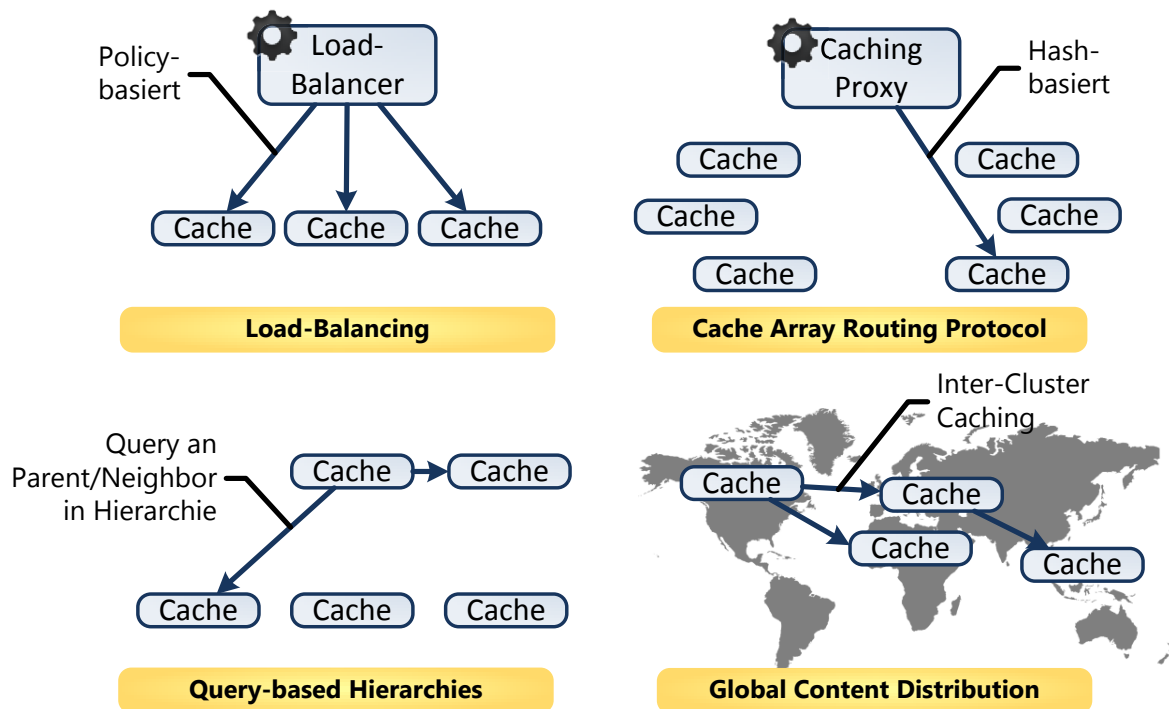


Abbildung 9 Skalierungsmechanismen von Web-Caches.

Eine interessante und bisher wenig beachtete Eigenschaft von Web-Caching ist: Web-Caches skalieren auf die gleiche Weise wie Datenbanksysteme – durch Sharding und Replikation. Deshalb liegt der Ansatz nahe, Web-Caches für die Skalierung von Datenbanksystemen einzusetzen. Vier generelle Skalierungsmechanismen lassen sich bei Web-Caches unterscheiden (siehe Abbildung 9). Beim Load-Balancing über mehrere Web-Caches findet implizit eine Replikation statt. Bei jedem GET-Request über einen Web-Cache wird das entsprechende Objekt in dem jeweiligen Web-Cache als Replikat gespeichert. Beim Cache Clustering, wie es z.B. in dem Cache Array Routing Protocol (CARP) [43] implementiert ist, wird die Menge aller URLs durch eine einfache Hash-Partitionierung (wie $hash(URL) \bmod n_{server}$) disjunkt auf eine Menge von Web-Caches verteilt. Jeder Web-Cache hält also eine Partition (*Shard*) der gesamten Datenmenge. Eine Mischform aus Replikation und Sharding ist in Query-basierten Inter-Cache Protokollen wie ICP, Cache Digests und HTCP [44]–[46] implementiert. Dort haben Web-Caches die Möglichkeit, bei einem Cache-Miss benachbarte Web-Caches nach dem angeforderten Objekt zu befragen. So entstehen hierarchisch strukturierte, kooperative Web-Cache Cluster, in denen Caches bei Bedarf Objekte von Nachbarn nachladen können.

Content Delivery Networks verwenden zumeist ebenfalls Kombinationen aus Sharding und Partitionierung. Dort kommen neben den standardisierten Inter-Cache Protokollen zudem häufig proprietäre Spezialprotokolle zum Einsatz. Trifft dort beispielsweise durch den Request Routing Mechanismus des CDNs ein Request in einem Cache Cluster des CDNs ein, werden bei einem Cache-Miss zuerst benachbarte Server des gleichen Clusters befragt. Bei



einem Cache-Miss im Cluster werden dann geographisch weiter entfernte Cluster befragt. Nur wenn es dort ebenfalls zu einem Cache-Miss kommt, wird der Origin Server befragt.

	Datenbanksystem	Replikation	Sharding
Key-Value Stores	Redis	asynchron	Hash-basiert (Client)
	Riak	asynchron	Consistent Hashing
	Voldemort	asynchron	Consistent Hashing
	Couchbase	asynchron	Hash-basiert
	Dynamo	asynchron	Consistent Hashing
Dokumenten Datenbanken	CouchDB	asynchron	Consistent Hashing
	MongoDB	synchron/asynchron	Range-basiert (Autosharding)
Wide Column Stores	BigTable	asynchron	Range-basiert
	MegaStore	synchron	Explizite Parent-Child Tabellengruppierung (Entity Group)
	HBase	asynchron	Range-basiert
	Hypertable	synchron	Range-basiert
	Cassandra	asynchron	Consistent Hashing
	SimpleDB	asynchron	Shard per Table/Domain
	Azure Tables	asynchron	Expliziter Partitions-Key
Relationale Datenbanken	MySQL Cluster	synchron	Hash-basiert
	VoltDB/H-Store	synchron	Hash-basiert
	Oracle RAC	synchron	- (Shared Disk)
	IBM PureScale	synchron	- (Shared Disk)

Tabelle 2 Replikation und Sharding in horizontal skalierbaren Datenbanksystemen (cf. [1], [3], [4], [11], [21], [47]–[54])

In der Konsequenz zeigt diese Betrachtung also, dass Web-Caches die Skalierungsmechanismen Sharding und Replikation implementieren, die viele Datenbanksysteme neu implementieren. Tabelle 2 zeigt eine relevante Auswahl an horizontal skalierbaren Datenbanksystemen und ihre Verwendung von Replikation und Sharding. Replikation kann in zwei verschiedene Ansätze unterteilt werden: synchrone und asynchrone Replikation. In Abhängigkeit von den Verfügbarkeits- und Konsistenzgarantien des Systems werden beide Varianten umgesetzt. In neueren NoSQL Systemen überwiegt dabei häufig die asynchrone Replikation, da sie geringe Latenzen durch den Verzicht auf unmittelbare Konsistenz gestattet. Sharding (auch Partitionierung genannt) basiert meist auf einer Aufteilung des Schlüsselraums in zusammenhängende Bereiche (*Range Partitioning*) oder einer deterministischen Knotenzuteilung durch einen Hashwert der Schlüssel (*Hash Partitioning*) [55]. Erfolgt der lesende Zugriff stets als Punktzugriff über den Schlüssel, ist ein hash-basierter Ansatz vorteilhaft, da er die gleichmäßigste Aufteilung der Daten auf Knoten erzielt. Web-Cache Cluster (z.B. auf Basis von CARP) können stets nur hash-basiert arbeiten, da sie keine Kenntnis über die interne Struktur der Schlüssel besitzen und deshalb auf URLs und Metadaten (z.B. ETags) zurückgreifen müssen. Für ORESTES würde eine Range-basierte Partitionierung keinerlei weitere Vorteile bringen, da die Replikate (Web-Caches) ohnehin keine Transaktionen und Query-Verarbeitung unterstützen. In ORESTES werden deshalb Query-Ergebnisse als URL-Liste von

selektierten Objekten übertragen, die anschließend über direkten Schlüsselzugriff geladen werden.

Insgesamt werden in den diversen Umsetzungen von Sharding und Replikation Implementierungsfehler wiederholt, die erst durch jahrelangen praktischen Einsatz gefunden und behoben werden können. Wir vertreten deshalb die These, dass es vorteilhaft ist, auf die soliden und gut erforschten Web-Caches als Skalierungsmechanismus zurückzugreifen und den Nachteilen ihres simplizistischen Modells auf Protokollebene zu begegnen. Zwei Punkte unterscheiden Web-Cache (Cluster) von systemspezifischen Replikations- und Sharding-Techniken [31]:

1. Web-Caches garantieren für ad-hoc Änderungen keine Cache Kohärenz (*asynchrone Replikation / Eventual Consistency*).
2. Web-Caches können keine datenbankspezifische Logik implementieren (z.B. *Synchronisation, Locking, Logging, etc.*).

Um also Web-Caches für Datenbanksysteme nutzbar zu machen, lösen wir beide Probleme. Cache-Kohärenz erreichen wir durch den in dieser Arbeit entwickelten Ansatz zur bloomfilterbasierten Cache-Kohärenz. Des Weiteren setzt die Nebenläufigkeitskontrolle in ORESTES keine strikte Cache Kohärenz voraus, da optimistische Verfahren eingesetzt werden. Diese sind in der Lage, das Lesen von veralteten Cache-Einträgen und die damit einhergehende potentielle Verletzung der Serialisierbarkeit zu erkennen. Der Mangel an datenbankspezifischer Logik ist bei genauer Betrachtung kein Nachteil. Dadurch, dass Web-Caches auf das sehr schnelle Ausliefern von gecachten Objekten spezialisiert sind und keinerlei operativen Overhead wie Synchronisation ausführen müssen, entsteht maximaler Durchsatz. Die Kehrseite liegt für ORESTES darin, dass Clients selbst dafür zuständig sind, Versionsnummern von gelesenen Objekten in einem *Read-Set* zu aggregieren und beim Transaktionscommit an den Server zu übertragen. Auch unterstützten Web-Caches für schreibende Operationen nur eine Write-Through Semantik, d.h. jede Schreiboperation erreicht immer den/die ORESTES-Server. Da der ORESTES-Server nur als ein Wrapper des zugrundeliegenden Datenbanksystems fungiert, liegt die Verantwortung der Write-Skalierbarkeit bei diesem Datenbanksystem. Im Gegenzug kann die gesamte Last navigierender Lesezugriffe, d.h. alle Anfragen, die auf einem Primärschlüssel bzw. einer eindeutigen Objekt-ID basieren, durch Web-Caches beantwortet werden.

Die Arbeitsweise eines Web-Caches bei der Beantwortung eines Objektzugriffs ist in Abbildung 10 dargestellt. Der Client stellt seine Anfrage für ein Objekt als einen HTTP GET Request auf eine URL, die das Objekt identifiziert. Besitzt der Web-Cache das angefragte Objekt bisher nicht, leitet er die Anfrage an die Datenbank weiter. Stellt er jedoch einen Cache-Hit fest, überprüft er die gecachte Objektkopie anhand der Caching-Metadaten (*max-age=x*) auf Gültigkeit. Ist das Objekt abgelaufen, führt er eine Revalidierung an der Datenbank durch. Dazu wird mit einem *If-None-Match* Header eine konditionale Auslieferung gefordert, die nur erfolgt, wenn sich das Objekt tatsächlich geändert hat. Dies erspart einerseits den server-



seitigen Neuaufbau der Objektrepräsentation und andererseits die Übertragung des Objektes selbst. Ist die Objektkopie jedoch frisch (der zu erwartende Standardfall für Objekte des Working-Sets), so liefert der Web-Cache das gecachte Objekt direkt an den Client aus.

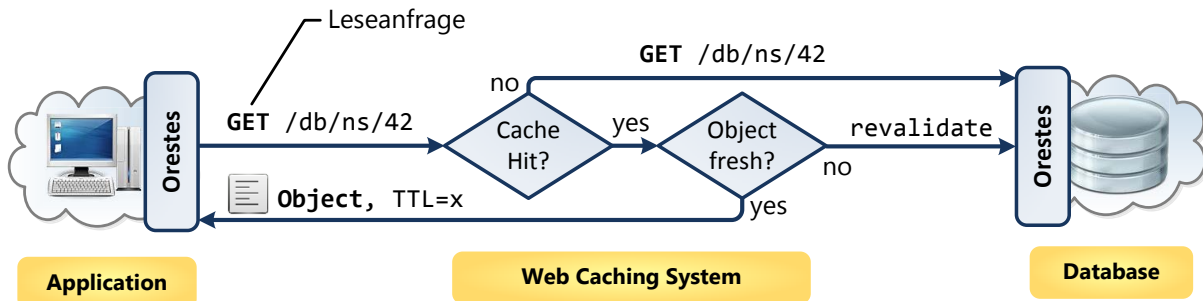


Abbildung 10 Arbeitsweise eines Web-Caches.

Wir demonstrieren die Arbeitsweise des Caches an einem sehr einfachen Beispiel. Dazu nutzen wir das Web-Interface, das ORESTES für die Datenbank bereitstellt (Abbildung 12). Über das Web-Interface kann u.a. über eine interaktive JavaScript-Konsole aus dem Browser heraus programmatisch mit der Datenbank kommuniziert werden. Die Persistenz-API ist die JavaScript Persistence API (JSPA), eine im Zuge dieser Arbeit entwickelte Portierung der Java Persistence API (JPA). JSPA wird in Kapitel 5 genauer beschrieben. Ein Beispielskript (Abbildung 11) soll die Extension der Klasse Coffee abfragen und das Attribut name ausgeben. Die Abfrage erfolgt über eine Query mit leerem Prädikat. Da die API asynchron und eventbasiert arbeitet, erfolgt die Ausgabe des Queryergebnisses in einem Callback.

```

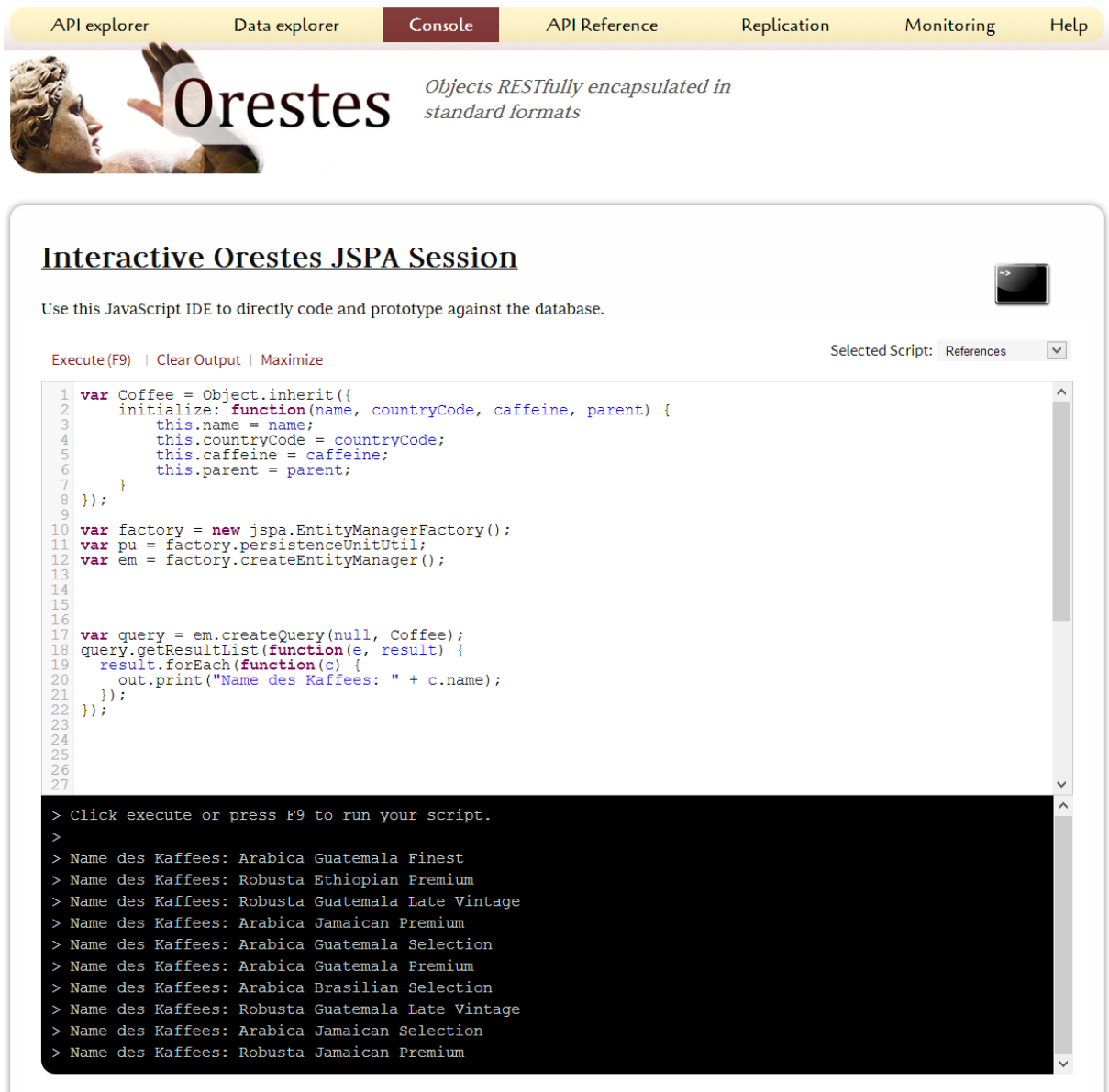
var Coffee = Object.inherit({
  initialize: function (name, countryCode, caffeine, parent) {
    this.name = name;
    this.countryCode = countryCode;
    this.caffeine = caffeine;
    this.parent = parent;
  }
});

var factory = new jspace.EntityManagerFactory();
var pu = factory.persistenceUnitUtil;
var em = factory.createEntityManager();

var query = em.createQuery(null, Coffee);
query.getResultList(function (e, result) {
  result.forEach(function (c) {
    out.print("Name des Kaffees: " + c.name);
  });
});

```

Abbildung 11 Ein einfaches Beispiel, das mit JSPA alle Objekte einer Klasse lädt.



API explorer Data explorer **Console** API Reference Replication Monitoring Help

Orestes

Objects RESTfully encapsulated in standard formats

Interactive Orestes JSPA Session

Use this JavaScript IDE to directly code and prototype against the database.

Execute (F9) | Clear Output | Maximize Selected Script: References

```

1 var Coffee = Object.inherit({
2   initialize: function(name, countryCode, caffeine, parent) {
3     this.name = name;
4     this.countryCode = countryCode;
5     this.caffeine = caffeine;
6     this.parent = parent;
7   }
8 });
9
10 var factory = new jspa.EntityManagerFactory();
11 var pu = factory.persistenceUnitUtil;
12 var em = factory.createEntityManager();
13
14
15
16
17 var query = em.createQuery(null, Coffee);
18 query.getResultList(function(e, result) {
19   result.forEach(function(c) {
20     out.print("Name des Kaffees: " + c.name);
21   });
22 });
23
24
25
26
27

```

```

> Click execute or press F9 to run your script.
>
> Name des Kaffees: Arabica Guatemala Finest
> Name des Kaffees: Robusta Ethiopian Premium
> Name des Kaffees: Robusta Guatemala Late Vintage
> Name des Kaffees: Arabica Jamaican Premium
> Name des Kaffees: Arabica Guatemala Selection
> Name des Kaffees: Arabica Guatemala Premium
> Name des Kaffees: Arabica Brazilian Selection
> Name des Kaffees: Robusta Guatemala Late Vintage
> Name des Kaffees: Arabica Jamaican Selection
> Name des Kaffees: Robusta Jamaican Premium

```

Abbildung 12 Ausführung eines einfachen Querys über das ORESTES Web-Interface.

Das Resultat der Abfrage sind auf REST/HTTP Ebene mehrere Requests. Im ersten Schritt wird die Extension der Klasse Coffee als Liste von URLs abgerufen (Abbildung 13). Im zweiten Schritt werden parallel alle in der Liste enthaltenen Objekte abgerufen, die alle aus dem Browser-Cache beantwortet werden (Abbildung 14). Für die zweistufige Abfrage gibt es zwei Gründe:

1. Queryergebnisse sind dynamisch und können deshalb nur sinnvoll von dem Datenbanksystem beantwortet werden. Bisherige Ansätze, Query-Ergebnisse dediziert zu cachen, basieren auf vollständigen Datenbanksystemen in der Nähe der Clients und waren wenig erfolgreich (DBProxy [56], DBCache [57]).
2. Damit Objekte so schnell wie möglich in die Web-Caches gelangen, ist es notwendig, sie einzeln abzufragen und dadurch separat cachebar zu machen.



```

GET /db/test.persistent/Coffee/all_objects HTTP/1.1
Accept: application/json
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding: gzip,deflate, sdch
Accept-Language: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.11
(KHTML, like Gecko) Chrome/23.0.1271.64 Safari/537.11
Host: try.ORESTES.info
Referer: http://try.ORESTES.info/web/console

HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0
Date: Tue, 13 Nov 2012 14:47:47 GMT
Server: Restlet-Framework/2.1snapshot
Vary: Accept
Content-Length: 1500
Content-Type: application/json; charset=UTF-8

[
  "/db/test.persistent/Coffee/1125899906847742",
  "/db/test.persistent/Coffee/1125899906847757",
  "/db/test.persistent/Coffee/1125899906847739",
  "/db/test.persistent/Coffee/1125899906847754",
  "/db/test.persistent/Coffee/1125899906847736",
  "/db/test.persistent/Coffee/1125899906847751",
  "/db/test.persistent/Coffee/1125899906847733",
  "/db/test.persistent/Coffee/1125899906847748",
  "/db/test.persistent/Coffee/1125899906847745",
  "/db/test.persistent/Coffee/1125899906847760"
]

```

Abbildung 13 Abfrage der Extension der Klasse „Coffee“.

Trotz des Umstandes, dass Objekte für die Dauer einer statisch festgelegten Zeitspanne aus Web-Caches ausgeliefert werden, müssen die transaktionalen Garantien erfüllt werden. Sofern das zugrundeliegende Datenbanksystem ACID Transaktionen bereitstellt, werde diese auf Ebene des ORESTES-Protokolls ebenfalls bereitgestellt. Ohne bloomfilterbasierte Cache Kohärenz werden ACID Garantien unter einer gesteigerten Gefahr von Transaktionsabbrüchen hergestellt. Bei der Validierung des Read- und Write-Sets des Clients ermittelt der Server beim Transaktionscommit, ob der Client zuvor veraltete Objekte gelesen hat. Objekte sind veraltet, falls die beim Schreiben monoton ansteigende Versionsnummer eines Objektes in der (konsistenten) Datenbank größer ist, als die entsprechende Versionsnummer des Read-Sets. Die Details der Transaktionsverwaltung in ORESTES werden in Kapitel 3 dargestellt.

Mit den Techniken, die in dieser Arbeit entwickelt wurden, sind in ORESTES vier Konsistenzmodi wählbar (die Bezeichnungen sind an Ramakrishnan [58] orientiert):

1. **Read-Any:** Eine solche Anfrage kann mit Objekten aus Web-Caches beantwortet werden. Es gibt keine Garantie für die Aktualität des Objekts.
2. **Read-Up-To-Date:** Eine nichttransaktionale konsistente Anfrage wird von der Datenbank beantwortet und garantiert deshalb die aktuellste konsistente Sicht.
3. **Classic Transaction:** In einer herkömmlichen Transaktion können potentiell alle Objekte aus Web-Caches geladen werden. Stale Reads werden zum Commitzeitpunkt erkannt.
4. **Coherent Transaction:** Bei dieser bloomfiltergestützten Transaktion lädt der Client beim Transaktionsbeginn einen Bloomfilter. Dieser repräsentiert Änderungen in der letzten Caching-Zeitspanne. Der Client fragt die im Bloomfilter enthaltenen, geänderten Objekte stets direkt vom Server an und vermeidet so Stale Reads.

```

GET /db/test.persistent/Coffee/1125899906861158 HTTP/1.1
Host: try.ORESTES.info
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.11
(KHTML, like Gecko) Chrome/23.0.1271.64 Safari/537.11
Content-Type: application/json
accept: application/json
Referer: http://try.ORESTES.info/web/console
Accept-Encoding: gzip,deflate,sdch
Accept-Language: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

HTTP/1.1 200 OK
Cache-Control: max-age=604800, public, stale-if-error=3600, stale-
while-revalidate=30
Date: Tue, 13 Nov 2012 15:11:35 GMT
Server: Restlet-Framework/2.1snapshot
Vary: Accept
Content-Length: 1500
Content-Type: application/json; charset=UTF-8
ETag: "3"

{
  "_objectInfo": {
    "oid": "/db/test.persistent/Coffee/1125899906861158",
    "class": "/db/test.persistent/Coffee",
    "version": "3"
  },
  "/db/test.persistent/Coffee": {
    "caffeine": false,
    "name": "Arabica Ethiopian Late Vintage",
    "countryCode": 175,
    "parent": "/db/test.persistent/Coffee/1125899906861149"
  }
}

```

Abbildung 14 Abfrage einer Instanz der Klasse „Coffee“.



1.1.2.3 Implementierung

Die Systeme, die an einer Anfrage über ORESTES beteiligt sind, sollen nun anhand eines Beispiels betrachtet werden (Abbildung 15). Eine Anfrage soll die schon vorhandene Kopie eines Objektes revalidieren. Dies ist beispielsweise notwendig, wenn nach Ende einer Transaktion Objekte lokal gespeichert bleiben und zu einem späteren Zeitpunkt in einer neueren Transaktion weiterverwendet werden sollen. Die Persistenz-API des Client wird in diesem Fall eine Revalidierung durchführen, um die Objekte von ihrem potentiell inkonsistenten Zustand (*detached*) in einen konsistenten Zustand (*persistent clean*) zu überführen.

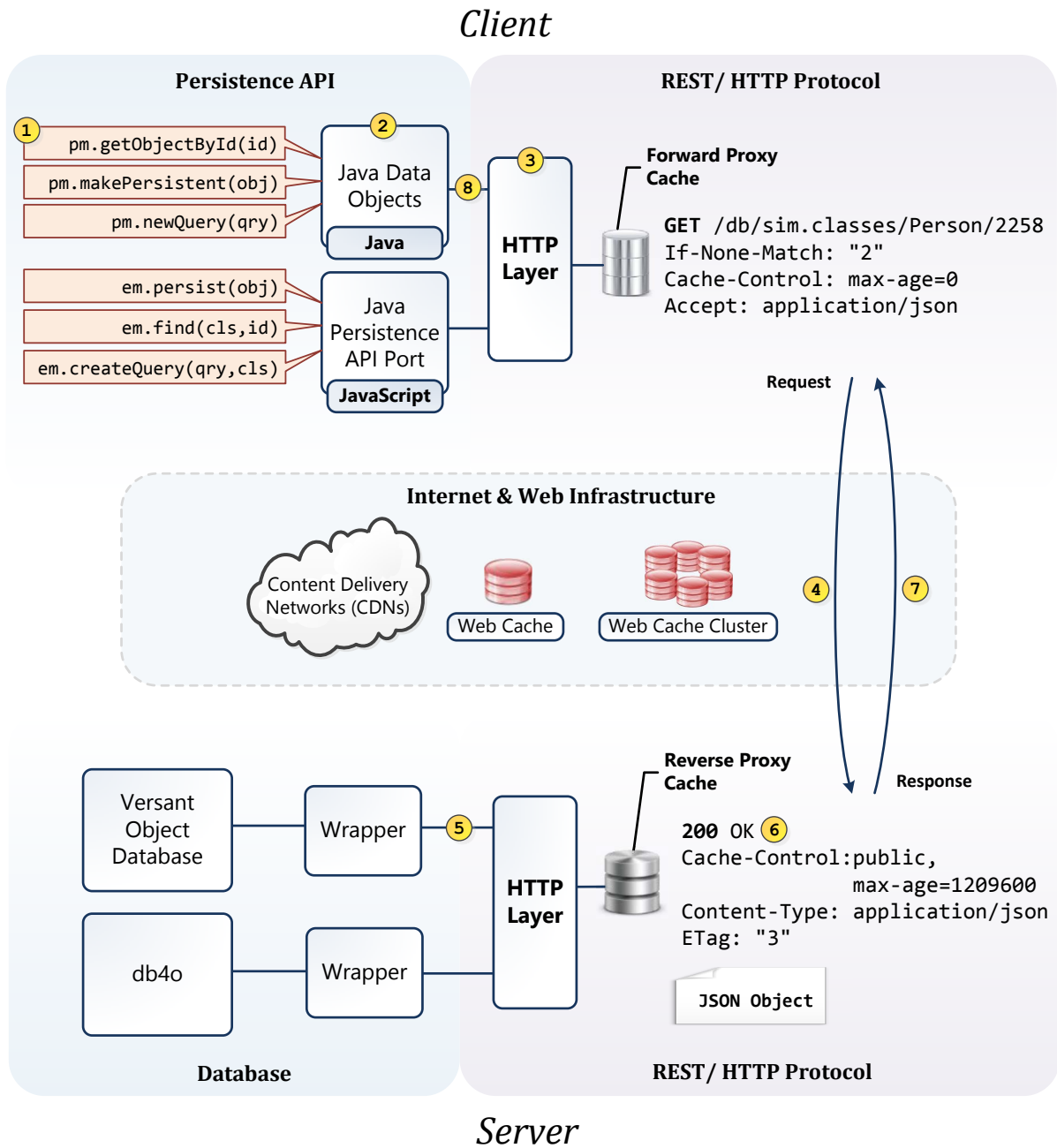


Abbildung 15 Übersicht beteiligter Systeme bei einer konditionalen Leseanfrage für ein Objekt.

Die Revalidierungsanfrage vollzieht sich wie folgt:

1. Die Applikation ruft an der Persistenz-API eine Operation auf, die eine Revalidierung eines schon vorhandenen Objekts erfordert (z.B. `getObjectById(id)`).
2. Die Persistenz-API leitet diesen Aufruf an das ORESTES Framework weiter. Das ORESTES Framework ist eine Implementierung des ORESTES REST/HTTP Protokolls in Java.
3. Das ORESTES Framework sendet einen konditionalen HTTP GET Request, um das übergebene Objekt zu revalidieren.
4. Forward-Proxy Caches im Netzwerk des Client leiten diesen Requests unverändert weiter. Analog verfahren CDNs und Web Proxy Cache in Netzwerken von ISPs und Reverse-Proxy Caches im Servernetzwerk. So erreicht der Aufruf die HTTP Schicht der Datenbank, d.h. das ORESTES Framework.
5. Das ORESTES Framework leitet die Revalidierung an den datenbankspezifischen Wrapper weiter. Dieser überprüft, ob das Objekt noch valide ist und gibt das Ergebnis zurück an das ORESTES Framework.
6. In Abhängigkeit des Revalidierungsergebnisses überträgt das ORESTES-Framework ein `304 Not Modified` an den Client oder ein `200 OK` mit dem aktuellen Objekt im JSON Format.
7. Jedes Caching-System, das die Antwort weiterleitet, aktualisiert ggf. seine gespeicherte Kopie und die Caching Metadaten (d.h. Versionsnummer und Cachingdauer).
8. Der Aufruf erreicht den Client, wo das ORESTES Framework das Revalidierungsergebnis an die Persistenz-API zurückgibt.

Wie Abbildung 15 zeigt, unterstützt die Implementierung von ORESTES derzeit zwei Persistenz-APIs. Persistenz APIs bilden Aufrufe an das Datenbanksystem auf das ORESTES REST/HTTP Protokoll ab:

- **Java Data Objects (JDO):** JDO war über viele Jahre die dominierende standardisierte API, um in Java Objekte persistent zu speichern [27]. Häufig wurde es in Kombination mit einem objekt-relationalen Mapper wie Hibernate, EclipseLink, DataNucleus etc. eingesetzt. Mit der Java Persistence API (JPA), die JDO in fast allen Punkten sehr ähnlich ist, existiert seit einigen Jahren eine weitere standardisierte Persistenz-API für Java. Die JDO Implementierung von ORESTES basiert auf der Codebasis von Versant Inc. und bildet dort alle Operationen auf das generische ORESTES Framework ab. Das ORESTES Framework überführt diese Aufrufe wiederum in die REST/HTTP Netzwerkkommunikation des ORESTES Protokolls.
- **JavaScript Persistence API (JSPA):** Da JavaScript sich einer rasant zunehmenden Beliebtheit im Umfeld von Web Apps, Mobile Apps, NoSQL und serverseitiger Webentwicklung erfreut, haben wir eine objektorientierte Persistenz-API für JavaScript entworfen [59]. Sie entspricht dabei in allen Punkten, die es erlauben, der Spezifikation der Java Persistence API [28]. JSPA kann sowohl in allen gängigen Browsern als auch in dem eventbasierten Webserver Node.JS ausgeführt werden [60].



Auf der Serverseite agiert ORESTES als Wrapper der Datenbank und bildet das REST/HTTP Protokoll auf die native Datenbankschnittstelle ab. Derzeit existieren drei Anbindungen, die im Zuge dieser Masterarbeit alle weiter- oder neu entwickelt wurden. Die Redis Implementierung wurde explizit implementiert, um die Übertragbarkeit der Ergebnisse dieser Arbeit auf andere Datenmodelle und Datenbanksysteme zu demonstrieren:

- **Versant Object Database (VOD):** VOD ist der Marktführer unter den objektorientierten Datenbanken [32]. Die Anbindung an VOD entstand im Zuge einer Projekt-Kooperation.
- **Db4o:** Db4o ist ein objektorientiertes Open-Source Datenbanksystem [33]. Es erzielt nicht die gleiche Leistung wie VOD, ist allerdings portabel und kann auch direkt in Anwendungen eingebettet werden.
- **Redis:** Redis ist eine neuere Open-Source NoSQL Datenbank der Kategorie *Key-Value Store*. Da Redis Operationen In-Memory durchführt, ist es hochperformant.

Die Details der ORESTES Implementierung werden in Kapitel 5 diskutiert.

1.2 Cloud Computing

Wie in dem Anforderungskatalog bereits dargestellt, muss ein Datenbanksystem mehrere Eigenschaften erfüllen, damit es gewinnbringend in Cloud Umgebungen eingesetzt werden kann. Auch klassische Datenbanksysteme (z.B. DB2, Oracle, MSSQL, PostgreSQL, Sybase, Teradata, etc.) können in Cloud Umgebungen eingesetzt werden. Da sie jedoch nicht explizit für einen derartigen Einsatz entwickelt wurden, haben sie dort viele Nachteile und Einschränkungen. Neuere Datenbanksysteme, die meist unter der Bezeichnung *NoSQL* geführt werden, greifen die Entwicklung des Cloud Computings auf und unterstützen sie aktiv. Der wichtigste Punkt, in dem sich dies äußert, ist das Zugriffsprotokoll. Für sprachübergreifende Interoperabilität und lose-gekoppelte Service-orientierte Architekturen haben sich dabei REST-APIs durchgesetzt. Dies ist ein gewaltiger Vorteil gegenüber klassischen Protokollen, die ausschließlich proprietäre Binärschnittstellen umsetzen.

Der Unterschied beider Ansätze ist in Abbildung 16 gezeigt. Eine Java Anwendung, die mit einem relationalen Datenbanksystem kommuniziert, benötigt dafür einen Treiber des jeweiligen Herstellers. Dieser Treiber bildet typischerweise Methoden der Java Database Connectivity API (JDBC) auf Aufrufe an einer auf dem Client installierten herstellerspezifischen Bibliothek ab (sogenannter Typ-2 Treiber). Diese überführt die Aufrufe in das Binärprotokoll des Datenbanksystems. Die Binärprotokolle wurden über Jahre hinweg erweitert und sind durchweg auf die Implementierung des Datenbanksystems zugeschnitten und damit sehr eng an Hersteller und Datenbanksystem gekoppelt. Moderne NoSQL Datenbanken, die REST APIs anbieten, unterscheiden sich grundlegend von diesem Ansatz. Dort kann aus jeder beliebigen Programmiersprache direkt mit der Datenbank kommuniziert werden. Ermöglicht wird dies durch eine native REST API der Datenbanken, die für Clients sehr leicht zu konsumieren ist.

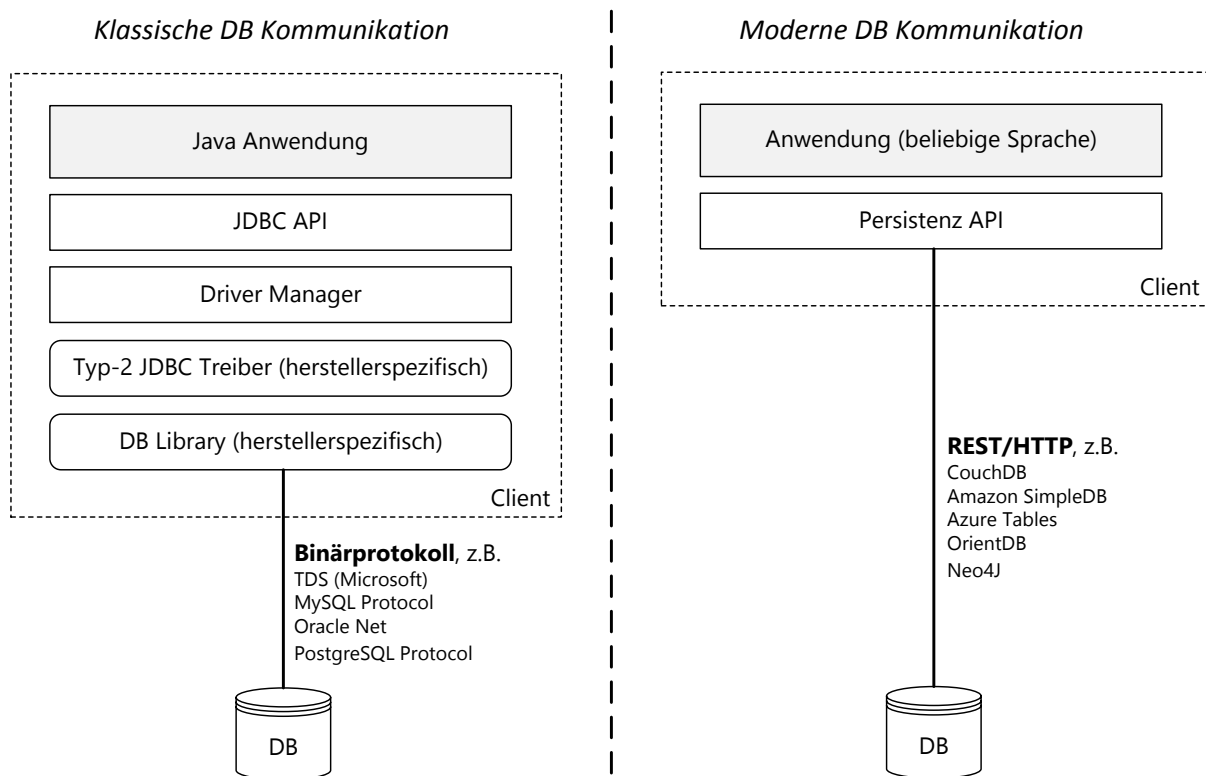


Abbildung 16 Klassische und moderne Datenbankkommunikation

In Bezug auf Cloud Computing haben REST Schnittstellen überwältigende Vorteile gegenüber binären, proprietären TCP-Protokollen:

- Nutzung der Datenbank aus Platform-as-a-Service (PaaS) Clouds, die meist binäre TCP Verbindungen verbieten jedoch HTTP Kommunikation gestatten (z.B. Google App Engine [2]).
- Verwendung der Datenbank aus jeder gängigen Programmiersprache, da HTTP Teil jeder Standardbibliothek ist und die Programmiersprachenvielfalt zunimmt [61].
- Nutzung existierender, offener Standards (z.B. JSON und XML) mit den dazugehörigen Tools, Frameworks und Monitoring Werkzeugen (z.B. Pingdom [2]).
- Verwendung der Datenbank aus mobilen Geräten (HTML5 und native Apps) und Webbrowsern (JavaScript/Ajax), da JavaScript HTTP Kommunikation unterstützt.
- Leichtes Deployment, Monitoring, etc. über die existierende HTTP Schnittstelle der Datenbank und Webbrowser möglich.
- Nutzung der Web-Infrastruktur: Load-Balancer und Web-Caches.
- Weiterentwicklung der Schnittstelle im laufenden Betrieb durch neue Ressourcen und erweiterte Medientypen.
- Simple Versionierung der Schnittstelle durch fehlertolerante Medientypen.
- Lose Kopplung zwischen Datenbank und Client API, die weitgehende Implementierungsfreiheiten einräumt.
- Erleichtertes Debugging, da die Kommunikation menschenlesbar ist.
- Einfache Entwicklung von Administrationswerkzeugen.



Bisher werden meist wenige der aufgeführten Vorteile von REST-APIs in NoSQL Datenbanken genutzt. Mit ORESTES wollen wir zeigen, dass dies jedoch möglich ist. Im Mai 2012 brachte *ProgrammableWeb*, das größte Verzeichnis von Web Services, einen Artikel über gelistete Service-Schnittstellen für Datenbanken mit dem Titel „123 Database APIs“ [62]. Wie Abbildung 17 zeigt, sind 61% dieser APIs als REST-APIs umgesetzt. Während nicht alle dieser Schnittstellen reine Datenbankschnittstellen sind (z.B. Freebase als semantisches Begriffsverzeichnis), entfällt ein Großteil auf Database-as-a-Service Schnittstellen für NoSQL Datenbanken. Dies zeigt einen großen Nachteil der REST-APIs verschiedener Datenbanken – es fehlt an Universalität. Da REST als Architekturstil Erweiterbarkeit und Modularisierung als Kernkonzept besitzt, wollen wir das ORESTES REST Protokoll mit der notwendigen Universalität ausstatten. So können beispielsweise die gleichen Ressourcen und Formate für Transaktionen verwendet werden, unabhängig davon, ob ein objektorientiertes oder Key-Value Datenmodell vorliegt. Kapitel 5 beschreibt die Umsetzung dieser universellen Schnittstelle.

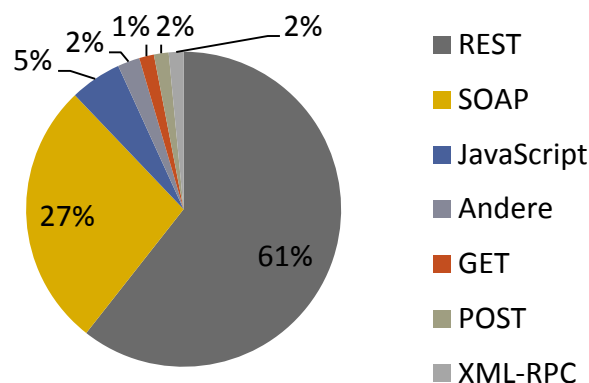


Abbildung 17 Protokolle von 123 Datenbank-APIs (aus [62]).

Tabelle 3 zeigt eine Untersuchung von REST APIs der zuvor auf Sharding und Replikation geprüften Datenbanksysteme, erweitert um diejenigen NoSQL Datenbanken, die eine REST Schnittstelle anbieten. Die Untersuchung zeigt, dass nahezu alle neueren NoSQL Datenbanken REST Schnittstellen bereits anbieten. Für alle übrigen bestehen meist Pläne, künftig eine REST-API anzubieten oder die Community hat experimentelle REST Schnittstellen bereits entwickelt. Der Fokus vieler NoSQL Systeme auf geringe Latenzen und hohen Durchsatz räumt überdies das Vorurteil aus, dass REST Schnittstellen nicht für Performance-kritische Anwendungen geeignet sind. Die untersuchten REST-API variieren stark in der Güte ihrer Umsetzung. Auch die ausgereifteren Umsetzungen haben teilweise Einschränkungen, die wir mit dem ORESTES REST Protokoll umgehen wollen. Nachfolgend sind einige der anzutreffenden Einschränkungen aufgeführt:

- Fehlende Unterstützung von **Web-Caching**. Dies gilt für jede der untersuchten REST Schnittstellen.
- Verletzung der Statuslosigkeit, die **Load-Balancing** der HTTP Requests ermöglichen würde.
- Fehlende **Erweiterbarkeit** auf neue Daten- und Konsistenzmodelle.

- Mangelhafte Umsetzung der **REST Constraints**, z.B. Hypermedia. Dies führt u.a. zu inkompatiblen API Umstellungen und schlechter Resilienz der Clients.

	Datenbanksystem	REST-Schnittstelle
Key-Value Stores	Redis	nein
	Riak	ja
	Voldemort	nein
	Couchbase	nein (nur für Administration)
	Dynamo	ja (Amazon DynamoDB)
	Simple Storage Service	ja
	Azure Blobs	ja
Dokumenten Datenbanken	CouchDB	ja
	MongoDB	nur wenige Leseoperationen, volle REST APIs (Prototypen) extern
	ArangoDB	ja
Wide Column Stores	RavenDB	ja
	BigTable	nein (Google intern)
	MegaStore	nein (Google intern)
	HBase	ja (externes StarGate Projekt)
	Hypertable	nein
	Cassandra	nein (einige Prototypen; DBaaS mit REST API: Cassandra.io)
	SimpleDB	ja
Graphen Datenbanken	Azure Tables	ja
	OrientDB	ja
	Neo4j	ja
	InfoGrid	ja
Relationale Datenbanken	AllegroGraph	ja
	MySQL Cluster	nein
	VoltDB/H-Store	nein (nur für Administration)
	Oracle RAC	nein
	IBM PureScale	nein

Tabelle 3 Datenbanksysteme und REST/HTTP Schnittstellen, Stand November 2012 (cf. [3], [11], [30], [42], [51], [63]–[65]).

Ein wichtiges Ziel dieser Arbeit besteht also darin, das ORESTES Protokoll von den Mängeln der untersuchten REST Schnittstellen zu bereinigen und damit fähig für den Einsatz in Cloud Computing Umgebungen zu machen. Der besondere Fokus gilt dabei dem Alleinstellungsmerkmal des Umgangs mit Web-Caching und Load-Balancing.

1.2.1 Database-as-a-Service

Bei der Nutzung von Datenbanksystemen im Cloud Computing (*Cloud Data Management*) unterscheiden wir zwei Freiheitsgrade: den Ort der Anwendung und das Deployment des Datenbanksystems. Eine Anwendung kann entweder in der gleichen Cloud wie das Datenbanksystem ausgeführt werden oder außerhalb der Cloud. Die Datenbank selbst kann über



ein Database-as-a-Service Modell bereitgestellt werden oder durch den Mandanten selbst aufgesetzt, administriert und gewartet werden.

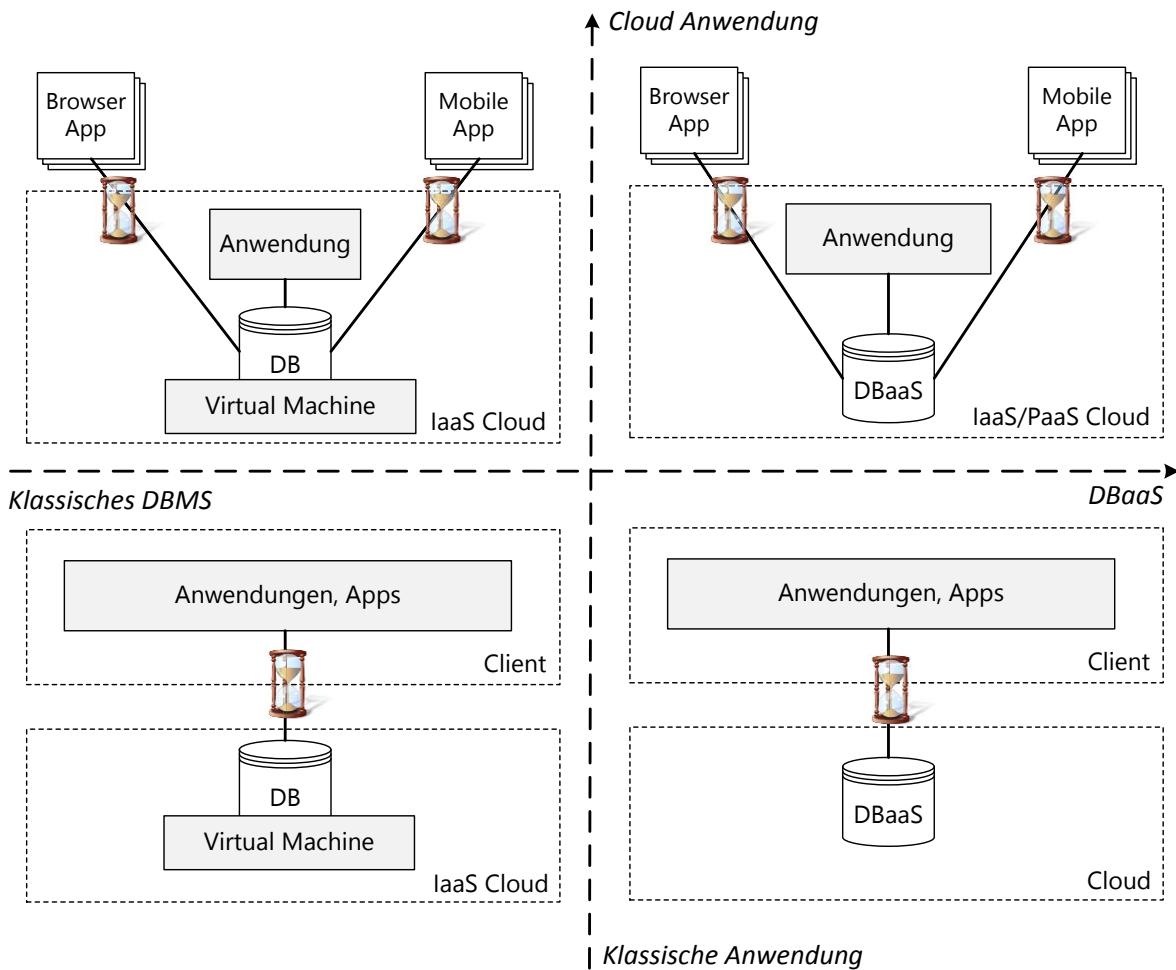


Abbildung 18 Die vier Szenarien für die Beziehung zwischen Anwendung und Datenbank.

Abbildung 18 verdeutlicht die vier möglichen Kombinationen. Jede der Kombinationen wird bereits praktisch eingesetzt [2]. Die Abbildung zeigt außerdem die Beziehungen, die potentiell durch eine hohe Netzwerklatenz gefährdet sind. Um die entsprechenden Szenarien dennoch nutzbar zu machen, gilt es das Latenzproblem zu lösen. Durch ein Deployment der Anwendung in geographischer Nähe zum Datenbanksystem lässt sich für klassische Anwendungen das Latenzproblem meist lösen. Für Datenbanken in Cloud Umgebungen ist die Situationen jedoch anders – nicht jede Anwendung kann in der Cloud ausgeführt werden. So gibt es Anwendungen, die auf bestehender Infrastruktur aufbauen oder Daten verwenden, die aus legalen Gründen oder aufgrund von Governance-Bestimmungen nicht in die Cloud verlagert werden können. Derartige Anwendungen sind also zwangsläufig einer höheren Latenz ausgesetzt, sobald sie mit einer Clouddatenbank arbeiten. Gleiches gilt für Browser Anwendungen und mobile Apps, die unmittelbar mit der Datenbank kommunizieren. Diese Anwendungsarchitektur, die ohne Application Server zur Kapselung der Datenbank auskommt, gewinnt zunehmend an Verbreitung [13]. Dies äußert sich einerseits in DBaaS Angeboten, die auf Schnittstellen für mobile und Browseranwendungen setzen (z.B. MongoLab

und Cassandra.io) und andererseits durch NoSQL Datenbanksysteme, die das Konzept clientseitiger Persistenz direkt in das Design der Datenbank aufnehmen (z.B. CouchDB). In der Konsequenz heißt das also, dass sich trotz des ungelösten Latenzproblems bereits neue Anwendungsarchitekturen auf Basis von Clouddatenbanken durchsetzen. Diese Architekturen würden von einer Lösung der Latenzproblems ungemein profitieren.

In den letzten zwei Jahren sind zahlreiche kommerzielle DBaaS Angebote entstanden. Die DBaaS Provider spezialisieren sich zumeist auf ein Datenbanksystem. Sie bieten für das Datenbanksystem ein automatisiertes Deployment in einer wählbaren Cloud bzw. Region eines Cloud Providers. Der Kunde erhält einen Service Endpunkt (also in fast allen Fällen die URL einer REST-API) und kann sich aus der Persistenz API seiner Anwendung direkt mit der Datenbank verbinden. Alle DBaaS Provider bieten darüber hinaus ein Browser Konsole, in der die einzelnen Datenbanken verwaltet werden können. Einige bieten zusätzlich darüber hinausgehende Funktion an, wie das Erstellen neuer Replikate und Shards, automatische Archivierung und Disaster Recovery durch Failover Kopien der Datenbank. Der typische Prozess zur Nutzung eines DBaaS Angebots ist in Abbildung 19 am Beispiel von MongoHQ gezeigt, einem DBaaS Provider für die Dokumentendatenbank MongoDB:

1. Die meisten DBaaS Provider arbeiten nach dem Freemium Modell, d.h. sie bieten Basisdienste kostenlos an und alle darüber hinausgehenden Leistungen gegen eine monatliche Gebühr (Pay-as-you-go). Eine Anmeldung ist meist erforderlich.
2. Nach der Anmeldung kann über eine Web-Konsole eine Datenbankinstanz erstellt werden.
3. Die URL der Datenbankinstanz kann aus der Anwendung benutzt werden, um eine Verbindung mit dem DBaaS herzustellen.

The screenshot illustrates the typical steps for using a DBaaS offering at MongoHQ. It shows the web console interface where a database instance named 'orestes' is managed. A table lists the collections, showing 'test_collection' with 39 documents and 8 KB size. Below the console, a code snippet demonstrates connecting to the instance via a REST API and inserting a document. The terminal output shows the duration of the operation as 113.999843597 ms.

```

c = Connection('mongodb://orestes:BrotBrot@alex.mongohq.com:10008/orestes')
obj = { "User" : "Felix", "message" : "Testing MongoDB", "test" : [1,2,3,4] }
c.orestes.test_collection.insert(obj)

with timer:
    c.orestes.test_collection.find_one( {"User":"Felix" })
timer.show()

Duration: 113.999843597 ms

```

Abbildung 19 Typische Schritte zur Nutzung eines DBaaS Angebots am Beispiel von MongoHQ [66].



Die in Abbildung 19 gezeigte Anwendung lädt ein Dokument aus der Datenbank und misst dabei die Latenz. Das Ergebnis sind ca. 114 ms. Als Vergleichswert für die Latenz eines Web-Caches eines CDNs diene Google mit seinem *Global Cache CDN*. Die Netzwerklatenz (gemessen in Hamburg) beträgt dort 12 ms. Die Latenz zu einem typischen DBaaS unterscheidet sich von der eines CDN Web-Caches also um eine Größenordnung. In Tabelle 4 sind gemessene Netzwerklatenzen für verschiedene Cloud Storage Provider angegeben. Diese wurden mithilfe einer JavaScript Test Suite von Cloudharmony.com ermittelt, die direkt aus dem Browser die Latenz über die REST-API des jeweiligen Anbieters ermittelt (in diesem Fall aus Hamburg). Cloud Storage (z.B. Amazon Simple Storage Service oder Azure Storage) ist ein Spezialfall der DBaaS Kategorie, bei dem das DBaaS nur das Speichern von Blobs (Binary Large Objects) unter einem Schlüssel (URL) erlaubt. Die Bezeichnungen Storage-as-a-Service und Data-as-a-Service sind ebenfalls üblich. Die Latenzmessungen sind jedoch repräsentativ für andere DBaaS Angebote, da diese aus denselben Data-Centern der Cloud Provider bedient werden (meist Amazon Web Services und Windows Azure) und deshalb sehr ähnliche Latenzzeiten aufweisen. Die Ergebnisse von Amazon S3 und Azure Storage fallen am besten aus, da diese Data-Center in Irland besitzen und damit geographisch recht dicht am Messungsort Hamburg liegen. Überdies erlauben beide Anbieter das statische Caching der abgelegten Dateien in ihren jeweiligen CDNs (Amazon Cloudfront, Azure CDN) [67], [68].

Service	Ort	#Samples	Min (ms)	Max (ms)	Std Dev	Median (ms)	Avg (ms)
Simple Storage Service (S3)	CA, US	3	197	203	1.72%	203	201
Internap Cloud Storage		3	190	419	49.3%	192	267
HP Cloud Object Storage	AZ, US	3	191	199	2.25%	192	194
Zetta Enterprise Cloud Storage		3	177	183	1.67%	180	180
Google Storage for Developers		5	148	160	3.35%	155	155
Windows Azure Storage	IL, US	4	143	228	24.28%	150	167.25
Windows Azure Storage	TX, US	4	143	149	1.82%	146	145.5
Simple Storage Service (S3)	VA, US	5	128	136	2.5%	130	131.6
Simple Storage Service (S3)	IE	5	61	73	6.78%	65	66

Windows Azure Storage	NL	7	53	60	5.19%	57	56.57
Windows Azure Storage	IE	4	36	40	4.46%	39	38.25

Tabelle 4 Netzwerklatenz einiger Cloud Storage Provider (ermittelt über CloudHarmony [69] aus Hamburg).

1.2.2 Analyse des DBaaS Ökosystems

Nach der populären NIST Taxonomie für Cloud Computing (cf. [70]) sind Database-as-a-Service Dienste auf gleicher Ebene angesiedelt wie *Software-as-a-Service* (SaaS) und damit eine Schicht über *Platform* und *Infrastructure-as-a-Service* (PaaS und IaaS) Modellen. Die derzeit häufigste Realisierungsform von DBaaS greift auf die Ressourcen eines IaaS Providers zurück und nutzt deren virtuelle Infrastruktur, um automatisiert Datenbankinstanzen bereitzustellen. Einige DBaaS Angebote sind direkt integriert in PaaS Systeme (z.B. Heroku Postgres oder Google App Engine DataStore) und können dort mit geringem Konfigurationsaufwand aus einer PaaS Anwendung genutzt werden.

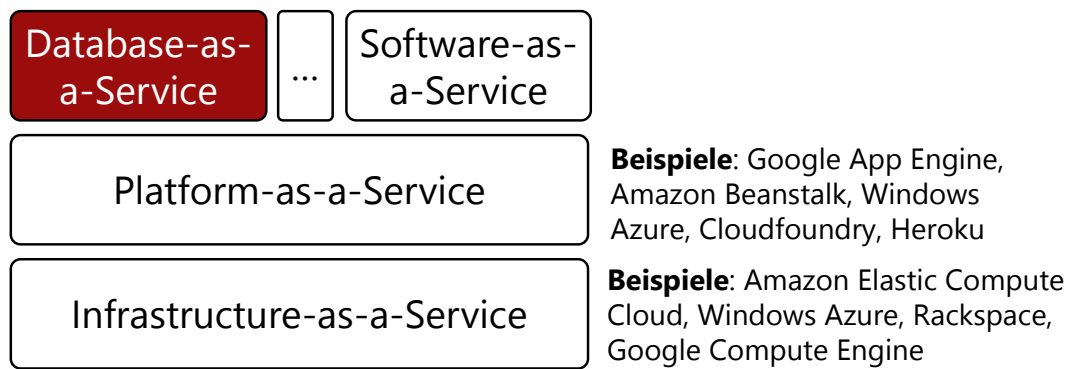


Abbildung 20 Einordnung von DBaaS gemäß der NIST Definition [70].

Tabelle 5 zeigt eine umfassende Analyse existierender DBaaS Angebote, klassifiziert nach Datenmodell und Datenbanksystem. Auffällig ist die große Anzahl von DBaaS Angeboten auf Basis von Redis, MongoDB und MySQL. Die hohe Beliebtheit dieser Angebote ist zurückzuführen auf Web Entwickler, die häufig Erfahrung mit Open-Source Datenbanksystemen haben und nach einer Lösung suchen, um alle administrativen Aufgaben auf ein Minimum zu reduzieren. Die Kehrseite einer hohen Netzwerklatenz wird dabei als Tradeoff meist billigend in Kauf genommen. Einige Anbieter haben zudem erkannt, dass mobile und Browser Apps DBaaS nur dann nutzen können, wenn die Datenbank eine REST/HTTP Schnittstelle anbietet, die aus Programmiersprachen wie JavaScript genutzt werden kann. MongoLab [71] und Cassandra.io [72] bieten deshalb eigene (rudimentäre) REST/HTTP Schnittstellen für MongoDB bzw. Cassandra an, um dieses Nutzungsszenario zu ermöglichen, obwohl das zugrundeliegende Datenbanksystem keine native REST/HTTP Schnittstelle besitzt. Allen aufgeführten DBaaS Ansätzen ist gemeinsam, dass die einzige Art Einfluss auf die Netzwerklatenz zu nehmen in der expliziten Auswahl des IaaS Data-Centers liegt, in dem die Datenbankinstanz ausgeführt wird. Die Netzwerklatenz ist also für alle Angebote



zwangsläufig ein Engpass, sobald die zugreifenden Anwendungen geographisch verteilt sind. Keines der DBaaS Angebote bietet zudem ein einheitliches Protokoll für eine Polyglot Persistence Lösung auf Basis unterschiedlicher Datenbanksysteme. Auch automatische Skalierbarkeit ist in sehr wenigen Systemen anzutreffen, die Ausnahme bilden DynamoDB [73], Azure Tables [74] und BigTable/GAE DataStore [48], [75], denen die breite Infrastruktur von Amazon, Microsoft bzw. Google direkt zu Verfügung steht. Die anderen Dienste arbeiten mit der Abstraktion virtueller Maschinen für die Datenbankknoten mit zugesicherten Ressourcen wie RAM, CPU Zeit und Kommunikationsvolumen. Sie bieten daher keine automatische Skalierung von Reads und Writes. Auch die explizite Erfassung von Service Level Objectives (SLOs) wie Latenz, Verfügbarkeit und Durchsatz, sowie ihre Zusicherung in Service Level Agreements (SLAs) wurde bisher nicht umgesetzt. Die Abrechnung erfolgt i.d.R. über Pläne, d.h. definierte Ressourcenkonfigurationen, die den Knoten des Datenbanksystems zur Verfügung stehen. Die Auswahl eines leistungsfähigeren Plans erfordert dann meist das Stoppen und Klonen des Datenbanksystems zur Provisierung auf einer leistungsfähigeren virtuellen Maschine. Nur wenige Anbieter (z.B. Cloudant [76]) setzen ein Pay-as-you-go Kostenmodell um, das auf Requests und gelesenem/modifiziertem Datenvolumen basiert.

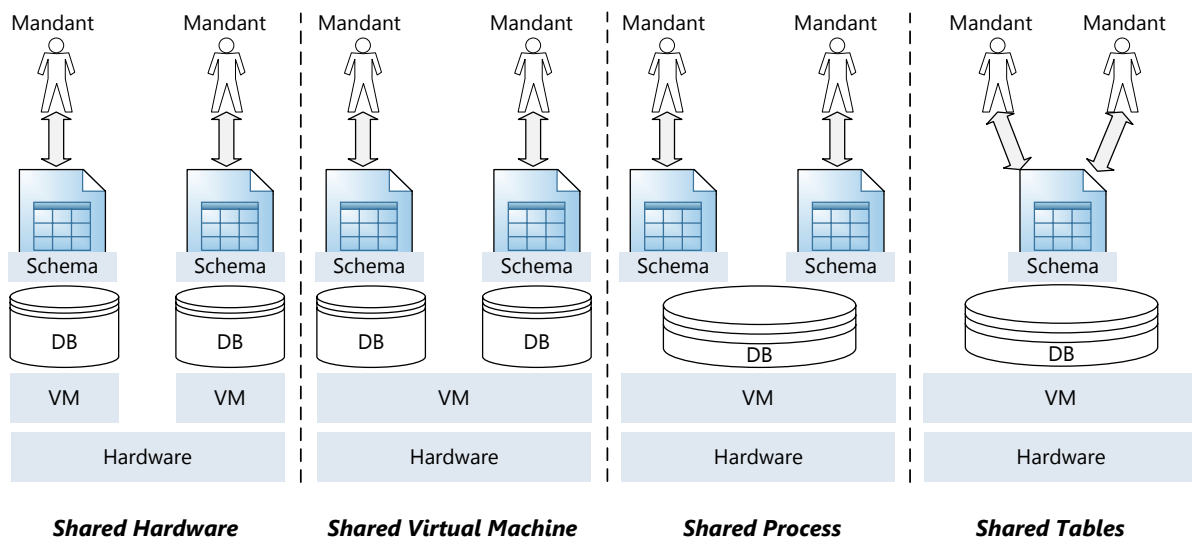


Abbildung 21 Konsolidierungsstrategien für Multitenancy Datenbanken (cf. [77], [78]).

Um die Ressourcennutzung und Kosten zu optimieren, müssen DBaaS Anbieter ihre Ressourcen möglichst effizient nutzen. Deshalb ist es notwendig, mehrere Mandanten auf der gleichen Plattform zu konsolidieren (*Mandantenfähigkeit*, engl. *Multitenancy*). Auf diese Weise können Mandanten mit komplementären Arbeitslasten zusammengelegt werden, ohne dass stets die Kapazität für Spitzenlasten vorgehalten werden müsste [77]. Es lassen sich mehrere Konsolidierungsstrategien unterscheiden (cf. [77]–[79], siehe Abbildung 21):

- **Shared Hardware:** die Hardware ist in virtuelle Maschinen (VMs) aufgeteilt, jedem Mandanten wird eine dedizierte VM mit zugehöriger Datenbank zugewiesen.
- **Shared Virtual Machine:** auf einer VM wird für jeden Mandanten ein eigener Datenbankprozess ausgeführt.

- **Shared Process/Database:** in einem Datenbankprozess steht jedem Mandanten ein eigener Teil des Schemas zur Verfügung. Entweder das Datenbanksystem oder eine dedizierte Schicht des Services sorgt dafür, dass jedem Mandanten nur der eigene Ausschnitt des Schemas zur Verfügung steht. Durch den Shared Database Ansatz treten Synergien auf, da das Datenbanksystem viele Prozesse wie Logging, Pufferverwaltung, Group-Commit u.ä. effizienter durchführen kann.
- **Shared Tables:** der Overhead bezüglich der Metadatenverwaltung und Indexstrukturen kann weiter minimiert werden, wenn allen Mandanten ein gemeinsames Schema bereitgestellt wird [77]. Dies bietet sich für Standard- und SaaS-Anwendungen wie ERP und CRM an, die auf einem definierten Schema basieren.

ORESTES ist mit jeder der vier genannten Konsolidierungsstrategien kombinierbar. Da mandantenfähige Datenbanken bisher selten sind, ist die ORESTES Architektur darauf ausgelegt, Mandantenfähigkeit auch für Datenbanksystem bereitzustellen, die nicht unter diesen Gesichtspunkten konzipiert wurden. Dies wird durch die Entkopplung der REST/HTTP Kommunikationsschicht und dem Datenbanksystem erreicht. Der Wrapper, der mithilfe des ORESTES Frameworks die REST/HTTP Schnittstelle bereitstellt, kann durch den Shared Database Ansatz das mandantenfähige Schemamapping vornehmen (cf. [80]). Ein herkömmliches Datenbanksystem wird dadurch mandantenfähig gekapselt und das Schema logisch partitioniert. Dabei ist entscheidend, dass dieser Shared Database Ansatz auch für schemalose NoSQL Datenbanken (z.B. Dokumentendatenbanken und Key-Value Stores) geeignet ist. Denn auch schemalose NoSQL Datenbanken bieten eine modularisierende Einteilung der Datenbank in Dokumentensammlungen (Collections) bzw. Keyspaces (Buckets) [11]. Dieses Aufteilungsgranulat ist geeignet, um einzelne Mandanten durch ein Schema-Mapping im serverseitigen ORESTES-Datenbankwrapper voneinander zu isolieren.

Unter den in Tabelle 5 aufgeführten DBaaS Angeboten überwiegt die Shared Hardware Konsolidierung, da sie auf Basis von IaaS sehr einfach umgesetzt werden kann. Viele der nach dem Freemium Modell arbeitenden DBaaS Provider bieten eine kostenlose Datenbankinstanz als Einstieg. Diese werden dann zumeist durch eine Shared Database oder Shared Virtual Machine Konsolidierung bereitgestellt. Die kostenpflichtigen Pläne basieren hingegen meist auf dedizierten VMs wählbarer Größe. Ein Randfall von DBaaS ist die Bereitstellung vorkonfigurierter Datenbank VMs in einer IaaS Cloud, wie es Amazon es z.B. für die SAP Hana DB bietet [81]. Da die Verantwortung für Konfiguration, Skalierung, Backup und Zugangskontrolle weiterhin beim Mandanten liegt, betrachten wir dieses Modell nicht als DBaaS. Zusammenfassend zeigen die untersuchten DBaaS Systeme drei besonders schwerwiegende Einschränkungen, zu deren Lösung wir mit dieser Arbeit beitragen wollen:

1. Hohe Netzwerklatenzen für verteilte, mobile und Web Anwendungen
⇒ CDNs und Web-Caching
2. Fehlende Mechanismen zur elastischen und automatischen Skalierbarkeit
⇒ Web-Cache Cluster, CDNs, Workload-gesteuertes Erzeugen von Web-Caches



3. Fehlende oder eingeschränkte REST/HTTP Schnittstellen
 ⇒ ORESTES REST/HTTP Protokoll für Polyglot Persistence

	Datenbanksystem	DBaaS Provider
Key-Value Stores	Redis	Redis2Go [82] RedisGreen [83] RedisCloud [84] OpenRedis [85]
	Memcache	Heroku Memcache [86]
	Simple Storage Service (Amazon Web Services) [42]	
	Azure Blobs (Windows Azure) [64]	
Dokumenten Datenbanken	CouchDB	Cloudant [76] IrisCouch [87]
	MongoDB	MongoHQ [66] MongoLab [71] HostedMongo [88] ObjectRocket [89] MongoOd [90]
	RavenDB	RavenHQ [91] CloudBird [92]
	OrientDB	NuvolaBase [93]
	BigTable	App Engine Data Store [94]
	Treasure Data (Data Warehousing, Tabellen in S3, Hive Queries) [95]	
Wide Column Stores	Cassandra	Cassandra.io [72]
	SimpleDB	Amazon SimpleDB [96]
Graphendatenbank	DynamoDB	Amazon DynamoDB [73]
	Neo4j	Heroku Neo4j [97]
Relationale Datenbanken	MySQL	Amazon Relational Database Service [98] Google Cloud SQL [99] Xeround [100] ScaleDB [101] ClearDB [102] ScaleBase [103]
	Microsoft SQL Server	SQL Azure [104] Amazon Relational Database Service
	Oracle	Oracle Database Cloud Service [105] Amazon Relational Database Service
	PostgreSQL	EnterpriseDB [106] Heroku Postgres [107]
	Clustrix (Sierra Database Engine Appliance, MySQL kompatibel) [108]	
	Zoho (Point and Click Formulare, Business Rules, Analytics, nicht für API Zugriff konzipiert) [109]	
	Database.com (basiert auf Oracle, relationales Schema, REST und SOAP APIs für viele Operationen, Salesforce.com Integration) [110]	

Tabelle 5 Kommerzielle Database-as-a-Service Provider, klassifiziert nach Datenmodell und Datenbanksystem.

1.2.3 Benchmarkergebnisse für ORESTES in einem Cloud Szenario

Um die Eignung des ORESTES Ansatzes für das Cloud Computing zu untersuchen, haben wir Benchmarks entworfen, die das Verhalten des ORESTES Protokolls mit einem klassischen TCP-basierten Datenbankprotokoll verglichen, das nicht im Hinblick auf DBaaS Szenarien entwickelt wurde. Dazu haben wir ein Social Network Klassenmodell entwickelt (Abbildung 22), das vollständig in der Datenbank persistiert wird. Die Benchmark Clients laden transaktional Objekte des Klassenmodells, aktualisieren sie und führen Querys aus. Die Leseoperationen werden dabei sequentiell ausgeführt, um das Muster des navigierenden Zugriffs zu studieren. Der Benchmark besitzt mehrere Freiheitsgrade, zu den wichtigsten zählt das Verhältnis von Reads zu Writes (ausgedrückt durch eine *Read-Ratio*), die Anzahl insgesamt persistent gespeicherter Objekte, die Anzahl an Lese/Schreib Operationen und die Wahrscheinlichkeitsverteilung, die dem zufälligen Auswählen von Objekten zugrunde liegt (diskrete Gleichverteilung gegenüber einer Zipfverteilung). Für weitere Details verweisen wir an dieser Stelle auf unser Paper „ORESTES: a REST protocol for horizontally scalable cloud database access“, das sich derzeit im Review-Prozess befindet.

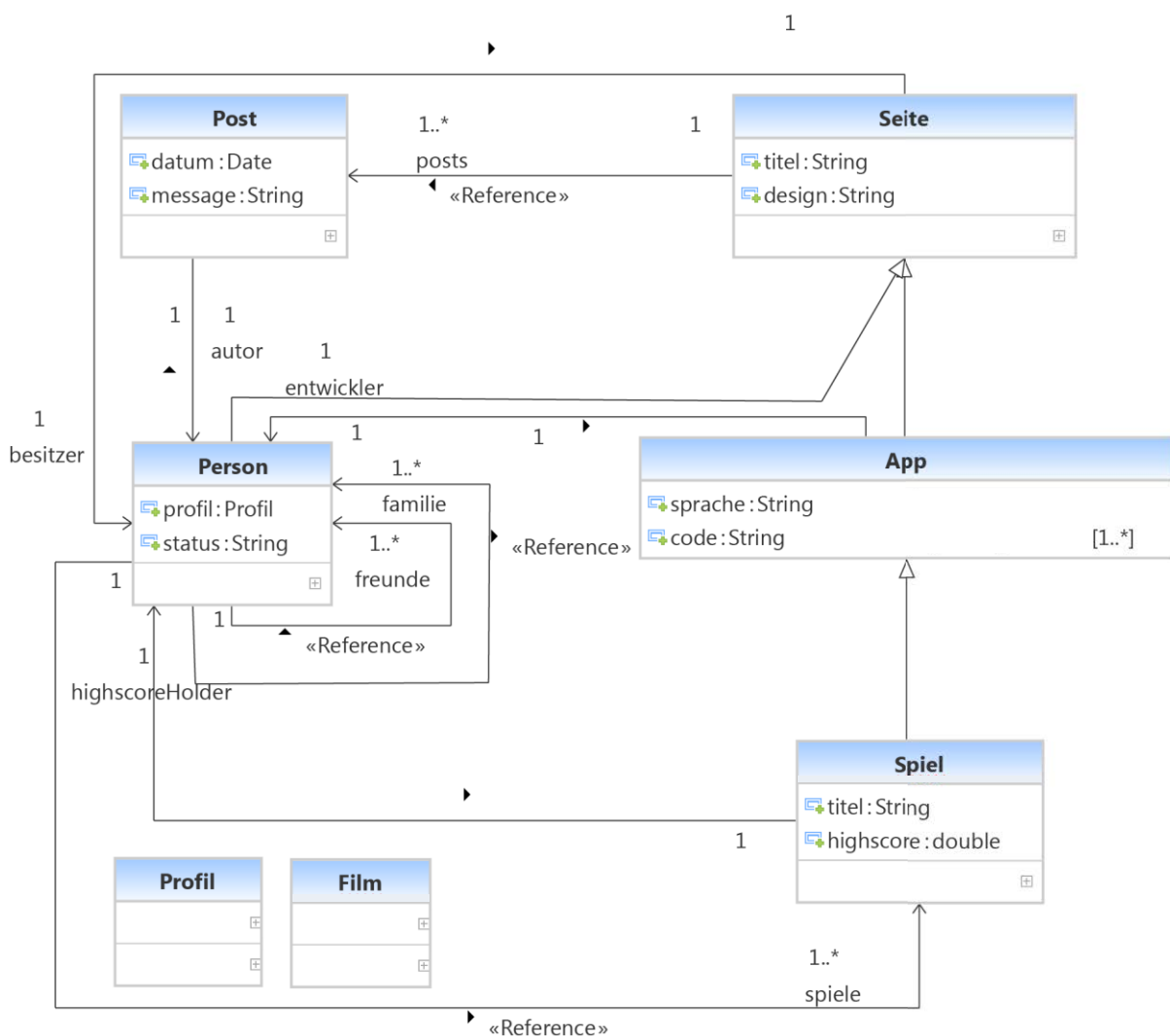


Abbildung 22 UML Klassendiagramm der persistenten Objekte des Benchmarks.



Das Testbed zur Durchführung der Benchmarks wurde in der Amazon Elastic Compute Cloud (EC2) angelegt [42]. Der Aufbau für das massiv-parallele Testszenario ist in Abbildung 23 gezeigt. In der EC2 Region Irland werden 50 Clients ausgeführt, die jeweils über eine eigene VM verfügen. Die Rechenkapazität von VMs in EC2 wird durch *EC2 Compute Units* (ECUs) angegeben, die etwa der Leistung eines 1,0-1,2 GHz 2007 Xeon Prozessors entsprechen. Die Benchmark-Clients verfügen jeweils über eine ECU, 1,7 GByte RAM und Windows Server 2008 R2 als Betriebssystem. Über eine REST Schnittstelle können alle Benchmark Clients gleichzeitig gestartet werden. In der Region der Clients befindet sich außerdem ein Web-Cache (*Squid*), der die Auswirkungen des Web-Cachings demonstrieren soll. In der EC2 Region Kalifornien befindet sich das Datenbanksystem (*Versant Object Database*) und der dazugehörige ORESTES-Server (der Wrapper). Sowohl die VM des Squid als auch der Datenbank verfügen über 4 ECUs und 7,5 GB RAM. Die Netzwerklatenz zwischen beiden Regionen beträgt etwa 165 ms.

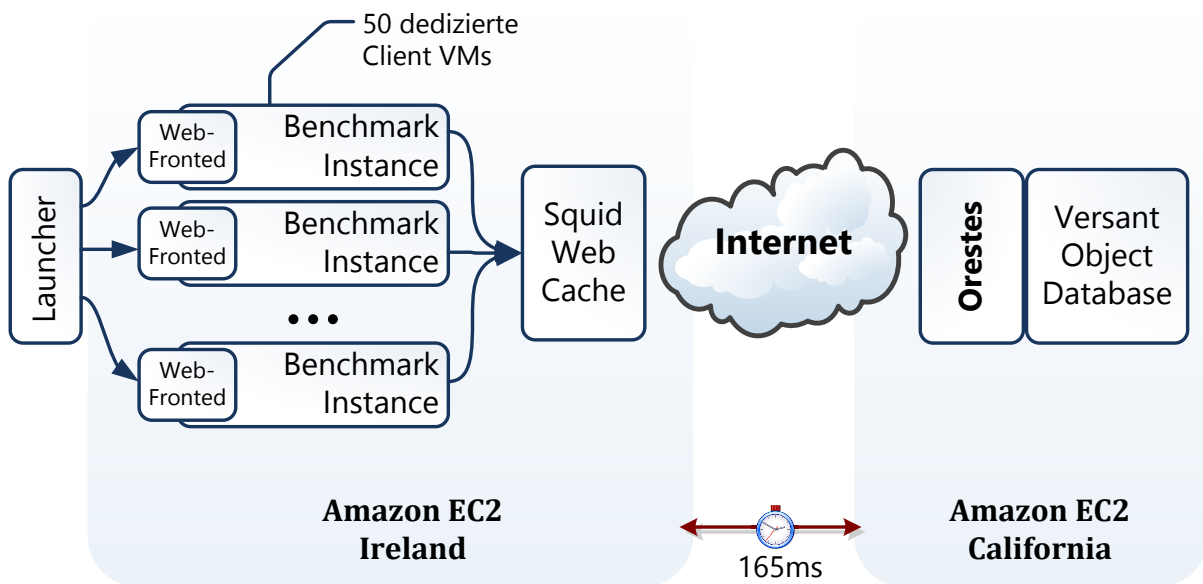


Abbildung 23 Testbed für die Evaluierung von ORESTES unter massiv paralleler Anfragelast.

Als Persistenz-API kommt Java Data Objects (JDO) [27] zum Einsatz. Der Test wird in Java ausgeführt. Bei der Ausführung vergleichen wir den Unterschied zwischen dem TCP-basierten VOD Protokoll und dem ORESTES-Protokoll. Dazu wird der Client instrumentiert und die Ausführungszeiten verschiedener Methodenaufrufe werden gemessen und protokolliert. Jeder der 50 parallel ausgeführten Clients erzeugt diese Messergebnisse und schreibt die Logdatei in einen geteilten Storage Service (Dropbox). Die JDO Implementierung von VOD und ORESTES unterscheidet sich in einem Punkt zu Gunsten von VOD. Auf Clientseite besitzt VOD einen sogenannten Level-2-Cache, der transaktionsübergreifend aber transient Objekte cacht. Die ORESTES JDO Implementierung besitzt bisher keinen solchen clientseitigen Cache.

Abbildung 24 zeigt die Gesamtlaufzeiten für VOD und ORESTES im Vergleich. Ein Test wird dabei für eine bestimmte Anzahl insgesamt in der Datenbank gespeicherter Objekte (300,

3.000, 30.000) dreimal hintereinander ausgeführt. Da der clientseitige Cache von VOD transient ist und jeweils zu genau einem Client gehört, verringert sich die Ausführungszeit für VOD nicht über die drei Durchläufe. ORESTES hingegen profitieren von der Lokalität des Zugriffs und dem sich füllenden Web-Cache, so dass sukzessive Ausführungen schrittweise schneller werden. Dabei ist zu beachten, dass hier Objekte aus einer diskreten Gleichverteilung gezogen werden, die für das Caching den Worst-Case darstellt (kein häufig angefragtes Working-Set). Dennoch ist der Effekt überaus deutlich: das ORESTES Protokoll und Web-Caching beschleunigt die Ausführung um einen Faktor von 2-8.

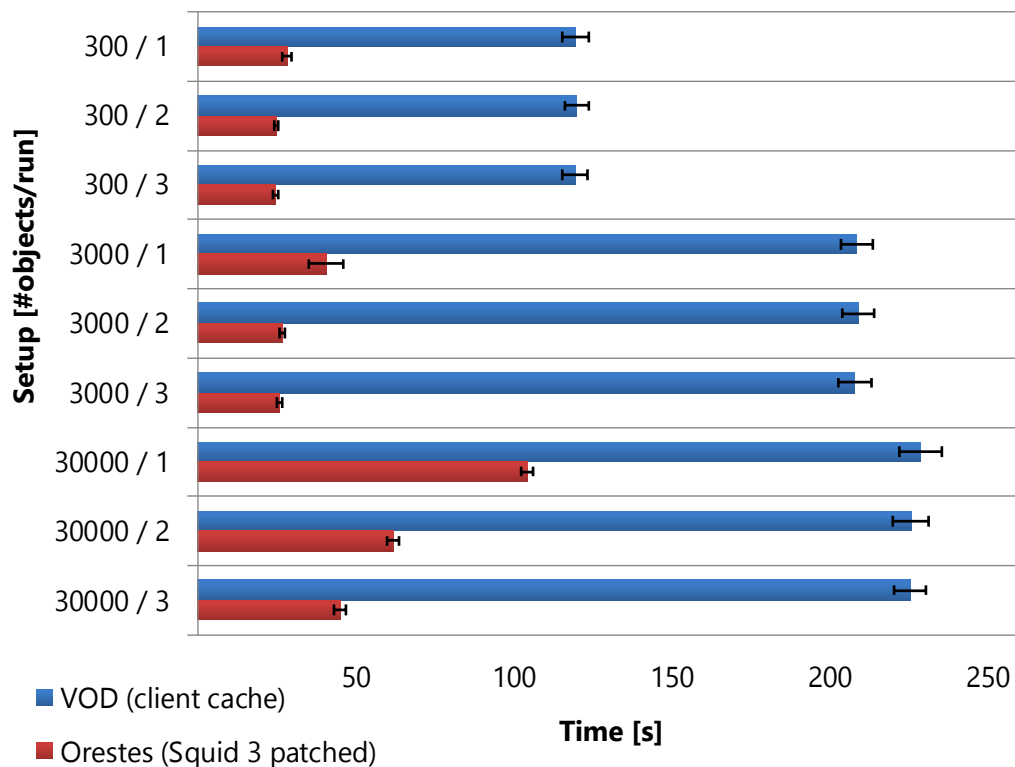


Abbildung 24 Ergebnisse für die Ausführungszeit von 50 parallelen Clients. Die statistische Standardabweichung ist durch die schwarzen Balken gekennzeichnet.

Der gemittelte Anteil verschiedener Operationsarten (Read, Write, Query, Transaktionen, etc.) ist in Abbildung 25 gezeigt. Da der getestete Workload leseintensiv ist (Read/Write Verhältnis 10/1), entfällt der größte Teil der Ausführungszeit auf Reads. Es ist deutlich erkennbar, wie das Web-Caching in ORESTES die Reads beschleunigt. Auf der anderen Seite fällt ein geringer Overhead bei den Writes auf. Im ORESTES-Protokoll werden Schreibvorgänge als einzelne PUT Requests ausgeführt, während VOD diese Operationen wenn möglich als Batchoperation zum Transaktionscommit durchführt. Der Grund für die separate Übertragung der Änderungen ist der *Invalidation by Side Effect* Mechanismus. Jeder Web-Cache, der einen ändernden Request (PUT, POST, DELETE) weiterleitet, invalidiert seine lokale Kopie. Der Mechanismus sorgt damit dafür, dass Einträge in Web-Caches aktuell bleiben, wenn Lese- und Schreibvorgänge über die gleichen Web-Caches weitergeleitet werden. Um diesen



Effekt zu nutzen, werden im ORESTES Protokoll alle Schreibvorgänge separat ausgeführt, wodurch im Vergleich ein gewisser Overhead beim Schreiben entsteht.

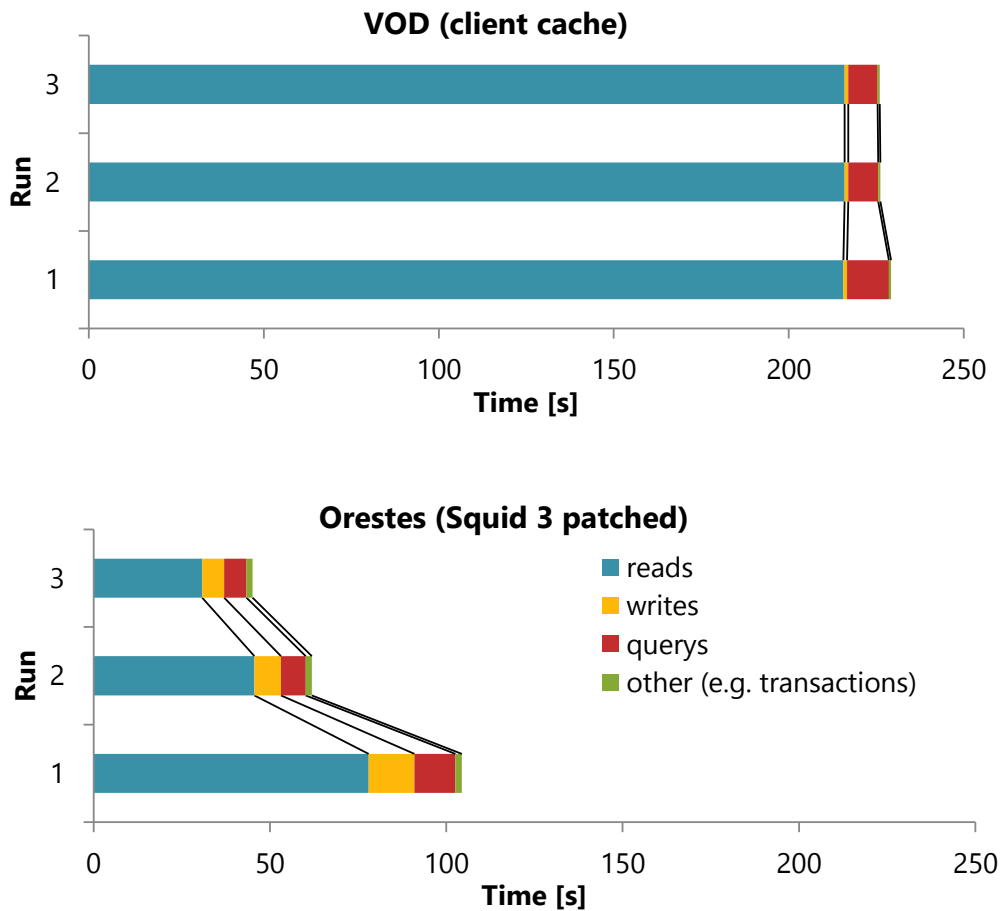


Abbildung 25 Gemittelter Anteil verschiedener Arten von Operationen an der Ausführungszeit (Read/Write Verhältnis 10/1, 30.000 Datenbankobjekte).

1.2.4 Problematik der Cache Kohärenz

Die Benchmarks belegen also, dass für realistische Workloads in Cloud Kontexten durch das ORESTES Protokoll erhebliche Performance Vorteile erzielt werden können. Die Benchmarks vernachlässigen durch den einfachen Aufbau jedoch den möglichen Effekt fehlender Cache-Kohärenz: kommen mehrere Web-Caches zum Einsatz, ist nicht garantiert, dass die Web-Caches wie in diesem Testbed durch Invalidation by Side Effect ihre Einträge invalidieren können. Es treten dann Stale Reads auf. Diese führen je nach Wunsch des Clients entweder zu Transaktionsabbrüchen oder herabgesetzter Konsistenz und haben damit negative Auswirkungen auf die Performance. Das Problem ist in Abbildung 26 verdeutlicht. Wenn zwei oder mehr Clients auf derselben Datenbank Lesen und Schreiben, wird die Cache Kohärenz zu einem Problem:

1. Angenommen ein Client **A** liest ein Objekt der Version **v1** über einen Web-Cache. Dieser Web-Cache speichert das Objekt und betrachtet es für eine statische Zeitspanne als aktuell.

2. Ändert ein Client **B** das Objekt vor Ablauf der Zeitspanne auf Version **v2**, ist die ge-
cachte Kopie nicht mehr aktuell.
3. Fragt Client **A** (oder ein anderer Client) über den Web-Cache das Objekt im Rahmen
einer Transaktion erneut an, erhält er die veraltete Kopie. Diese notiert der Client in
seinem Read-Set.
4. Beim Commit der Transaktion hat der Client zwei Möglichkeiten
 - a. Er sendet das Read-Set und gibt darin an, Version **v1** des Objektes gelesen zu
haben. Der Server stellt bei der Validierung fest, dass die Version veraltet ist
und veranlasst einen Rollback/Abort der Transaktion.
 - b. Der Client nimmt in Kauf, möglicherweise veraltete Objekte gelesen zu haben
und überträgt die gelesenen Versionsnummern nicht. Der Server weist die
Transaktion nur zurück, wenn ein Schreib/Schreib Konflikt mit einer Transak-
tion auftrat. Das Ergebnis der Transaktion ist möglicherweise inkonsistent, da
sie auf veralteten Daten beruht.

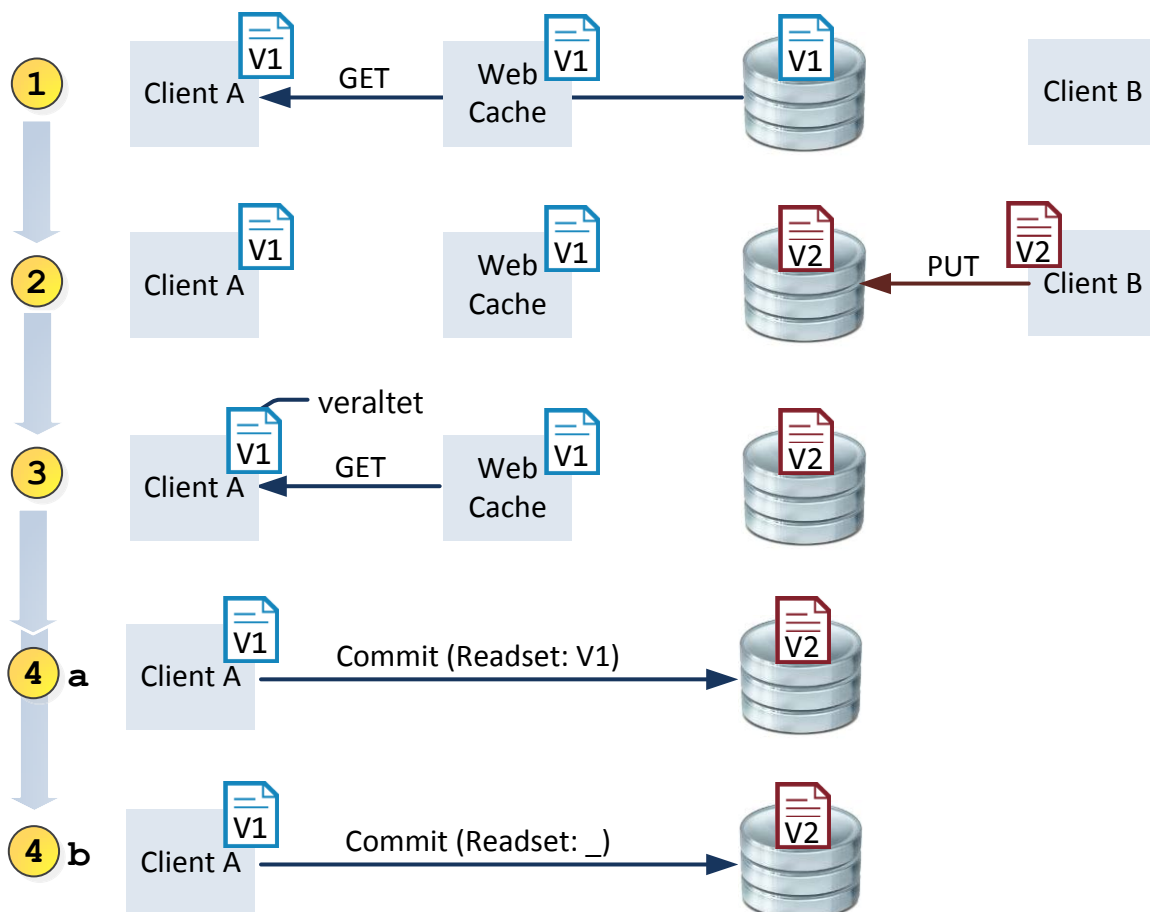


Abbildung 26 Die Problematik der Stale Reads.

Die geschilderte Problematik soll im Folgenden gelöst werden.



2 Related Work

Unsere Arbeit stützt sich auf mehrere Kategorien verwandter Forschung. Das DBaaS Modell verspricht, das Problem der Konfiguration, Skalierung, Bereitstellung, Überwachung, Sicherung und Zugangskontrolle auf die entsprechenden Service Provider zu verlagern [79], [111]. Der kommerzielle Erfolg von DBaaS Angeboten wie SQL Azure, Amazon RDS und SimpleDB zeigt das Potential dieses Ansatzes [10], [112]. Die Idee des DBaaS ist auf Hacigumus et al. [111] zurückzuführen. Einige Aspekte sind bisher jedoch ungelöst. Insbesondere Zugriff mit geringer Latenz, elastische Skalierbarkeit, effektive Mandantenfähigkeit und Datenschutz für Datenbanken bedürfen neuer Ansätze [79], [113]. In dieser Arbeit beschreiben wir, wie die beiden ersten Probleme gelöst werden können: geringe Latenz und elastische Skalierbarkeit. Verwandte Forschung zu elastischer Skalierbarkeit gibt meist entweder Konsistenz oder die Mächtigkeit des Datenmodells zugunsten der Skalierbarkeit auf, z.B. in ElasTraS und den meisten NoSQL Systemen [48], [112]–[114]. Unser Ansatz zielt darauf ab, weder die Ausdrucksstärke noch die Konsistenz einzuschränken, bietet dafür aber keine explizite Lösung für die Skalierbarkeit von Schreibzugriffen. Das Problem skalierbarer Writes bildet einen Grundpfeiler der NoSQL Bewegung. Auslöser für viele der neuen Systeme sind die Ideen aus Amazons *Dynamo* System [114], Googles *BigTable*, *Filesystem*, *Megastore*, *Spanner* und *Dremel* [17], [19], [47]–[49] und Yahoos *PNuts* [115]. Erste praktische Strategien zur Lösung der Mandantenfähigkeit und des Datenschutzes wurden u.a. in den Systemen wie *Relational Cloud* [79] und *M-Store* [116] vorgestellt. Einen wichtigen Meilenstein in Richtung verschlüsselter Datenbanken legten jüngst Popa et al. mit *CryptDB* und ihrer *Onions of Encryption* Technik [117]. Die Einhaltung von SLOs unter gemischten Workloads ist für DBaaS Provider ein wichtiger Faktor, um die drohenden Vertragsstrafen einer SLA-Verletzung zu vermeiden. Für das Problem existiert bisher keine zufriedenstellende Lösung, Ansätze wie *ActiveSLA* machen jedoch erste Schritte [118], [119]. Kemper et al. [77], [80] haben einige interessante Ansätze entwickelt, um SaaS Anwendungen mandantenfähig durch relationale Datenbanken zu unterstützen.

Der Kern von ORESTES ist sein REST/HTTP Protokoll. Der REST (Representational State Transfer) Architekturstil wurde von Fielding eingeführt [34] und gilt zunehmend als bester Weg, um Services mit einer klaren Struktur und breiten Zugänglichkeiten für unterschiedlichste Systeme und Programmiersprachen umzusetzen. Neben der weiten Verbreitung für unterschiedliche Cloud Services beginnt REST sich, wie in der Einführung belegt, für NoSQL Datenbanken durchzusetzen wie *CouchDB*, *RavenDB*, *Riak*, *Azure Table*, *Neo4J*, *HBase*, *SimpleDB*, *OrientDB*, *ArangoDB* andeuten [2], [3], [10]. Das Problem dieser Implementierungen liegt in ihrer fehlenden Kombinierbarkeit mit Web-Caching und Load-Balancing, sowie den fehlenden Konzepten für Schema- und Transaktionsmanagement. Ein bemerkenswerter Ansatz ist Microsoft's *OData* Protokoll [120]. Es besitzt zwar ebenfalls die zuvor genannten Einschränkungen, ist aber im Prozess ein OASIS Standard zu werden und zeigt eine saubere Umsetzung der REST Prinzipien. Es ist jedoch in seiner Ausrichtung primär auf lesenden Zugriff für heterogene Datenquellen fokussiert. Google Data (*GData*) ist ein vergleichbarer

Ansatz von Google [121]. Für ein vollwertiges Datenbanksystem sind beide REST APIs jedoch unzureichend. Für relationale Datenbanksysteme wurde bisher kein Versuch unternommen, eine vollständige REST API zu implementieren. Erste theoretische Überlegungen sind bei Haselman et al. [122] zu finden. Sakr et al. [10] geben einen guten Überblick über die erstrebenswerten Eigenschaften von Cloud Data Management Systemen: Verfügbarkeit, Skalierbarkeit, Elastizität, Performance, Mandantenfähigkeit, Last- und Mandanten-Balancierung, Fehlertoleranz, Fähigkeit, in heterogenen Systemen ausgeführt zu werden und flexible Query Schnittstellen. Mit ORESTES wollen wir die meisten dieser Probleme lösen.

Web-Caching ist das Skalierungsmodell des Webs und im HTTP Protokoll implementiert [31], [35]. Es gibt verschiedene Strategien, Web-Caches in Netzwerken anzuordnen [123]. Panthan und Buyya [22] geben einen umfassenden Überblick über Web-Caching in CDNs. Neuere CDNs sind z.T. direkt mit Cloud Plattformen integriert, wie Amazons Cloudfront und Microsofts Azure CDN [67], [68]. Wie in der Einführung erläutert, können Web-Caches in verschiedenen Topologien angeordnet werden. Skalierbare Hierarchien können mit den Protokollen ICP, HTCP und Cache Digests [124] umgesetzt werden. Shared-nothing Web-Cache Cluster können durch das CARP Protokoll implementiert werden [31]. Eine weitere Möglichkeit zur Skalierung von Web-Caches ist Load-Balancing, über das Gilly et al. eine gute Übersicht geben [125]. Große Bemühungen fließen noch immer in die Optimierung von Cacheverdrängungsstrategien und Prefetching Algorithmen von Web-Caches [126]. Es gibt diverse Implementierungen von Load-Balancing und Web-Caching sowohl in Hard- als auch Software [24]. Wir haben in unseren Benchmarks sowohl den Squid Proxy Cache [127], Varnish als auch den Microsoft TMG [40] deployt und mit dem Aufdecken zweier schwerwiegender Protokollfehler einen Beitrag zu ihrer Weiterentwicklung geleistet [128], [129].

Verwandte Forschung zum Thema Datenbank Caching kann in zwei Kategorien eingeteilt werden: *Buffer Management* und *Netzwerk Caching* [5], [130]. Ziel des Buffer Managements ist es, die Zugriffslücke zwischen RAM und Festplatte zu überbrücken. In-Memory Datenbanksysteme (z.B. Redis, HanaDB und VoltDB [50], [131]–[133]) lösen dieses Problem durch vollständigen Verzicht auf Externspeicher und Buffermanagement. Andere Datenbanksysteme implementieren z.T. sehr komplexe Buffer Management Strategien [55]. Dieses Problem ist jedoch orthogonal zu unseren Betrachtungen, da es sich ausschließlich auf die interne Implementierung eines einzelnen Datenbankknotens bezieht. Ansätze zu netzwerkbasiertem Caching nutzen zumeist vollständige Datenbanksysteme als Caches. Ein solches System ist DBProxy [56], das Query Ergebnisse in einer Datenbank cacht und für die Beantwortung weiterer Quers nutzt, falls die gecachten (materialisierten) Views prädikatsvollständig sind. DBCache [57] ist ein weiterer netzwerkbasierter Ansatz, der auf sogenannten Cache Groups basiert, die Regeln angeben, gemäß welcher Assoziation bei einem Cache-Miss Tupel nachzuladen sind. DBProxy und DBCache haben beide den Nachteil, dass sie ein vollständiges relationales Datenbanksystem als Cache benötigen und synchrone Replikation zur Sicherstellung der Konsistenz voraussetzen [38].



Verschiedene nichtsperrende Verfahren wurden in der Forschung zur Nebenläufigkeitskontrolle vorgeschlagen und in verschiedensten Datenbanksystemen implementiert [55]. ORESTES basiert auf solchen Techniken, da Objekte in Web-Caches nicht gelockt werden können, ohne ihre Cachebarkeit aufzugeben. Nennenswerte Ansätze nichtsperrender Nebenläufigkeitskontrolle sind *Optimistic Locking*, *Serialization Graph Testing*, *Timestamp Ordering* und *Forward/Backward Oriented Optimistic Concurrency Control* [8], [134]. Ihre Kombinierbarkeit mit dem Web-Caching Ansatz von ORESTES wird im nächsten Kapitel diskutiert. Für Querys und Writes kann der Datenbankserver weiterhin sperrende Verfahren nutzen, da diese Operationen von Web-Caches stets weitergeleitet werden. Auch Multi-Version Concurrency Control / Snapshot Isolation [135] sind mit ORESTES kombinierbar, vorausgesetzt, sie werden mit einem der zuvor genannten Verfahren kombiniert, um Stale Reads zu detektieren.

Unser Ansatz zur Transaktionskontrolle mit starker Cache-Kohärenz basiert auf Bloomfiltern. Bloomfilter wurden 1970 von Burton Bloom eingeführt [136]. Zahlreiche Varianten von Bloomfiltern wurden seither erforscht, u.a. Counting Bloomfilters [137], [138], *Scalable Bloomfilters* [139], *Bloomier Filters* [140], *Attenuated Bloomfilters* [141], *Bitwise Bloomfilters* [142], *Spectral Bloomfilters* [143] und *Compressed Bloomfilters* [144]. Wir werden diese Varianten auf ihre Nutzbarkeit zur bloomfilterbasierten Cache-Kohärenz untersuchen. Hashfunktionen, von deren Güte die Leistungsfähigkeit einer Bloomfilter-Implementierung und seine theoretischen Garantien abhängen, wurden in verschiedensten Arbeiten vorgeschlagen. Konheim [145] bietet eine gute, wenn auch unvollständige Monographie.

Nach unserem Wissen wurde die Kombination aus Web-Caching und Datenbanksystemen bisher nicht erforscht und stellt damit einen neuen Ansatz da.

3 Kohärentes Web-Caching

In diesem Kapitel beschreiben wir eine neue Technik, die verhindern kann, dass veraltete Objekte aus Caches gelesen werden. Wir wollen zuerst eine generelle Taxonomie für Caches einführen. Wir unterscheiden zwischen *Out-of-Path* und *In-Path* Caches sowie den Kohärenzmechanismen *Expiration* und *Invalidation* (Abbildung 27):

Out-of-Path Cache: ein solcher Cache liegt nicht in einem Kommunikationspfad, über den alle Einträge gelesen und geschrieben werden (z.B. aufgrund von Load Balancing). Web-Caches sind typischerweise Out-of-Path Caches.

In-Path Cache: ein In-Path Cache liegt an einer Stelle des Kommunikationspfades, die von allen Lese- und Schreibvorgänge passiert wird. Ein typischer In-Path Cache ist ein Festplattencache, der Lese- und Schreibvorgänge zwischenpuffert.

Expiration: die Cache-Kohärenz durch Expiration geht von der Annahme aus, dass Objekte stets für eine statisch bestimmte Lebensdauer gültig sind, die sich a-priori festlegen lässt. Dieses Modell ist sehr einfach, da es auf komplizierte Protokolle zur Cache-Kohärenz verzichtet. Neben dem Web-Caching ist es für das In-Memory Caching hochskalierbarer verteilter Anwendung gebräuchlich und beispielsweise in Memcache und Redis implementiert [131].

Invalidation: ändert sich ein gecachter Eintrag, kann dieser in den entsprechenden Caches invalidiert werden. Dies kann zum einen über spezialisierte Cache-Kohärenz Protokolle geschehen, die den Kommunikationskanal überwachen (z.B. MESI für speichergekoppelte Multiprozessorsysteme [146]), oder der datenhaltende Server und der Client invalidieren Caches, sobald sie eine Änderung vornehmen oder registrieren (im Web-Caching).

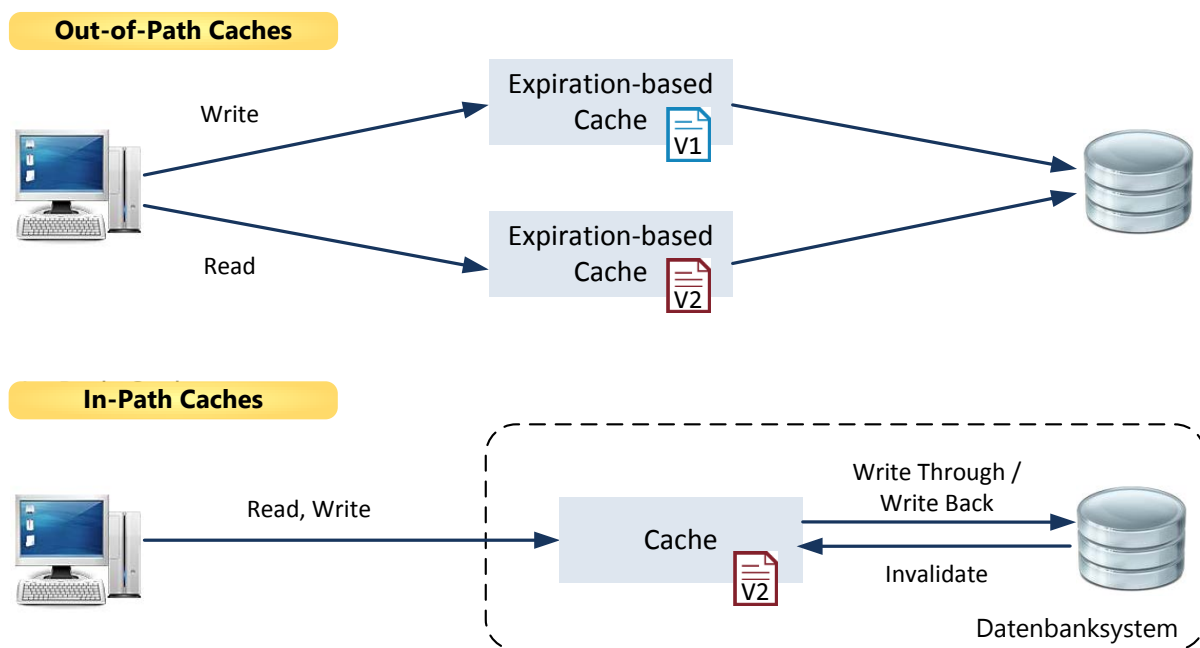


Abbildung 27 Taxonomie für Caches.



Kapitel 4 befasst sich mit der Sicherstellung der Cache-Kohärenz für Caches, die Invalidierungen unterstützen. Im Kontext von Web-Caching sind dies Reverse-Proxy Caches und Caches von CDNs. Das Modell lässt andere Caches ebenfalls zu, sofern diese ebenfalls Invalidierungen unterstützen. So könnten In-Memory Caches wie Memcache mit dem gleichen Mechanismus auf der ORESTES Serverseite explizit eingesetzt werden, um das Datenbanksystem zu entlasten. Dieses Kapitel befasst sich mit expirationsbasierten Out-of-Path Caches. An diese Caches werden die geringsten Anforderungen gestellt, was eine allgemeingültige Lösung diffizil gestaltet. In-Path Caches (z.B. der Seitenbuffer eines Single-Sever Datenbanksystems) stellen für die Cache-Kohärenz eine wesentlich geringere Herausforderung dar, da diese direkte Kenntnis von Änderungsoperationen besitzen.

Stale Reads aus expirationsbasierten Caches können auf drei verschiedene Weisen verhindert werden:

1. Änderungen an Objekten, die noch in Caches gespeichert sein können, werden verboten.
2. Cache-Einträge werden invalidiert, wenn ein Objekt geschrieben wird.
3. Der Client verlangt bei seiner Anfrage eines potentiell veralteten Objekts eine Revalidierung oder umgeht den Cache.

Option 1 ist inakzeptabel für einen Datenbanksystem, da es jederzeit Änderungen umsetzen können muss. Option 2 ist keine allgemeingültige Lösung und wird in Kapitel 4 behandelt. Option 3 ist die einzig verbleibende Möglichkeit, um Stale-Reads für jede Topologie und Klasse von Web-Caches zu verhindern. Wir bezeichnen mit $WTS(x)$ (*Write Timestamp*) den Schreibzeitstempel eines Objektes x . Der Zeitstempel $Now()$ bezeichne den Zeitstempel der Gegenwart. Ein Objekt ist dann potentiell veraltet, wenn seine letzte Änderung vor Ablauf der Expirationsdauer E geschehen ist:

$$stale(obj) := \begin{cases} \mathbf{true}, & \text{wenn } Now() < WTS(obj) + E \\ \mathbf{false}, & \text{sonst} \end{cases}$$

Damit das Caching seine Funktion erfüllen kann, dürfen Clients nur eine Revalidierung fordern, wenn Objekte potentiell veraltet sind. Andernfalls würde das Caching umgangen und die positiven Effekte auf Latenz und Skalierbarkeit aufgehoben. Da lediglich der ORESTES-Server alle Änderungen registriert, muss die Information über Änderungen vor Ablauf der Expiration vom Server zum Client übertragen werden. Das heißt also, dass auf Serverseite die Objekt-IDs aller Änderungen eines Zeitfensters erfasst werden müssen, dessen Länge der Expirationsdauer von Objekten entspricht. Wir gehen vorerst davon aus, dass diese Expirationsdauer für alle Objekte identisch ist. Die geschilderten Verfahren könnten jedoch auf objektspezifische Expirationsdauern erweitert werden. Eine Randbedingung für die Sammlung von Änderungen auf dem Server erwächst aus dem Umstand, dass im ORESTES Protokoll nicht gefordert wird, dass alle Änderungen durch einen ORESTES Server behandelt werden. Vielmehr kann es eine Reihe von ORESTES-Servern geben, die Zugriff auf dieselbe, möglich-

erweise verteilte Datenbank haben. Dennoch müssen alle geänderten Objekt-IDs an den Client übertragen werden.

Die Anforderungen für die serverseitige Erfassung von geänderten Objekten sind also:

1. **Effizienz.** Die Änderungen müssen effizient in Bezug auf Zeit- und Speicherbedarf erfasst werden können.
2. **Shared Datastructure.** Eine verteilte Feststellung und Speicherung der Änderungen muss möglich sein.
3. **Schnelle Generierung.** Die Datenstruktur zu Erfassung der Änderungen muss sehr schnell generiert werden können, da sie möglicherweise sehr häufig angefragt wird.
4. **Garbage Collection.** Objekte, deren Expirationsdauer abgelaufen ist, müssen aus der Datenstruktur entfernt werden. Das Objekt wird erst wieder aufgenommen, wenn es geschrieben wurde.

Da der Client vor jeder Anfrage prüfen können muss, ob das angefragte Objekt mit einer Revalidierungsaufforderung geladen werden muss, ist Effizienz von oberste Priorität für die Datenstruktur, an der diese Prüfung durchgeführt wird. Auch muss die Änderungsdatenstruktur bei der Übertragung eine sehr geringe Größe haben, um den Overhead zu reduzieren. Die zentrale Garantie, die dem Client durch das Abrufen der Änderungsdatenstruktur gewährt wird, ist:

Clients sehen keine Objektversionen, die vor Abrufen der Änderungsdatenstruktur ungültig geworden sind.

Das heißt, dass der Abruf der Änderungsstruktur dem Client eine Sicht auf die Datenbank gibt, die mindestens so aktuell wie der Zeitpunkt des Abrufs ist. Die Anforderungen an die Änderungsdatenstruktur für den Client sind:

1. **Effizienz.** Der Client muss sehr effizient - in $O(1)$ - prüfen können, ob ein Objekt mit einer Revalidierung angefragt werden muss. Darüber hinaus muss die Änderungsdatenstruktur sehr speichereffizient sein, um auch für ressourcenarme Clients (z.B. mobile Geräte) nutzbar zu sein.
2. **Kompakte Darstellung.** Da der Client die Änderungsdatenstruktur vom Server abrufen muss, muss sie bei der Übertragung eine sehr geringe Größe besitzen.

Das Problem ist in Abbildung 28 zusammengefasst. Eingehende Schreiboperationen können von einem beliebigen ORESTES Server entgegengenommen werden. Dieser registriert die Änderung an der geteilten Änderungsdatenstruktur. Durch die festgelegte Expirationsdauer entsteht ein *Sliding Window* über den Änderungsoperationen. Dieses Sliding Window repräsentiert potentiell veraltete Objekte. Ein Client kann das Sliding Window über die REST/HTTP Schnittstelle abrufen.



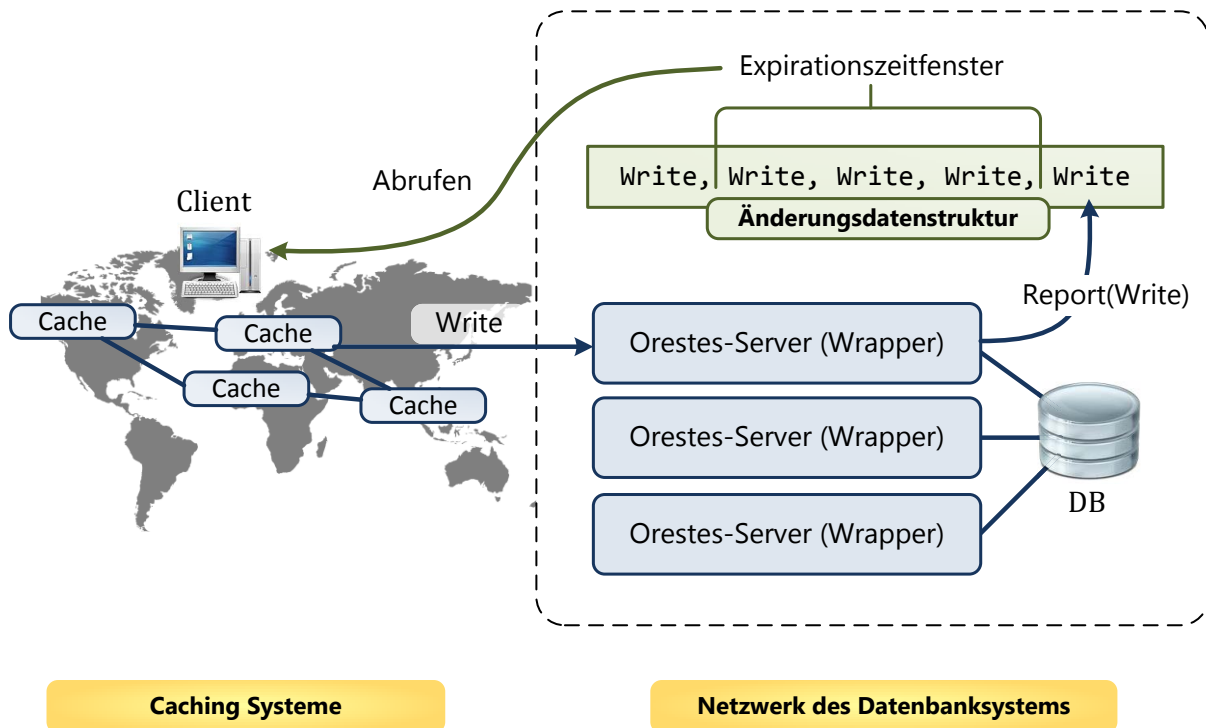


Abbildung 28 Änderungsdatenstruktur. Schreibvorgänge können von beliebigen ORESTES-Servern bedient werden, die Änderungen an die Änderungsdatenstruktur propagieren.

Dieses Kapitel widmet sich der Umsetzung einer Datenstruktur, die sowohl die Anforderungen des Servers als auch des Clients erfüllen kann. Dazu führen wir zuerst eine Analyse der Transaktionsverfahren durch, die mit Web-Caching kompatibel sind. Anschließend leiten wir das Konzept der bloomfilterbasierten Cache-Kohärenz her. Dazu gehen wir zuerst auf die mathematischen Eigenschaften ein und erklären verschiedene algorithmische Varianten und Anwendungen. Nach der Untersuchung bestehender Bloomfilter-Implementierungen beschreiben wir unsere Implementierung eines Bloomfilters und Counting Bloomfilters und ihre Vorteile. Anschließend suchen wir statistisch geeignete Hashfunktionen, die eine verlässliche Grundlage für die mathematischen Eigenschaften der Bloomfilter bilden. Das Gesamtergebnis vergleichen wir mit einer anderen Open-Source Implementierung und erklären, warum bisherige Implementierungen für unsere Zwecke unzureichend sind. Aufbauend auf der Bloomfilter Implementierung führen wir die Änderungsdatenstruktur für Schreibvorgänge und ihre Integration in ORESTES ein. Das bloomfilterbasierte Transaktionsmodell unterziehen wir einem Vergleich mit der zuvor verwendeten bloomfilterlosen Technik und stellen fest, dass viele Transaktionsabbrüche nun verhindert werden können. Die verbesserte Konsistenz der Web-Caches ordnen wir bezüglich des CAP Theorems ein.

3.1 ACID Transaktionen ohne Cache Kohärenz

Der Ablauf einer ORESTES Transaktion auf Protokollebene ist in Abbildung 29 gezeigt. Transaktionen werden durch Clients aktiv gestartet und committet. Wenn der Client die Validierung von gelesenen Objekten bezüglich Stale Reads wünscht, gibt es zwei Möglichkeiten: 1. die Objekte werden stets direkt vom Server angefragt (kein Caching), 2. der Client

protokolliert gelesene Objekte in einem Read-Set. Die 2. Variante erlaubt die Nutzung von Web-Caching und ist deshalb sinnvoller. Beim Commit überträgt der Client sein Read-Set, damit es durch den Sever validiert werden kann.

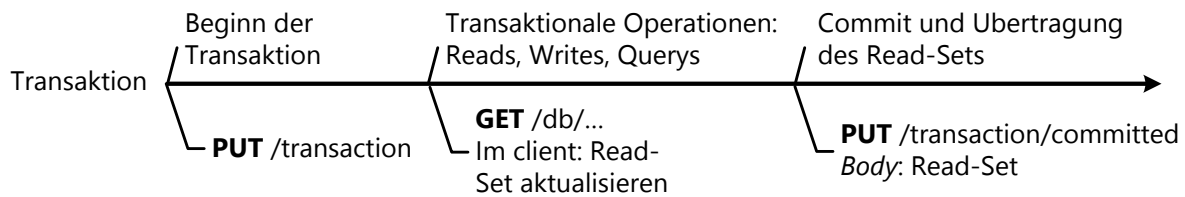


Abbildung 29 Ablauf einer ORESTES Transaktion auf Protokollebene.

Um die Transaktionsverwaltung in ORESTES umzusetzen, gibt es zwei Realisierungsmöglichkeiten. Die Komponente, in der die transaktionale Logik umgesetzt ist, kann entweder Teil der Datenbank sein (wenn diese es nativ unterstützt) oder ein Teil des Datenbankwrappers. Wenn die Datenbank ohnehin nichtsperrende Nebenläufigkeitskontrolle unterstützt, bietet sich die erstere Variante an. In dem Fall, dass die Datenbank keine Erweiterung des Read-Sets um die vom Client übermittelten Versionen erlaubt, kann diese Validierung auch im Wrapper stattfinden. Abbildung 30 zeigt den generischen Commit Algorithmus des Datenbankwrappers als Pseudocode. Unterstützt das Datenbanksystem die Validierung, wird die Aufgabe, das Read-Set auf Stale Reads zu überprüfen, an das Datenbanksystem übergeben. Andernfalls wird atomar überprüft, ob das Read-Set aus aktuellen Versionen besteht, d.h. ob gilt:

$$\forall(obj.id, obj.version) \in transaction.readSet : db.version(obj.id) = obj.version$$

Anschließend wird durch einen Algorithmus zur optimistischen Nebenläufigkeitskontrolle überprüft, ob Nebenläufigkeitskonflikte aufgetreten sind. Diese Validierung kann eingespart werden, wenn das Read-Set vor der Überprüfung auf Stale Reads mit dem Write-Set vereint wird. Die konkrete Implementierung des Commit Algorithmus hängt jedoch von dem Transaktionsmodell und dem Datenmodell des gekapselten Datenbanksystems ab. Beispielsweise kann in einem System, das sperrende Transaktionen unterstützt, die explizite Synchronisierung im Wrapper entfallen und stattdessen eine performantere implizite Synchronisierung über eine pessimistische Transaktion des Datenbanksystems genutzt werden. Auch kann es Unterschiede darin geben, ob das Datenbanksystem die Objekte des Write-Sets direkt bei ihrer Änderung schreibt (und zur Verhinderung von Dirty Reads sperrt) und die Änderung bei einem Rollback zurücknimmt oder die Objekte einer Transaktion immer privat geändert werden (z.B. durch Multiversion Concurrency Control) und erst bei einem erfolgreichen Commit freigegeben werden. Im ersteren Fall ist eine Prüfung des Write-Sets unnötig, da Write/Write Konflikte durch Sperren verhindert werden (auf Kosten möglicher Deadlocks). Beim endgültigen Schreiben der Änderungen einer Transaktion müssen die Versionen der geschriebenen Objekte inkrementiert werden. Auch dies kann entweder direkt durch das Datenbanksystem geschehen (z.B. in VOD, db4o, MongoDB, CouchDB, Riak, Cassandra, etc.) oder explizit durch den Wrapper.



```

def commit(read_set):
    if(optimistic_concurrency_supported):
        #Das DBMS führt die Validierung durch
        db.put_read_set(this_transaction, read_set)
        db.validate(this_transaction)
    else:
        synchronized {
            #Validieren, dass gelesene Objekte noch aktuell sind
            stale = check_if_stale(read_set)
            #Nebenläufigkeitskontrolle (z.B. BOCC)
            conflict = check_for_conflicts(read_set, write_set)

            if(stale or conflict):
                rollback(this_transaction)
            else:
                make_persistent(write_set)
        }

```

Abbildung 30 Grundlegender Commit Algorithmus des ORESTES Datenbankwrappers .

In der Literatur wurde für verschiedene Annahmen untersucht, ob sperrende oder nicht-sperrende Verfahren überlegen sind [147]. Dabei herrscht Einigkeit darüber, dass nicht-sperrenden Verfahren für Workloads mit geringer Konfliktwahrscheinlichkeit überlegen sind, da der Overhead für die Sperrverwaltung entfällt. Pessimistische Verfahren wiederum sind vorteilhaft, wenn die Wahrscheinlichkeit von Konflikten hoch ist, d.h. Clients tendenziell die gleichen Daten lesen und ändern. Stonebraker et al. haben 2006 bei der Untersuchung von relationalen Datenbanksystem festgestellt, dass diese einen substantiellen Teil ihrer Leistung auf die Verwaltung von Sperrern verwenden [148]. Zu dieser Feststellung gelangten sie durch die Untersuchung von Shore, einem repräsentativen relationalen Datenbanksystem, unter der Last eines Standardbenchmarks für relationale Datenbanken (TPC-C). Ihr Ergebnis ist, dass der Anteil nützlicher Instruktionen, d.h. solcher die transaktionale Operationen des Clients ausführen, sehr gering ist (Abbildung 31). Vier Kategorien sind maßgeblich für den Overhead [148]:

- **Logging.** Das Erstellen eines Datenbank Logs nach dem Write-Ahead-Log Prinzip ist aufwendig, erlaubt aber Recovery.
- **Locking.** Sperrende Verfahren erfordern die Verwaltung verschiedenster Metadaten (z.B. einer Lock-Tabelle), die großen Einfluss auf die Performance des Systems haben.
- **Latching.** Geteilte Datenstruktur und Multithreading müssen innerhalb des Datenbanksystems durch Synchronisation gelöst werden.
- **Buffer Management.** Die Pufferverwaltung schmälert auf Kosten eines großen Overheads die Zugriffslücke zwischen RAM und Externspeichern.

Stonebraker et al. leiten daraus die Notwendigkeit einer verteilten In-Memory Datenbank (kein Buffer Management) mit ausschließlichen Stored Procedures (kein Locking) in einem Single-Threaded-Design (kein Latching) und dem Verzicht auf Logging her [50], [149]. Wir denken jedoch, dass die Probleme anders zu lösen sind. In ORESTES wird das Buffer Management i.d.R. nicht zum Overhead, da die lesenden Zugriffe bereits durch Web-Caches

beantwortet werden, d.h. in der vielstufigen Cache Hierarchie eine Stufe über dem Buffer Pool. Der Locking Overhead kann in ORESTES durch nichtsperrende Verfahren komplett entfallen. Da ORESTES keinerlei Restriktionen bezüglich der Architektur des Datenbanksystems auferlegt, kann dieses auch ein Single-Threaded In-Memory Design verfolgen, wie wir praktisch am Beispiel von Redis belegen werden. Latching und Logging sind somit orthogonal zu ORESTES und können durch die Architektur des Datenbanksystems gelöst werden. Je nach Anforderung (Polyglot Persistence) kann sich der Overhead einer bestimmten Kategorie jedoch auszahlen (z.B. Logging). Ein geeignetes Datenbankbackend von ORESTES sollte deshalb auch unter diesen Gesichtspunkten ausgewählt werden.

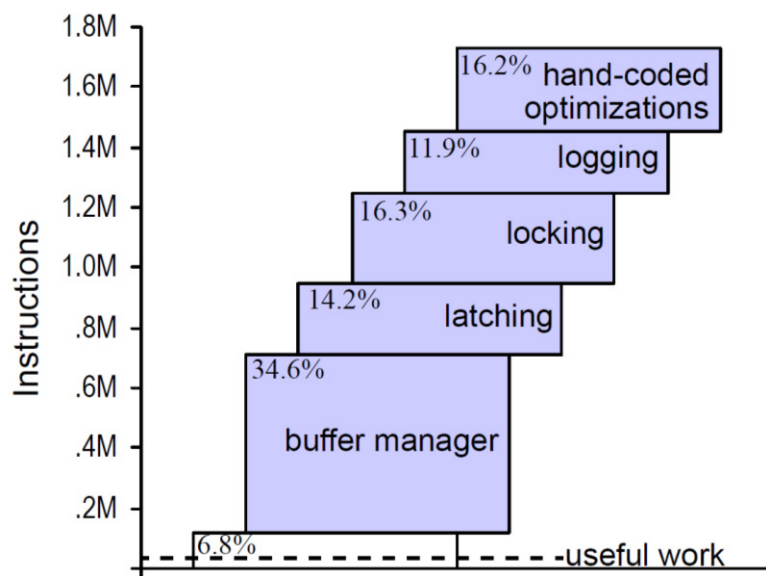


Abbildung 31 Ausgeführte Instruktionen eines relationalen Datenbanksystem unter der Last eines Standardbenchmarks (TPC-C). Die gestrichelte Linie gibt den Anteil nützlicher Operationen an, ermittelt durch die Ausführung auf einem modifizierten Overhead-freien Kernel mit nur einer Transaktion. Ergebnisse und Grafik aus [148].

Untersuchungen, die auf Basis stochastischer und statistischer Annahmen Ergebnisse über die Performance von sperrenden und nichtsperrenden Algorithmen herleiten, gehen stets von unrealistisch vereinfachten Annahmen aus. Zwei populäre Ergebnisse dieser Art sind:

Gray et al. [150] zeigten, dass die Wahrscheinlichkeit von Deadlocks (d.h. Transaktionsabbrüchen) P bei sperrenden Verfahren mit der vierten Potenz der Transaktionsdauer D und mit der zweiten Potenz der Anzahl paralleler Transaktionen T zunimmt. Unrealistische vereinfachende Annahmen sind in diesem Modell beispielsweise die gleiche Länge aller Transaktionen und der gleichverteilte Zugriff über alle Datenbankobjekte (cf. [5]).

$$P(D) \sim D^4 \text{ und } P(T) \sim T^2$$

Franaszek et al. [151] zeigten für optimistische Transaktionen den „quadratischen Effekt“, der besagt, dass die Wahrscheinlichkeit von Transaktionsabbrüchen mit der zweiten Potenz ihrer Größe, d.h. der Anzahl der von ihnen gelesenen oder geschriebenen Objekte, zunimmt.



Auch dieses Modell geht von einer Gleichverteilung aller Objktanfragen aus. Die wesentlich unrealistischere Annahme ist jedoch ein sogenanntes *Preclaiming*, d.h. dass die zugegriffenen Objekte bereits zu Beginn der Transaktion ausgewählt werden. Wenn in einer Datenbank der Größe D eine committende Transaktion m Objekte ändert und eine potentiell konfliktierende Transaktion n Objekte zugreift, ist die Wahrscheinlichkeit eines Konflikts die Gegenwahrscheinlichkeit der Laplace-Wahrscheinlichkeit für nicht-konfliktierende Möglichkeiten [147]:

$$P(n, m) = 1 - \frac{\binom{D-n}{m}}{\binom{D}{m}} \cong 1 - \left(1 - \frac{n}{D}\right)^m \cong \frac{nm}{D}$$

Unter der naiven Annahme, dass alle gelesenen Objekte auch geändert werden und jede Transaktion auf k Objekte zugreift, ergibt sich der quadratische Effekt:

$$P(k, k) \cong \frac{k^2}{D}$$

In diesem sehr einfachen Modell für optimistische Verfahren nimmt die Konfliktwahrscheinlichkeit mit der zweiten Potenz der Transaktionsgröße zu und ist invers proportional zu der Größe der Datenbank. D.h. dass in großen Datenbanken die Konfliktwahrscheinlichkeit geringer ist.

Die Beispiele für analytische Herleitungen von Abbruchraten zeigen, dass die Güte eines Verfahrens nur sehr schwer durch die generischen Annahmen eines statistischen Modells erfasst werden kann. Vielmehr kann nur ein konkreter Use-Case bestimmen, ob ein sperrendes oder ein nicht-sperrendes Verfahren eine bessere Wahl ist. Der Einsatz von sperrenden Verfahren in ORESTES ist durch das Web-Caching limitiert. Es sind jedoch gemischte Verfahren möglich, bei denen der Scheduler im Wrapper eine optimistische Validierung auf Basis eines sperrenden Datenbanksystems durchführt. Dieses Verfahren kommt in gängigen objekt-relationalen Mappern wie Hibernate, EclipseLink und OpenJPA ebenfalls zum Einsatz, da diese optimistische Transaktionen auf Basis von relationalen Datenbanken implementieren [152]. Nahezu alle gängigen relationalen Datenbanksysteme wurden jedoch – inspiriert durch System R – für sperrende Transaktionen entwickelt [149]. Wir denken, dass eine allgemeingültige Aussage darüber, ob sperrende oder nicht-sperrende Verfahren überlegen sind, weder möglich noch sinnvoll ist. Eine Option könnte jedoch sein, die Auswahl explizit dem Client zu überlassen und ein Verfahren als Default festzulegen. Dies scheint der Ansatz zu sein, der für praktische Anwendungen am sinnvollsten ist, wie z.B. die Umsetzung in der industriedominierenden Java Persistence API Spezifikation andeutet [28].

```
//Auswahl des Concurrency-Control Verfahrens beim Laden
Employee e = em.find(Employee.class, 1, LockModeType.OPTIMISTIC);

//Explizites Locking
em.lock(e, LockModeType.PESSIMISTIC_WRITE);

//Mögliche Transaktionsabbrüche:
```

```

//Transaktionscommit einer optimistischen Transaktion
try {
    em.commit();
} catch (OptimisticLockException ole) {
    //Ein Konflikt ist aufgetreten, da
    //eines der Objekte nebenläufig geändert wurde
}

//Deadlock bei pessimistischen Sperren
try {
    Employee e = em.find(Employee.class, 1, LockModeType.PESSIMISTIC_WRITE);
}
catch (LockTimeoutException lte) {
    //Ein Sperrkonflikt ist aufgetreten, d.h. eine andere Transaktion
    //besitzt eine Lese-/Schreibsperre auf dem zu sperrenden Objekt
}

```

Abbildung 32 Die explizite Offenlegung der Nebenläufigkeitskontrolle in JPA.

Eine Integration von expliziten Sperren in das ORESTES REST/HTTP Protokoll ist möglich, aber stets auf Kosten der Performance, da das Web-Caching umgangen wird. Deshalb beschränkt sich unsere Ansatz bisher auf ausschließlich nichtsperrende Transaktionen.

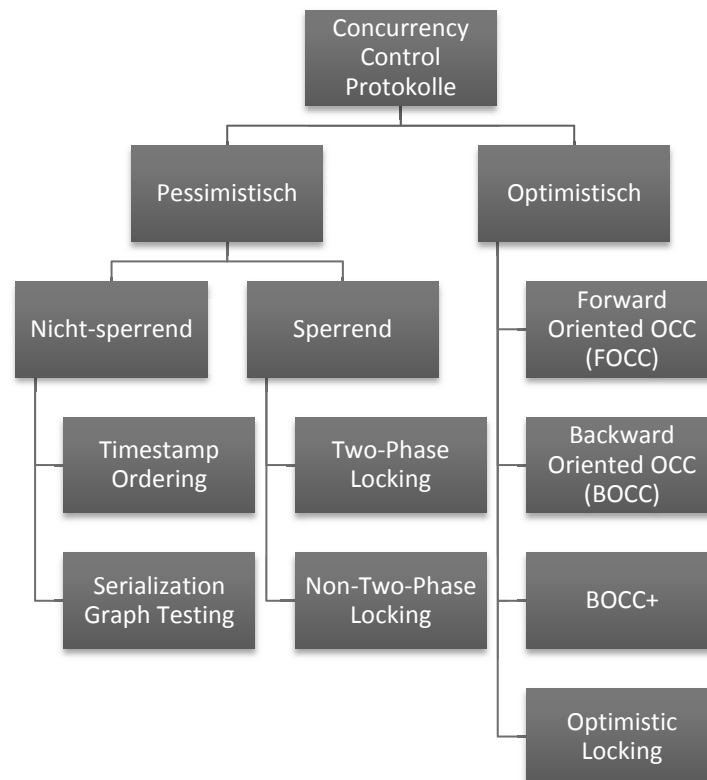


Abbildung 33 Algorithmen zur Nebenläufigkeitskontrolle in Datenbanksystemen.

In der wissenschaftlichen Literatur wurden eine Reihe nichtsperrender Verfahren zur Nebenläufigkeitskontrolle vorgeschlagen: *Optimistic Locking*, *Serialization Graph Testing*, *Timestamp Ordering* und *Forward/Backward Oriented Optimistic Concurrency Control (FOCC/BOCC)*. Abbildung 33 zeigt eine Einordnung dieser Verfahren. Generell kann zwischen op-



timistischen und pessimistischen Verfahren unterschieden werden. In pessimistischen Verfahren prüft der Transaktionsscheduler für jede Operation (Lesen, Schreiben), ob diese Operation zugelassen werden kann, ohne die Serialisierbarkeit der Transaktionen zu gefährden. Bei optimistischen Verfahren erfolgt die Prüfung auf Verletzungen der Serialisierbarkeit erst zum Zeitpunkt des Transaktionscommit, d.h. diese Verfahren gehen optimistisch davon aus, dass Konflikte unwahrscheinlich sind. Eine Verletzung der Serialisierbarkeit führt in einer optimistischen Transaktion beim Commit zum Abbruch einer oder mehrerer Transaktionen, die die Serialisierbarkeit verletzen.

Im Folgenden soll untersucht werden, welche dieser Verfahren mit ORESTES und dem Web-Caching Modell kompatibel sind. Damit ein Algorithmus mit ORESTES kombinierbar ist, müssen drei Bedingungen gelten:

1. Stale Reads müssen erkannt werden.
2. In Web-Caches ist keine Logik (z.B. Sperren) erforderlich.
3. Das vollständige Read-Set steht erst beim Commit zur Verfügung.

3.1.1 Serialization Graph Testing (SGT)

Der Serialization Graph Testing Algorithmus ist die unmittelbarste Umsetzung des Serialisierbarkeitstheorems [8]:

Ein Schedule ist serialisierbar, wenn der Konfliktgraph keinen Zyklus enthält.

Der SGT Algorithmus pflegt den Konfliktgraph explizit als Datenstruktur und fügt für jede eingehende Operation (Read/Write) wenn nötig eine Kante in den Konfliktgraphen ein. Der Algorithmus ist in Abbildung 34 gezeigt (cf. [8]).

```

scheduleOperation(transaction, operation(object)):
  wenn dies die erste Operation von transaction:
    Füge eine Knoten transaction in den Konfliktgraph G ein

  Füge Kanten (otherTransaction, transaction) in G ein für jede Operation
  und Transaktion otherTransaction, die zuvor eine Konfliktoperation zu
  operation(object) ausgeführt hat

  Wenn G anschließend azyklisch:
    operation(object) akzeptieren
  sonst (G zyklisch):
    Würde die Operation angenommen, wäre die Serialisierbarkeit ver-
    letzt
    --> Ablehnung von operation(object) und Abbruch von transaction

```

Abbildung 34 Der SGT Algorithmus.

Der Algorithmus hat mehrere praktische Probleme, z.B. die Schwierigkeit zu entscheiden, wann eine committete Transaktion aus dem Graph **G** entfernt werden kann. Schwerwiegender ist jedoch, dass der Algorithmus unsere Forderung 1 und 3 nicht erfüllt: Der Algorithmus muss jede lesende Operation explizit annehmen oder ablehnen. Wie Abbildung 35 zeigt,

kann dies nicht gewährleistet werden, wenn Leseanfragen aus Web-Caches beantwortet werden können. Der SGT Scheduler hat keine Kenntnis von den Leseoperationen und kann deshalb Konflikte zwischen Reads und Writes nicht zuverlässig erkennen.

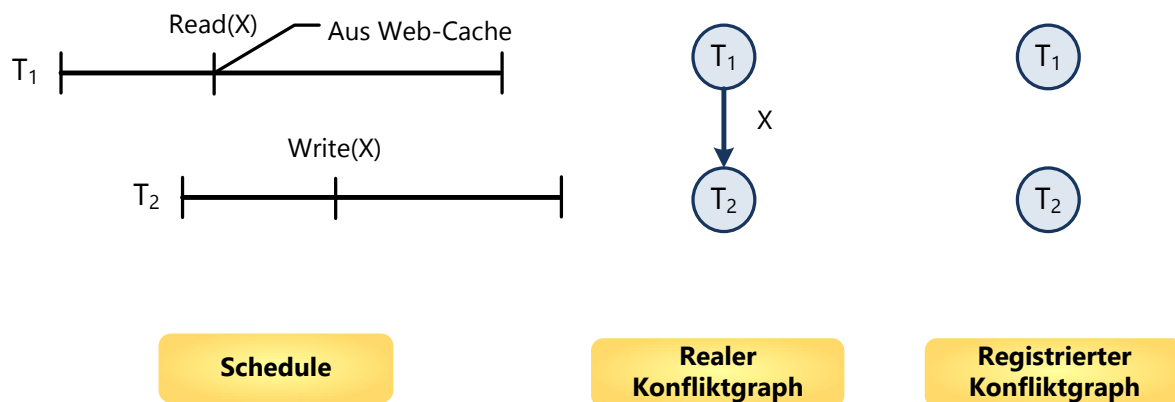


Abbildung 35 Die Unvereinbarkeit von SGT und Web-Caching.

3.1.2 Timestamp Ordering

Eines der einfachsten Verfahren zur Nebenläufigkeitskontrolle ist das Zeitmarkenverfahren (Basic Timestamp Ordering, BTO) [5]. In diesem Verfahren wird die Serialisierungsreihenfolge von Transaktionen a priori durch die Zeitstempel der Transaktionen festgelegt, die beim Transaktionsbeginn vergeben werden. Eine Transaktion darf deshalb nie Änderungen von jüngeren Objekten sehen oder Objekte ändern, die bereits von jüngeren Transaktionen geändert oder gelesen wurden. Sollte dies geschehen, wird die betreffende Transaktion zurückgesetzt. Um diese Forderung umzusetzen, trägt jedes Objekt zwei Zeitstempel:

- Der **Read Timestamp** (RTS) ist der Transaktionszeitstempel derjenigen Transaktion, die das Objekt zuletzt gelesen hat
- Der **Write Timestamp** (WTS) ist der Transaktionszeitstempel derjenigen Transaktion, die das Objekt zuletzt geschrieben hat

Der BTO Scheduler lässt eine Leseoperation $Read(x)$ einer Transaktion T mit dem Zeitstempel $TS(T)$ nur zu, wenn gilt:

$$TS(T) < WTS(x)$$

D.h. keine jüngere Transaktion darf das Objekt geschrieben haben. Eine Schreiboperationen $Write(x)$ wird analog nur zugelassen, wenn gilt:

$$TS(T) < \max(WTS(x), RTS(x))$$

D.h. eine jüngere Transaktion darf das Objekt weder gelesen noch geschrieben haben. Es ist offensichtlich, dass dieses relativ primitive Verfahren nur einen geringen Teil aller serialisierbaren Schedules akzeptiert. Trotz einiger Optimierungen (z.B. Thomas' Write Rule [8]) ist es deshalb von wenig praktischer Relevanz. Auch ist offensichtlich, dass BTO unsere Forderungen nicht erfüllt: um den Lesezeitstempel RTS zu pflegen, ist es nötig, dass jede Leseope-



ration den Scheduler erreicht. Dies kann wie für SGT im Web-Caching Szenario nicht sichergestellt werden. BTO ist deshalb wie SGT nicht mit ORESTES kombinierbar. Nehmen wir sperrende Verfahren in diese Betrachtung auf, die wegen der offensichtlichen Unfähigkeit von Web-Caches mit Sperren umzugehen bereits ausscheiden, stellen wir also fest, dass alle pessimistischen Synchronisationsverfahren inkompatibel mit dem Web-Caching Modell sind.

3.1.3 FOCC und BOCC

Die Idee optimistischer Verfahren ist, dass Konflikte zwischen Transaktionen selten auftreten und sperrende Verfahren deshalb einen unvermeidbaren Overhead darstellen [5], [153], [154]. Ein optimistisches Verfahren lässt deshalb alle Operationen zu und überprüft erst beim Commit, ob Verletzungen der Serialisierbarkeit aufgetreten sind. Diese werden dann durch Rollback einer Transaktion gelöst. Die Ausführung einer optimistischen Transaktion unterteilt sich in drei Phasen (siehe Abbildung 36):

1. **Read Phase:** In der Read Phase führen Transaktionen ihre Arbeit aus (Lesen, Schreiben, Querys, etc.). Anstatt Änderungen jedoch sofort in das Datenbanksystem zu übernehmen, werden diese zuerst privat gespeichert. Dies kann auf verschiedene Weisen geschehen:
 - a. Die Transaktion hält geänderte Objekte in einem Transaktionspuffer, der auf Client oder Serverseite alle geänderten Objekte zwischenspeichert.
 - b. Das Datenbanksystem verwendet Multiversion Concurrency Control, bei dem Änderungen zu Erzeugung einer separaten Objektversion führen und laufende Transaktionen weiterhin die Sicht zum Beginn ihrer Transaktion wahrnehmen (Snapshot Isolation) [135].
 - c. Das Datenbanksystem benutzt einen hybriden Ansatz und sperrt geänderte Objekte, die vor Abschluss der Transaktion auf den Server übertragen werden.
2. **Validation Phase:** Die Validation Phase wird gestartet, wenn eine Transaktion committet. Es wird dann in einem kritischen Bereich überprüft, ob Konflikte zu parallel laufenden Transaktionen aufgetreten sind. Ist es zu einem derartigen Konflikt gekommen, gibt es zwei mögliche Strategien:
 - a. **Kill/Broadcast-OCC:** alle laufenden Transaktionen, die im Konflikt mit der committenden Transaktion stehen, werden zurückgesetzt.
 - b. **Die:** Die committende Transaktion wird zurückgesetzt.
3. **Write Phase:** War die Validierung erfolgreich, werden alle Änderungen der Transaktion persistent und sichtbar gemacht. Ggf. werden dazu auch Log Einträge geschrieben, um die Wiederholbarkeit der Transaktion sicherzustellen (Recovery).

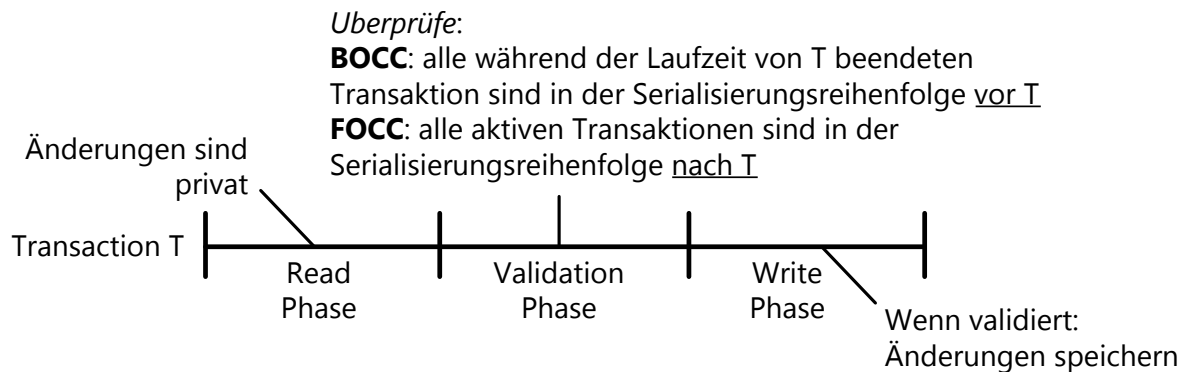


Abbildung 36 Die drei Phasen in optimistischen Synchronisationsverfahren.

Anhand des Validierungsalgorithmus unterscheidet man zwischen zwei Varianten optimistischer Nebenläufigkeitkontrolle: *Forward Oriented* und *Backward Oriented Optimistic Concurrency Control* (FOCC und BOCC). Bei FOCC wird während der Validierung einer Transaktion T sichergestellt, dass gilt:

$$\forall T_{other} \in RunningTransactions : T < T_{other}$$

wobei $<$ die Serialisierungsrelation, d.h. die logische Reihenfolge der Transaktionen kennzeichnet. Für BOCC wird bei Validierung hingegen überprüft, dass gilt:

$$\forall T_{other} \in CommittedTransactionsSinceTStarted : T_{other} < T$$

d.h., dass während der Ausführung von T committete Transaktionen in der Serialisierungsreihenfolge vor der validierenden Transaktion stehen.

```

BOCC_Validation(transaction):
    Für alle während der Laufzeit von transaction beendeten Transaktionen t:
        wenn RS(transaction) ∩ WS(t) ≠ ∅:
            //Eine parallel abgeschlossene Transaktion hat ein Objekt
            geändert, dass in dieser Transaktion gelesen wurde (potentieller Stale Read)
            Validierung fehlgeschlagen
        sonst:
            Write_Phase(transaction)

FOCC_Validation(transaction):
    Für alle anderen aktiven Transaktionen t:
        wenn WS(transaction) ∩ RS(t) ≠ ∅:
            //Eine laufende Transaktion hat ein Objekt gelesen, dass in
            dieser Transaktion geändert wurde und deshalb schon hätte
            sichtbar sein müssen
            Validierung fehlgeschlagen
        sonst:
            Write_Phase(transaction)

```

Abbildung 37 Die Validierung im BOCC und FOCC Algorithmus.

Zur Überprüfung dieser Bedingungen werden im klassischen Ansatz Read- und Write-Sets (RS, WS) vom Datenbanksystem gesammelt. Diese Mengen enthalten die gelesenen bzw.



geänderten Objekte einer Transaktion. Die Validierungsalgorithmen ergeben sich dann wie in Abbildung 37 gezeigt. In dieser Darstellung wird angenommen, dass alle Elemente des Write-Sets auch Teil des Read-Sets sind, d.h. $WS(T) \subseteq RS(T)$. In ORESTES ist dies stets der Fall, da Objektänderungen immer auf bereits gelesenen Objekten basieren (keine *Blind Writes*). Die Validierungen werden in einem kritischen Abschnitt ausgeführt, so dass stets nur eine einzige Transaktion validiert wird. Einige Systeme verzichten bei BOCC auf die Validierung von Read-Sets und beschränken sich auf die Write-Sets. Dies verletzt jedoch die Serialisierbarkeit, da auf diese Weise *Nonrepeatable-Reads* (NRRs) zugelassen werden. Abbildung 38 zeigt ein Beispiel für eine derartige Situation: eine Transaktion liest ein Objekt, das anschließend von einer zweiten Transaktion geändert wird. Nach der Schreibphase der zweiten Transaktion kann das geänderte Objekt gelesen werden. Bei der Validierung wird dieser Nonrepeatable-Read (auch *Inconsistent Read* genannt [5]) nur detektiert, wenn das Read-Set gegen das Write-Set der schon abgeschlossenen Transaktion validiert wird. In ORESTES werden deshalb auch die Read-Sets validiert (wenn der Client dies aktiviert).

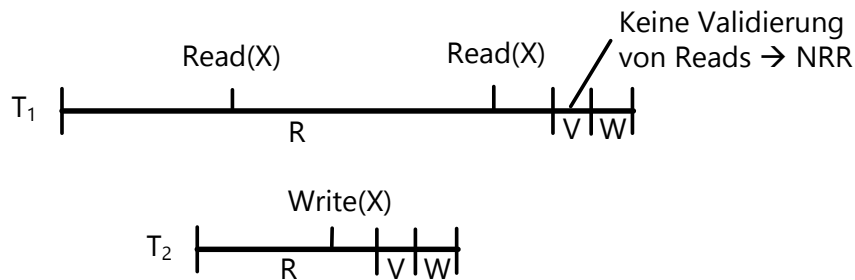


Abbildung 38 Die Gefahr von Nonrepeatable-Reads bei fehlender Validierung des Read-Sets.

Von den beiden optimistischen Verfahren eignet sich lediglich BOCC für das Web-Caching Szenario. Da FOCC bei der Validierungsprozedur das Write-Set der comittenden Transaktion gegen das Read-Set der laufenden Transaktionen prüft, ist dieses Verfahren ungeeignet – es setzt voraus, dass die Read-Sets aller laufenden Transaktionen bekannt sind. Diese sind jedoch erst zum Zeitpunkt des Commits bekannt, da vorher Lesezugriffe für den Server unbemerkt aus Web-Caches beantwortet werden können. Bei BOCC hingegen wird das Read-Set der comittenden Transaktion gegen die Write-Sets der bereits abgeschlossenen Transaktionen validiert. Da einerseits der Client sein Read-Set überträgt und andererseits Schreiboperationen stets den Server erreichen, ist BOCC in dieser Hinsicht mit dem Web-Caching Modell kompatibel (Forderung 3). Dass Stale Reads ebenfalls verhindert werden können, zeigen wir im nächsten Kapitel mit der Verbesserung BOCC+ (Forderung 1). BOCC benötigt darüber hinaus keine spezielle Logik für Leseoperationen, die deshalb aus Web-Caches beantwortet werden können (Forderung 2).

Wir zeigen das nebenläufige Verhalten des BOCC Verfahrens an einem Beispiel (Abbildung 39). Zuerst schreibt Transaktion T2 das Objekt x , das anschließend von Transaktion T1 gelesen wird. Dabei liest T1 die Version von x , die vor dem Transaktionsbeginn von T2 gültig war, da in der Lesephase alle Änderungen privat für jede Transaktion erfolgen. Der Lesezugriff kann zudem aus einem Web-Cache beantwortet werden. Die Validierung von T1 ist er-

folgreich, da keine parallele Transaktion zur Laufzeit von T_1 comittet wurde. Anschließend validiert T_2 erfolgreich. Wären die Transaktion mit einem klassischen RX-Sperrverfahren und Two-Phase-Locking durchgeführt worden, wäre es beim Lesen von x in Transaktion T_1 zu einem Sperrkonflikt gekommen. Durch BOCC wird dieser Overhead durch die optimistische Grundannahme der Konfliktfreiheit verhindert.

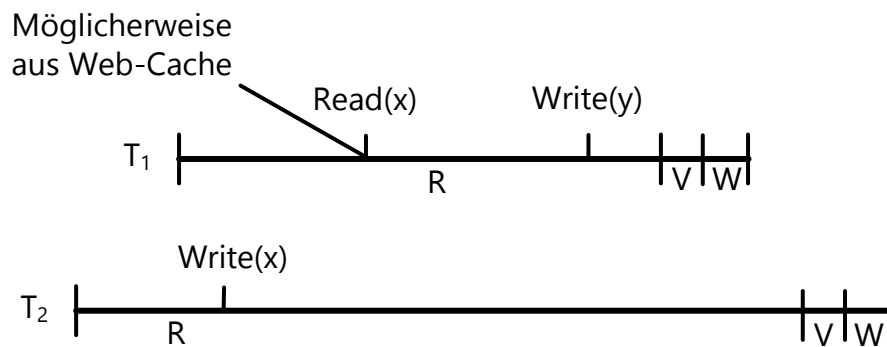


Abbildung 39 Beispiel für die Arbeitsweise von BOCC in Kombination mit Web-Caching.

Zu den Nachteilen von FOCC und BOCC gehört die Gefahr des Verhungerns (*Transaction Starvation* [153]). Verhungern bezeichnet die Situation einer Transaktion, die mehrfach zurückgesetzt wurde, da sie entweder besonders langlebig ist oder auf häufig geänderten Objekten arbeitet. Für langlebige Transaktionen ist zudem das Maß unnötig verrichteter Arbeit groß, da ein Zurücksetzen (bei einer Die-Strategie) erst beim Commit erfolgt. Andererseits sind langlebige Transaktion für alle klassischen Synchronisationsverfahren ein Problem, weshalb für derartige Transaktionen in der Forschung alternative Modelle wie beispielsweise Sagas [155] und Open Nested Transactions [55] vorgeschlagen wurden. Für OLTP (Online Transaction Processing) und OLAP (Online Analytical Processing) Systeme, zu denen ORESTES wie fast alle Datenbanksysteme zählt, sind diese Modelle jedoch kaum relevant, sondern in ihrer Anwendung weitestgehend beschränkt auf Workflow Management Systeme.

3.1.4 BOCC+ und Optimistic Locking

Ein Nachteil von BOCC nach dem in Abbildung 37 dargestellten Algorithmus sind sogenannte *unechte Konflikte* [5]. Diese treten auf, wenn eine Transaktion zurückgesetzt wird, obwohl sie keine veralteten Objekte gelesen hat. Bei einer mengenbasierten Prüfung des Write-Sets comitteter Transaktionen gegen das Read-Set der comittenden Transaktion kann dies geschehen, da jede Information über die zeitliche Korrelation der konkreten Operationen verloren geht. Abbildung 39 zeigt einen derartigen Konflikt. Transaktion T_1 wird nach Transaktion T_2 gestartet und ändert Objekt x . Nachdem T_1 erfolgreich comittet hat, liest T_2 Objekt x . Die Validierung von T_2 ist anschließend nicht erfolgreich, da sich das Read-Set von T_2 mit dem Write-Set von T_1 überlappt. Transaktion T_2 wird deshalb zu Unrecht zurückgesetzt, denn tatsächlich hat T_2 die aktuelle Version von Objekt x gelesen. Dieses Problem lässt sich leicht durch Versionsnummern lösen (nach Rahm BOCC+ genannt [156]). Dazu erhält jedes Objekt eine monoton ansteigende Versionsnummer, die bei jedem erfolgreichen



Schreiben inkrementiert wird. Anstatt beim Commit die anspruchsvoll zu ermittelnde Schnittmenge zwischen dem Read-Set und den Write-Sets zu bilden, muss für jedes Objekt des Read-Sets lediglich geprüft werden, ob die jeweils gelesene Version des Objektes aktuell, d.h. identisch mit der konsistenten Datenbanksicht ist. Auf diese Weise lässt sich auch einfach unsere erste Forderung (das Erkennen von Stale Reads) erfüllen: wenn ein veraltetes Objekt aus einem Web-Cache geladen wurde, trägt es auch eine veraltete Versionsnummer. Bei der Validierung des Read-Sets durch BOCC+ wird dies entdeckt. Bei einer mengenbasierten Prüfung würde dies nicht zuverlässig erkannt werden, da die Stale Reads aus Web-Caches ggf. auch Objekte produzieren können, die nicht während der Ausführung der Transaktion geändert wurden, sondern bereits davor.

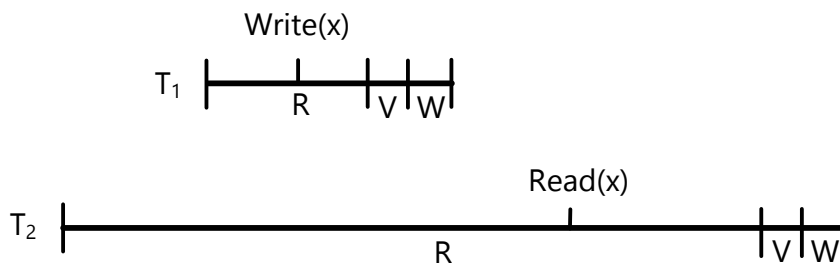


Abbildung 40 Die Problematik unechter Konflikte im mengenbasierten BOCC.

Optimistische Synchronisation durch BOCC+ ist nicht durch den ursprünglichen Ansatz von Rahm bekannt geworden. Vielmehr hat erst der Aufstieg des objekt-relationalen Mappings ein konkretes Bedürfnis an optimistischen Synchronisationsverfahren offenbart. Der BOCC+ Ansatz wird in diesem Kontext *Optimistic Locking* genannt [152]. Optimistic Locking setzt BOCC+ auf Basis eines sperrbasierten Datenbanksystems um. Die Idee dabei ist, ein Versionsfeld explizit ins Schema einer jeden Tabelle oder Klasse aufzunehmen und zur Validierung zu benutzen. In seiner ursprünglichen und naiven Form führt dabei die Applikation selbst die Validierung und den möglichen Rollback oder Commit durch. Abbildung 41 zeigt ein Beispiel für eine in Applikationslogik umgesetzte Transaktion mit Optimistic Locking für eine relationale Datenbank.

```

SELECT Name, Age, Version
FROM Accounts
WHERE Id = 42

--Gelesenes Tupel: ("Felix", 24, 1)
--Am Ende der Transaktion: Werte aktualisieren, Version inkrementieren und Änderung speichern

UPDATE Accounts
SET AGE = 25, Version = 2
WHERE Id = 42
AND Version = 1

```

Abbildung 41 Die ursprüngliche Form des Optimistic Locking in expliziter Form (cf. [8]).

Dieser manuelle Ansatz von Optimistic Locking wurde z.T. stark kritisiert, u.a. von Weikum et al. [8]. Der Hauptkritikpunkt am manuellen Optimistic Locking ist, dass die Applikation selbst dafür verantwortlich ist, die Transaktion zu committen, wenn alle konditionalen Updates erfolgreich sind und sie zurückzurollen, wenn eine davon fehlschlägt. Des Weiteren muss das Schema explizit erweitert werden. Um diese unnötigen Fehlerquellen auszuschließen, sollte die Validierungslogik nach unserer Einschätzung einerseits transparent und andererseits Teil des Servers sein. Im ORESTES REST/HTTP Protokoll besitzen deshalb alle Objekte eine Versionsnummer, die Teil der Objekt-Metadaten ist. Zudem ist der ORESTES-Server dafür zuständig, die Validierung einer Transaktion durchzuführen.

Der Vorteil von Optimistic Locking gegenüber dem klassischen BOCC+, das ohne Sperren auskommt, liegt vor allem darin, dass die Validierung nicht in einem kritischen Bereich ausgeführt werden muss. Vielmehr müssen nur beim abschließenden Schreiben Sperren für das Write-Set benutzt werden, die nur für eine sehr kurze Zeit gehalten werden. Im genannten Beispiel für manuelles Locking (Abbildung 41) werden die Sperren implizit allokiert, da Schreibvorgänge bis zum Commit verzögert werden, um dann als gewöhnlich SQL-UPDATE Statements ausgeführt zu werden. Da wie beschrieben die Deadlock-Wahrscheinlichkeit nach Gray et al. [150] für kurze sperrende Transaktionen sehr gering bleibt, ist das Locking in diesem Fall kein Performance-Bottleneck.

```
try {
    //Transaktion starten
    em.getTransaction().begin();
    //Objekt laden
    Account a = em.find(Account.class, 42, LockModeType.OPTIMISTIC);
    //Änderung vornehmen
    a.setAge(a.getAge() + 1);
    //Transaktion committen
    em.getTransaction().commit();
}
catch (LockTimeoutException lte) {
    //Fehlgeschlagene Validierung
}
```

Abbildung 42 Optimistic Locking in objektorientierten Persistenz-APIs am Beispiel von JPA.

Wie eine adäquate Umsetzung von Optimistic Locking ohne explizite Transaktionsverwaltung in der Applikationslogik aussehen kann, zeigen objektorientierte Persistenz-APIs wie JPA (Abbildung 42). Dort ist zwar das Anlegen eines Versionsfeldes explizit, die Validierungslogik nach dem BOCC+ Prinzip wird jedoch durch die Implementierung der Persistenz-API durchgeführt. Diese wiederum kann die Validierung entweder clientseitig durch Umschreiben von Querys vornehmen oder sie an den Server delegieren. Ersterer Ansatz wird von allen gängigen objektrelationalen Mappern verfolgt (z.B. Hibernate, EclipseLink, Entity Framework [152], [157]). Dazu muss jede Schreiboperation wie im Beispiel für manuelles Optimistic Locking (Abbildung 41) um ein Prädikat erweitert werden, das sicherstellt, dass die Objektversion unverändert ist. Anschließend wird überprüft, ob alle Änderungen



erfolgreich waren. Nur dann wird die Transaktion committet, andernfalls zurückgesetzt. ORESTES hingegen verfolgt den letzteren Ansatz, d.h. der Client delegiert die Validierung an den ORESTES-Server. Dort kann die Überprüfung erstens mit höherer Geschwindigkeit ausgeführt werden und zweitens ohne die Gefahr eines Applikations- oder Konnektivitätsfehlers, der aufgrund von verwaisten Sperren zu Blockierungen und Lock-Timeouts führen würde.

3.1.5 Zusammenfassung

Wie unsere Analyse zeigt, ist das Web-Caching Modell von ORESTES mit den optimistischen Synchronisationsverfahren **BOCC+** und **Optimistic Locking** kompatibel. Das ORESTES REST/HTTP Protokoll stellt dem Datenbankwrapper frei, wie optimistische Transaktionen implementiert werden. So kann der Wrapper die Validierung entweder in einem kritischen Bereich ausführen (BOCC+) oder auf Basis von Sperren im gekapselten Datenbanksystem (Optimistic Locking). Beide Validierungen beruhen auf der Überprüfung von Versionsnummern. Auch Timestamps können in ORESTES als Versionsnummern dienen, vorausgesetzt, sie sind feingranular genug, um Eindeutigkeit zu garantieren. Der Vorteil des Optimistic Locking ist, dass der kritische Bereich durch die Schreibsperren beim Ändern des Read-Sets implizit in das Datenbanksystem verlagert wird. Dadurch sind mehrere parallele Validierungen möglich. Außerdem entfällt gegenüber BOCC+ die Komplexität einer Synchronisierung des kritischen Bereichs über mehr als einen Server für den Fall, dass mehrere ORESTES-Server dasselbe Datenbanksystem kapseln. Unsere ORESTES Implementierungen für VOD, db4o und Redis basieren alle auf Optimistic Locking, wobei das Sperrkonzept je nach Datenbanksystem stark variiert. Durch das ORESTES REST/HTTP Protokoll werden diese Uneinheitlichkeiten jedoch durch das einfache optimistische Transaktionsprinzip mit versionierten Objekten abstrahiert.

ORESTES hat keinen Einfluss auf die ACID Garantien des zugrundeliegenden Datenbanksystems. Insbesondere verhalten sich alle Isolation Level in ORESTES so, als gäbe es kein Web-Caching. Es werden keine Mehrbenutzeranomalien verursacht:

Lost Updates. Die optimistische Nebenläufigkeitskontrolle verhindert Lost Updates. Sie geht davon aus, dass Write/Write Konflikte selten sind, da sie zu einem Transaktionsabbruch führen.

Dirty Reads. Dirty reads können nur auftreten, wenn das zugrundeliegende Datenbanksystem die transaktionale Isolation verletzt. BOCC+ und Optimistic Locking gehen davon aus, dass geänderte Objekte bis zum Commit entweder privat oder gesperrt sind. ORESTES verhindert das Caching von schmutzigen Objekten explizit, indem geänderte Objekte von einer privaten, uncachbaren und transaktionsspezifischen Ressource (`dbview`) abgerufen werden.

Non-repeatable Reads. In ORESTES werden Nonrepeatable Reads (NRRs) ebenfalls durch die optimistische Nebenläufigkeitskontrolle verhindert. Durch die explizite Versionierung der Objekte können NRRs auch direkt von dem Client erkannt werden, der das Read-Set sam-

melt. Wird ein Objekt geladen, das sich bereits im Read-Set befindet, jedoch eine andere Version besitzt, ist das ein Zeichen für einen NRR. Wenn der Server die optimistische Nebenläufigkeit auf Basis von MVCC umsetzt, ergibt sich in ORESTES eine Besonderheit: eine Transaktion kann ein Objekt ein zweites Mal laden, nachdem es von einer bereits committeten Transaktion geändert wurde. Da die Objekt-Ressourcen des Servers zustandslos sind und unabhängig vom Client stets die neuste transaktional konsistente Objektversion repräsentieren, kann der Server keine älteren Versionen ausliefern. Um eine konsistente Sicht zu bewahren, kann ein Client deshalb ein Objekt von seiner privaten Ressource *dbview* laden, wenn ein Objekt transaktional ein zweites Mal geladen wird. Auf diese Weise kann der Transaktionsabbruch verhindert werden, der andernfalls bei der Validierung eines Read-Sets mit zwei abweichenden Objektversionen die Folge wäre.

Phantom Reads. Alle Querys, die über einfachen Objektzugriff anhand der ID hinausgehen, erreichen stets den Server. Phantom Reads können nur für Prädikate auftreten (z.B. beim Selektieren aller Angestellten mit einem bestimmten Gehalt). Es ist deshalb die Aufgabe des zugrundeliegenden Datenbanksystems, nichtwiederholbare Prädikatsanfragen zu verhindern (z.B. durch Tabellen-, Klassen- oder Prädikatssperren) – vorausgesetzt, der Isolationslevel ist *Serializable*.

Zusammenfassend lässt sich festhalten, dass ORESTES auch im Kontext von Web-Caching durch die Algorithmen BOCC+ und Optimistic Locking die Serialisierbarkeit von Transaktionen gewährleisten kann – potentiell auf Kosten einer höheren Transaktionsabbruchrate.

3.2 Grundkonzept

Ziel ist es, die Änderungen an der Datenbank von den ORESTES-Servern an die Clients zu übertragen, damit diese alle veralteten Objekte mit einer Revalidierung anfragen. Wir formalisieren das Problem, um potentielle Änderungsdatenstrukturen beurteilen zu können. Wir bezeichnen mit $WTS(x)$ wieder den Schreibzeitstempel eines Objektes und mit $Now()$ den Zeitstempel der Gegenwart. Die Expirationsdauer E gibt an, wie lange Web-Caches Objekte maximal speichern dürfen.

Die Änderungsdatenstruktur muss die Menge *StaleObjects* aller Objekt-IDs obj beinhalten, deren letzte Änderung weniger als E Zeiteinheiten in der Vergangenheit liegt:

$$StaleObjects = \{obj \in db \mid stale(obj)\} = \{obj_1, obj_2, \dots, obj_n\}$$

$$stale(obj) := \begin{cases} \mathbf{true}, & \text{wenn } Now() < WTS(obj) + E \\ \mathbf{false}, & \text{sonst} \end{cases}$$

Die Anfragen, die der Server auf der Datenstruktur durchführen muss, sind:

- *Insert*. Einfügen von Objekt-IDs bei eingehenden Änderungen.
- *Delete*. Löschen von Objekten aus der Datenstruktur, wenn sie seit mindestens E Zeiteinheiten nicht geändert wurden.
- *Contains*. Prüfen, ob eine Objekt-ID in der Datenstruktur enthalten ist.



Der Client benötigt die ändernden Operationen nicht, lediglich *contains* ist erforderlich. Die Definition der drei Operationen ist offensichtlich:

$$\text{Insert}(\text{StaleObjects}, \text{obj}) = \text{StaleObjects} \cup \{\text{obj}\}$$

$$\text{Delete}(\text{StaleObjects}, \text{obj}) = \text{StaleObjects} \setminus \{\text{obj}\}$$

$$\text{Contains}(\text{StaleObjects}, \text{obj}) = \begin{cases} \text{true}, & \text{wenn } \text{obj} \in \text{StaleObject} \\ \text{false}, & \text{sonst} \end{cases}$$

Von der Änderungsdatenstruktur wird auf Serverseite also die Semantik eines mutablen Sets gefordert, während der Client die Semantik eines immutablen Sets benötigt. Eine Besonderheit des Sets ist, dass die Extension der Menge nicht erfassbar sein muss, d.h. weder Client noch Server müssen über den Inhalt der Menge iterieren können. Wir stellen nun kurz drei mögliche Implementierungen der Datenstruktur vor: sortierte Listen, balancierte Suchbäume und Hashtabellen.

3.2.1 Sortierte Listen

Die *StaleObjects* Menge lässt sich sehr leicht als geordnete Liste implementieren. Dabei gibt es zwei prinzipielle Umsetzungen: Array-Listen und verkettete bzw. Skip-Listen. Beide Varianten haben unterschiedliche Zeitkomplexitäten für die geforderten Operationen. Für die Array-Liste ist das Einfügen aufwändig, da nach dem Suchen der richtigen Einfügeposition Elemente verschoben werden müssen, um einen Platz für das neue Element freizugeben. Gleiches gilt für Löschungen, bei denen ein freier Platz geschlossen werden muss. Der Lookup kann durch binäre Suche auf dem sortierten Array erfolgen. Für die Array-Liste ergibt sich also:

$$\text{Insert} \in O(n), \text{Delete} \in O(n), \text{Contains} \in O(\log(n))$$

Die Implementierung ist also außerordentlich ineffizient. Verkettete und Skip-Listen sind ebenfalls ineffizient, da sie keine binäre Suche erlauben. Dafür kann das Löschen in $O(1)$ durch einfache Referenzmanipulation implementiert werden. Das Einfügen benötigt ebenfalls lineare Zeit, da es die Suche nach der geeigneten Position voraussetzt. Damit ergibt sich:

$$\text{Insert} \in O(n), \text{Delete} \in O(1), \text{Contains} \in O(n)$$

Obwohl die Datenstruktur als sortierte Liste also leicht implementiert werden kann, ist schon die Laufzeit der Operationen ein Ausschlusskriterium.

3.2.2 Balancierte Suchbäume

Balancierte Bäume sind Suchbäume, die garantieren, dass die Höhe des Baumes logarithmisch in der Anzahl eingefügter Elemente ist [158]. Suchbäume sind vor allem im sortierten Durchwandern einer Liste effizient, bieten aber auch für schlüsselbasierte Suche gute Laufzeiten. Die verbreitetste Datenstruktur zur Datenbank-Indizierung für Felder mit effizientem Punkt und Bereichszugriffe ist deshalb der B-Baum in seinen zahlreichen Varianten und Erweiterungen [55]. Das Prinzip aller Suchbäume ist der Aufbau der Datenstruktur als Baum

mit Verzweigungen in Abhängigkeit einer Ordnungsrelation über den Schlüsseln der gespeicherten Objekte. Im Fall der *StaleObjects* Datenstruktur wären diese Schlüssel Objekt-IDs.

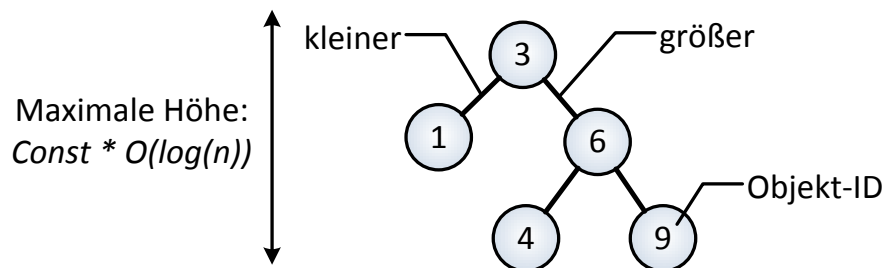


Abbildung 43 Balancierte Suchbäume für die Änderungsdatenstruktur bei n eingefügten Elementen.

Umsetzungen von Suchbäumen variieren vor allem im Verzweigungsgrad und in der Balancierungsstrategie. Die Balancierung ist für Suchbäume eine wichtige Strategie, da sie andernfalls zu einer Liste entarten können (Baumtiefe $O(n)$), was für Operationen zu einer linearen Laufzeit führt. Für balancierte binäre Suchbäume sind die populärsten Varianten AVL-Bäume, Rot-Schwarz-Bäume, Scapegoat-Trees und Splay Trees [158]. Diese Datenstrukturen besitzen, die für die drei geforderten Operationen dieselbe amortisierte Laufzeit:

$$\text{Insert} \in O(\log(n)), \text{Delete} \in O(\log(n)), \text{Contains} \in O(\log(n))$$

Bis auf konstante Faktoren unterscheiden sich Suchbäume mit höherem Verzweigungsgrad (z.B. Mehrwegbäume und B-Bäume) i.d.R. nicht von diesen Laufzeiten. Einige Suchbäume implementieren spezielle Optimierungen, beispielsweise können Van Emde Boas Bäume für Schlüssel der Länge m Bit eine Laufzeit von $O(\log(m)) = O(\log(\log(n)))$ garantieren. Da allgemeine Suchbäume auch das Traversieren performant unterstützen, können diese Operationen nicht schneller als in $O(\log(n))$ ausgeführt werden. Das Traversieren ist jedoch keine Operation, die für die *StaleObjects* Datenstruktur notwendig ist und Suchbäume deshalb keine geeignete Lösung. Hinzu kommt, dass Suchbäume sich nicht effizient serialisieren lassen (Forderung *schnelle Generierung*), da hierfür der gesamte Baum über Pointer Traversal durchschritten werden muss.

3.2.3 Hash-Tabellen

In den meisten Programmiersprachen ist die Set-Datenstruktur auf Basis einer Hash-Tabelle implementiert [159]. Hashtabellen basieren auf einem Array. Vollständige Objekte werden an einer Position gespeichert, die durch das Anwenden einer Hashfunktion auf den Schlüssel (d.h. die Objekt-ID) ermittelt wird (siehe Abbildung 44). Dabei kann es zu Kollisionen kommen, wenn die Hashfunktion Schlüssel auf den gleichen Hashwert abbildet. Es gibt mehrere etablierte Strategien, um mit diesen Kollisionen umzugehen. Zu ihnen zählen offene Adressierung mit linearer und quadratischer Sondierung sowie Double Hashing, Überlauf-listen, Coalesced Chaining, Cuckoo Hashing und Hopscotch Hashing [158], [160]–[162]. Die Kollisionsstrategie kann für die praktische Umsetzung große Unterschiede machen, das



prinzipielle Resultat ist jedoch das gleiche: das Einfügen erfordert $O(1)$ amortisierte Laufzeit [158]. Die Worst-Case Laufzeit variiert je nach Strategie der Kollisionsauflösung und beträgt bei einigen Verfahren $O(n)$ (z.B. wenn es bei offener Adressierung zum Clustering kommt) und bei anderen $O(1)$ (z.B. bei Überlaufbehältern mit Pointern auf das Listenende). Amortisierte Laufzeiten machen eine Aussage darüber, welche Laufzeit eine Operation in der kompletten Sequenz von Anweisungen durchschnittlich benötigt. Sie ist damit i.d.R. nicht identisch mit der Average-Case-Laufzeit, die nur eine Aussage über die durchschnittliche Laufzeit macht, ohne zusammenhängende Sequenzen einer Operation zu berücksichtigen. Da Hash-Tabellen auf einem Array basieren, kann es nach dem Einfügen vieler Elemente zu einem sogenannten *Rehashing* kommen, bei dem das zugrundeliegende Array vergrößert und alle Elemente neu verteilt werden. Das Rehashing ist sehr aufwendig. Da es jedoch in einer Sequenz von Einfügeoperationen nur selten zu einem Rehashing kommt, ist die amortisierte Laufzeit dennoch $O(1)$. Für den ORESTES-Server sind das Rehashing und die Worst-Case Laufzeit einer Einfügeoperation jedoch ein kritisches Problem, da es zu vorhersagbaren Antwortzeiten führt. Das Rehashing kann vermieden werden, indem eine Hash-Tabelle in einer Größe allokiert wird, die stets für alle Elemente Platz bietet.

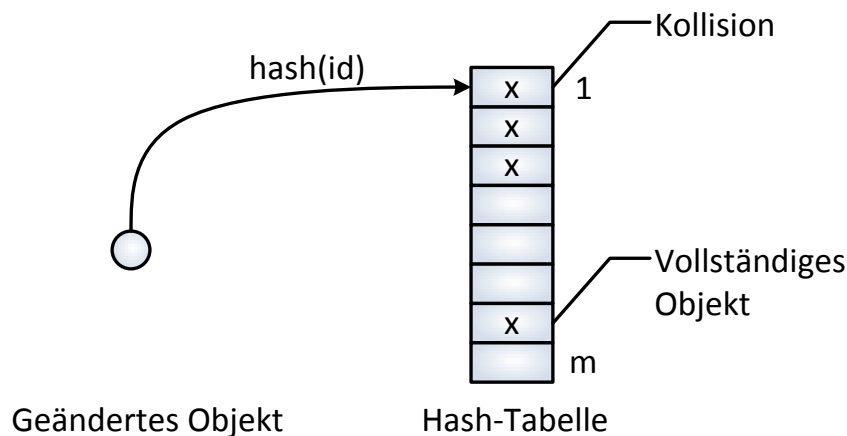


Abbildung 44 Nutzung einer Hashtabelle für die Änderungsdatenstruktur.

Das Auffinden eines Elements benötigt ebenfalls $O(1)$ amortisierte Laufzeit. Die Worst-Case Laufzeit beträgt jedoch $O(n)$, da jede Kollisionsstrategie versagt, wenn die gewählte Hashfunktion viele Kollisionen erzeugt. Das Löschen eines Elements verhält sich analog zum Suchen und benötigt deshalb ebenfalls $O(1)$ amortisierte Laufzeit und $O(n)$ im Worst-Case. Für ein Set auf Basis einer Hash-Tabelle sind die Laufzeiten also:

$Insert \in O(1), Delete \in O(1), Contains \in O(1)$ [Amortisierte Laufzeit]

$Insert \in O(1), Delete \in O(n), Contains \in O(n)$ [Worst – Case Laufzeit]

Die benötigten Operationen können im Durchschnitt also effizient durchgeführt werden. Die Datenstruktur hat dennoch mehrere Nachteile:

- Die schlechte Worst-Case Laufzeit und damit einhergehende Unvorhersagbarkeit von Laufzeiten macht es unmöglich, ein System zu entwickeln, das mit konstant niedriger Latenz antwortet und dies ggf. sogar als Service Level Objective zusichert.
- Der Platzbedarf einer Hashtabelle ist groß ($O(n) + \text{Overhead}$), da die Hashtabelle (bzw. das zugrundeliegende Array) nicht dicht besetzt sein darf, um gute Laufzeiten für *Contains* zu garantieren. Außerdem ist der Platzoverhead durch die meisten Strategien zur Kollisionsauflösung signifikant. Die gängigste Implementierung, die auf Überlaufbehältern basiert (z.B. in Java), würde für jede eingefügte Objekt-ID ein vollständiges Objekt mit Metadaten zur Umsetzung einer verketteten Liste allokiieren.
- Die serverseitige Anforderung zur *schnellen Generierung* ist nur mit offener Adressierung umsetzbar, da andernfalls ein Objekt-Graph traversiert werden muss, um die Hashtabelle zu serialisieren.

Hashtabellen weisen mit Hashing bereits den Weg zu einer geeigneten Datenstruktur, besitzen aber zu viele Nachteile, um als Änderungsdatenstruktur tauglich zu sein.

3.3 Bloomfilter und ihre mathematischen Eigenschaften

Ein einfaches Beispiel soll demonstrieren, dass die drei gezeigten Datenstrukturen bereits durch ihren Platzbedarf ungeeignet zur Speicherung der Änderungen sind. Angenommen, Objekte werden für $E = 3600$ Sekunden gecacht, um eine hohe Cache-Hit Ratio zu erzielen und der Server erfährt durchschnittlich $wps = 10$ Änderungen pro Sekunde. Für die Kardinalität der *StaleObjects* Menge ergibt sich dann eine Kardinalität von $|StaleObjects| = wps * E = 36.000$. Nehmen wir an, dass die Objekt-IDs – an die ORESTES als einzige Bedingung Eindeutigkeit stellt – als UUIDs mit einer Größe von 128 Bit implementiert sind. Eine Datenstruktur, die diese Änderungen speichern soll, benötigt also mindestens $Space = |StaleObjects| * 128 = 4.608.000$ Bits, also ca. 4,5 MByte Speicher. Der datenstrukturspezifische Overhead (z.B. Pointer in verketteten Listen, leere Plätze in einer Hashtabelle, Referenzen auf Kindknoten in Suchbäumen) erhöht den Speicherbedarf in der Praxis weit darüber hinaus. Dieser Speicherbedarf ist völlig inakzeptabel, da die Änderungsdatenstruktur bei jedem Transaktionsbeginn an den Client übertragen werden soll. Da die Entropie von zufälligen Objekt-IDs sehr hoch ist, kann auch über klassische Wörterbuch-Verschlüsselungen wie Lempel-Ziv-Welch oder Entropiecodierung wie Huffman-Encoding keine bedeutende Reduktion des Platzbedarfs für Übertragung erzielt werden. Für diese Datenstrukturen ist die einzige Strategie, den Speicherbedarf herabzusetzen, eine Verkürzung der Expirationdauer E . Die Wahl von E hat jedoch drastische Auswirkungen auf die Cache-Hit-Ratio, die mit der Expirationdauer ansteigt.

Um den Speicherbedarf der *StaleObjects* Menge dennoch gering zu halten, sind zwei Beobachtungen zentral:

- Die Objekt-ID muss nicht explizit gespeichert werden, da sie nie abgerufen wird.



- False-Positives sind für *Contains* Anfragen tolerierbar – die Konsequenz ist lediglich eine Anfrage, die statt Web-Caches stets den ORESTES-Server erreicht. Im Vergleich zu einem Transaktionsabbruch aufgrund eines Stale-Reads ist der Latenzoverhead durch False-Positives marginal.

Diese zwei Beobachtungen machen sich **Bloomfilter** zunutze. Bloomfilter sind eine probabilistische Datenstruktur für Mengen, deren Zeit- und Speicherkomplexität außerordentlich gering ist. Ein Bloomfilter erlaubt die Anfragen *Contains* und *Insert* mit einer Worst-Case Laufzeit von $O(1)$. Der Vorteil wird durch die Wahrscheinlichkeit p für False-Positives (*False-Positive-Rate*) erzielt. Abbildung 45 demonstriert diese Eigenschaft: ein Bloomfilter gibt für eine *Contains* Anfrage nie fälschlich *false* zurück, hat aber eine gewisse Wahrscheinlichkeit p , für ein nicht enthaltenes Element *true* zurückzugeben. Der Platzbedarf des Bloomfilters m (in Bits) ist eine Funktion der False-Positive-Rate p , d.h. $m = f(p)$. Bloomfilter unterstützen kein Löschen von Objekten. Wie wir jedoch näher beschreiben werden, gibt es ein Bloomfilter-Derivat (*Counting Bloomfilter*), das eine *Delete* Operation ebenfalls in $O(1)$ unterstützt.

```
//Objekte in einem Bloomfilter speichern
StaleObjects = {obj1, obj2, ...}
Bloomfilter bf = new Bloomfilter
for each obj in StaleObjects:
    bf.add(obj)

def contains(bf, obj)
    if obj in StaleObjects:
        //Objekt ist enthalten
        return true
    else:
        //Objekt ist nicht enthalten
        return true mit Wahrscheinlichkeit p, sonst false
```

Abbildung 45 Das Prinzip eines Bloomfilters.

Das Prinzip Bloomfilter für Mengen einzusetzen, wenn der Platzbedarf von hoher Wichtigkeit ist und False-Positives tolerierbar, wurde nach Broder et al. als *Bloomfilter Principle* bekannt (aus [163]):

Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.

Bloomfilter wurden bereits 1970 von Burton Bloom entwickelt [136]. Bloom benötigte für ein Silbentrennungsprogramm eine speichereffiziente Datenstruktur, um für ein Wort abzufragen, ob es sich wie 90% aller englischen Worte für eine regelbasierte Silbentrennung eignet oder es in einem Wörterbuch auf der Festplatte nachgeschlagen werden muss. Speichereffizienz war entscheidend, damit die Datenstruktur im Hauptspeicher gehalten werden konnte, der für das Wörterbuch bei Weitem zu klein war. Der negative Seiteneffekt eines False-Positives war dabei wie in ORESTES lediglich eine unnötige Operation – er resultierte im Nachschlagen eines Wortes, das eigentlich keines Nachschlagens bedurft hätte. Seit ihrer

Erfindung wurden Bloomfilter für unterschiedlichste Datenbankproblematiken eingesetzt. In den letzten 10 Jahren wurde ihre Nützlichkeit für viele weitere Gebiete wie Kollaboration, Routing und Datenanalyse entdeckt [163].

Ein Bloomfilter repräsentiert eine Menge $S = \{x_1, x_2, \dots, x_n\}$ mit n Elementen durch ein Bit-Array der Länge m . Es werden k unabhängige Hashfunktionen h_1 bis h_k benötigt, die jeweils von dem Universum der Menge S nach $\{1, \dots, m\}$ abbilden. Voraussetzung für die mathematischen Garantien des Bloomfilters ist, dass die Abbildung der Hashfunktion gleichverteilt über den Bildbereich $\{1, \dots, m\}$ erfolgt. Um ein Element x in dem Bloomfilter zu speichern, werden die k Hashwerte des Elements $h_1(x)$ bis $h_k(x)$ berechnet und als Positionen in dem Bit-Array auf 1 gesetzt. Ist eine Position in dem Bit-Array bereits auf 1 gesetzt, bleibt sie es auch weiterhin. Um abzufragen, ob ein Element y enthalten ist, werden wie zuvor die Hashwerte berechnet und überprüft, ob jede Position auf 1 gesetzt ist. Auf diese Weise können False-Positives entstehen: wurden alle Positionen von y durch das Einfügen von anderen Elementen auf 1 gesetzt, wird y fälschlich als enthalten erkannt. Abbildung 46 zeigt den Pseudocode für die *Insert* und *Contains* Methode.

```
def insert(obj):
    for each position in hashes(obj):
        #Alle Positionen auf 1 setzen
        bits[position] = 1

def contains(obj):
    for each position in hashes(obj):
        if bits[position] == 0:
            //Falls auch nur eine der Position 0 ist,
            //ist das Element nicht enthalten
            return false;
    return true
```

Abbildung 46 Die *Insert* und *Contains* Methode eines Bloomfilters.

Abbildung 47 zeigt ein Beispiel für das Einfügen und Abfragen von Elementen. Um die False-Positive Rate zu bestimmen, nutzen wir die Annahme, dass die Hashwerte zufällig (gemäß einer diskreten Gleichverteilung) sind. Die Herleitung orientiert sich an Broder et al. [163] und Mitzenmacher et al. [160]. Die Wahrscheinlichkeit, dass ein bestimmtes Bit durch eine spezifische Hashfunktion nicht auf 1 gesetzt wird, ist:

$$1 - \frac{1}{m}$$

Für k Hashfunktionen und nach dem Einfügen von n Elementen ist diese Wahrscheinlichkeit für ein Bit:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$



Die Approximation ist bis auf $O\left(\frac{1}{m}\right)$ präzise und damit eine sehr gute Näherung [163]. Mit dieser Näherung ergibt sich die Wahrscheinlichkeit, dass ein bestimmtes Bit nach n Einfügeoperationen 1 ist, als die Gegenwahrscheinlichkeit:

$$1 - p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right) \approx 1 - e^{-\frac{kn}{m}}$$

Damit ein False-Positive auftritt, müssen alle k Hashfunktionen eine Kollision erzeugen, d.h. die jeweiligen Bits auf 1 gesetzt worden sein. Unter der Annahme, dass die Wahrscheinlichkeit, für jedes Bit 1 zu sein unabhängig voneinander sind, ergibt sich die False-Positive Rate zu:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

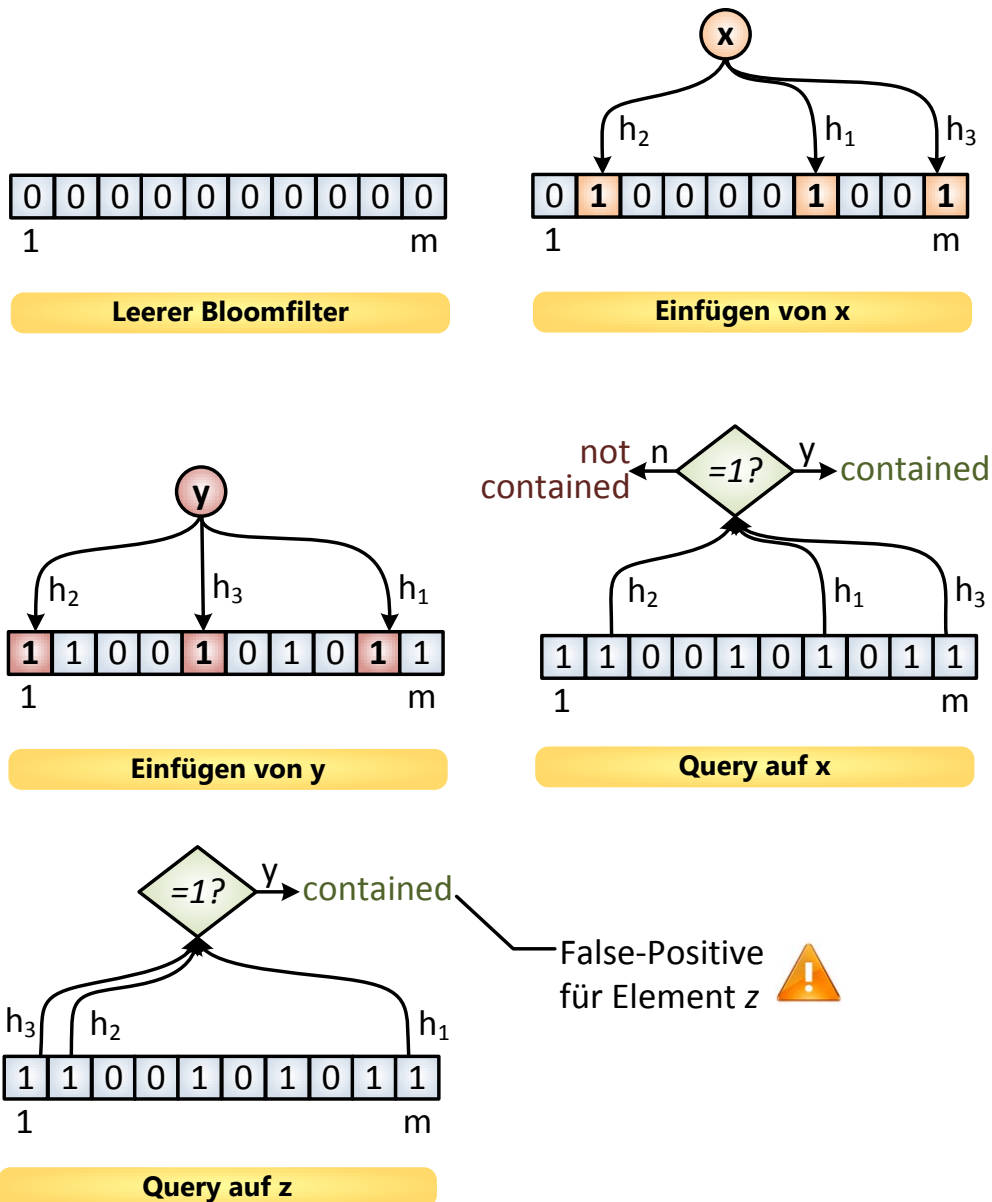


Abbildung 47 Beispiel für Einfügen und Abfragen auf einem Bloomfilter.

D.h. die Wahrscheinlichkeit eines False-Positives wird kleiner, wenn die Anzahl an Bits m zunimmt und wird größer, je mehr Elemente n eingefügt werden. Um die optimale Anzahl an Hashfunktionen k für ein gegebenes m und n zu finden, muss f nach k abgeleitet werden. Dazu kann f durch Anwendung von Potenzgesetzen ausgedrückt werden als $f = \exp(k \ln(1 - e^{-kn/m}))$ [163]. Dieser Ausdruck wird minimiert, wenn der Exponent $g = k \ln(1 - e^{-kn/m})$ minimiert wird:

$$\frac{\partial g}{\partial k} = \frac{k n}{\left(-1 + e^{-\frac{k n}{m}}\right) m} + \ln\left(1 - e^{-\left(\frac{k n}{m}\right)}\right) = 0$$

Die Ableitung wird 0, wenn für die Anzahl der Hashfunktionen gilt:

$$k = \ln(2) * \frac{m}{n} \text{ (cf. [163])}$$

Setzt man dieses globale Minimum in f ein, erhält man für die False-Positive Rate:

$$f \approx (1 - e^{-\ln(2)})^k = \left(\frac{1}{2}\right)^{\ln(2) * \frac{m}{n}} \approx 0,6185^{\frac{m}{n}}$$

Mitzenmacher zeigt in [144], dass die in der Herleitung benutzten Approximationen sehr genau sind. Bei einer gegebenen Anzahl zu erwartender Objekte n , einer gewünschten False-Positive Rate f und der optimalen Anzahl an Hashfunktionen k , ergibt sich anhand der obigen Formel die erforderliche Größe der Bloomfilters:

$$f \approx \left(\frac{1}{2}\right)^{\ln(2) * \frac{m}{n}} \Rightarrow \ln(f) \approx -\ln(2)^2 * \frac{m}{n} \Rightarrow m = -\frac{n \ln(f)}{\ln(2)^2} \approx -2.081 * n \ln(f)$$

Die Anzahl an Bits muss natürlich auf ganze Bits gerundet werden. Die angegebene Formel ist zentral für die Konfiguration eines Bloomfilters. In ORESTES berechnen wir anhand einer gegebenen Anzahl von Änderungen wps pro Sekunde, der Expirationsdauer E und einer gewünschten False Positive Rate f die notwendige Größe des Bloomfilters zu:

$$m = -2.081 * wps * E * \ln(f)$$

Für das eingangs genannt Beispiel von $wps = 10$ und $E = 3600$ (d.h. 36000 geschriebenen Objekten pro Stunde) ergibt sich bei einer False-Positive Rate von $f = 1\%$ ein Bloomfilter der Größe:

$$m = 345062 \text{ Bit} = 329 \text{ KByte}$$

Die Größe des Bloomfilters ist in diesem Szenario also vergleichbar mit der Größe eines komprimierten Bildes einer Website. Wird die Expirationszeit von einer Stunde auf 10 Minuten verringert, geht die Größe zurück auf:

$$m = 55 \text{ KByte}$$



Der Aufwand für die Übertragung einer Datenstruktur dieser Größe ist bei gängigen Datenraten vernachlässigbar. Wird der Bloomfilter beispielsweise von einem Mobilgerät über einen HSPA+ Link mit 21,1 MBit/s heruntergeladen, beträgt die Zeit den Bloomfilter zu empfangen $t = 20ms$. Da die Netzwerklatenz von mobilen Geräten zwischen $100ms$ und $1000ms$ beträgt (nach [164]), ist die Übertragungsdauer des Bloomfilters im Verhältnis zur Netzwerklatenz vernachlässigbar klein.

Die Formel für die Größe des Bloomfilters zeigt, dass die Anzahl benötigter Bits m proportional zu der erwarteten Anzahl eingefügter Elemente n ist. Tatsächlich benutzt der Bloomfilter bei einer gegebenen False-Positive Rate f :

$$bpe = 2.081 * \ln\left(\frac{1}{f}\right)$$

Bits für jedes Element. Wird die Größe unerwartet überschritten, steigt die False-Positive Rate. Würde in unserem Beispiel mit $wps = 10$ und $E = 3600$ sich die Aktivität sprunghaft vergrößern und 50% mehr Objekte pro Stunde geschrieben werden als zuvor, würde die False-Positive Rate von 1% auf 4,6% ansteigen. Dies ist in Abbildung 48 gezeigt. Die logarithmische x-Achse gibt den Faktor an, um den die Menge an eingefügten Objekten von der erwarteten Anzahl ($wps * E = 36000$) abweicht. Es wird deutlich, dass die False-Positive Rate für den wenig gefüllten Bloomfilter sehr schnell gegen 0 strebt.

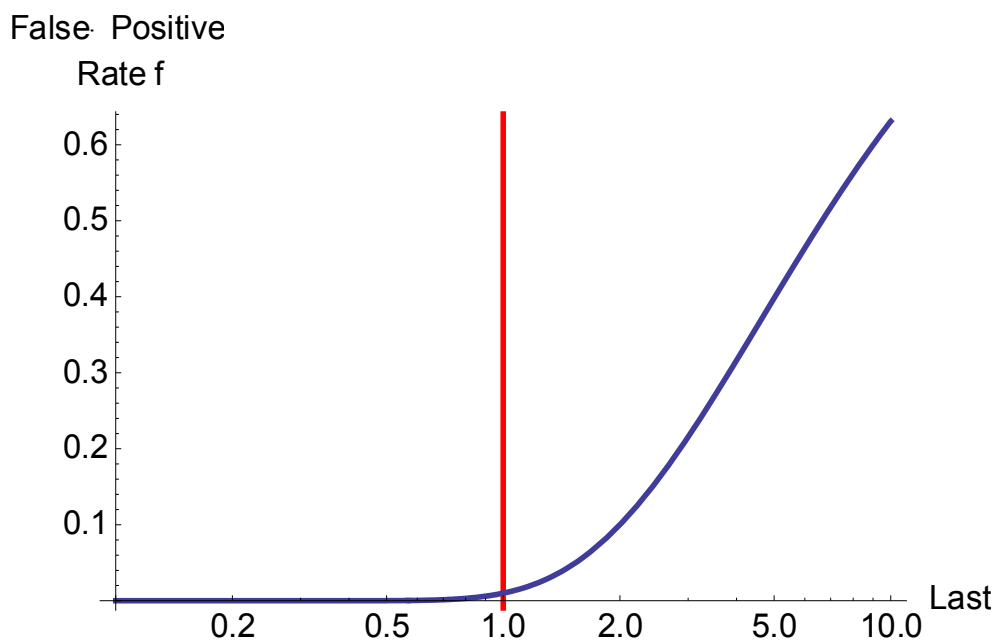


Abbildung 48 Die False-Positive Rate f für unterschiedliche Lasten, bei $m=345062$ Bit. Die erwartete Last von $WPS=10$ ist durch die rote Linie markiert. Die x-Achse (logarithmisch) gibt an, um welchen Faktor die tatsächliche Last von diesem Wert abweicht.

Die wichtigen Zusammenhänge zwischen den Parametern des Bloomfilters sind in Tabelle 6 zusammengefasst. Am Beispiel wurde bereits deutlich, dass der Platzbedarf eines Bloomfilters signifikant geringer ist als die von klassischen, nicht-probabilistischen Datenstrukturen für Mengen. Es lässt sich sogar zeigen, dass Bloomfilter nur einen 44% höheren Platzbedarf

haben als die (asymptotische) Untergrenze für eine Mengen-Datenstruktur mit gleicher False-Positive Rate [165]. Um dies zu beweisen, betrachten wir eine Menge X mit n Elementen aus einem Universum der Größe u , die durch ein Bit-Array der Länge m repräsentiert werden soll. Da False-Negatives verboten sind, muss jedes Element, das in X enthalten ist, als enthalten zurückgegeben werden. Durch das Erlauben einer False-Positive Rate f dürfen jedoch auch $f(u - n)$ weitere Elemente fälschlich als enthalten zurückgegeben werden. D.h. es werden insgesamt höchstens $n + f(u - n)$ Elemente als enthalten zurückgegeben. Ein konkretes Bit-Array der Länge m kann jede der $\binom{n+f(u-n)}{n}$ n -elementigen Mengen repräsentieren. Da es insgesamt $\binom{u}{n}$ mögliche Mengen und 2^m mögliche Bit-Arrays der Länge m gibt, muss gelten (cf. [163]):

$$2^m * \binom{n + f(u - n)}{n} \geq \binom{u}{n}$$

d.h. die Anzahl repräsentierbarer Möglichkeiten (unter Einbezug von False-Positives) muss größer sein als die Anzahl möglicher Mengen. Es gäbe andernfalls Mengen, die sich nicht darstellen lassen. Nach einigen algebraischen Umformungen kann der obige Ausdruck vereinfacht werden (cf. [163]):

$$m \geq n \log_2 \left(\frac{1}{f} \right) \Rightarrow bpe = \log_2 \left(\frac{1}{f} \right)$$

Eine optimale Datenstruktur benötigt also $\log_2(1/f)$ Bits für jedes Element. Vergleicht man dies mit dem Wert für den Bloomfilter $bpe = 2.081 * \ln(1/f) = 1.44 * \log_2(1/f)$, erhält man das gesuchte Ergebnis: der Bloomfilter ist bis auf einen konstanten Faktor von 44% platzoptimal.

Gegebene Parameter	Verbleibende Parameter
Größe in Bit m Erwartete Anzahl an Elementen n	$f \approx 0,6185 \frac{m}{n}$ $k = \ln(2) * \frac{m}{n}$ (optimale Anzahl)
False Positive Rate f Erwartete Anzahl an Elementen n	$m = \lceil -2.081 * n \ln(f) \rceil$ $bpe = 2.081 * \ln\left(\frac{1}{f}\right)$ (Bits pro Element)
(bei optimaler Anzahl an Hashfunktionen k)	
Größe in Bit m Erwartete Anzahl an Elementen n False Positive Rate f	$f \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$
Writes pro Sekunde wps Expirationsdauer E	$n = wps * E$

Tabelle 6 Die Zusammenhänge zwischen den Parametern des Bloomfilters.

Eine Datenstruktur mit dem theoretisch optimalen Platzbedarf lässt sich durch sogenanntes *Perfect Hashing* erzeugen [163]. Dazu wird eine Hashfunktion h_p gesucht, die jedes einzutragende Element aus $\{1, \dots, u\}$ eindeutig auf einen Wert aus $\{1, \dots, n\}$ abbildet. Eine zweite



Hashfunktion dient dazu, ein Element unter Inkaufnahme von False-Positives zu komprimieren, d.h. h_k bildet von $\{1, \dots, u\}$ nach $\{1, \dots, 2^j - 1\}$ vollständig gleichverteilt ab. Um ein Element x abzuspeichern, wird seine Position durch $h_p(x)$ bestimmt und an der betreffenden Stelle seine j -Bit-Repräsentation $h_k(x)$ gespeichert. Die untere Schranke für den Platzbedarf wird dadurch genau erreicht, da die Größe $m = n * j$ beträgt und die False-Positive Rate $f = 1/2^j$ (da h_k gleichverteilt abbildet). Eine einfache Umformung ergibt die untere Schranke $m = \log_2(1/f)$. Tatsächlich ist dieses Verfahren jedoch nahezu unbrauchbar, da es sehr aufwändig ist, ein h_p zu finden, das Elemente eindeutig abbildet und sogar praktisch unmöglich, ein h_k zu ermitteln, das vollständig gleichverteilt abbildet. Zudem muss beim Einfügen (also bei jeder Änderung eines Objektes) h_p vollständig neu berechnet werden. Perfect-Hashing ist damit zwar im Platzbedarf optimal, aber im Kontext unserer praktischen Anwendung ungeeignet.

In der Literatur wurden einige platzeffizientere Varianten von Bloomfiltern entwickelt. Eine interessante Technik wurde von Mitzenmacher [144] vorgestellt. Die Idee ist, den Bloomfilter für die Übertragung kompakter zu gestalten, indem der Bloomfilter auf die Länge $const * 2^k n$ vergrößert wird. Dadurch, dass nach Einfügen von Elementen das Bit-Array mit besetzten Bits sehr dünn besiedelt ist, nimmt die Entropie des Bit-Arrays ab und kann sehr gut mit gängigen Kompressionsverfahren verkleinert werden. Der Nachteil liegt in dem wesentlich erhöhten Speicherbedarf, der für den unkomprimierten Bloomfilter notwendig ist, auf dem das Einfügen und Abfragen vorgenommen wird. Eine andere platzeffizientere Bloomfiltervariante studieren Putze et al. [166]. Ihr Vorschlag sieht vor, nur eine Hashfunktion zu benutzen ($k = 1$). Dadurch ist erneut das Bit-Array mit gesetzten Bits dünn besiedelt. Da der Abstand zwischen gesetzten Bits einer geometrischen Verteilung folgt, kann durch sogenanntes Golomb Coding eine fast optimale Kompression erreicht werden. Der Platzbedarf pro Bit weicht dann nur um durchschnittlich ein halbes Bit vom theoretischen Optimum ab [166]. Da die Datenstruktur jedoch entpackt werden muss, um sie zu nutzen, schlagen Putze et al. vor, sie in Unterabschnitte zu zerlegen und jeden Unterabschnitt einzeln zu komprimieren und dekomprimieren. Der Platzgewinn wird auf diese Weise durch erhöhten Rechenaufwand erkauft. Beide Varianten sind damit für uns uninteressant, da sie erhöhten Speicherbedarf in dekomprimierter Form besitzen oder sogar einen Overhead in der Effizienz der *Contains* Operation. Da der klassische Bloomfilter ohnehin fast platzoptimal ist, scheint der Overhead einer komprimierenden Implementierung für die meisten Applikationen unnötig.

Dadurch, dass Bloomfilter aus einem einfachen Bit-Array bestehen, sind einige Operationen sehr einfach:

- **Vereinigung.** Wenn zwei Bloomfilter kompatibel sind, d.h. die gleichen Hashfunktionen benutzen und die gleiche Größe besitzen, kann die Vereinigung der Mengen, die sie repräsentieren, durch eine *OR* Verknüpfung der beiden Bit-Arrays erreicht werden. Der resultierende Bloomfilter entspricht dem Bloomfilter, der entstanden wäre, wenn alle Elemente direkt in einen Bloomfilter eingefügt worden wären [163].

- Schnittmenge.** Die Schnittmenge zweier kompatibler Bloomfilter kann analog zur Vereinigung durch die Verwendung von *AND* statt *OR* ermittelt werden ([163]). Die Schnittmenge ist präzise in dem Sinne, dass keine False Negatives entstehen. D.h. alle Elemente der Schnittmenge sind auch der Teil des vereinten Bloomfilters. Für die False-Positives gilt, dass ihre Anzahl höchstens so hoch ist wie die höhere False-Positive Anzahl der beiden vereinten Bloomfilter. Es werden keine neuen Bits auf 1 gesetzt, d.h. ein False-Positive muss bereits in beiden der vereinten Bloomfilter ein False-Positive gewesen sein. Die False-Positive Rate ist im Allgemeinen jedoch größer als die eines Bloomfilters, der direkt aus der Schnittmenge erzeugt wird. Der Grund dafür sind Elemente, die nicht zur Schnittmenge gehören, aber überlappende Hashwerte haben. Die Überlappungen werden bei der AND Verknüpfung übernommen und führen zu unnötigerweise gesetzten Bits und damit zu einer höheren False-Positive Rate [167].
- Halbierung.** Die Größe eines Bloomfilters kann sehr leicht halbiert werden, indem die obere und die untere Hälfte des Bit-Arrays mit *OR* verknüpft werden (*Folding*). Bei einem anschließenden Aufruf von *Contains* kann das oberste Bit jedes Hashwertes maskiert werden.

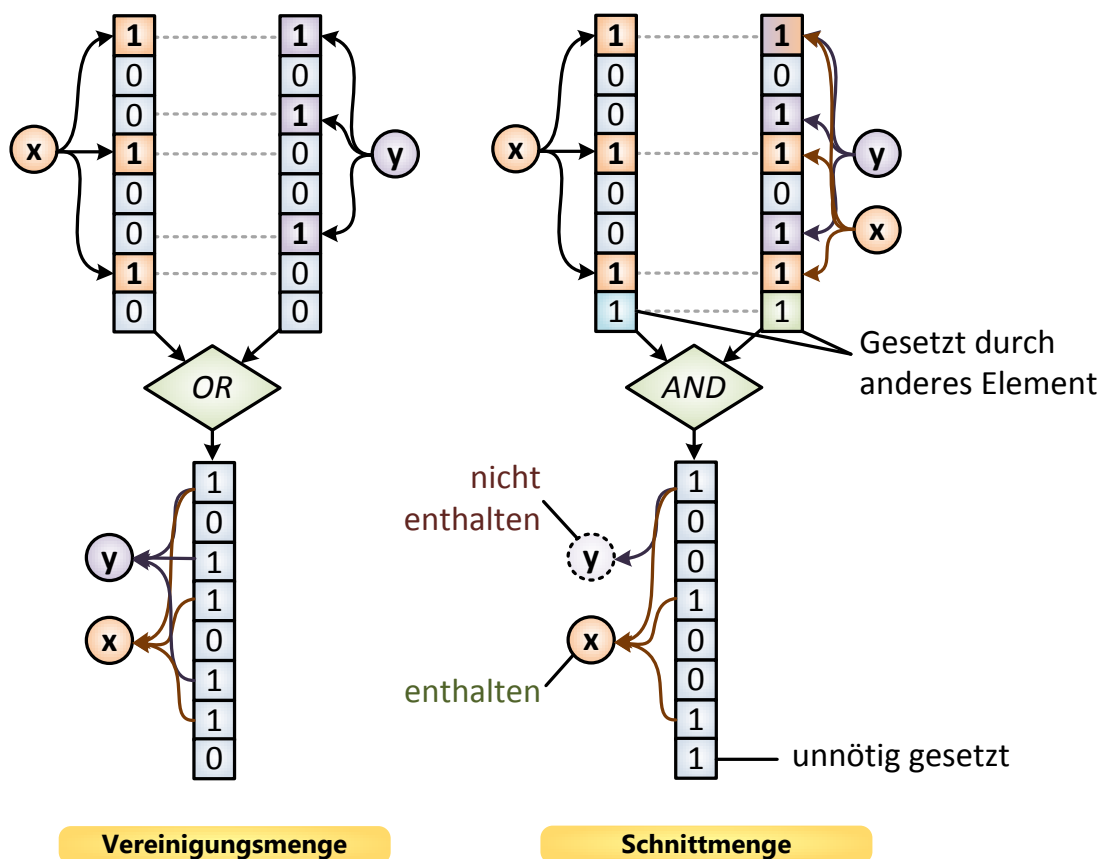


Abbildung 49 Implementierung von Vereinigungs- und Schnittmengen durch bitweise Operatoren.

Für ORESTES wäre die einfache OR-basierte Vereinbarkeit eine Möglichkeit, die von einzelnen ORESTES-Servern gesammelten Einträge zu vereinen. Da diese Vereinigung jedoch sehr



hochfrequent nötig ist – bei jeder Abfrage durch einen Client – werden wir den Ansatz bevorzugen, einen vereinten Bloomfilter zu pflegen.

Durch die bisherigen Betrachtungen ist offensichtlich, dass der Bloomfilter eine optimale *StaleObjects* Datenstruktur für die Clientseite ist. Er erfüllt alle gestellten Anforderungen:

- **Effizienz.** Der Bloomfilter unterstützt die Operationen *Contains* mit einer Worst-Case Laufzeit von $O(1)$.
- **Kompakte Darstellung.** Wie der Beweis gezeigt hat, kommt der Bloomfilter dem theoretischen Optimum bis auf 44% Overhead nahe.

3.4 Anwendungen

Die kompakte Darstellung von Mengen mit hoher Effizienz bei *Contains* Aufrufen ist für zahlreiche Anwendungen interessant. Generell gibt es zwei Muster, in denen Bloomfilter immer wieder auftauchen [163]:

1. Als kompakte Darstellung von Ressourcen eines Netzwerkknotens.
2. Als Filter zur Vermeidung teurer Operationen wie Festplatten I/O.

Wir führen eine kurze Analyse beider Muster durch und wählen als Use-Cases stellvertretend die thematisch mit ORESTES verwandten Ansätze *Summary Cache* (Cache Digests) [138] und *Google BigTable* [48].

3.4.1 Summary Cache

Mit Summary Cache beschrieben Fan et al. [138] 1998 ein Verfahren für kooperierende Web-Caches, das *Neighbour Queries* effizienter machen sollte. Wie In Kapitel 1 beschrieben, können Web-Caches zu einem Verbund zusammengefügt werden durch Protokolle wie das Inter-Cache Protocol (ICP) [44]. Es erlaubt einem Cache bei einem lokalen Cache-Miss seine Nachbarn nach der angefragten URL zu befragen (Neighbour Query). Da es sich bei dieser Anfrage um eine klassische *Scatter-and-Gather* Strategie handelt, ist sie anfällig für Latenz-Probleme. Insbesondere muss der anfragende Web-Cache bei der Antwort stets auf den langsamsten seiner Nachbarn warten.

Um dieses Problem zu umgehen, schlagen Fan et al. vor, ICP mit Bloomfiltern zu verbinden. Die Idee ist, die Menge aller gecachten URLs durch einen Bloomfilters (Summary) darzustellen und diesen periodisch zwischen den verbundenen Web-Caches auszutauschen. Ein Web-Cache, der eine Anfrage für eine URL erhält, muss bei einem Cache-Miss nun lediglich die lokal im Arbeitsspeicher vorliegenden Bloomfilter seiner Nachbarn abfragen. Nur wenn einer dieser *Contains* Aufrufe erfolgreich ist, wird der entsprechende Nachbar gefragt. Dies bringt zwei Vorteile:

- Bei einem Cache-Miss wird das Netzwerk nicht durch einen Broadcast geflutet.
- Unnötige Befragungen von Nachbarn werden in $1 - f$ aller Fälle vermieden.

Das Prinzip von Summary Cache ist in Abbildung 50 gezeigt. Bei der Konstruktion der Bloomfilter fiel Fan et al. auf, dass Einträge aus Bloomfiltern wieder entfernt werden müssen, um die Expirationszeiten von gecachten Objekten zu berücksichtigen. Sie entwickelten deshalb die Idee des *Counting Bloomfilters*, den wir im nächsten Abschnitt beschreiben. Dieser erlaubt auf Kosten eines Platzoverheads, eingetragene Objekte mit *Delete* in $O(1)$ zu entfernen.

Das Zusammenspiel von Web-Caches einer *Peer Group*, d.h. eines Verbunds, mit Summary Cache ist in Abbildung 50 gezeigt:

1. Eine Client stellt über HTTP eine GET Anfrage für eine URL.
2. Kommt es bei dem befragten Web-Cache zu einem Cache-Miss, prüft er die gespeicherten Bloomfilter seiner Peers.
3. Enthält einer der Bloomfilter die URL, wird die Anfrage an den entsprechenden Web-Cache weitergeleitet, der das gecachte Objekt mit hoher Wahrscheinlichkeit besitzt.

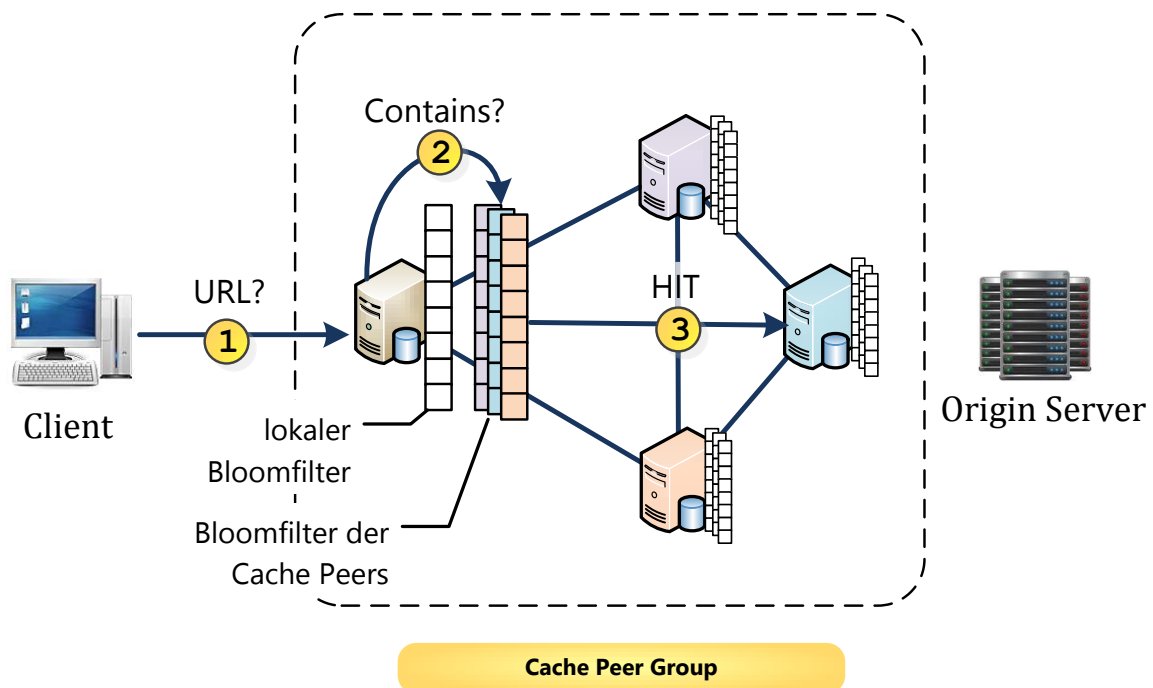


Abbildung 50 Die Funktionsweise von Summary Cache.

Der Squid Cache hat das Verfahren von Summary Cache unter dem Namen Cache Digests implementiert und eine Spezifikation der ICP-Protokollerweiterung veröffentlicht [46]. Die Implementierung ist seit 1998 erfolgreich im Einsatz. Für ORESTES ist diese Implementierung zwar auf Ebene des Web-Cachings nutzbar, für die Implementierung der *StaleObjects* Menge jedoch zu rudimentär. Dies hat mehrere Gründe:

- Die Cache-Digest Implementierung in Squid verzichtet auf Counting Bloomfilter und konstruiert in regelmäßigen Abständen (eine Stunde) den kompletten Bloomfilter neu, um veraltete Objekte zu entfernen.



- Die Implementierung ist in C++ geschrieben (ORESTES in Java) [168].
- Die Implementierung ist sehr eingeschränkt in ihrer Konfigurierbarkeit: die Anzahl der Hashfunktion ist statisch auf $k = 4$ festgelegt, was für die meisten Fälle weit vom tatsächlichen Optimum entfernt ist [168]. Außerdem kann nur eine Hashfunktion verwendet werden: MD5. MD5 bietet als kryptographische Hashfunktion zwar eine sehr gute Verteilung, ist aber sehr aufwendig zu implementieren. Für ORESTES ist die Konfigurierbarkeit der Hashfunktion sehr wichtig, da potentiell beliebig viele Programmiersprachen und Plattformen unterstützt werden, in denen die Hashfunktion zur Benutzung der Bloomfilter zur Verfügung stehen muss. So gibt es für JavaScript beispielsweise MD5 Implementationen, doch sind diese in ihrer Geschwindigkeit stark eingeschränkt und nicht vergleichbar mit einer Implementierung in C++, Java oder im Betriebssystem.

3.4.2 BigTable

Das BigTable Paper von Google [48] gilt als eine der einflussreichsten Datenbankveröffentlichungen des letzten Jahrzehnts. Es beschreibt ein spaltenorientiertes, skalierbares, verteiltes Datenbanksystem, das bei Google unter anderem den gesamten Suchindex speichert. Auf das Datenmodell soll hier jedoch nicht näher eingegangen werden, sondern stattdessen die Speicherstruktur beleuchtet werden, deren Effizienz auf Bloomfiltern basiert. Da das System auf die Speicherung von Daten in der Größenordnung von Petabytes ausgerichtet ist, kommen für das Speichermedium aus Kostengründen nur klassische Festplatten in Frage, die über das verteilte *Google File System* angesprochen werden. Die Index- und Speicherstruktur, die BigTable verwendet, ist ein sogenannter *Log-Structured Merge-Tree (LSM)* [169], der bereits lange vor der Veröffentlichung von BigTable bekannt war.

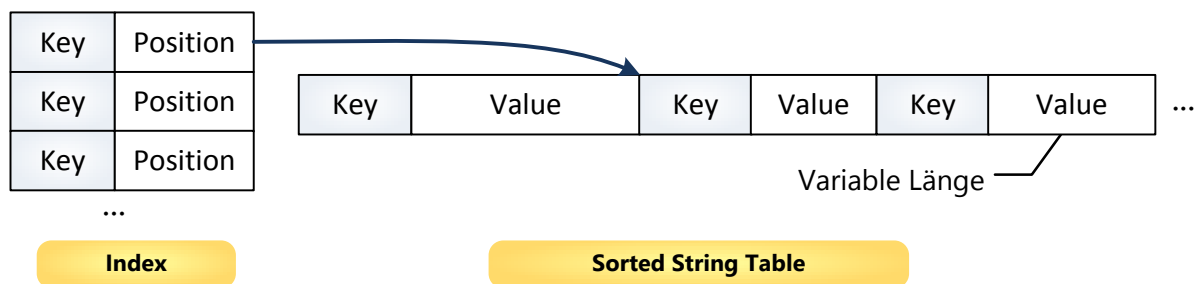


Abbildung 51 Die Sorted-String Table Datenstruktur.

Die zentrale Datenstruktur des LSM ist die Sorted-String Table (*SSTable*). Eine *SSTable* enthält eine sortierte Liste von Key-Value Paaren (siehe Abbildung 51). In BigTable sind die Keys eine Kombination aus Primärschlüssel, Spaltenname und Timestamp. Der Value kann ein beliebiger Wert sein. Durch die Sortierung lässt sich eine *SSTable*-Datei in $O(\log(n))$ mit binärer Suche als Index benutzen. Ist dies ungenügend, kann die *SSTable* durch einen simplen Index erweitert werden, der den Key auf den Speicherort abbildet. Die *SSTable* und ihr Index werden in Dateien auf dem Sekundärspeicher abgelegt. Durch die Sortierung ist eine *SSTable* als Datei gespeicherte *SSTable* effektiv unveränderlich – ein Schreibzugriff würde

ein Überschreiben von nahezu der gesamten Datei erfordern. Für Schreibzugriffe wird die SSTable deshalb im Hauptspeicher gehalten, wo sie den Namen *Memtable* trägt. Um also das schnelle Lesen aus SSTables mit einer hohen Schreibgeschwindigkeit zu vereinen, werden unveränderliche SSTables auf Festplatten gespeichert und zum Lesen verwendet, während eine Memtable die Schreibzugriffe bearbeitet. Für die Forderung nach Durability werden Writes zudem in einem Log protokolliert, das durch Group Commits [55] effizient auf die Festplatte geschrieben werden kann. Leseanfragen werden beantwortet, indem zuerst die Memtable geprüft wird und dann alle auf der Festplatte gespeicherten SSTables. In periodischen Abständen wird die Memtable als SSTable auf die Festplatte geschrieben. Um die Anzahl an SSTables zu beschränken, werden die auf der Festplatte gespeicherten SSTables periodisch zusammengefügt, damit die Anzahl der SSTables logarithmisch beschränkt in der Größe der Daten bleibt.

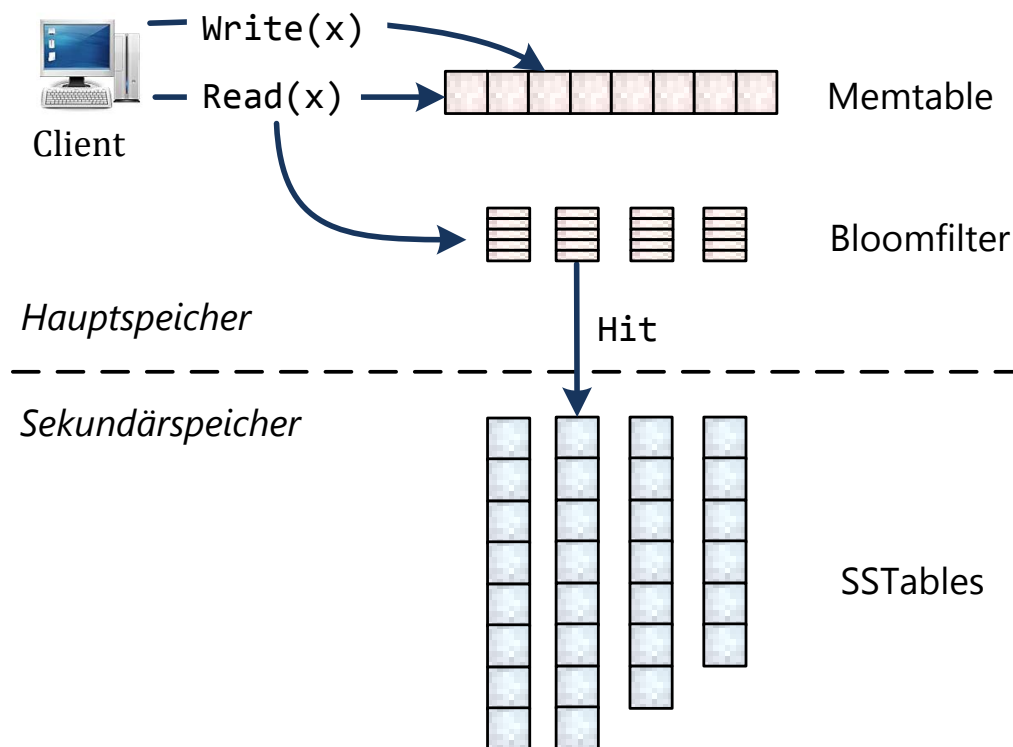


Abbildung 52 Die Verwendung von Bloomfiltern zur Vermeidung eines Festplattenzugriffs.

Das Problem des Ansatzes ist, dass für jeden Read, der nicht direkt aus der Memtable beantwortet werden kann, mehrere SSTables abgefragt werden müssen. Dies resultiert jeweils in einem wahlfreien Festplattenzugriff, der sehr ineffizient ist und von der Rotationsgeschwindigkeit des Mediums (*Seek Time*) abhängt. Auch die Indexe der SSTable sind bei weitem zu groß, um sie im Hauptspeicher zu halten. Als Lösung kommen Bloomfilter zum Einsatz. Für jede SSTable wird ein Bloomfilter angelegt, der die gespeicherten Keys enthält. Die Bloomfilter sind klein genug, um im Hauptspeicher gehalten zu werden. Bei jeder Leseanfrage werden so zuerst die Bloomfilter der SSTables effizient im Hauptspeicher geprüft und nur bei einem Treffer der teure Festplattenzugriff ausgeführt.



Das Design von BigTable hat zahlreiche NoSQL Datenbanksysteme inspiriert. Das BigTable Modell wurde in nahezu identischer Weise in HBase [51] und Cassandra [63] implementiert, die ebenfalls Bloomfilter für die SSTables nutzen. Beide Implementierungen sind als Open-Source frei verfügbar. Die Implementierungen haben jedoch ebenfalls mehrere Nachteile, die ihre Verwendung in ORESTES ausschließen:

- Der HBase Bloomfilter und der Cassandra Bloomfilter benutzt eine von Mitzenmacher et al. [170] eingeführte Optimierung, die statt k Hashfunktionen k Linearkombinationen zweier Hashfunktion bildet. Zwar lässt sich für eine perfekte Hashfunktion die asymptotisch gleiche False-Positive Rate beweisen, in der Praxis leidet die False-Positive Rate jedoch sehr unter dieser Optimierung [171].
- Der HBase Bloomfilter unterstützt zudem nur zwei Hashfunktionen, der Cassandra Bloomfilter nur eine [172], [173].
- Die HBase Implementierung ist beschränkt auf einen einfachen Bloomfilter, nur Cassandra bietet auch einen Counting Bloomfilter. Dieser ist jedoch statisch konfiguriert, d.h. er erlaubt insbesondere keine Konfiguration der Counter-Größe. Bei beiden fehlen Bloomfilteroperationen, die in anderen Kontexten sinnvoll sind, z.B. die Bildung der Vereinigungs- und Schnittmenge.
- Beide Implementierungen führen statt mathematisch korrektem *Rejection Sampling* eine Reduktion der Hashwerte durch eine Modulo Berechnung durch. Auf diesen Fehler werden wir im Abschnitt über Hashing genauer eingehen.

Die Verwendung dieser Bloomfilter-Implementierungen würde also im Prinzip zu einer kompletten Neuimplementierung führen. Deshalb verzichten wir auf ihre Verwendung und schaffen stattdessen eine allgemeingültige Implementierung.

3.5 Varianten

Wir wollen nun verschiedene Erweiterungen des Bloomfilters untersuchen, um eine geeignete Bloomfiltervariante für den Server zu finden. Der Server muss einerseits Elemente mit *Delete* aus dem Bloomfilter entfernen können und andererseits schnell einen herkömmlichen Bloomfilter für den Client generieren können.

3.5.1 Counting Bloom Filter

Der Counting Bloomfilter, der von Fan et al. [138] für Summary Cache entwickelt wurde, ist eine naheliegende Erweiterung der Bloomfilters. Statt beim Einfügen die Positionen des Bit-Arrays auf 1 zu setzen, wird ein Counter erhöht. Dieser kann dekrementiert werden, um das Löschen zu unterstützen. Abbildung 54 zeigt das Grundprinzip und Abbildung 53 den Pseudocode für die *Insert*, *Delete* und *Contains* Methoden. Jeder Counter hat eine Größe von c Bits, so dass das resultierende Bit-Array $c * m = \lceil \log_2(\maxCount) \rceil * m$ Bits benötigt, wobei \maxCount die größte Zahl bezeichnet, die ein Counter darstellen kann. Die Herausforderung bei der Konfiguration eines Counting Bloomfilter liegt in der Wahl einer geeigneten Countergröße.

```

def insert(obj):
    for each position in hashes(obj):
        #Alle Positionen auf 1 setzen
        counters[position] += 1

def delete(obj):
    if contains(obj) == false:
        error("Element nicht enthalten")
    else:
        for each position in hashes(obj):
            #Alle Positionen auf 1 setzen
            counters[position] += 1

def contains(obj):
    for each position in hashes(obj):
        if counters[position] == 0:
            //Falls auch nur eine der Positionen 0 ist,
            //ist das Element nicht enthalten
            return false;
    return true

```

Abbildung 53 Die *Insert*, *Contains* und *Delete* Methode eines Counting Bloomfilters.

Fan et al. [138] konnten zeigen, dass für fast alle Applikationen eine Countergröße von $c = 4$, d.h. $maxCount = 15$ ausreicht. Ihre Analyse geht davon aus, dass die Einfügeoperationen einer Gleichverteilung über allen Objekten genügen. Wenn $c(i)$ den Wert des i ten Counters bezeichnet, ist die Wahrscheinlichkeit, dass $c(i)$ insgesamt j mal inkrementiert wird eine binomialverteilte Zufallsvariable [163]:

$$P(c(i) = j) = \binom{n * k}{j} * \left(\frac{1}{m}\right)^j * \left(1 - \frac{1}{m}\right)^{n*k-j}$$

Dabei ist $n * k$ die Anzahl sämtlicher Countererhöhungen und $\frac{1}{m}$ die Wahrscheinlichkeit, dass ein spezifischer Counter erhöht wird. Der interessanteste Wert für k ist sein Optimum, also $k = \ln(2) * \frac{m}{n}$. Damit ist die Wahrscheinlichkeit, dass einer der Counter den Wert j übersteigt:

$$P(c(i) \geq j) \leq \binom{n * \ln(2) * \frac{m}{n}}{j} * \left(\frac{1}{m}\right)^j$$

Die Wahrscheinlichkeit, dass ein konkreter Counter j übersteigt, ist größer als die Wahrscheinlichkeit, dass der maximale Counter j übersteigt:

$$P(\operatorname{argmax}_j c(i) \geq j) \leq P(c(i) \geq j)$$

Werden 4 Bits verwendet, tritt ein Overflow auf, sobald einer der Counter $j = 16$ erreicht. Die Wahrscheinlichkeit hierfür ist durch obigen Ausdruck beschränkt:

$$P(\operatorname{argmax}_j c(i) \geq j) \leq 1,37 * 10^{-15} * m$$



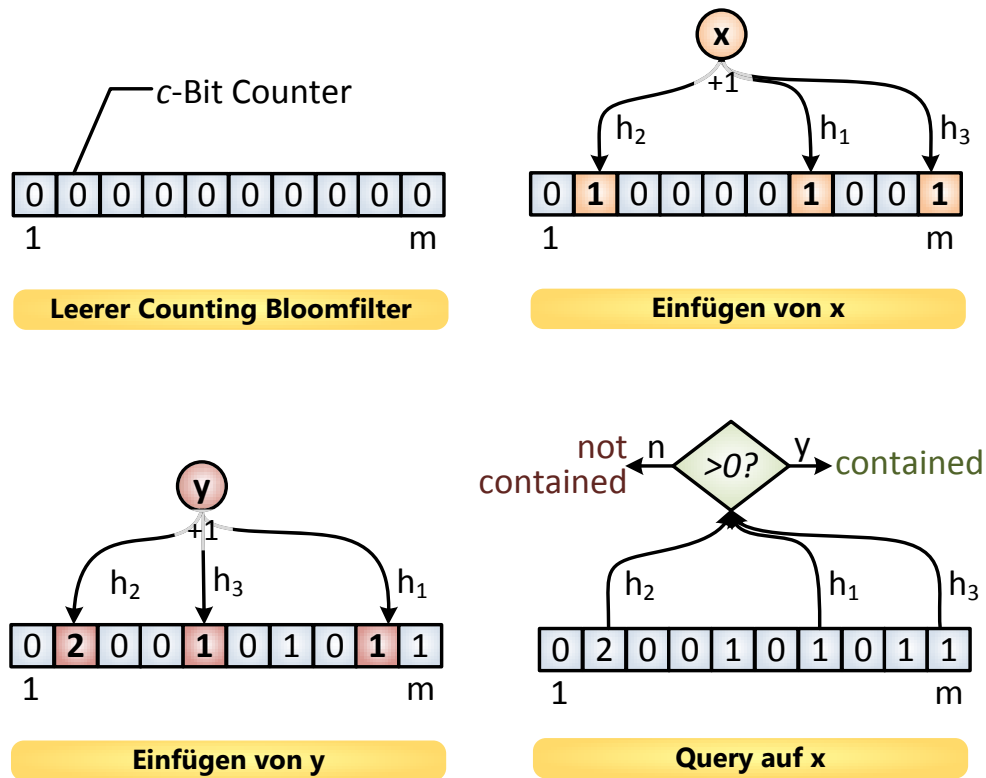


Abbildung 54 Beispiel für die Funktionsweise eines Counting Bloomfilters.

Die Wahrscheinlichkeit für einen Overflow bei 4-Bit Countern und einem zufälligen Zugriffsmuster ist also außerordentlich gering. Die verbreitetste Strategie mit einem Overflow eines Counters umzugehen besteht darin, den Wert unverändert auf dem Maximum zu belassen [163]. Dies birgt jedoch die Gefahr von False-Negatives, sobald einer der vormals übergelaufenen Counter (zu schnell) 0 erreicht. Die Größe der Counter sollte deshalb so gewählt werden, dass ein Overflow sehr unwahrscheinlich ist. Für ORESTES ist dies eine Herausforderung, da Objekte mehrfach in den Counting Bloomfilter eingefügt werden. Dabei werden stets dieselben Counter erhöht. Es gibt zwei Strategien, damit umzugehen:

- Man verhindert, dass Elemente mehrfach eingefügt werden. Dazu kann nicht der Counting Bloomfilter verwendet werden, da False-Positives auftreten können. Es ist also notwendig, ein separates Set zu pflegen, das darauf befragt wird, ob ein Element bereits enthalten ist.
- Die Countergröße wird so konservativ gewählt, dass selbst beim Einfügen von identischen Objekten die Counter nicht überlaufen. Bei einer Countergröße von 20 Bit müsste ein Element beispielsweise bereits $2^{20} = 1.048.576$ eingefügt werden, um einen Überlauf zu verursachen.

In der Implementierung für ORESTES haben wir uns für den zweiten Ansatz entschieden, da der Speicherbedarf für den Bloomfilter selbst bei großen Countern kein Engpass werden kann. Der Counting Bloomfilter ist bei einer Countergröße von $c = 32$ Bit nur 32-mal größer als der tatsächliche Bloomfilter, der an die Clients übertragen wird. Da dieser aus praktischen Gründen in seiner Größe beschränkt ist auf Werte, die deutlich unter 1 Mbyte liegen,

ist der Speicherbedarf des c -mal größeren Counting Bloomfilters sehr moderat. Damit der Counting Bloomfilter eine ideale Wahl für die serverseitige Implementierung der *StaleObjects* Menge ist, muss auch das Abrufen des einfachen Bloomfilters sehr einfach sein. Hierfür nutzen wir eine einfache Erweiterung des Counting Bloomfilters, die wir *Materialized Counting Bloomfilter* nennen und die vermutlich in dieser Weise bereits von Fan et al. [138] gedacht war.

3.5.2 Materialized Counting Bloomfilter

Mit dem Materialized Counting Bloomfilter erweitern wir den Bloomfilter auf sehr einfache Weise: neben den Countern wird in einem Bit-Array für jeden Counter materialisiert, ob der Wert des jeweiligen Counters größer oder gleich 0 ist. Die entsprechenden *Delete* und *Insert* werden dazu so erweitert, dass sie den materialisierten Bloomfilter aktualisieren, wenn ein Counter 0 erreicht oder überschreitet. *Contains* kann sehr effizient direkt gegen den materialisierten Bloomfilter ausgeführt werden, ohne die Counter zu prüfen. Der Counting Bloomfilter enthält jederzeit einen vollwertigen Bloomfilter. Dieser kann deshalb auf Anfrage sehr schnell ausgeliefert werden, da er nicht ad-hoc aus den Countern generiert werden muss. Unsere Counting Bloomfilter Implementierung setzt diese Materialisierung um.

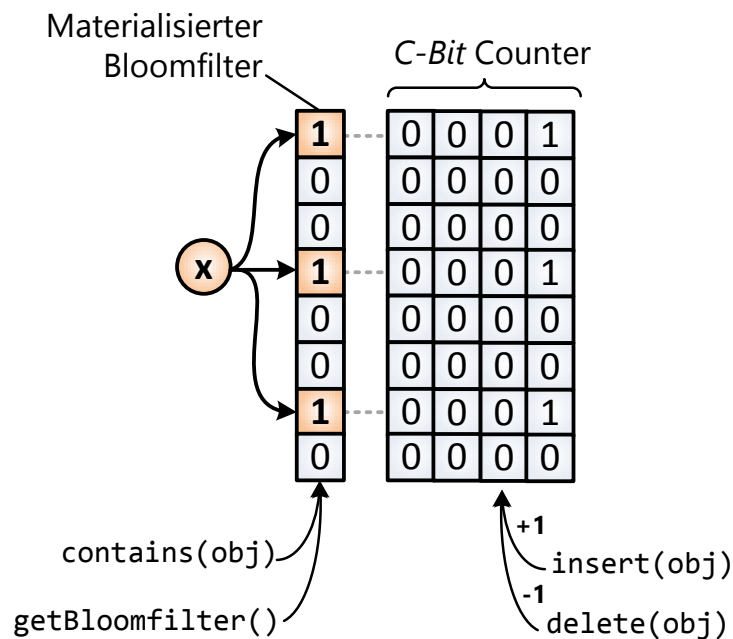


Abbildung 55 Der Materialized Counting Bloomfilter.

Wie in Abbildung 55 dargestellt, erkaufte der Materialized Counting Bloomfilter also durch einen Speicheroverhead von m Bit eine Beschleunigung von *Contains* um den Faktor c , da kein Element der Bit-Arrays geprüft werden muss. Dieser Vorteil ist jedoch für ORESTES wenig bedeutend, da der Server keinen Grund hat, den Bloomfilter abzufragen. Entscheidend für den ORESTES-Server ist vielmehr, dass wir die Forderung zur schnellen Generierung optimal erfüllen können: eine Generierung findet nicht statt, der Bloomfilter liegt bei der An-



frage bereits vor. Darüber hinaus ist der geringe Speicheroverhead für den Server unerheblich.

3.5.3 Bitwise Bloomfilter

Der Bitwise Bloomfilter [142] versucht das Problem überlaufender Counter im Counting Bloomfilter zu beheben. Dazu kombiniert der Bitwise Bloomfilter l Counting Bloomfilter. Jeder dieser Counting Bloomfilter CBF_i hat eine eigene Größe m_i , Countergröße c_i und Hashfunktionen k_i [174] (siehe Abbildung 56). Um ein Element in den Bitwise Bloomfilter einzufügen, wird zuerst versucht, es in den ersten Bloomfilter CBF_0 einzufügen. Wenn dabei ein Counter-Overflow auftritt, wird das Element entnommen und stattdessen in CBF_1 eingefügt. Dort wird ebenso verfahren. Durch das Hinzufügen von neuen Counting Bloomfiltern ist die Anzahl der Counter unbeschränkt und kann dynamisch wachsen. Der Nachteil liegt darin, dass für eine Abfrage stets alle l Counting Bloomfilter befragt werden müssen. Der Wert eines Counter ergibt sich dadurch als:

$$count(i) = \sum_{i=0}^l CBF_i \cdot c_i$$

Für ORESTES ist der Bitwise Bloomfilter kein Gewinn, da er serverseitigen Speicherbedarf auf Kosten höheren Rechenaufwandes für das Einfügen und Löschen umsetzt. Außerdem können in dem Bitwise Bloomfilter False Negatives auftreten [142]. Dies ist immer dann der Fall, wenn beim Löschen in CBF_i ein False Positive auftritt und das Element deshalb fälschlich aus CBF_i anstelle des richtigen Counting Bloomfilters gelöscht wird.

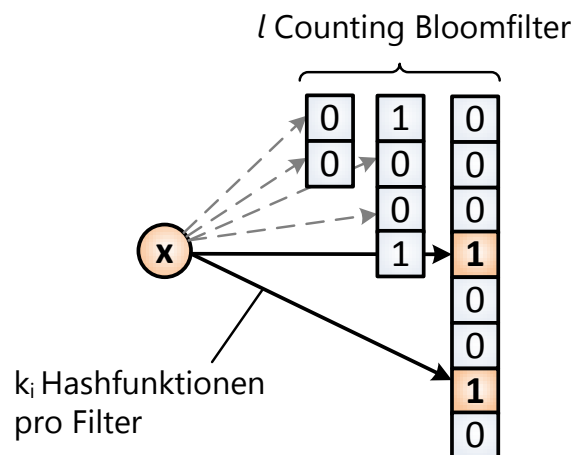


Abbildung 56 Der Bitwise Bloomfilter.

3.5.4 Spectral Bloomfilter

Spectral Bloomfilter [143] versuchen den Counting Bloomfilter so zu erweitern, dass er in der Lage ist, Schätzungen für die Multiplizitäten von eingefügten Elementen zurückzugeben, d.h. Multimengen zu repräsentieren. Der Bloomfilter ist spektral in dem Sinne, dass er das

Filtern von Elementen anhand einer Multiplizität eines gewünschten Bereichs (Spektrums) erlaubt. Cohen et al. [143] schlagen für Multiplizitätsabschätzung drei Algorithmen vor, sowie eine recht komplizierte Datenstruktur (*String-Array Index*) als Alternative zu dem klassischen Counting Bloomfilter mit festen Countergrößen und einem etwas geringeren Speicherbedarf. Die Schätzung von Multiplizitäten ist für viele Anwendungen sinnvoll. Ein Beispiel ist die performante Durchführung von Aggregations-Querys, beispielsweise [143]:

```
SELECT count(a1) FROM R WHERE a1 = v
```

Ein Spectral Bloomfilter kann in diesem Fall als ein Aggregatsindex über dem Attribut *a1* dienen. Die durch den Spectral Bloomfilter geschätzte Multiplizität des Attributs liefert dann eine effizient ermittelbare Schätzung des tatsächlichen Werts. Die Algorithmen, die Cohen et al. [143] vorschlagen, basieren konzeptionell auf der Datenstruktur des Counting Bloomfilters:

- **Minimum Selection.** Der Minimum Selection Algorithmus macht eine Schätzung \hat{x} der Multiplizität eines Elements durch seinen niedrigsten Counter im Counting Bloomfilter. Cohen et al. zeigen, dass die tatsächliche Multiplizität x stets kleiner als die Schätzung \hat{x} ist. Die Wahrscheinlichkeit einer Fehlschätzung ist genau die Wahrscheinlichkeit eines False-Positives, d.h. $P(\hat{x} \neq x) = f$. Das liegt daran, dass für eine Fehlschätzung genau jeder Counter eines Elements einmal Opfer einer Kollision werden muss, was exakt dem Szenario eines False-Positives für dieses Element entspricht. Die Operationen *Insert*, *Delete* und *Contains* sind identisch mit dem klassischen Counting Bloomfilter.
- **Minimum Increase.** Die Idee des Minimum Increase Algorithmus ist es, beim Einfügen stets nur den niedrigsten Counter zu erhöhen, da dieser die präziseste Schätzung der Multiplizität widerspiegelt. Dies erhöht zwar die Präzision der Schätzung \hat{x} , hat aber fatale Auswirkungen auf das Löschen von Elementen. Da beim Löschen stets alle Counter verringert werden müssen, können False Negatives entstehen. Der Minimum Increase Algorithmus sollte deshalb nicht verwendet werden, wenn Elemente wieder entfernt werden müssen.
- **Recurring Minimum.** Der Recurring Minimum Algorithmus basiert auf der empirischen Beobachtung, dass nur 20% der Elemente eines Counting Bloomfilters einen eindeutigen minimalen Counter besitzen [143]. Alle anderen Elemente besitzen mehrere identische minimale Counter (ein *Recurring Minimum*). Besonders False-Positives neigen dazu, nur ein Minimum zu besitzen. Der Recurring Minimum Algorithmus nutzt diese Beobachtung aus, indem zwei Counting Bloomfilter CBF_1 und CBF_2 gepflegt werden. Die Elemente mit einem eindeutigen minimalen Counter werden in CBF_2 gehalten, während CBF_1 wie zuvor alle Elemente enthält. Beim Einfügen eines Elements in CBF_1 wird geprüft, ob das Element einen eindeutigen minimalen Counter besitzt. Ist dies der Fall, wird das Element zusätzlich in CBF_2 eingefügt. Ist es dort bereits vorhanden, werden seine Counter lediglich erhöht. Ist es neu, wird es mit ei-



ner Multiplizität eingefügt, die dem minimalen Counter aus CBF_1 entspricht. Beim Abfragen der Multiplizität eines Elements wird zuerst in CBF_1 nachgeschlagen. Besitzt das Element dort ein Recurring Minimum, wird dieser Wert zurückgegeben. Andernfalls wird in CBF_2 nachgeschlagen. Ist das Element dort nicht vorhanden, d.h. sein Counter 0, wird das eindeutige Minimum aus CBF_1 zurückgegeben. Andernfalls wird der minimale Wert aus CBF_2 zurückgegeben. Da der Recurring Minimum Algorithmus Minimum Selection für CBF_1 benutzt, sind seine Ergebnisse mindestens so gut, wie die des reinen Minimum Selection Algorithmus.

Für ORESTES ist die Abschätzung von Multiplizitäten nicht zwingend erforderlich. Da der Minimum Selection Algorithmus jedoch keine Änderung an dem Counting Bloomfilter erfordert, sondern lediglich eine neue Operation einführt, könnte der Algorithmus genutzt werden, um zu analysieren, welche Elemente besonders hochfrequent geändert werden. Diese Elemente tauchen mit einer hohen Multiplizität im Counting Bloomfilter auf. Auf diese Weise könnten automatisiert Hotspot Objekte identifiziert werden. Für Hot-Spot Objekte sind sinnvolle Sonderbehandlungen denkbar, z.B. die Erfordernis, bei ihrer Änderung ein Lock anzufordern oder ein Herabsetzen ihrer Expirationsdauer. Die Implementierung der sehr komplexen String-Array Index Datenstruktur als Counting Bloomfilter Ersatz macht wenig Sinn für ORESTES, da der gewonnene Speicherplatz ohnehin kein Engpass ist.

3.5.5 Bloomier Filter

Chazelle et al. [140] schlagen den Bloomier Filter vor, eine Erweiterung des Bloomfilters, um mit ihm eine probabilistische Map (auch assoziatives Array, Hash oder Dictionary genannt) umzusetzen. D.h. der Bloomfilter besitzt statt einer *Contains* Methode eine *Get* Methode, die jedes Element (Key) aus dem Universum U auf seinen verknüpften Wert (Value) aus V abbildet oder *False* zurückgibt, wenn das Element nicht im Bloomfilter vorhanden ist:

$$GET: U \rightarrow V \cup \{False\}$$

Dabei besteht eine False-Positive Wahrscheinlichkeit, dass ein Key auf einen Value abgebildet wird, obwohl das Element nicht in der Map ist. Chazelle et al. schlagen eine sehr einfache Implementierung des Bloomier Filters vor und erklären es am Beispiel von $V = \{0,1\}$ [140]. Es werden zwei Bloomfilter gepflegt, A_0 und B_0 , die jeweils alle Keys enthalten, die auf 0 bzw. 1 abgebildet werden. Da die Gefahr von False-Positives besteht, kann ein Element sowohl in A_0 als auch B_0 enthalten sein. Es werden deshalb zwei weitere Bloomfilter eingeführt, A_1 und B_1 , wobei A_1 alle Elemente enthält, die in A_0 enthalten sind und gleichzeitig ein False-Positive in B_0 sind. B_1 ist analog dazu, es enthält Elemente aus B_0 , die ein False-Positive in A_0 sind. Da ein Element erneut sowohl in A_1 als auch B_1 enthalten sein kann, liegt ein rekursives Problem vor, das durch weitere Schichten A_i und B_i gelöst werden kann.

Da A_{i+1} und B_{i+1} nur Elemente enthalten, die in A_i und B_i ein False-Positive sind, fällt die Größe dieser Bloomfilter exponentiell ab und ab einer bestimmten Stufe können diese Werte sehr einfach in einer herkömmlichen, deterministischen Hashtabelle gespeichert werden. Da

fast alle Abfragen direkt durch das erste Bloomfilter-Paar beantwortet werden können, ist die Average-Case Laufzeit (gemittelt über alle Keys aus U) in $O(1)$ und die Worst-Case Laufzeit $O(\log \log n)$. Um einen Key zu suchen, wird er zunächst im ersten Bloomfilter-Paar gesucht. Ist er dort nicht vorhanden, wird *False* zurückgegeben. Ist er in einem der Filter enthalten, wird der korrespondierende Value zurückgegeben. Ist er in beiden Filtern enthalten, wird die Suche im nächsten Bloomfilter-Paar fortgesetzt. Um ein Key-Value Paar in den Bloomier Filter einzufügen, wird der Key zuerst in dem zum Value gehörigen Bloomfilter der ersten Stufe abgelegt. Gleichzeitig wird geprüft, ob das Element im anderen Bloomfilter ein False-Positive ist. Ist dies der Fall, wird der Algorithmus rekursiv auf die nächste Stufe angewendet.

Der Algorithmus kann leicht auf beliebige Values aus einer größeren Menge V erweitert werden. Dazu wird jeder Value als Bit-Repräsentation betrachtet und für jedes Bit der beschriebenen Algorithmus mit Bloomfilter-Paaren angewendet. Der Platzbedarf für einen solchen Bloomier Filter mit Values aus V beträgt etwas mehr als $\lceil \log_2 |V| \rceil * 2 * m$. Eine einfache Möglichkeit, um beliebige Values auf eine mögliche kompakte n -Bit Repräsentation abzubilden, ist Minimal Perfect Hashing. Da V statisch ist, kann eine Hashfunktion ermittelt werden, die jedes Element aus V auf das Intervall $\{1, \dots, |V|\}$ abbildet und damit die kompakteste denkbare Darstellung für V ist. Im Bloomier Filter selbst müssen dann lediglich die Keys mit den Hashwerten der Values verknüpft werden.

Auf den ersten Blick scheinen Bloomier Filter keine unmittelbare Anwendung für ORESTES zu besitzen. Tatsächlich ist der Bloomier Filter jedoch eine überaus interessante Darstellung für die Clientseite, da er erlaubt, Informationen über geänderte Objekte zu speichern. Eine denkbare Value-Menge wäre $V = \{HotspotObject, NormalObject\}$. Ein Client, der einen Bloomier Filter mit dieser Zusatzinformation erhält, könnte beispielsweise explizit ein Lock für das Hotspot Objekt allokalieren, um optimistische Transaktionsabbrüche zu verhindern. Da Hotspot Objekte von dem aktuellen Workload abhängen, ist es sogar ausgesprochen sinnvoll, Hotspot Objekte nicht statisch zu deklarieren, sondern anhand des Sliding Windows der Expirationsdauer zu ermitteln. So werden nur Hotspot Objekte als solche markiert, die auch zum aktuellen Zeitpunkt viel geschrieben werden.

Um einen Bloomier Filter mit dieser Information auf der Serverseite generieren zu können, wäre eine Kombination aus Counting, Spectral und Bloomier Filtern notwendig. Der Bloomier Filter müsste erweitert werden, um assoziative Multimengen bzw. das Counting zu unterstützen. Dieser Versuch wurde bisher nie unternommen. Wahrscheinlich ließe sich diese Erweiterung mit dem Ersetzen der gewöhnlichen Bloomfilter durch Counting Bloomfilter in dem Bloomier Filter erreichen. Dann könnte jedes Objekt erst mit dem Value *NormalObject* eingefügt werden. Sobald einer der spektralen Algorithmen zur Multiplizitätsabschätzung (z.B. Minimum Selection) einen kritischen Wert von Änderungen feststellt, wird das Objekt aus dem Bloomier Filter entfernt und mit dem Value *HotspotObject* eingefügt. Ein materialisierter Bloomier Filter könnte dabei stets mitgeneriert werden, um effizient



vom Client abgerufen werden zu können. Der Speicheroverhead durch einen Bloomier Filter würde sich etwa auf den Faktor 2 belaufen. Es wäre eine interessante Forschungsperspektive, Counting Bloomfilter und Bloomier Filter zu kombinieren und den beschriebenen Mechanismus zur Hotspot Erkennung von Objekten umzusetzen.

3.5.6 Stable Bloomfilter

Zwei Erweiterungen von Bloomfiltern gehen explizit mit dem Problem eines zeitlichen Sliding Windows über einen Datenstrom um. In ORESTES ist dieser Datenstrom eine Sequenz von Objekt-IDs, deren zugehöriges Objekt geändert wurde. Das Problem bei einem Sliding Window ist, dass die alten Elemente wieder aus dem Bloomfilter entfernt werden müssen. In ORESTES ist dieses Problem nicht schwerwiegend, da es über alle ORESTES-Server parallelisiert werden kann. Jeder ORESTES-Server kann die Elemente aus dem Counting Bloomfilter wieder entfernen, die über ihn geschrieben wurden. Problematisch wäre nur der Ausfall eines ORESTES-Servers, da die von ihm verwalteten Objekte nicht mehr aus dem Counting Bloomfilter entfernt würden.

Der Stable Bloomfilter [175] löst die Alterung von Einträgen auf sehr einfache Weise. Der Stable Bloomfilter besteht aus einem normalen Counting Bloomfilter mit einer modifizierten *Insert* Methode. Beim Einfügen eines Elements werden vor dem Einfügen d zufällige Counter ausgewählt und dekrementiert. Die Counter des eingefügten Elements werden anschließend auf ihr Maximum $2^c - 1$ gesetzt. Deng et al. [175] haben gezeigt, dass ein solcher Filter die *Stabilitätseigenschaft* erfüllt: ab einem Zeitpunkt wird die Anzahl von Counter mit dem Wert 0 konstant. Sobald dieser stabile Punkt erreicht ist, beträgt die False-Positive Wahrscheinlichkeit (cf. [174]):

$$f = \frac{1}{\frac{d}{k} - \frac{d}{m} + 1}$$

Um eine gewünschte False-Positive Wahrscheinlichkeit zu erreichen, muss also eine geeignete Kombination der Parameter gefunden werden. Da über den praktischen Einsatz von Stable Bloomfiltern und insbesondere das Erreichen der Stabilität wenig bekannt ist, erscheint es uns verfrüht, diese Datenstruktur für ORESTES einzusetzen. Des Weiteren ist nicht garantiert, dass ein Objekt, dessen Expirationsdauer abgelaufen ist, auch direkt aus dem Stable Bloomfilter entfernt wird. Dies würde zu unnötigen Revalidierungen und einer erhöhten Last des Servers führen.

3.5.7 A²Bloomfilter

Eine weitere Bloomfilter-Variante mit der Unterstützung zur Verdrängung von alten Elementen ist das Active-Active Buffering (A² Bloomfilter) [176]. Es pflegt zwei Bloomfilter BF_1 und BF_2 . Ein Element gilt als enthalten, wenn es in mindestens einem der beiden Bloomfilter enthalten ist. Ein Element wird beim Einfügen in BF_1 gespeichert. Sobald BF_1 seine *Kapazität* erreicht hat - d.h. die maximale Anzahl an eingefügten Elementen erreicht ist, für die eine False-Positive Rate f noch garantiert werden kann - wird BF_2 zurückgesetzt und mit BF_1

getauscht. Die Elemente überleben anschließend so lange in BF_2 , bis der nächste Tausch vorgenommen wird. Diese Datenstruktur ist sehr speichereffizient, da ein Bloomfilter stets vollständig gefüllt ist [174]. Wie der Stable Bloomfilter gibt jedoch auch der A^2 Bloomfilter keine Garantie dafür, dass Elemente nach Ablauf einer bestimmten Zeitspanne verdrängt werden. Die Verdrängung erfolgt stets in Abhängigkeit der geänderten Elemente. Der A^2 Bloomfilter ist damit ungeeignet für ORESTES, da er zur Folge hätte, dass in Phasen geringer Änderungsaktivität Objekte länger im Bloomfilter bleiben und damit lange nicht gecacht ausgeliefert werden können.

3.6 Existierende Implementierungen

Drei Implementierungen von Bloomfiltern haben wir bereits untersucht: die Umsetzung in HBase, Cassandra und Squid. Alle drei Umsetzungen haben schwerwiegende Einschränkungen. Zwei besonders populäre Implementierungen wollen wir zusätzlich untersuchen: den Java Bloomfilter von Magnus Skjogstad [177] und die Ruby Bloomfilter von Ilya Grigorik [178].

3.6.1 Bloomfilter von Magnus Skjogstad

Der in Java implementierte Bloomfilter von Magnus Skjogstad [177] ist vor allem deshalb populär, weil seine Benutzung lediglich die Einbindung einer einzigen Java-Datei erfordert. Die Implementierung besteht aus einem einfachen Bloomfilter. Trotz seiner Popularität ist der Bloomfilter sehr eingeschränkt:

- Beschränkung auf eine einzige Hashfunktion (MD5). Der Hashwert wird in 32-Bit Fragmente zerlegt und diese Fragmente als Hashwerte benutzt. Die Implementierung ist mathematisch fehlerhaft, da statt Rejection Sampling eine einfache Modulo-Reduktion auf die Größe m des Bloomfilters durchgeführt wird. Wir haben diesen Fehler in einem Bug-Report gemeldet [177].
- Keine Unterstützung der Mengenvereinigung und Schnittmenge.
- Keine Erweiterung auf Counting Bloomfilter.
- Handwerkliche Fehler. Die Bloomfilter-Implementierung ist nicht Thread-safe (was sinnvoll ist), benutzt aber zur Durchführung der Hashwert-Berechnung die Synchronisationsprimitive von Java, die stets einen signifikanten Overhead darstellen – im besagten Fall aber völlig unnötig sind.

3.6.2 Bloomfilter von Ilya Grigorik

Ilya Grigorik, ein bekannter Ruby Entwickler von Google, hat eine sehr bekannte Open-Source Bibliothek mit Bloomfiltern veröffentlicht [178]. Die Implementierung enthält sowohl einen einfachen Bloomfilter als In-Memory Datenstruktur für Ruby, sowie einen einfachen und einen Counting Bloomfilter als persistente Struktur für den Key-Value Store Redis. Die bemerkenswerte Idee einer Implementierung auf Basis von Redis nehmen wird als Inspiration für unsere Implementierung, räumen aber mit den Fehlern und Einschränkungen auf:



- Die (einzig verfügbare) Hashfunktion ist sehr schlecht gewählt. Es handelt sich um eine CRC32-Prüfsumme. CRC32 wurde jedoch zur Detektion von zufälligen Korruptionen entwickelt und ist als Hashfunktion nahezu unbrauchbar [179]. Insbesondere folgen die generierten Hashwerte keiner diskreten Gleichverteilung. Darüber hinaus ist der sogenannte *Avalanche-Effekt* sehr schlecht, d.h. die Änderung eines Bits in der Eingabe hat deterministische und minimale Auswirkungen auf die Änderung des Hashwerts. Die Implementierung von Grigorik benutzt für die k Hashfunktionen jedoch eine Iteration von 1 nach k , um den Schleifenzähler in der Eingabe der Hashfunktion zu konkatenieren. So kommen beide Nachteile von CRC32 direkt zum Tragen und die Hashwerte sind nicht zufällig.
- Die In-Memory Bloomfilter-Portierung ist schlecht portabel, da sie Operationen lediglich auf einen in C implementierten Counting Bloomfilters abbildet. Obwohl die C-Implementierung als Counting-Bloomfilter ausgelegt ist, unterstützt die Bibliothek in Ruby nur einen einfachen Bloomfilter.
- Keine automatische Konfiguration des Bloomfilters durch die Angabe gewünschter Parameter.
- Die Redis-Bloomfilter bilden die Bloomfilter-Operationen unperformant ab. Wir gehen auf diesen Punkt bei der Beschreibung unserer Redis-Bloomfilter ein. Insbesondere wird keine konsistente Verwendung von Pipelining, optimistischen Transaktionen und Lua-Scripting gemacht.
- Die Nützlichkeit des Redis Counting Bloomfilters ist stark reduziert, da die Implementierungen nicht Thread-safe ist. D.h. trotz eines gemeinsamen nutzbaren Key-Value Stores kann der Bloomfilter nicht von mehreren Clients genutzt werden, da es Race-Conditions zwischen den Einfüge- und Löschoptionen gibt.
- Die Methode *clear* zum Zurücksetzen der Redis-Bloomfilter löscht neben dem Bloomfilter auch alle anderen Daten der Redis-Instanz.
- Der einfach Redis Bloomfilter besitzt eine *Delete* Methode, die eine Löschung durchführt, indem alle betroffenen Bits auf 0 zurückgesetzt werden. Dieser Ansatz ist natürlich falsch und führt zu False-Negatives.

3.7 Implementierung

Unsere Untersuchung der Bloomfilter-Implementierungen ergibt, dass keine der existierenden Lösungen ausreichend ist. Unsere Umsetzung greift aber zwei der guten Ideen auf:

- Die Bloomfilter sollen ohne externe Abhängigkeit zu Third-Party Libraries funktionsfähig sein (wie der Bloomfilter von Magnus Skjegstad [177]).
- Die Bloomfilter sollen als hochperformante, persistente und geteilte Datenstruktur nutzbar sein (wie die Redis-Bloomfilter von Ilya Grigorik [178], erweitert um Nebenläufigkeit).

Wir implementieren die Bloomfilter in Java. Abbildung 57 gibt eine vereinfachte Übersicht über die implementierten Bloomfilter. Die Schnittstelle eines Bloomfilters wird in der Basis-

klasse `BloomFilter` spezifiziert. Die Methoden sind so gewählt, dass sie mit der `Set` Schnittstelle des Java Collections-Frameworks übereinstimmen, dieses jedoch nicht implementieren, da ein semantischer Unterschied besteht – Bloomfilter speichern keine Elemente, weshalb nicht über sie iteriert werden kann. Die Basisklasse implementiert außerdem verschiedene Hashfunktionen, die von abgeleiteten Klassen genutzt werden können. Außerdem enthält sie statische Methoden für die Ermittlung optimaler Parameterkonfigurationen. Zur Speicherung des Bit-Arrays kommt ein `BitSet` zum Einsatz, die performanteste Implementierung, die auf der Java Plattform hierzu zur Verfügung steht. Die Umsetzung des einfachen Bloomfilters für Redis, `RedisBloomFilter`, leitet sich von `BloomFilter` ab. Durch die saubere Kapselung kann der Großteil der Funktionalität der Basisklasse weiterverwendet werden. Dies wird dadurch erzielt, dass die `BitSet` Implementation durch unser `RedisBitSet` ausgetauscht wird, das als Fassade für ein in Redis gespeichertes Bit-Array dient.

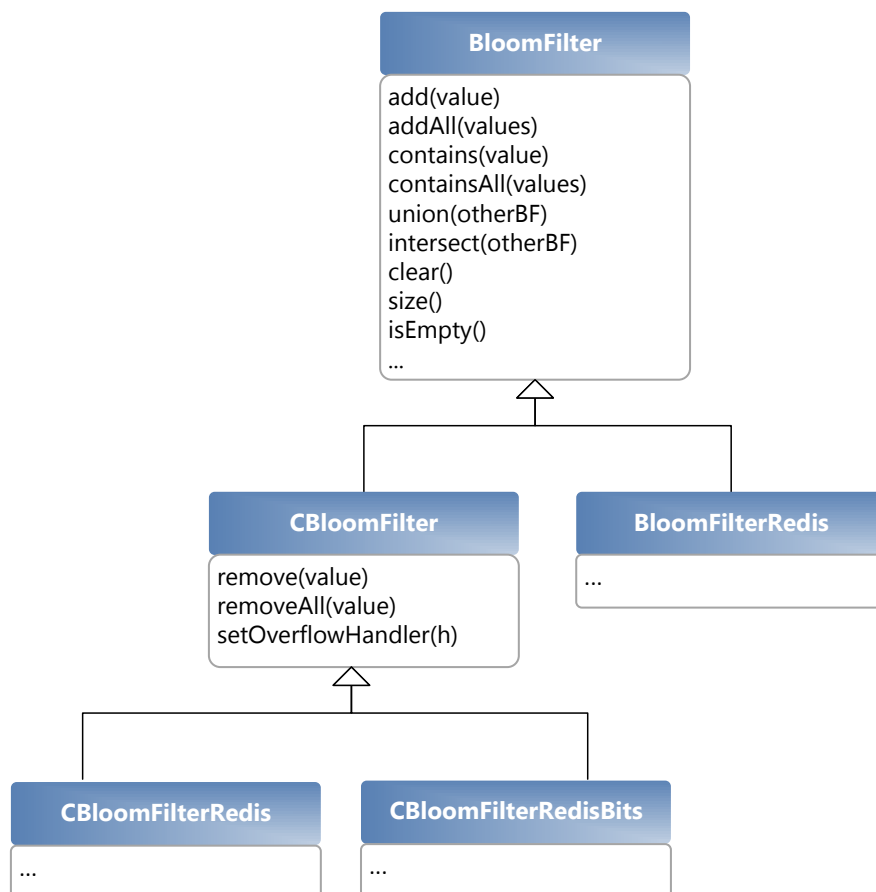


Abbildung 57 Ein stark vereinfachtes Klassendiagramm der Bloomfilter.

Der Counting Bloomfilter, `CBloomFilter`, ist ebenfalls eine Erweiterung der `BloomFilter` Basisklasse. Es besitzt einige zusätzliche Methoden, die wichtigste von ihnen ist `remove`, zum Löschen eingefügter Elemente. Der Counting Bloomfilter wird mit einer frei wählbaren Countergröße c initialisiert. Die Counter sind effizient als Bit-Array implementiert und `CBloomfilter` enthält Methoden, um performant arithmetische Operationen auf den kompakt abgelegten Countern auszuführen. Der `CBloomfilter` wiederum wird erweitert durch



zwei Implementierungen für Redis. Die Redis Implementierungen erläutern wir im nächsten Kapitel.

Alle Bloomfilter-Klassen und ihre Methoden sind umfassend dokumentiert und durch mehrere Unit-Tests geprüft. Wir werden die Bloomfilter Bibliothek in naher Zukunft als Open-Source Projekt unabhängig von ORESTES veröffentlichen. Wir haben insbesondere darauf geachtet, die Fehler und Einschränkungen der HBase, Cassandra, Squid, Grigorik und Skjogstad Bloomfilter Implementierung zu umgehen bzw. zu beheben. Das heißt insbesondere:

- Die Bloomfilter können durch die gewünschten Parameter initialisiert werden, die mathematisch optimalen, fehlenden Parameter werden nach den Formeln aus Kapitel 3.3 berechnet.
- Das Hashing (das wir im übernächsten Kapitel beschreiben) kann umfassend konfiguriert werden und verschlechtert nicht durch Tricks wie Double Hashing und fehlendes Rejection Sampling die praktische False-Positive Rate. Diverse Hashfunktionen stehen zur Verfügung.
- Der Bloomfilter und der Counting Bloomfilter besitzen eine einheitliche Schnittstelle mit allen Methoden, die sich sinnvoll für Bloomfilter implementieren lassen (u.a. auch Vereinigung und Schnittmengen). Eine Bloomfilter-Halbierung ist bisher nicht implementiert, könnte aber leicht hinzugefügt werden.

Als einen ersten Anhaltspunkt dafür, dass die wesentlich umfangreichere Funktionalität unserer Bloomfilter-Implementierung keine negativen Auswirkungen auf die Geschwindigkeit hat, benutzen wir den Bloomfilter-Benchmark, den Skjogstad mit seiner Java Bloomfilter-Implementierung ausliefert [177]. Der Test wurde unter gleichen Bedingungen auf einer Windows 8 Maschine mit einem 4-Core Intel i7 und 8 GByte RAM ausgeführt. Der Bloomfilter von Skjogstad erlaubt keine Hash-Konfiguration. Wir haben für den Benchmark unsere Default-Einstellung benutzt, die ebenfalls MD5 verwendet, aber auf mathematische Korrektheit achtet. Für andere Hashfunktionen (z.B. den Java RNG) ist unsere Implementierung um den Faktor 4-6 schneller.

Unsere Implementierung	Skjogstad Bloomfilter [177]
Testing 50000 elements k is 10 p is 0.0010000189402619613 m is 718880 add(): 0.668s , 74850.2994 elements/s contains(), existing: 0.436s , 114678.8991 elements/s containsAll(), existing: 0.416s , 120192.3077 elements/s contains(), nonexisting: 0.426s , 117370.892 elements/s containsAll(), nonexisting: 0.429s , 116550.1166 elements/s	Testing 50000 elements k is 10 p is 9.765579999352291E-4 m is 721348 add(): 0.818s , 61124.6944 elements/s contains(), existing: 0.543s , 92081.0313 elements/s containsAll(), existing: 0.575s , 86956.5217 elements/s contains(), nonexisting: 0.512s , 97656.25 elements/s containsAll(), nonexisting: 0.504s , 99206.3492 elements/s

Abbildung 58 Benchmarkergebnisse für die Default Einstellung unserer Bloomfilters bei der Ausführung des Benchmarks von Skjogstad [177].

3.7.1 Die Redis Bloomfilter

Um die Bloomfilter mehrerer ORESTES-Server konsolidieren zu können, Persistenz zu schaffen und dabei noch immer eine sehr hohe Geschwindigkeit zu erreichen, benutzen wir die Idee von Ilya Grigorik: das Bit-Array des Bloomfilters wird in einem In-Memory Key-Value Store gespeichert. Eine naheliegende und sehr lange etablierte Lösung für einen solchen In-Memory Key-Value Store ist Memcache [61]. Sie hat jedoch mehrere Nachteile. So ist die API einerseits zwar sehr einfach, erlaubt aber keine komplexen Operationen. Insbesondere eine Manipulation auf Bit-Ebene ist ausgeschlossen. Deshalb müsste eine auf Memcache basierende Implementierung entweder stets das gesamte Bit-Array lesen und zurückschreiben, was einen signifikanten Synchronisationsoverhead zur Vermeidung von Race-Conditions nach sich ziehen würde. Oder jedes Bit bzw. jeder Counter müsste als einzelnes Key-Value gespeichert werden, wodurch die Materialisierung und effiziente Generierung des Bloomfilters unmöglich wäre. Redis ist ein sehr populärer Vertreter der NoSQL Szene, der versucht, die Einschränkungen von Memcache durch eine reichhaltigere API bei gleicher Performance zu lösen.

Redis (für *Remote Dictionary Server* [131]) implementiert einen Key-Value Store mit den zusätzlichen Datenstrukturen *List*, *Set*, *SortedSet*, *Hash*, sowie der Unterstützung für Message Queuing, serverseitiges Scripting und Optimistische Batch-Transaktionen. Redis ist ein Open-Source System, dessen Hauptentwickler von VMWare finanziert werden. Diverse Benchmarks (z.B. [180]) wurden durchgeführt, die Redis mit anderen In-Memory Key-Value Systemen vergleichen. Das Ergebnis ist, dass der Performance-Unterschied zwischen sehr simplen Systemen wie Memcache gegenüber Redis marginal (ca. 10%) ist. Tatsächlich erreicht Redis auf durchschnittlicher Hardware etwa einen Durchsatz von 100.000 einzeln gestellten *Get* und *Set* Operationen pro Sekunde; auf gängiger Server-Hardware das doppelte. Selbst in einer virtuellen Linux-Maschine mit nur einem Prozessorkern und 1 GByte RAM konnten wir etwa 40.000-60.000 Operationen pro Sekunde erreichen.

Besonders interessant für die Implementierung von Bloomfiltern ist Redis aufgrund seiner Operationen für die bitweise Manipulation der unter einem Key gespeicherten Daten:

- *GETBIT(key, offset)*. Gibt das Bit an Position *offset* für den unter *key* gespeicherten Wert zurück.
- *SETBIT(key, offset)*. Setzt analog zu *GETBIT* ein Bit.
- *BITOP(op, key1, key2)*. Verknüpft zwei Bit-Arrays durch AND, OR oder XOR.
- *BITCOUNT(key)*. Zählt gesetzte Bits für den Wert unter *key*.
- *GET(key)*. Gibt den gesamten Inhalt zurück, der unter *key* gespeichert ist.

GETBIT und *SETBIT* können also zur Implementierung eines Bit-Arrays genutzt werden, während mit *GET* das gesamte Bit-Array bei Anfrage eines Clients sehr schnell abgerufen werden kann. Redis verwendet zur Kommunikation zwischen Client und Server ein textbasiertes TCP-Protokoll. Aufgrund des Datenmodells ist dieses Protokoll so einfach, dass die



Open-Source Community schnell für verschiedenste Programmiersprachen Zugriffsbibliotheken entwickelt hat. Redis unterstützt die persistente Speicherung von Änderungen anhand definierter Regeln, z.B. in periodischen Abständen oder nach großen Änderungen (*Snapshot Mode*), aber auch programmatisch über die API oder für jede Operation (*Append Only File Mode*).

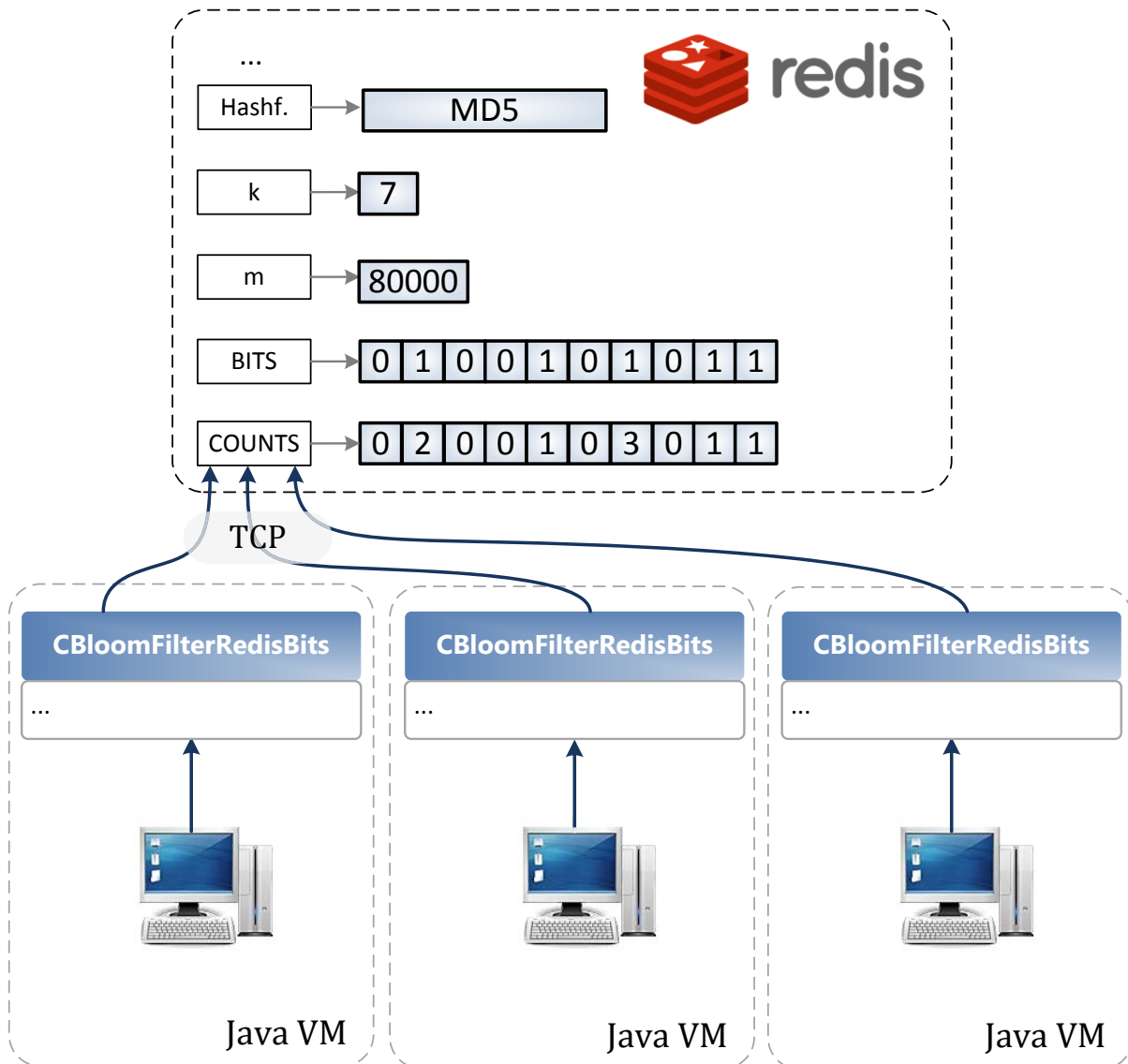


Abbildung 59 Das Prinzip der Redis Bloomfilter.

Die Netzwerk-Architektur unserer Redis Bloomfilter ist in Abbildung 59 gezeigt. Jeder Client (z.B. der ORESTES-Server), der auf einem geteilten Bloomfilter arbeiten will, erzeugt eine Instanz der gewünschten Redis Bloomfilter-Klasse. Diese verbindet sich über TCP mit einer angegebenen Redis Instanz und nutzt sie als Backend für die Datenstrukturen. Die Redis-Instanz kann auch auf dem System eines Clients ausgeführt werden, wodurch die lokale Kommunikation auf IPC (*Inter Process Communication*) z.B. durch Unix Domain Sockets reduziert werden kann. Alle Redis Bloomfilter arbeiten nebenläufigkeitsbewusst auf der Redis-Instanz, so dass beliebig viele Clients einen Bloomfilter teilen können.

Horizontale Skalierung kann in Redis bisher nur über intelligente Client Bibliotheken vorgenommen werden, die eine Hash-basierte Partitionierung der Keys vornehmen und auf eine Menge von Redis-Servern abbilden. Viele Bibliotheken unterstützten dieses *Client Sharding*. Die native Unterstützung impliziter Partitionierung (Redis Cluster) wird für das kommende Jahr (2013) erwartet [3]. Um hohe Verfügbarkeit zu gewährleisten, unterstützt Redis asynchrone Master-Slave-Replikation. Die Master-Slave Beziehungen können, wie in Abbildung 60 gezeigt, beliebig geschachtelt werden. Schreibenfragen werden stets nur gegen den Master gerichtet, um die Konsistenz des Masters stets sicherzustellen. Die Master-Slave Replikation bietet für ORESTES eine interessante Option. So kann der Master so konfiguriert werden, dass er ganz auf Persistenz verzichtet, aber einer oder mehrere seiner Slaves alle Änderungen direkt persistiert. So wird einerseits hohe Verfügbarkeit durch eine Option zum Failover beim Ausfall des Masters erreicht und gleichzeitig die Antwortzeit des Masters durch die ausschließlich RAM-basierte Speicherung minimiert. Das Lesen aus den Slaves sollte für die Redis Bloomfilter jedoch vermieden werden, da durch die asynchrone Natur der Replikation nicht garantiert wird, dass Änderungen am Master direkt in den Slaves sichtbar werden.

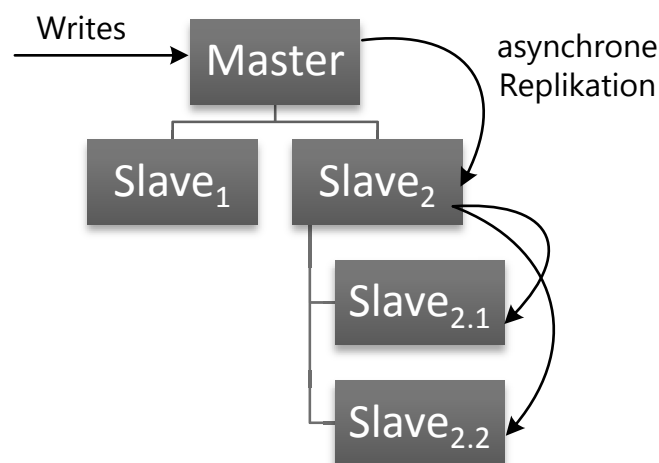


Abbildung 60 Mehrstufige asynchrone Replikation in Redis.

Wir beschreiben im Folgenden knapp die Implementierung unserer Redis Bloomfilter.

3.7.1.1 Einfacher Redis Bloomfilter

Der einfache Redis Bloomfilter unterscheidet sich von dem Java Bloomfilter prinzipiell nur durch die Implementierung des Bit-Arrays. Das `RedisBitSet` kapselt ein Bit-Array in Redis als gewöhnliches `BitSet`. Diese Abstraktion alleine ist jedoch unperformant – das Einfügen eines Elements in den Bloomfilter erfordert das Setzen von k Bits. In seiner naiven Form würde ein Einfügen deshalb k Roundtrips zwischen Client und Redis erfordern. Diesen signifikanten Performance-Bottleneck gilt es (anders als in der Grigorik Implementierung des Counting Bloomfilters) unbedingt zu vermeiden. Um Operationen zu gruppieren, bietet Redis zwei Mechanismen:



- **Pipelining.** Ähnlich dem HTTP-Pipelining können Operationen auch über das Redis Protokoll versendet werden, ohne auf die Antwort der jeweiligen Operation zu warten. Der Server beantwortet die Anfrage mit den gebündelten Ergebnissen der Einzeloperationen (*Multi-Bulk Reply*).
- **MULTI Transaction.** Eine Transaktion in Redis unterscheidet sich signifikant von klassischen ACID Transaktionen. Eine *MULTI* Transaktion in Redis gruppiert beliebige Operationen und führt sie atomar und isoliert als Batchoperation auf dem Server aus. Es können jedoch zwischen den Operationen keine Abhängigkeiten bestehen, da dies einerseits einen Roundtrip zum Client erfordern würde und andererseits verhindert, dass die Transaktion in Redis als Batch-Operation ausgeführt wird. Das Ergebnis einer Transaktion ist ebenfalls die gebündelte Ausgabe der Einzeloperationen. Die Transaktion wird durch die *MULTI* Operationen eingeleitet und durch die *EXEC* Operation gebündelt ausgeführt.

Transaktionen in Redis unterstützen außerdem Optimistic Locking in seiner einfachsten Form: durch einen oder mehrere *WATCH(key)* Befehle wird die Ausführung der Batch Transaktion konditional. Nur wenn der unter *key* gespeicherte Wert zum Zeitpunkt des *EXEC* Befehls noch unverändert ist, wird die Transaktion ausgeführt, ansonsten verworfen. Unsere Redis Bloomfilter Implementierungen nutzen eine Kombination aus Pipelining und *MULTI* Transactions. Das *RedisBitSet* erlaubt deshalb das Setzen eines Ausführungskontextes, d.h. einer Pipeline oder Transaktion (siehe Abbildung 61).

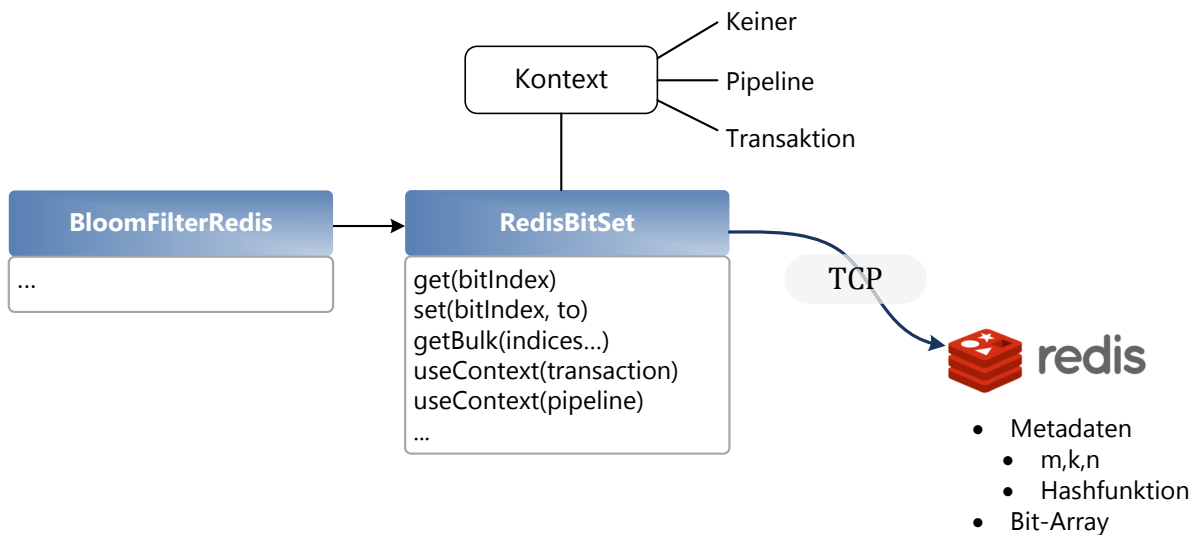


Abbildung 61 Vereinfachte Architektur des Redis Bloomfilters.

Das Zusammenspiel von Pipelining und Batch-Transaktionen ist als Pseudocode für die *add* Methode in Abbildung 61 gezeigt. Strenggenommen ist für das Setzen der Bits eine Batch Transaktion nicht erforderlich, da das Setzen eines Bits sowohl kommutativ als auch idempotent ist und eine verschränkte Ausführung zweier *add* Aufrufe deshalb zu dem gleichen Ergebnis führt. Da die Batch-Transaktion jedoch keinen Overhead darstellt ([131]), ist sie eine

gute Option, um das Verhalten des einfachen Bloomfilters dem der Counting Bloomfilter anzugleichen, wo die Atomarität und Isolation von *contains*, *add* und *delete* unerlässlich ist.

```
def add(element):
    hashes = hash(element)
    redis.pipeline()
    redis.multi()
    for each hash in hashes:
        redis.setBit("bits", hash, true)
    redis.exec()
    redis.flushPipeline()
```

Abbildung 62 Die *Insert*, *Contains* und *Delete* Methode eines Counting Bloomfilters.

3.7.1.2 Redis Bloomfilter mit Bit-Array Countern

Wir haben zwei Counting Bloomfilter für Redis entwickelt, die sich in der Implementierung der Counter unterscheiden:

- `CBloomFilterRedisBits`, der die Counter als Bit-Array verwaltet.
- `CBloomFilterRedis`, der jeden Counter als separaten Key pflegt.

Die große Schwierigkeit einer nebenläufig nutzbaren Implementierung des Counting Bloomfilters ist der transaktionale Charakter einer Änderung. Das Lesen ist trivial, da es gegen den materialisierten Bloomfilter gerichtet werden kann. Das Hinzufügen und Entfernen erfordert jedoch Atomarität und Isolation während des Zugriffs auf die Bit-Arrays des Counters und des Bloomfilters. Werden diese Operationen unsynchronisiert durchgeführt, können verschiedenste schwer zu ermittelnde Mehrbenutzeranomalien auftreten. So könnte beispielsweise eine naive Umsetzung von *delete(x)* so aussehen:

1. Berechne die k Hashwerte von x .
2. Frage einzeln die Werte der Counter an den berechneten Positionen ab.
3. Addiere 1 auf die gelesenen Counter und schreibe sie zurück.
4. Setze das Bit-Array des Bloomfilters gemäß der neuen Counter-Werte.

Der beschriebene Algorithmus ist nicht nebenläufigkeitssicher. Abbildung 63 gibt ein Beispiel für eine Lost-Update Anomalie, die auftreten kann, weil zwischen dem Abfragen der Counter und ihrer Änderung weitere Operationen ausgeführt werden können. Auch das gebündelte Abfragen aller Counter ist keine Lösung, da das Zurückschreiben dennoch die Änderungen anderer Clients überschreibt, wenn diese zwischen dem Abfragen und Zurückschreiben überlappende Änderungen vornehmen. Auch das Aktualisieren des materialisierten Bloomfilters ist anfällig für Lost Update Anomalien. Eine verbesserte Lösung könnte Optimistic Locking und Batch Transaktionen nutzen:

1. Berechne die k Hashwerte von x .
2. Führe `WATCH(counterBits, bloomBits)` aus.
3. Frage gebündelt die Werte der Counter an den berechneten Positionen ab.



4. Beginne eine Batch Transaktion (*MULTI*).
5. Ändere das Bit-Array der Counter und das Bit-Array des Bloomfilters gemäß der neuen Counter-Werte und führe die Transaktion aus (*EXEC*).
6. Wenn die Transaktion nicht erfolgreich war, d.h. *counterBits* oder *bloomBits* zwischenzeitlich geändert wurden, wiederhole die Prozedur bei Schritt 2.

Der beschriebene Algorithmus ist nebenläufigkeitssicher – immer wenn eine Verletzung der seriellen Reihenfolge auftritt, wird die Transaktion neu gestartet. Der Algorithmus hat jedoch ein sehr schwerwiegendes Problem: selbst unter mittlerer Last wird er sehr lange brauchen, um zu terminieren (*Transaction Starvation*). Das Problem besteht darin, dass durch den Roundtrip zur Datenbank zwischen dem Lesen der Counter und dem Zurückschreiben viel Zeit für konkurrierende Zugriffe bleibt. Jeder erfolgreiche Zugriff sorgt für einen Transaktionsabbruch und –neustart der anderen nebenläufigen Transaktionen. Der beschriebene Fall stellt das Worst-Case Szenario für optimistische Mehrbenutzerkontrolle dar, nämlich das ausschließliche Lesen und Schreiben auf einem Hotspot Objekt.

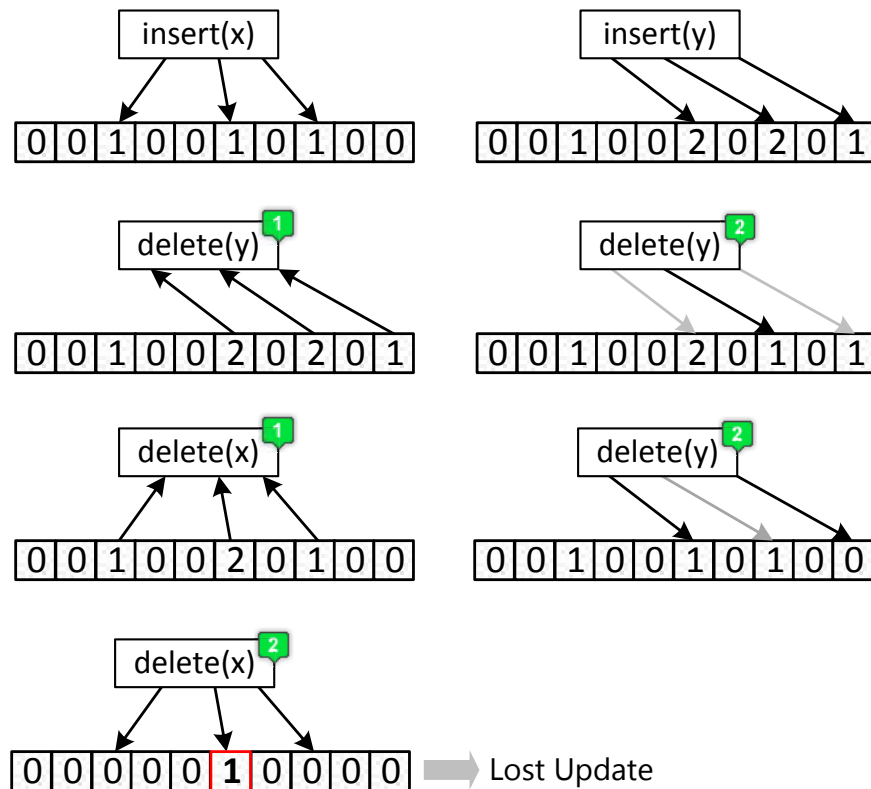


Abbildung 63 Die Problematik nebenläufiger Änderungen des Bit-Arrays (Leserichtung: von links nach rechts und von oben nach unten).

Es liegt also nahe, statt einer optimistischen Transaktion eine pessimistische, sperrbasierte zu verwenden. Redis bietet nativ keine Sperrmechanismen. Diese müssen deshalb in der Applikationsschicht nachgebaut werden. Da alle benutzten Objekte (*bloomBits*, *counterBits*) bereits zu Beginn der Transaktion bekannt sind, können für diese Objekte durch *Preclaiming* beim Transaktionsbeginn Sperren angelegt werden. Dies schränkt jedoch die Parallelität ein. Da das Sperrmuster bekannt ist, kann der Algorithmus so designet werden, dass keine Dead-

locks auftreten. Auf das Preclaiming kann dann verzichtet werden. Mit der Annahme, dass keine Abbrüche auftreten, muss überdies kein striktes Zwei-Phasen Sperrprotokoll eingehalten werden, da ohne Abbrüche keine kaskadierenden Rollbacks auftreten können. Der Algorithmus für *add* sähe dann mit einem einfachen Sperrverfahren wie folgt aus:

1. Berechne die k Hashwerte von x .
2. Setze ein Lock auf *counterBits*.
3. Frage gebündelt die Werte der Counter an den berechneten Positionen ab.
4. Erhöhe die Counter in *counterBits*.
5. Setze ein Lock auf *bloomBits*. Gib das Lock für *counterBits* frei.
6. Schreibe die Bits in *bloomBits*.
7. Gib das Lock für *bloomBits* frei.

Dieser Ansatz hat vor allem den Nachteil, dass er die Parallelität signifikant einschränkt. Höchstens zwei Clients können gleichzeitig ein Objekt einfügen, da der erste und der zweite Abschnitt des *add* Algorithmus jeweils eine exklusive Sperre erfordern. Dies ist vor allem deshalb problematisch, weil Schritt 3 und 4 zusammen einen Roundtrip von Client zum Server erfordern. Für die Dauer des Roundtrips bleibt *counterBits* gesperrt. Damit wird die Netzwerklatenz zu einem Bottleneck für den Durchsatz. Dies gilt es zu vermeiden. Darüber hinaus muss das Lock in der Applikationslogik umgesetzt werden, was besonders die Vermeidung von endlos gehaltenen Locks sehr erschwert.

Die Lösung für das Problem liegt in serverseitigen Stored-Procedures. Diese werden in Redis jüngst (seit Version 2.6) unterstützt [3]. Sie werden in der Dokumentation ([181]) als Mechanismus für Transaktionen empfohlen, die nicht auf Optimistic Locking zurückgreifen können oder wollen. Die Stored-Procedures werden in der Programmiersprache *Lua* definiert. *Lua* ist eine leichtgewichtige, interpretierte Multiparadigmensprache, die 1993 vor allem von Roberto Ierusalimsky entwickelt wurde und für eine nahtlose Integration mit C und C++ bekannt ist [182]. Redis erlaubt es, *Lua*-Skripte direkt auf dem Server auszuführen und von dort die Redis-Schnittstelle zu benutzen. Definierte *Lua*-Skripte werden in Redis von einem Script-Cache in optimierter Form gecacht. Bei der Ausführung des Skripts garantiert Redis insbesondere, dass die Ausführung sowohl atomar als auch isoliert ist, d.h. die Semantik einer Transaktion besitzt. Da Redis ein Single-Thread und Event-Loop Design besitzt, verursacht die atomare Ausführung des Skripts in einem *Lua*-Interpreter des Servers nur minimalen Overhead für kurze Skripte [181].

Die Umsetzung der *add* Methode vereinfacht sich damit zu:

1. Berechne die k Hashwerte von x .
2. Rufe das serverseitige *Lua*-Skript *IncrementAndSet* auf und übergebe dabei die berechneten Hashwerte.



Das *IncrementAndSet* Skript ist in Abbildung 64 gezeigt. Es setzt für jeden übergebenen Index (Hashwert) das entsprechende Bit im materialisierten Bloomfilter und führt eine binäre Addition auf dem Bit-Array der Counter durch. Als Parameter erwartet jedes Lua-Skript die Keys, die in dem Skript durch Redis-Operationen benutzt werden. Dies sichert vor allem die Kompatibilität zu der kommenden Redis Cluster Unterstützung. Alle übrigen Parameter des Skripts können frei gewählt werden. *IncrementAndSet* erhält als zusätzliche Parameter die Anzahl der Counter-Bits *c*, die Anzahl der Hashfunktionen sowie die Hashwerte selbst.

```
-- IncrementAndSet: countsBits, bloomBits, c, indexcount, index1, index2, ...
-- ARGV, übergebene Parameter
-- KEYS, Schlüssel, die das Lua-Skript benutzt
for index = 3, ARGV[2] + 2, 1 do
  --Setzen des Bits im materialisierten Bloomfilter
  redis.call('setbit', KEYS[2], ARGV[index], 1)
  --Binäres Addieren auf dem Bit-Array der Counter
  local low = ARGV[index] * ARGV[1]
  local high = (ARGV[index] + 1) * ARGV[1]
  local incremented = false
  for i = (high - 1), low, -1 do
    if redis.call('getbit', KEYS[1], i) == 0 then
      redis.call('setbit', KEYS[1], i, 1)
      incremented = true
      break
    else
      redis.call('setbit', KEYS[1], i, 0)
    end
  end
end
end
```

Abbildung 64 Das Einfügen in den Counting Bloomfilter als serverseitiges Lua Script.

Das Löschen eines Elements wird durch ein analog gestaltetes Lua-Skript gelöst. Dadurch, dass das Lua-Skript Operationen lokal und ohne Netzwerk-Roundtrips und Synchronisationsoverhead ausführt, ist es hocheffizient. Im Gegensatz zu *delete* und *add* ist die *contains* Methode von *CBloomFilterRedisBits* sehr einfach und erfordert kein Lua-Skript. Der Abruf erfolgt direkt als Batch-Transaktion mit Pipelining gegen das Bit-Array des materialisierten Bloomfilters.

3.7.1.3 Redis Bloomfilter mit expliziten Countern

Anstatt die Counter explizit in einem Bit-Array zu verwalten, können die Counter auch als separate Keys gehandhabt werden. *CBloomFilterRedis* verfolgt diesen Ansatz. Ein Key, der einen Integer speichert, unterstützt in Redis drei Operationen, die ihn für einen Bloomfilter interessant machen:

- *INCR(key)*. Inkrementiert atomar den Zähler und gibt den neuen Wert zurück.
- *DECR(key)*. Dekrementiert atomar den Zähler und gibt den neuen Wert zurück.
- *EXPIRE(key, seconds)*. Löscht den Key nach *seconds* Sekunden.

Mithilfe der INCR Operationen kann die add Methode vereinfacht werden, da das Lesen des Counters nicht länger notwendig ist, um ihn zu schreiben:

1. Berechne die k Hashwerte von x .
2. Beginne eine Batch-Transaktion (*MULTI*).
3. Führe *INCR(hash)* für jeden berechneten Hashwert durch und setze in *bloomBits* die entsprechenden Bits.
4. Führe die Transaktion aus (*EXEC*).

Das Löschen kann nicht auf dieselbe Weise vereinfacht werden, da das Setzen des Bits in *bloomBits* davon abhängt, ob der Counter 0 erreicht hat und für das Löschen eines Elements zuerst validiert werden muss, dass es auch enthalten ist. Um sowohl die Stored-Procedures als auch einen sperrenden Ansatz studieren zu können, haben wir in *CBloomFilterRedis-Bits Locking* verwendet, um *remove* zu implementieren:

1. Berechne die k Hashwerte von x .
2. Setze ein globales Lock.
3. Lade die Werte aller Counter mit einer Operation (*MGET*).
4. Für jeden Counter, der größer gleich eins ist, führe *DECR* aus und setze das entsprechende Bit in *bloomBits*.
5. Gib das Lock wieder frei.

Der gleiche Algorithmus hätte genauso durch ein Lua-Skript implementiert werden können. Die Umsetzung des Locks ist eine Herausforderung, da das Anfordern eines Locks implizit erfordert, atomar zu überprüfen, ob das Lock derzeit gehalten wird, um es dann zu akquirieren. Zudem sollte das Lock im Falle eines Applikationsfehlers nach einem Timeout wieder freigegeben werden. Abbildung 65 zeigt unsere Implementierung eines Locks durch ein serverseitiges Lua-Skript. Zuerst wird geprüft, ob das Lock existiert. Wenn es nicht existiert, wird es angelegt und dabei gleichzeitig eine Expiration des Schlüssels (Lock Timeout) festgelegt. Redis entfernt so expirierte Locks automatisch. Das Skript gibt 0 zurück, wenn das Lock bereits gehalten wird und 1, wenn das Lock erfolgreich gesetzt wurde.

```

--lock lockkey, timeout
local lock = redis.call('get', KEYS[1])
if not lock then
    return redis.call('setex', KEYS[1], ARGV[1], "locked")
end
return false

```

Abbildung 65 Das Einfügen in den Counting Bloomfilter als serverseitiges Lua Script.

Der Vorteil des Skripts ist, dass kein zusätzlicher Roundtrip beim Sperren erforderlich ist und Lock-Timeouts von Redis und nicht in der Applikation behandelt werden. Ein verbreitetes Locking Pattern für Redis ist [131]:

1. *WATCH(lock)*. Optimistic Locking auf dem Key *lock* anwenden.



2. *GET(lock)*. Abfrage, ob das Lock gesetzt ist.
 - a. Warten oder Abbrechen, wenn das Lock gesetzt ist.
3. *MULTI* Transaktion starten, *SETEX(lock, timeout)* ausführen, Transaktion ausführen.

Der beschriebene Algorithmus hat gegenüber der Lock-Allokation durch das Lua-Skript den Nachteil eines zusätzlichen Roundtrips, selbst in Kombination mit Pipelining.

Die Verwaltung von Countern in einzelnen Keys hat den Vorteil, dass durch Expiration ein automatischer Garbage-Collection Mechanismus zur Verfügung steht, ähnlich dem Stable Bloomfilter und dem Active-Active Buffering. `CBloomFilterRedis` erlaubt es, bei der Instanziierung eine Expirationsdauer festzulegen, nach der die Counter automatisch freigegeben werden, falls sie nicht zwischenzeitlich geändert wurden. Dafür besitzt jeder Counter einen signifikanten Speicheroverhead von ca. 80-90 Byte. Wie die Berechnungen für Counting Bloomfilter gezeigt haben, benötigt der Counter praktisch jedoch höchstens eine Größe von höchstens 20 Bit. Ob der zusätzliche Speicheroverhead einen Nachteil darstellt, hängt von der Applikation ab - für die ORESTES-Server wäre es nicht problematisch. Für die ORESTES-Server bietet `CBloomFilterRedis` jedoch keine Vorteile gegenüber dem Bit-Array-basierten `CBloomFilterRedisBits`.

Für einen nicht-materialisierten Bloomfilter wäre die Verwaltung von Keys als Counter eine interessante Option. Sie würde es erlauben, Insert und Delete auf ein *INCR* und *DECR* Befehl in einer Pipeline zu reduzieren. Es gäbe zudem keinen „Hotspot-Key“ wie *bloomBits* mehr, alle Keys könnten durch Client Sharding oder Redis Cluster partitioniert werden. Da ein derartiger Counting Bloomfilter jedoch nicht das effiziente Abrufen des Bloomfilters erlaubt, ist er für unsere Problemstellung irrelevant.

3.7.1.4 Vergleich

Um zu prüfen, dass unsere Redis Bloomfilter tatsächlich nebenläufigkeitssicher auf denselben in Redis gespeicherten Daten arbeiten können, haben wir einen einfachen Test entworfen. Eine feste Anzahl von N Threads führt jeweils folgende Schritte auf einer Thread-lokalen Counting Bloomfilter Instanz aus:

1. Generiere ein zufälliges Element und füge es in den Counting Bloomfilter ein.
2. Prüfe, ob das Element enthalten ist. Falls nicht ⇒ Fehler.
3. Lösche das Element.
4. Prüfe, ob das Element enthalten ist. Falls ja ⇒ False Positive oder Fehler. Wiederhole ab 1.

Dadurch, dass die JVM Threads an beliebigen Stellen pausiert und einen Kontextwechsel zu einem der anderen Threads ausführt, wird bei ausreichenden Wiederholungen der kurze Algorithmus an fast allen denkbaren Stellen unterbrochen. Wenn dennoch keine Fehler, Endlosschleifen oder Deadlocks auftreten, ist die Wahrscheinlichkeit sehr hoch, dass der Bloomfilter nebenläufigkeitssicher ist. Sowohl `CBloomFilterRedis` als auch `CBloomFilterRedis-`

Bits bestehen diesen Test und erfüllen deshalb unsere Anforderung als eine hochperformante geteilte Datenstruktur.

Die Effektivität unserer Redis Bloomfilter lässt sich am besten exemplarisch in einem Benchmark belegen. Dazu führen wir eine einfache Benchmark-Prozedur aus:

1. Generiere ein zufälliges Element.
2. Prüfe, ob das Element im Bloomfilter enthalten ist, obwohl es noch nicht eingefügt wurde \Rightarrow False-Positive registrieren.
3. Füge das Element in den Counting Bloomfilter ein. Wiederhole *inserts* Mal ab 1.

CBloomFilterRedisBits		Grigorik Bloomfilter [178]	
<i>Konfiguration: m = 100.000, c = 4, k = 10, inserts = 30.000</i>			
False Positives: 2965		False Positives: 2979	
False-Positive Rate = 9.88		False-Positive Rate = 9.93	
real	0m9.297s	real	1m16.006s
user	0m2.216s	user	0m30.646s
sys	0m0.988s	sys	0m14.061s
6453.69 operations/s		789.474 operations/s	

Abbildung 66 Ergebnisse für einen exemplarischen Durchlauf des einfachen Benchmarks.

Die Ergebnisse für einen Durchlauf des Benchmarks sind in Abbildung 66 gezeigt. Wir vergleichen **CBloomFilterRedisBits** mit dem populären Counting Bloomfilter für Redis von Grigorik [178]. Da die Eingabe aus zufällig generierten Elementen besteht, treten für Implementierungen etwa die gleiche Anzahl an False-Positives auf. Der signifikante Unterschied liegt in der benötigten Ausführungsdauer – die Geschwindigkeit unserer Implementierung ist um knapp eine Größenordnung höher. Der Test wurde unter gleichen Bedingungen in einer VM mit Ubuntu Linux, einem virtuellen Core und 1.5 GByte RAM ausgeführt. Die Ausführungsgeschwindigkeit wird limitiert durch die Roundtrips, die in Schritt 2 und 3 des sequentiellen Benchmarks auftreten. D.h. Redis ist nicht der limitierende Faktor für die Geschwindigkeit des Benchmarks, sondern die Latenz einer Socket-Verbindung über das Loopback-Interface des Betriebssystems. Dies lässt sich leicht belegen, indem wir mit Pipelining den Durchsatz von einfachen Operationen wie GET, SET und INCR auf der gleichen VM messen:

```
SET: 303030.31 operations/s
GET: 333333.34 operations/s
INCR: 277777.78 operations/s
```

D.h. alle simplen Operationen können selbst mit geringen Hardware-Ressourcen mit einer Frequenz von 300.000 Requests pro Sekunde ausgeführt werden. Für unsere Redis Bloomfilter bedeutet dies, dass die Performance nur durch die Latenz eines Roundtrips beschränkt ist. Da unsere Redis Bloomfilter anders als die Implementierung Grigoriks Nebenläufigkeitssicherheit gewährleisten, können deshalb mehrere Threads bzw. Clients gleichzeitig in einen



Redis Bloomfilter Elemente einfügen und wieder lesen, um die Geschwindigkeit nahezu beliebig anzuheben. Darüber hinaus wollen wir diese Ergebnisse zum Anlass nehmen, eine *addAll*, *removeAll* und *containsAll* Methode zu implementieren, die auf Pipelining zurückgreift und deshalb nicht durch die Latenz eines Roundtrips eingeschränkt wird. Mit dem von uns verwendeten Java Redis Client (Jedis [183]) war dies bisher nicht möglich, da die (neuen) Lua-Befehle erst unvollständig in die API aufgenommen wurden.

Wir stellen also zusammenfassend fest, dass unsere Implementierung sehr gute Benchmarkergebnisse erzielt, da sie einerseits in Java umgesetzt ist und außerdem die *contains*, *add* und *remove* Methoden mit einem einzigen Roundtrip umsetzt. Die übrigen Vorteile durch Multi-Client Fähigkeit, konfigurierbare Hashfunktionen und die Bloomfilter-Materialisierung deckt der Benchmark nicht ab.

3.7.2 Hashing

Die False-Positive Rate eines Bloomfilters ist von der Güte der verwendeten Hashfunktion abhängig. Die mathematische Analyse des Bloomfilters geht von der Annahme aus, dass die Hashwerte der Hashfunktion gleichmäßig über $\{1, \dots, m\}$ verteilt sind. Das Maß, in dem verschiedene Hashfunktionen diese Forderung nach *Gleichverteilung* erfüllen, variiert sehr stark. Auch die Verteilung und die Datentypen der Eingabe haben entscheidenden Einfluss auf die Gleichverteilung der Ergebnisse. So ist beispielsweise das gleichmäßige Hashing von zufälligen Integer-Werten sehr viel leichter zu erreichen, als bei ähnlichen Strings.

Gemäß Knuth wurde die Idee des Hashing 1953 erstmals in einem internen IBM Memo von Hans Peter Luhn artikuliert [184]. Seitdem wurden viele verschiedene Hash-Algorithmen vorgestellt. Das Ermitteln guter Hashfunktionen ist noch immer ein sehr aktives Forschungsgebiet [145]. Neben dem Einsatz in hash-basierten Datenstrukturen wie dem Bloomfilter und Hashtabellen, kommt Hashing u.a. ebenfalls bei der Ähnlichkeitssuche (Locality Sensitive Hashing [185]), Substring Matching und Computergrafik (Geometric Hashing [186]) zum Einsatz. Für die Hashfunktion des Bloomfilters sind für ORESTES besonders zwei Eigenschaften entscheidend:

- Hohe Qualität der Hashwerte für Strings, da Objekt-IDs Strings sind.
- Auswahl zwischen verschiedenen Hashfunktionen mit unterschiedlichen Trade-offs. Beispielsweise ist die Wahl einer kryptographischen Hashfunktion für Szenarien, in denen die Clients Java benutzen, gut geeignet, da dort effiziente Implementierungen dieser Hashfunktion zu Verfügung stehen. Für mobile Clients oder Programmiersprachen wie JavaScript ist die Wahl einer performanten und leicht implementierbaren Hashfunktion geeigneter.

Als Beispiel für eine sehr einfache Hashfunktion führen wir die *hashCode* Implementierung von String in Java an (Abbildung 67). Die primäre Anwendung der *hashCode* Methode, die jedes Objekt in Java besitzt, ist die Unterstützung hash-basierter Collections, wie die Dokumentation dieser Methode suggeriert (cf. [187]):

This method is supported for the benefit of hashables such as those provided by `java.util.Hashtable`.

Die Hashfunktion ist sehr simpel. Sie benutzt die arithmetische Repräsentation jedes Bytes des Eingabestrings, um dieses mit der Primzahl 31 zu multiplizieren und auf das bisherige Ergebnis zu addieren. Die Auswahl von 31 in der Schleife hat mehrere Gründe. Zum einen hat die empirische Erfahrung mit dieser Hashfunktion gezeigt, dass sie in der Praxis gut funktioniert. Zum anderen lässt sich eine Multiplikation mit 31 effizient in Hardware ausführen, da $31 * i = (i \ll 5) - i$ gilt [187]. Für viele praktische Anwendung ist eine derart simple Hashfunktion deshalb eine effiziente Wahl. Es lässt sich jedoch leicht zeigen, dass `hashCode` Implementierung keineswegs eine gleichmäßige Verteilung der Hashwerte erzielt [188]. Seit einem jüngsten Update von Java (Java7u6) wird deshalb auch in Java eine aufwändigere Hashfunktion mit besserer Gleichverteilung angeboten [189].

```
int h = 0;
for (int i = 0; i < input.length(); i++) {
    h = 31 * h + input.charAt(i);
}
```

Abbildung 67 Die `hashCode` Implementierung von `String` in Java (cf. [187]).

Die Kollision von Hashwerten hat meist schwerwiegende Folgen für die Performance. Im Fall von Hashtabellen sind effiziente Kollisionsauflösungen mit linearer Laufzeit nötig, beispielsweise durch Überlaufbehälter. Für Bloomfilter resultieren Kollisionen in einer höheren False-Positive Rate. Generell sind neben der Effizienz die zwei wichtigsten Eigenschaften von Hashfunktionen:

- **Gleichverteilung.** D.h. eine Hashfunktion sollte Hashwerte so produzieren, dass alle Hashwerte mit gleicher Häufigkeit auftreten, d.h. $P(\text{hash}(x) = i) = \frac{1}{m} \forall i \in \{1, \dots, m\}$.
- **Determinismus.** Eine Hashfunktion muss für eine spezifische Eingabe stets denselben Hashwert erzeugen.

Um ein Element mit einem Hashwert in einer Hashtabelle oder einen Bloomfilter einzufügen, ist es erforderlich, dass der Wertebereich der Hashwerte mit den möglichen Indexpositionen der Datenstruktur übereinstimmt, d.h. $M = \{1, \dots, m\}$ für eine Hashfunktion $\text{hash}: N \rightarrow M$ und eine Datenstruktur mit m Slots. Fast alle Hashfunktionen generieren jedoch Hashwerte einer bestimmten Größe, z.B. 32 Bit. In dem Fall muss der Hashwert auf das Intervall $\{1, \dots, m\}$ reduziert werden. Dafür gibt es drei gängige Strategien:

1. **Reduktion durch Modulo Berechnung.** Die Hashwerte werden durch $\text{hash}(x) \% m$ auf den Wertebereich $\{1, \dots, m\}$ reduziert.
2. **Reduktion durch Shifts.** Wenn m eine Zweierpotenz ist, kann die Reduktion der Hashwerte durch Maskieren der höherwertigen Bits erreicht werden, d.h. $\text{hash}(x) \& (m - 1)$. Dies hat den Vorteil, dass Shifts sehr effizient sind gegenüber



Modulo Berechnungen, die meist ineffizient in Software durchgeführt werden müssen. Dieser Ansatz ist den Java Collections implementiert [187].

3. **Rejection Sampling.** Durch Rejection Sampling wird eine mathematisch korrekte Reduktion beliebiger Hashwerte auf das Intervall $\{1, \dots, m\}$ erreicht, d.h. wenn die Hashwerte auf ihrem Ursprungsintervall gleichverteilt waren, sind sie auch auf dem reduzierten Intervall $\{1, \dots, m\}$. Die Berechnung ist ineffizienter als die Brechnung durch Ansatz 1 und 2, erzielt aber die Hashwerte mit der höchsten Qualität.

Die erste Strategie ist in Praxis absolut dominierend. Alle Bloomfilter-Implementierungen, die wir in den vorangehenden Abschnitten untersucht haben, verwenden die Modulo-basierte Reduktion. Die Reduktion durch Shifts ist speziell, da sie voraussetzt, dass $m = 2^j$ mit $j \in \mathbb{N}$. Sie funktioniert gut, wenn die niederwertigen Bits der Hashwerte gleichmäßig verteilt sind. Für Strings mit Zeichen ist die `hashCode` Methode aus Java beispielsweise bis etwa zum 20ten Bit grob gleichverteilt und danach sehr ungleichmäßig [188].

Das Problem der dominierenden Modulo-basierten Reduktion auf das Intervall $\{1, \dots, m\}$ mit beliebigem m besteht darin, dass zuvor gleichverteilt Hashwerte aus $\{1, \dots, N\}$ anschließend nicht länger gleichverteilt sind. Dieser Effekt nimmt zu, je stärker m sich N nähert. Die Folge ist eine erhöhte False-Positive Rate des Bloomfilters, der seine mathematischen Garantien aufrecht erhalten kann (z.B., dass gilt $f \approx 0,6185^{\frac{m}{n}}$). Die Problematik rührt daher, dass N im allgemeinen kein Vielfaches von m ist. Dies sorgt dafür, dass eine Unregelmäßigkeit der Größe $N \% m * \lfloor \frac{N}{m} \rfloor$ entsteht. Ist beispielsweise $N = 2.000.000$ und $m = 1.500.000$, werden bei einer Reduktion der Hashwerte von $\{1, \dots, N\}$ auf $\{1, \dots, m\}$ die Werte aus $\{1, \dots, \lfloor \frac{m}{3} \rfloor\}$ doppelt so häufig auftreten wie die Werte aus $\{\lfloor \frac{m}{3} \rfloor, \dots, m\}$. Dies ist in Abbildung 68 visualisiert. Auf die False-Positive Rate hat der Effekt fatale Auswirkungen.

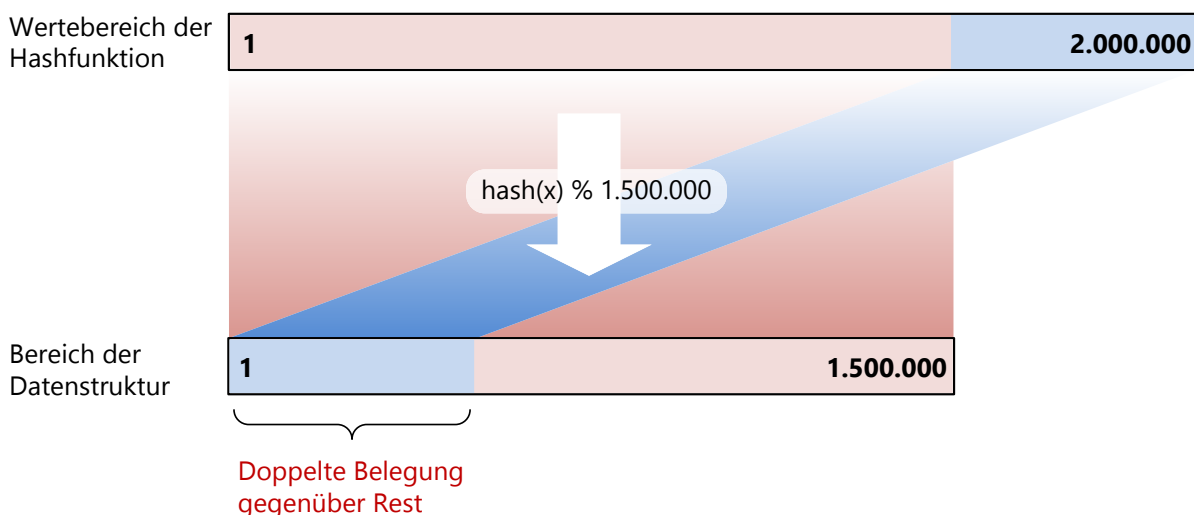


Abbildung 68 Das Problem der ungleichmäßigen Verteilung bei der Verwendung der Modulo Reduktion.

Das Problem der Modulo-Reduktion kann durch *Rejection Sampling* (deutsch *Verwerfungsmethode*) gelöst werden. Die intuitive Idee des Rejection Samplings ist in Abbildung 69 gezeigt:

anstatt die Ungleichmäßigkeit in Kauf zu nehmen, wird der Anteil von $\frac{N \% m}{N}$ Hashwerten, der eine Ungleichverteilung in $\{1, \dots, m\}$ verursacht, verworfen und stattdessen ein neuer Hashwert erzeugt. Um einen neuen Hashwert zu erzeugen, kann der zu hashende Wert x durch eine deterministische Funktion $update(x)$ aktualisiert werden, bevor erneut ein Hashwert gebildet wird. Dies ist eine spezielle Form des Rejection Sampling, die davon ausgeht, dass $hash(x)$ als eine gleichverteilte Zufallsvariable betrachtet werden kann. Die allgemeine Form des Rejection Sampling produziert auf ähnliche Weise F -verteilte Zufallszahlen unter Zuhilfenahme G -verteilter Zufallszahlen mit $k * g(x) < f(x)$ und kontinuierlich gleichverteilten Zufallszahlen [190]. Da wir jedoch aus einer diskreten gleichverteilten „Zufallsvariable“ aus $\{1, \dots, N\}$ eine gleichverteilte Zufallsvariable aus $\{1, \dots, m\}$ erzeugen wollen, ist die Rejection Sampling Prozedur sehr einfach:

1. Generiere einen Hashwert $hash(x)$ in $\{1, \dots, N\}$.
 - a. Wenn $hash(x) \leq \lfloor \frac{N}{m} \rfloor * m$ akzeptiere $hash(x)$.
 - b. Modifiziere x durch eine deterministische Funktion: $x := update(x)$. Fahre bei Schritt 1 fort.
2. Gib $hash(x) \% m$ als reduzierten Hashwert aus.

Damit das Rejection Sampling terminiert, muss $update(x) \neq x$ für alle möglichen Eingabewerte x gelten.

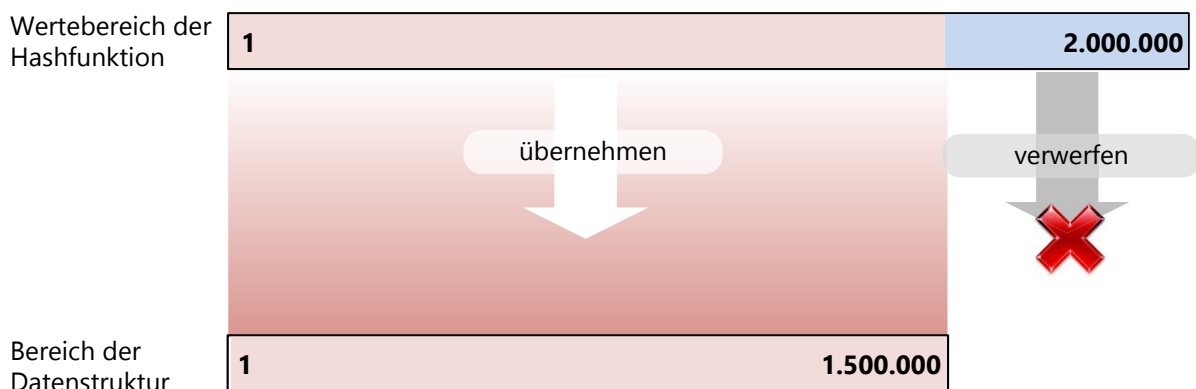


Abbildung 69 Lösung der ungleichmäßigen Verteilung durch Rejection Sampling.

Das Generieren von Hashwerten für Bloomfilter unterscheidet sich grundlegend von anderen Hash-basierten Datenstrukturen, da der Bloomfilter k unabhängige Hashwerte benötigt. Wie Broder et al. anmerken, ist die Wahl einer geeigneten Hashfunktion noch immer ein offenes Forschungsproblem [163]. Praktische Implementierungen von Bloomfiltern greifen i.d.R. auf eine Hashfunktion zurück, um mit ihr k Hashwerte zu erzeugen. Drei kombinierbare Strategien sind dabei denkbar:

1. **Zerlegung.** Der Hashwert der Größe X Bit wird in X/s Blöcke der Größe s Bit zerlegt, wobei gelten muss, dass $2^s \geq m$. Jeder Block der Größe s wird als ein unabhängiger



Hashwert betrachtet. Falls $X/s < k$, muss ein zusätzlicher Hashwert generiert werden, z.B. mit der zweiten Strategie oder einer weiteren Hashfunktion.

2. **Modifizierung.** Durch eine deterministische Modifizierung des zu hashenden Wertes können mit einer Hashfunktion k Hashwerte modifiziert werden. Dazu wird ein Element x in k Varianten gehasht. Beispielsweise kann nach der Berechnung von $hash(x)$ die nächste Eingabe als $x := hash(x) \text{ xor } x$ oder $x := x + 1$ bestimmt werden.
3. **Linearkombination.** Mitzenmacher et al. [170] haben gezeigt, dass statt k Hashfunktionen auch k Linearkombinationen $hash_1(x) + i * hash_2(x)$ mit $i \in \{1, \dots, k\}$ zweier Hashfunktion verwendet werden können. Sie konnten für perfekte Hashfunktion die asymptotisch gleiche False-Positive Rate wie für k unabhängige Hashfunktionen nachweisen. In der Praxis leidet die False-Positive Rate jedoch sehr unter dieser Optimierung [171].

In Kombination mit Rejection Sampling müssen statt k Hashwerten eine nicht vorher bekannte Anzahl $h \geq k$ Hashwerte bestimmt werden, wenn $h - k$ Hashwerte zur Erhaltung der Gleichverteilung abgelehnt werden. In unserer Implementierung benutzen wir eine Kombination aus Zerlegung und Modifizierung, d.h. ein großer Hashwert, z.B. ein SHA-512 Hash mit 512 Bit, wird in Blöcke zerlegt und die einzelnen Blöcke als Hashwerte betrachtet und ggf. durch Modifizierung ein neuer SHA-512 Hash gebildet. Die Zerlegung ist gerechtfertigt, da bei gleichverteilten Hashwerten alle Bits unabhängig voneinander eine Wahrscheinlichkeit von $1/2$ haben, 0 oder 1 zu sein. Die Modifizierung ist gerechtfertigt, wenn die Hashfunktion einen guten *Avalanche Effekt* besitzt (auch *No-Funnels* Eigenschaft genannt) [145], [179], [191]. Der von Webster et al. [192] eingeführte Avalanche Effekt besagt, dass die Änderung von einem Bit in der Eingabe einer Hashfunktion zu einem völlig unterschiedlichen Ergebnis führen muss. Ein perfekter Avalanche Effekt liegt dann vor, wenn jedes Bit des Hashwerts eine Wahrscheinlichkeit von $1/2$ hat, zu kippen, wenn ein Bit der Eingabe geändert wird. Insbesondere für kryptographische Hashfunktionen ist ein guter Avalanche Effekt eine Voraussetzung. Bei einem perfekten Avalanche Effekt ist die Modifizierung $x := x + 1$ ausreichend, um vollständig unabhängige Hashwerte zu erzeugen.

In unserer Bloomfilter-Umsetzung wenden wir Rejection Sampling zur Erhaltung der Gleichverteilung in Kombination mit Modifizierung und Zerlegung an. Im Folgenden wollen wir die implementierten Hashfunktionen beschreiben und stochastisch analysieren. Tabelle 7 zeigt die verfügbaren Hashfunktionen im Vergleich zu den übrigen untersuchten Bloomfilter-Implementierungen. Es wird deutlich, dass diese Implementierungen sehr beschränkte Umsetzungen des Hashings bieten. Zwar verwenden HBase und Cassandra die leistungsfähige Murmur Hashfunktion, generieren die k Hashwerte jedoch durch Linearkombinationen. Keine der Implementierungen verwendet Rejection-Sampling.

Die stochastische Analyse, die wir für die Hashfunktionen durchführen, untersucht, welche der Hashfunktionen eine mathematisch nachweisbare Gleichverteilung erzeugen. Für Hashfunktionen, deren Gleichverteilung sich mathematisch nachweisen lässt, gelten die Garantien des Bloomfilters, insbesondere die Größe der False-Positive Rate. Diese ist für ORES-

TES entscheidend, da jeder False-Positive unnötige Last auf dem Server erzeugt und einer hohen Netzwerklatenz ausgesetzt ist.

Bloomfilter-Implementierung	Typ der Hashfunktion	Hashfunktion	Erzeugung
Squid	kryptographisch	MD5	statische Zerlegung in $k = 4$ Blöcke
HBase	nicht-kryptographisch	Murmur	Linearkombination
Cassandra		Jenkins	
Magnus Skjestad	kryptographisch	MD5	Zerlegung und Modifizierung
Ilja Grigorik	Prüfsumme	CRC32	Modifizierung
ORESTES	kryptographisch	MD2	Zerlegung, Modifizierung, Rejection Sampling
		SHA-1	
		MD5	
		SHA-256	
		SHA-384	
	SHA-512		
	nicht-kryptographisch	Java Secure RNG	
	Pseudorandom Number Generator	Murmur	
FNV			
Java Random, LCG			
Universelle Hashfunktion	Carter-Wegman		
Prüfsumme	CRC32		
	Adler32		

Tabelle 7 Unterstützte Hashfunktionen unterschiedlicher Bloomfilter-Implementierungen.

Bevor wir die Gleichverteilung stochastisch untersuchen, führen wir die Hashfunktionen aus Tabelle 7 knapp ein.

3.7.2.1 Kryptographische Hashfunktionen

Kryptographische Hashfunktion sind besonders starke Hashfunktionen mit zahlreichen Anwendungen in der Kryptographie für verschiedene Formen der Authentifikation (z.B. digital Signaturen). Drei Forderungen müssen kryptographische Hashfunktionen erfüllen [145]:

- **Preimage Resistance.** Gegeben einen Hashwert $hash(x)$, muss es praktisch unmöglich sein, das Urbild x (Preimage) zur ermitteln, das diesen Hashwert erzeugt hat.
- **Second-Preimage Resistance.** Gegeben einen Hashwert $hash(x_1)$, muss es praktisch unmöglich sein, ein zweites Urbild x_2 zu ermitteln, das denselben Hashwert hat.
- **Collision Resistance.** Es muss praktisch nahezu unmöglich sein, zwei Werte x_1 und x_2 zu finden, die denselben Hashwert $hash(x_1) = hash(x_2)$ besitzen (Kollisionsfreiheit).



Die erste Forderung ist lediglich für kryptographische Anwendungen interessant. Die beiden Forderungen nach Kollisionresistenz spiegeln jedoch die Anforderung der Gleichverteilung wider – nur wenn die Hashwerte gleichverteilt sind, ist es in allen Fällen schwierig, zu einem gegebenen Hashwert zwei Urbilder oder für zwei Urbilder denselben Hashwert zu erzeugen. Wie die meisten leistungsfähigen Hashfunktionen benutzen kryptographische Hashfunktion meist die Merkle–Damgård Konstruktion [145], die nach folgendem Prinzip funktioniert:

1. Initialisiere einen internen Zustand $State = State_0$.
2. Für jeden Block B der Eingabedaten wiederhole:
 - Wende eine Kompressionsfunktion an, um den Zustand durch den gelesenen Block zu aktualisieren $State := Reduce(State, B)$.
3. Wende eine Finalisierungsfunktion an, um aus dem Zustand einen Hashwert zu erzeugen $result := finalize(State)$.

Kryptographische Hashfunktionen sind unterschiedlichsten praktischen Angriffen ausgesetzt, weshalb viele kryptographische Hashfunktionen sehr gut untersucht sind. Der Nachteil kryptographischer Hashfunktionen besteht einzig darin, dass die Berechnungen sehr kompliziert und damit zeitintensiv sein müssen, um eine solide *Preimage Resistance* zu erreichen. In den ORESTES Bloomfiltern erlauben wir die Wahl einer beliebigen Hashfunktion, die durch die Java Security Extension unterstützt wird (MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512). Der Algorithmus für die Generierung von k Hashwerten ist in Abbildung 70 gezeigt. Der Algorithmus benutzt für die Modifizierung eine deterministische Sequenz aus Pseudozufallszahlen, um die Eingabe zu aktualisieren. Für die meisten Hashfunktion wäre eine Aktualisierung durch einen einfachen Schleifenzähler aufgrund des guten Avalanche Effekts ebenfalls möglich. Diese Lösung zeigt jedoch bessere Ergebnisse für mittelmäßige oder alte Hashfunktionen.

```
def hashCryptographic(bytes, hashFunction):
    //Leere Liste für k Hashwerte, Zähler für berechnete Hashwert
    hashes = [], computedHashes = 0
    //Sequenz aus Pseudofallszahlen für die Modifikation
    initialisiere salt = neue Pseudozufallssequenz(startWertKonstante)
    while computedHashes < k:
        //Hashwert mit kryptographischer Hashfunktion berechnen
        hash = hashFunction(bytes + salt)
        Für jeden Block B mit  $2^{bits(B)} > m$ :
            //Wert nur akzeptieren, wenn die Gleichverteilung erhalten bleibt
            accepted = RejectionSample(B)
            if accepted:
                hashes.append(B % m)
                computedHashes++
        //Modifikation für nächste Runde durchführen
        salt = nächste Pseudozufallszahl
```

Abbildung 70 Nutzung kryptographischer Hashfunktionen für Bloomfilter.

3.7.2.2 Nicht-kryptographische Hashfunktionen

Wird die Forderung nach Preimage Resistance aufgegeben, lassen sich Hashfunktionen effizienter implementieren, ohne dabei die Gleichverteilung aufzugeben. In der Praxis besitzen zwei sehr einfache Hashfunktionen eine enorme Verbreitung: die Multiplikationsmethode und die Divisionsmethode. Sie sind sehr einfach zu implementieren, beispielsweise besteht die Divisionsmethode nur aus einer Berechnung (cf. [145]):

$$\text{hash}(\text{bytes}) := \text{asNumber}(\text{bytes}) \% m$$

wobei m vorzugsweise als Primzahl gewählt wird. Trotz ihrer großen Verbreitung sind Divisions- und Multiplikationsmethode für viele performancekritische und anspruchsvolle Anwendung, aufgrund ihrer ungleichmäßigen Verteilung ungeeignet. Im Laufe der letzten 10 Jahre wurden verschiedenste nicht-kryptographische Hashfunktionen vorgeschlagen. Eine umfassende, vergleichende Analyse ihrer statistischen Güte wurde bisher jedoch nur vereinzelt durchgeführt. Henke et al. [193] führen z.B. eine Untersuchung effizient in Hardware implementierbarer Hashfunktionen durch und testen ihre Gleichverteilung und ihren Avalanche Effekt. Die kryptographischen Hashfunktionen MD5 und SHA zeigen in den Ergebnissen das beste Verhalten, sind aber um zwei Größenordnungen langsamer als die nicht-kryptographischen Hashfunktionen. Zwei nichtkryptographische Hashfunktionen von Robert Jenkins [191] schneiden mit vergleichbar guten Ergebnissen ab.

Eine sehr junge Hashfunktion von 2008 mit sehr guten Ergebnissen für Avalanche- und Gleichverteilungstests ist Murmur von Austin Appleby [194]. Wir haben diese Hashfunktion integriert und durch das Modifikationsschema auf das Generieren von k Hashfunktionen erweitert. Auf die Zerlegung haben wir verzichtet, da die Hashwertberechnung so schnell ist, dass dieser Schritt eher zu einem unnötigen Overhead führen würde. Intern arbeitet der Algorithmus nach der Merkle–Damgård Konstruktion und führt insgesamt nur sehr wenige Shifts und Multiplikationen aus.

Für die Reduktion von beliebigen Byte-Arrays auf einen 32-Bit Integer verwenden wir eine weitere nicht-kryptographische Hashfunktion, Fowler–Noll–Vo (FNV) [191], die wir von C nach Java portiert haben. Java bietet für die Reduktion von Byte-Arrays auf 32 Bit Integer zwar eine Methode (*Arrays.hashCode*), doch ist diese erstens nicht so erprobt in Bezug auf eine gute Gleichverteilung wie FNV [191] und zweitens schlechter auf andere Programmiersprachen portierbar. Die Reduktion eines Byte-Arrays auf einen Integer wird benötigt, da einige Hashfunktionen (z.B. LCG) als Eingabe eine 32-Bit-Zahl erwarten.

3.7.2.3 Zufallsgeneratoren

Zufallsgeneratoren (*Pseudo Random Number Generators*, RNGs) können auch als Hashfunktionen eingesetzt werden. Für sie gilt wie für Hashfunktionen, dass die Sequenz der von ihnen generierten Pseudozufallswerte gleichmäßig über den gesamten Wertebereich verteilt sein sollte. In unserer Bloomfilter-Implementierung können die RNGs von Java genutzt werden: Java Random und Java Secure Random. Java Secure Random basiert wie viele RNGs mit



starken Forderungen bezüglich einer gleichmäßigen Verteilung auf kryptographischen Hashfunktionen, im Fall von Secure Random i.d.R. auf SHA-1 Hashes. Java Random basiert hingegen auf der verbreitetsten Form von RNGs, sogenannten *Linear Congruential Generators* (LCGs). Sie ähneln der Divisionsmethode, produzieren aber Zufallszahlen, die für viele praktische Anwendungen absolut adäquat sind. LCGs produzieren eine Sequenz x_0, x_1, \dots, x_k auf Basis einer einfachen Rekurrenzgleichung [184]:

$$x_{i+1} = (a * x_i + b) \% m$$

wobei ein Wert x_0 als Initialisierungsvektor (*seed*) gewählt werden muss. Während für Zufallszahlen als Initialisierung häufig der aktuelle Timestamp dient, muss für die Hashwertberechnung eine deterministische Funktion des zu hashenden Wertes *input* als Initialisierung verwendet werden, d.h. $x_0 = f(input)$. Die Konstanten a und b können gewählt werden. Die Güte eines LCGs ist jedoch empfindlich von der Wahl der Konstanten abhängig. Jeder LCG ist ab einem bestimmten Wert $j > J$ periodisch, d.h. $x_j = x_{j+P}$, wobei P als Periode bezeichnet wird [145]. Die Periode kann maximal $P = m$ betragen. Damit ein LCG die volle Periode besitzt, muss für die Konstanten a und b gelten [145]:

1. b und m sind relativ prim (teilerfremd).
2. $a - 1$ ist durch alle Primfaktoren von m teilbar.
3. $a - 1$ muss ein Vielfaches von 4 sein, wenn m ein Vielfaches von 4 ist.

Wir haben einen einfachen LCG implementiert, der die gut erprobten Konstanten von Java benutzt ($a = 25214903917, b = 11, m = 2^{48} - 1$) und so einfach ist, dass er leicht in andere Programmiersprachen portiert werden kann.

3.7.2.4 Universelle Hashfunktionen

Eine spezielle Klasse von Hashfunktionen stellen die sogenannten universellen Hashfunktionen dar. Sie beruhen auf der Idee, dass selbst bei ungünstig verteilten Eingabedaten mit einer randomisiert parametrisierten Hashfunktion im Durchschnitt ein gutes Ergebnis erzielt werden kann. Sei H eine endliche Menge von Hashfunktionen, die den Definitionsbereich U und den Wertebereich $\{1, \dots, m\}$ besitzen. H heißt universell, wenn für jedes eindeutige Paar von Keys k, l die Zahl der Hashfunktionen $hash \in H$ mit einer Kollision $hash(k) = hash(l)$ höchstens $|H|/m$ ist [158]. Für universelle Hashfunktionen lassen sich interessante Eigenschaften beweisen. So gelten bei zufälliger Auswahl von $hash \in H$ und einer beliebigen, gegebenen Menge von n Keys drei Eigenschaften [158]:

1. Für je zwei Keys k und l ist die Wahrscheinlichkeit einer Kollision kleiner $1/m$.
2. Für jeden Key ist die erwartete Anzahl an Keys mit demselben Hashwert $< 1 + n/m$.
3. Der Erwartungswert für die Anzahl kollidierender Keys ist durch $n * (n - 1)/(2m)$ beschränkt.

Diese Eigenschaften sind nicht an bestimmte Verteilungen der Keys geknüpft, sondern gelten aufgrund des zufälligen Ziehens einer universellen Hashfunktion immer. Eine bekannte

Familie von universellen Hashfunktionen wurde von Carter und Wegman vorgeschlagen [158]:

$$\text{hash}_{a,b}(x) := ((a * x + b) \% p) \% m$$

Dabei ist p eine Primzahl größer n und a und b wählbare Parameter aus $\{0, \dots, p - 1\}$. Wir haben die universelle Carter-Wegman Hashfunktion implementiert. Dabei interpretieren wir das zu hashende Byte-Array als große natürliche Zahl und initialisieren mit ihr die Parameter a und b , die für die Berechnung der k Hashwerte deterministisch durch einen LCG aktualisiert werden.

3.7.2.5 Prüfsummen

In viele Anwendungen werden Prüfsummen als Hashfunktionen eingesetzt. Wir unterstützen zwei der bekanntesten: CRC32 und Adler32. Beide Hashfunktionen sind darauf optimiert, Korruptionen von Nutzdaten zu erkennen. Einer Anforderung nach Gleichverteilung genügen sie nicht, darüber hinaus ist der Avalanche Effekt nahezu nicht vorhanden, da Änderungen der Eingabe nur ganz spezifische Auswirkungen auf die Bits der Ausgabe besitzen [179]. Während CRC32 (Cyclic Redundancy Check) auf Polynomdivision beruht und in vielen Standards (z.B. Ethernet) verankert ist, beruht Adler32 (nach Mark Adler) auf einer wesentlich einfacheren Berechnungsvorschrift, die explizit der Geschwindigkeit den Vorzug gegenüber der Zuverlässigkeit gibt. Beide Algorithmen lassen sich effizient in Hardware implementieren.

3.7.2.6 Hash Qualität

Wir überprüfen die Qualität der beschriebenen Hashfunktionen durch einen χ^2 Test (Chi-Quadrat Test). Der Chi-Quadrat Test ist ein etabliertes Verfahren um einen *Goodness of Fit* Test (Anpassungstest) durchzuführen, d.h. zu überprüfen, ob eine Stichprobe, bei der für die N möglichen Ergebnisse jedes Versuchs die Häufigkeiten C_0, \dots, C_{n-1} beobachtet wurden, einer Verteilungsannahme widerspricht. Bei T Versuchen ist die erwartete relative Häufigkeit f_i für jedes mögliche Ergebnis O_i $f_i = C_i/T$. Das Gesetz der großen Zahlen sichert zu, dass der Grenzwert von f_i zu der tatsächlichen Wahrscheinlichkeit konvergiert:

$$\lim_{T \rightarrow \infty} \frac{C_i}{T} = P(O_i) = q_i$$

Die χ^2 Statistik ist ein quadratisches Maß für die Abweichung der beobachteten Häufigkeiten von den tatsächlichen Häufigkeiten [145]:

$$\chi_N^2 = \sum_{i=0}^{N-1} \frac{(C_i - T * q_i)^2}{T * q_i} = \sum_{i=0}^{N-1} \frac{T}{q_i} (f_i - q_i)^2$$

Wählt man die Stichprobe (also die Anzahl zu prüfender Hashwerte) sehr groß, strebt jeder Summand entweder gegen 0 oder ∞ [145]:

$$\lim_{T \rightarrow \infty} f_i - q_i = \begin{cases} 0 & \text{wenn die Stichprobe der angenommenen Verteilung gehorcht} \\ \infty & \text{sonst} \end{cases}$$



Karl Pearson konnte zeigen, dass die Verteilungsfunktion von χ_N^2 existiert und die beobachteten Häufigkeiten $N - 1$ Freiheitsgrade haben, da gilt:

$$\sum_{i=0}^{N-1} C_i = T$$

Es existiert deshalb genau eine Randbedingung. Die Chi-Quadrat Verteilung gibt an, welcher Verteilung die Chi-Quadrat Statistik gehorcht und ist einzig davon abhängig, wie viele Freiheitsgrade das untersuchte Problem besitzt:

$$P(\chi^2 < x) = F(x) = \frac{2^{-v/2} e^{-x/2} x^{-1+v/2}}{\Gamma\left(\frac{v}{2}\right)}$$

Dabei ist $v = N - 1$ die Anzahl der Freiheitsgrade und Γ die Gammafunktion (eine Erweiterung der Fakultätsfunktion auf reelle Zahlen). Ein Beispiel für die Chi-Quadrat Verteilung ist in Abbildung 71 für drei verschiedene Freiheitsgrade gezeigt. Die Verteilung hat die blaue Form für die Freiheitsgrade 1 und 2, für größere Freiheitsgrade nimmt sie die Form der gelben und violetten Kurve an.

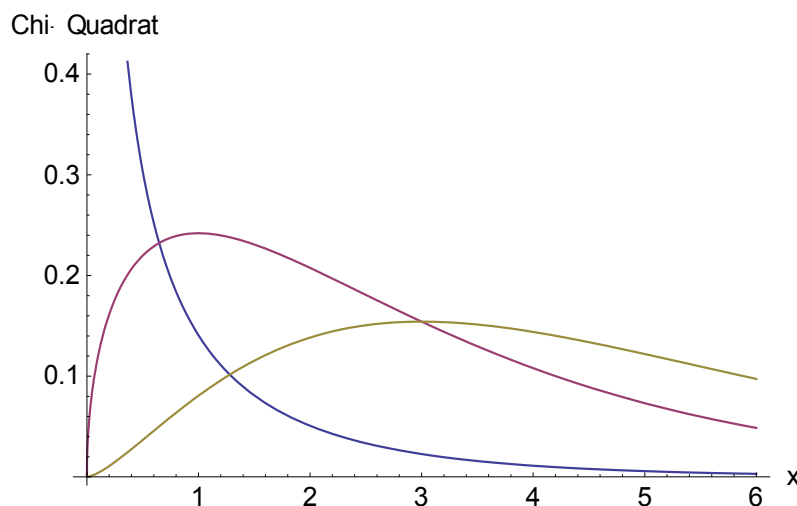


Abbildung 71 Die Chi-Quadrat Verteilung für drei verschiedene Freiheitsgrade.

Um mit der Chi-Quadrat Verteilung einen Hypothesentest auf das Zutreffen einer Verteilungsannahme vorzunehmen, ist folgendes Vorgehen nötig:

1. Hypothesenformulierung:
 - a. H_0 , die Nullhypothese besagt, dass die beobachteten Häufigkeiten der erwarteten Wahrscheinlichkeitsverteilung genügen.
 - b. H_1 , die Alternativhypothese besagt, dass die beobachteten Häufigkeiten **nicht** der erwarteten Wahrscheinlichkeitsverteilung genügen.
2. Festlegung des Signifikanzniveaus α auf z.B. 0,05 und der Freiheitsgrade v .
3. Durchführung des Experiments, d.h. Bestimmung der beobachteten Häufigkeiten und der Chi-Quadrat-Statistik.

4. Bestimmung des kritischen Werts x_{crit} mit $P(\chi_v^2 \geq x_{crit}) = \alpha$ unter mithilfe der Quantile der Chi-Quadrat Verteilung.
5. Wenn die Chi-Quadrat-Statistik größer als x_{crit} ist, wird die Nullhypothese H_0 zurückgewiesen, d.h. es gibt einen statistisch signifikanten Grund davon auszugehen, dass die beobachteten Werte **nicht** der vermuteten Verteilung entstammen.

Im Kontext der Hashwertberechnung gilt es also, N Hashwerte aus $\{1, \dots, m\}$ zu berechnen und mit dem Ergebnis einer Gleichverteilung zu vergleichen (Tabelle 8). Aus diesen Ergebnissen kann die Chi-Quadrat Statistik berechnet werden. Unsere Hypothesen sind dabei:

1. H_0 , die Nullhypothese, besagt, dass die Hashwerte einer diskreten Gleichverteilung über $\{1, \dots, m\}$ genügen.
2. H_1 , die Alternativhypothese, besagt, dass die beobachteten Hashwerte **nicht** einer Gleichverteilung genügen.

Beobachtete Häufigkeiten	Erwartete Häufigkeiten
Häufigkeit von $hash(x) = 1$	$1/m * N$
Häufigkeit von $hash(x) = 2$	$1/m * N$
...	
Häufigkeit von $hash(x) = m$	$1/m * N$

Tabelle 8 Bestimmung der Häufigkeit von Hashwerten.

Wir bestimmen anschließend den kritischen Wert x_{krit} . Ist die Hashfunktion nicht gleichverteilt – sie besitzt ein *Bias* – so wird die Chi-Quadrat Statistik in den meisten Fällen größer als x_{krit} sein und wir können die Nullhypothese zurückweisen. Der Beweis sagt aus, dass eine Gleichverteilung nur mit einer Wahrscheinlichkeit kleiner α einen so schlechten Wert für die Chi-Quadrat Statistik erbracht hätte. Wir bestimmen außerdem den sogenannten p-Wert, der angibt, wie groß genau die Wahrscheinlichkeit ist, dass eine Gleichverteilung eine mindestens so schlechte Chi-Quadrat-Statistik ergibt wie die geprüfte Hashfunktion. Der p-Wert ist also die bedingte Wahrscheinlichkeit:

$$pValue = P(\chi^2 > \chi_{Hashfunktion}^2 | H_0)$$

Wenn der p-Wert kleiner als α ist, ist dies analog zu einer Überschreitung des kritischen Werts und die Nullhypothese kann verworfen werden. Es ist wichtig zu beachten, dass die Nullhypothese nur verworfen werden kann. Es gibt keinen Weg, um sie zu beweisen, es kann lediglich keinen ausreichenden Grund für ihre Ablehnung geben.

Häufig werden Chi-Quadrat Tests nur für eine Serie an Beobachtungen ausgeführt. Dies führt jedoch dazu, dass eine korrekte Nullhypothese mit einer Wahrscheinlichkeit α zu Unrecht zurückgewiesen wird. Wir erweitern deshalb das Test-Schema und führen für jede Hashfunktion 100 unabhängige Chi-Quadrat Tests durch. Die Erwartung für eine gute Hashfunktion ist dann, dass der Test in etwa 5 Fällen die Nullhypothese zurückweist und in 95 Fällen kein ausreichender Grund für eine Zurückweisung vorliegt. Für schlechte Hashfunk-



tion erwarten wir hingegen, dass der Test in deutlich mehr als 5 Fällen zu einer Ablehnung führt.

Wir haben Chi-Quadrat Tests für verschiedene Konfigurationen von m, k und N durchgeführt. Wir können hier jedoch nicht alle Ergebnisse zeigen und beschränken uns deshalb auf einige Beispiele. Zur Generierung der Hashdaten haben wir 5 verschiedene Klassen von Eingabedaten getestet, die sich an dem erwarteten Use-Case für ORESTES orientieren, dem Einfügen von Objekt-IDs:

- **Random Strings.** Zufällige 128 Bit String (generiert mit einem LCG).
- **Random Integers.** Zufallszahlen (generiert mit einem LCG).
- **Increasing Integers.** Fortlaufend ansteigende Integer.
- **Increasing Ids.** Konkatenation aus einem festen String und einer fortlaufenden Zahl.
- **Random Bytes.** Zufällige 200-Byte-Arrays (generiert mit einem LCG).

In Abbildung 72 ist das Ergebnis einer Konfiguration für *Random Strings* gezeigt. Der Box-Whisker Plot enthält die Information über die Chi-Quadrat Statistik aller 100 Tests. Die gesamte Spannweite der Statistik ist durch die grauen Linien angegeben. Die Box enthält jeweils den Bereich vom 25% bis zum 75% Quantil. Die weiße Linie in der Mitte markiert den Median. Die gestrichelte rote Linie gibt den kritischen Wert von χ^2 für $\alpha = 0.05$ an.

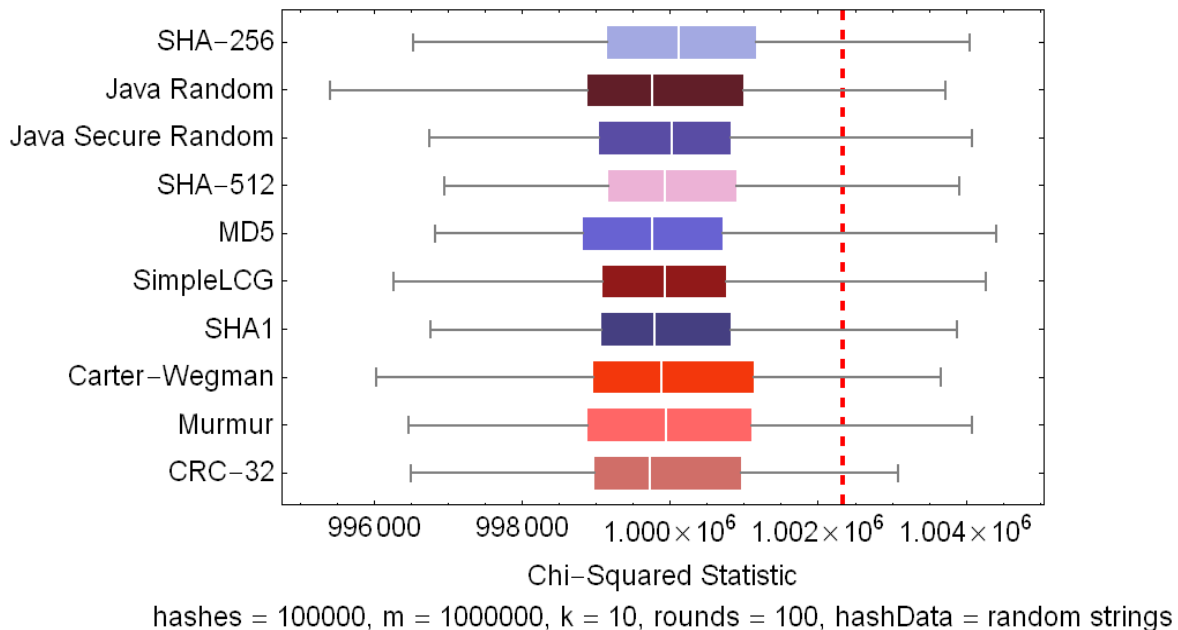


Abbildung 72 Ergebnisse für den Chi-Quadrat Test bei Random Strings.

Die korrespondierenden p-Werte sind in Abbildung 73 gezeigt. Alle Hashfunktionen zeigen sehr gute Ergebnisse – die Nullhypothese wird tatsächlich nur in etwa 5% aller Fälle zurückgewiesen. Die guten Ergebnisse sind jedoch primär darauf zurückzuführen, dass bereits die Eingabe zufällig verteilt war. In einem solchen Szenario erzeugt fast jede denkbare Hashfunktion eine ausreichend gleichverteilte Ausgabe. Eine Ausnahme bildet die Adler32 Funktion, die wir im Zuge der Lesbarkeit entfernen mussten. Selbst für Random Integers hat die

Adler32 Funktion eine ungleiche Verteilung, so dass der Chi-Quadrat Test die Nullhypothese zurückweist, die Funktion also mit hoher Signifikanz keine gleichverteilten Hashwerte erzeugt. Abbildung 74 zeigt die Ergebnisse von Adler-32 im Vergleich zu Java Secure Random. Die Ablehnung der Nullhypothese erfolgt mit sehr hoher Signifikanz, d.h. der p-Wert ist nahezu 0 für jeden der 100 Durchläufe.

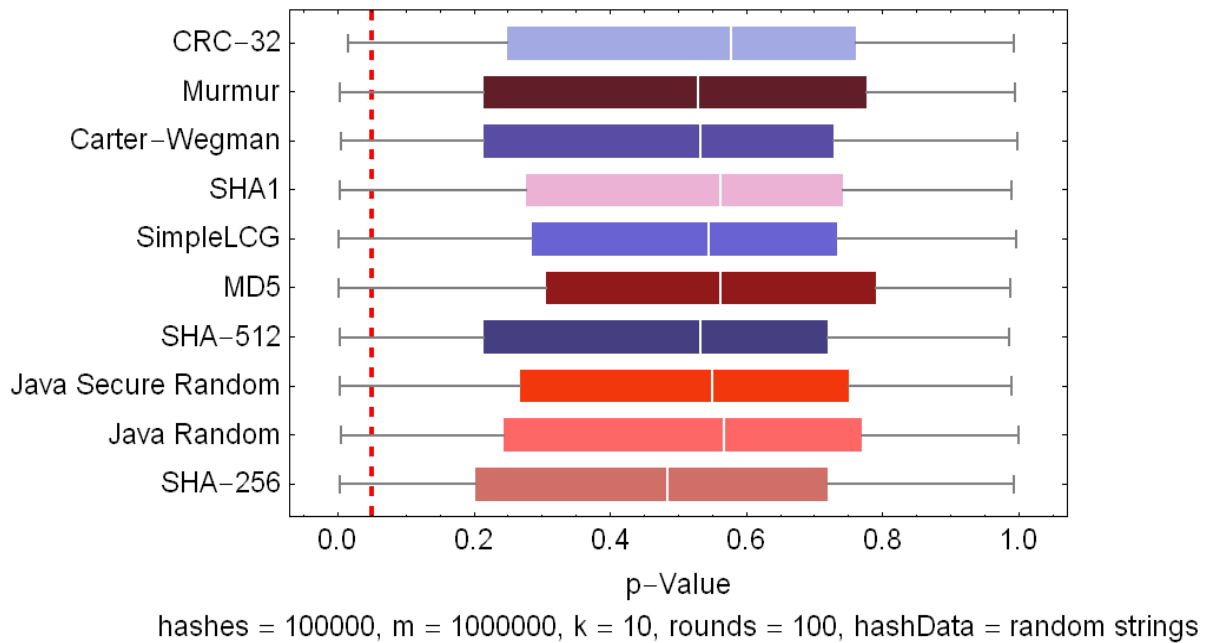


Abbildung 73 Die p-Werte für den Chi-Quadrat Test bei Random Strings.

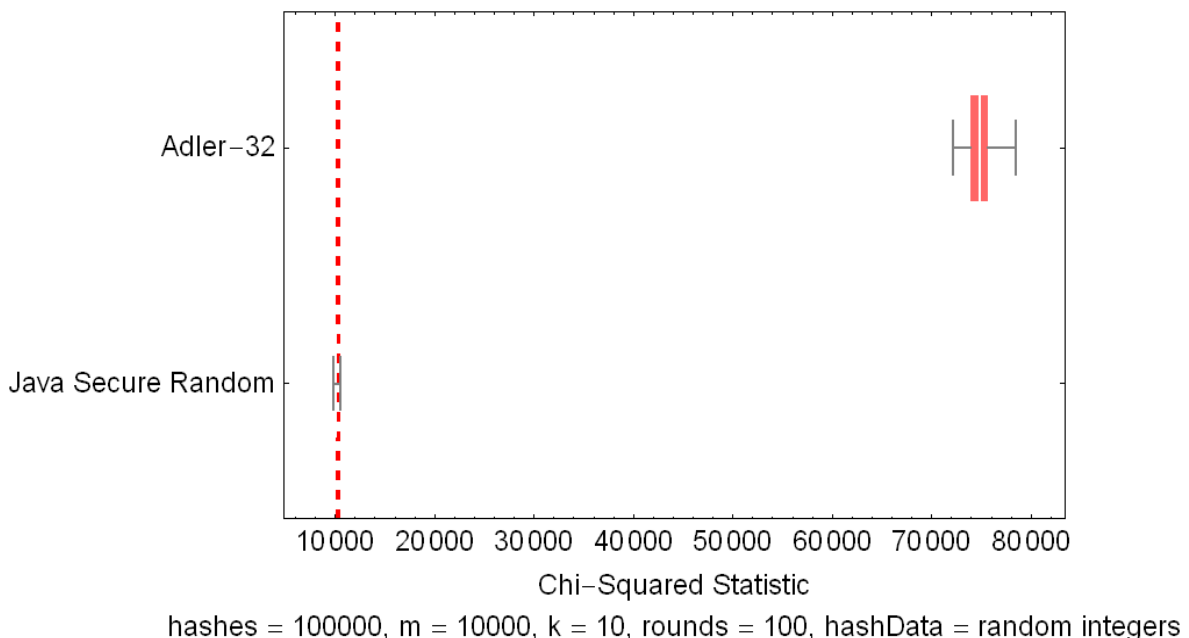


Abbildung 74 Die deutlich ungleichmäßige Verteilung von Adler-32.

Ein weiteres interessantes Ergebnis ist in Abbildung 75 für *Increasing Integers* gezeigt. Es zeigt eine Anomalie des einfachen LCGs. Dieser besteht jeden der 100 Tests deutlich. Dies ist ein Indiz dafür, dass die Ausgabewerte sehr gleichverteilt sind – womöglich zu gleichver-



teilt. Ein perfekter Zufallsgenerator hätte dieses Ergebnis nicht erzielt, der Test wäre in 5% aller Fälle fehlgeschlagen. Wir vermuten, dass der Grund hierfür in dem schlechten Avalanche Effekt von FNV liegt. Bevor eine Eingabe (stets ein Byte-Array) mit dem LCG gehasht wird, erzeugt FNV aus der Eingabe einen Integer. Da FNV nachgewiesenermaßen zwar eine recht gut Gleichverteilung besitzt (nachgewiesen durch Chi-Quadrat-Tests), aber einen sehr schlechten Avalanche-Effekt [179], ist es plausibel, dass die Eingabe aus ansteigenden IDs dafür sorgt, dass FNV ebenfalls ansteigende Werte zurückgibt. Der LCG mit voller Periode würde diese Werte mit modularer Arithmetik annähernd sukzessive auf die m verfügbaren Hashwerte abbilden. Dabei werden die Hashwerte durch die sukzessive Belegung gleichmäßiger belegt, als dies bei einer zufälligen Auswahl der Fall wäre. Deshalb ist das Ergebnis dieses wiederholten Chi-Quadrat Tests ein Hinweis für eine Verteilung, die nicht einer Gleichverteilung entspricht, ohne dass der Hypothesentest in der Lage ist, dies durch eine Ablehnung der Nullhypothese zu bestätigen.

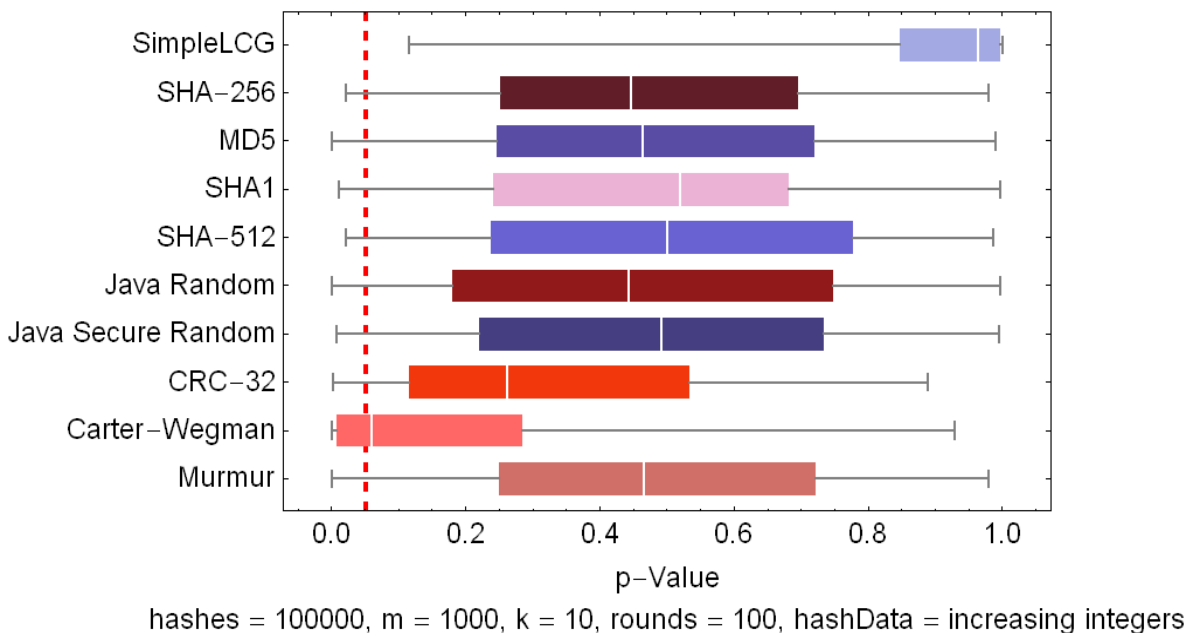


Abbildung 75 Ergebnis für Increasing Integers.

Ein weiteres wichtiges Ergebnis der Tests ist die Erkenntnis, dass die Vernachlässigung von Rejection Sampling nur dann zu ungleichmäßig verteilten Hashwerten führt, wenn die Anzahl möglicher Hashwerte m sehr dicht an der Größe der zu reduzierenden Hashwerte ist. Für ausreichend große Hashwerte, z.B. 32 oder 64 Bit, kann auf Rejection Sampling verzichtet werden, ohne die Gleichverteilung der Hashwerte merklich zu beeinträchtigen. Dies ist in Abbildung 76 gezeigt. Es sind die Ergebnisse der Hashfunktion aus der Bloomfilter Implementierung von Skjegstad (kein Rejection Sampling) und die Ergebnisse für Java Secure Random (Rejection Sampling) gezeigt. Die Implementierung Skjegstads zerteilt einen MD5 Hash in 32 Bit Blöcke und führt eine Reduktion auf $m = 1000$ mit einer Modulo Berechnung durch. Da $m \ll 2^{32}$, wird die Gleichverteilung nicht verletzt und eine Ablehnung der Nullhypothese findet nicht statt. Da die Bloomfilter in ORESTES hochfrequent zum Client übertragen werden müssen, wird $m \ll 2^{32}$ meist erfüllt sein. Für die leichte Portierbarkeit einer

Hashfunktionen für verschiedene Programmiersprachen und ihre Persistenz-APIs schien es uns deshalb sinnvoll, auf das Rejection Sampling im einfachen LCG Algorithmus zu verzichten. Der einfache LCG Algorithmus zeigt gute Ergebnisse für die Gleichverteilung und ist aufgrund der einfachen Prozedur leicht in wenig mächtige Sprachen wie JavaScript portierbar.

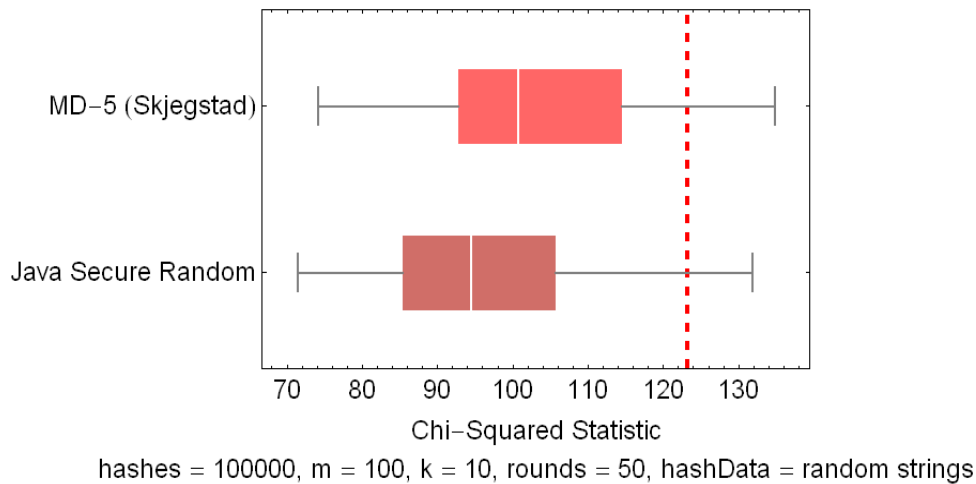


Abbildung 76 Test-Ergebnisse mit und ohne Rejection Sampling.

Hashfunktion	Geschwindigkeit (ms)	FPs während Einfügen	f bei Einfügen	f final
Cryptographic (MD5)	247,80	51,00	0,0017	0,0097
Cryptographic (SHA-256)	265,28	36,00	0,0012	0,0103
Cryptographic (SHA-512)	232,22	45,00	0,0015	0,0111
Cryptographic (SHA1)	257,99	47,00	0,0016	0,0105
CarterWegman	606,76	40,00	0,0013	0,0101
Java RNG	37,05	38,00	0,0013	0,0101
SecureRNG	263,20	38,00	0,0013	0,0101
CRC32	114,40	36,00	0,0012	0,0102
Murmur	57,94	50,00	0,0017	0,0104
SimpleLCG	26,83	48,00	0,0016	0,0095
Adler32	116,73	27846,00	0,9282	0,9860

Tabelle 9 Ergebnisse für die False-Positives für verschiedene Hashfunktionen beim Einfügen in einen Bloomfilter mit $m = 300000$ und $n = 30000$.

Tabelle 9 zeigt die Ergebnisse für das Einfügen in einen Bloomfilter unter Verwendung der verschiedenen Hashfunktion und *Increasing IDs* ($m = 300000$ und $n = 30000$). Dabei wurden während des Einfügens False-Positives gezählt (FPs) und nach Abschluss des Einfügens für n nicht eingefügte Objekte überprüft, ob sie einen False-Positive erzeugen. Dies ergibt die finale False-Positive Rate f , deren erwarteter Wert $f = 1,0185$ beträgt. Das Ergebnis der finalen False-Positive Rate ist in Abbildung 77 gezeigt. Adler-32 wurde zur besseren Lesbarkeit entfernt. Das Ergebnis macht deutlich, dass die Hashfunktionen in ihrem realen Kontext der Bloomfilter tatsächlich Werte erzeugen, die zu einem niedrigen f führen.



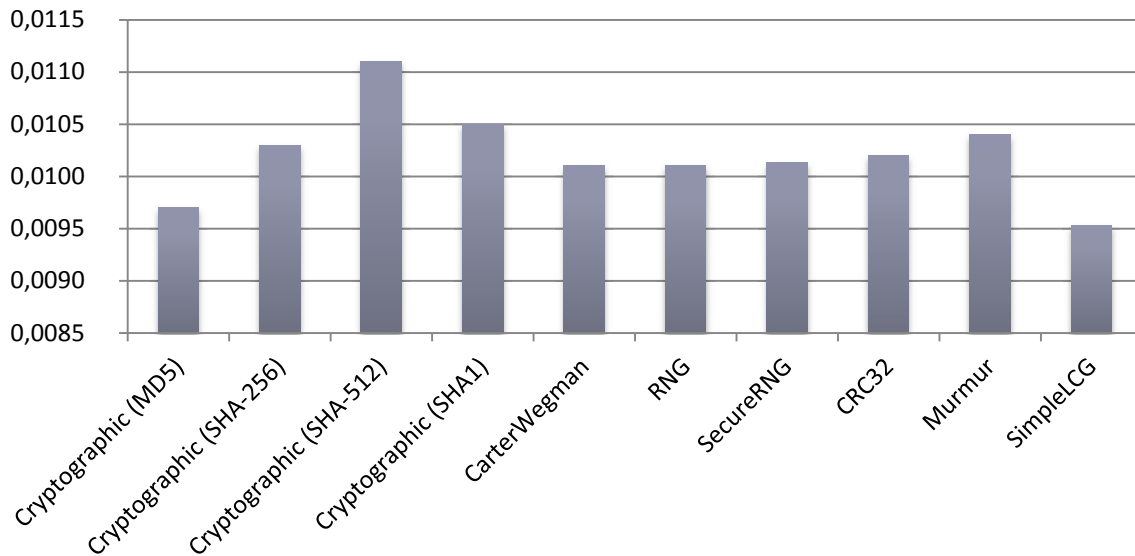


Abbildung 77 Finale False-Positive Rate beim Einfügen in Bloomfilter $m = 300000$ und $n = 30000$.

Für *Increasing IDs*, also künstliche Objekt-IDs, wie wir sie in ORESTES erwarten, zeigen alle Hashfunktionen außer Adler32, CRC32 und Carter-Wegman gute Ergebnisse. Die Mittelwerte der Chi-Quadrat Statistiken sind für die Konfiguration $m = 10000$ in Abbildung 78 gezeigt. Die schlechten Ergebnisse von Carter-Wegman sind vermutlich auf die langen Präfixe zurückzuführen, die für jeden Wert gleich sind. Das Interpretieren der Eingabe als Integer Wert führt dann dazu, dass sich die Eingaben lediglich in den niederwertigen Bits unterscheiden. Ist der Parameter p der Carter-Wegman Funktion nicht genügend groß gewählt, führt dies anschließend zu einer ungleichmäßigen Verteilung, da die Annahme verletzt ist, dass die Anzahl möglicher Eingaben kleiner p ist.

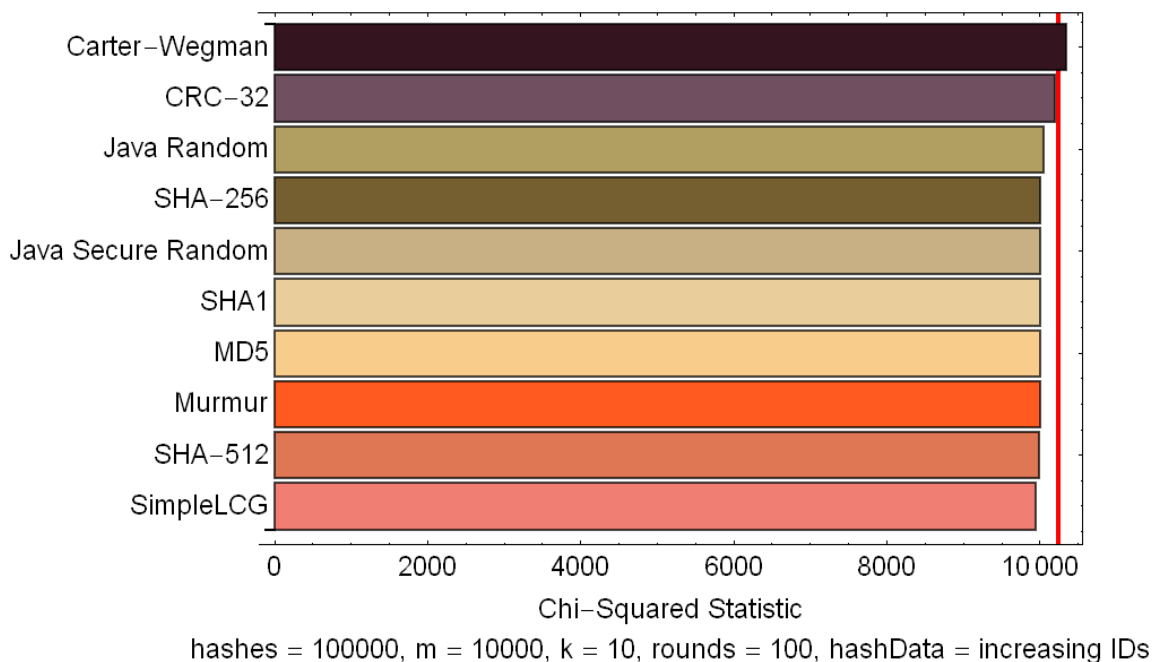


Abbildung 78 Qualität von Hashwerten auf künstlichen Objekt-IDs anhand des Mittelwerts der Chi-Quadrat-Statistik.

3.7.2.7 Performance Aspekte

Für die praktische Nutzung ist die Geschwindigkeit des Lookups für den In-Memory Bloomfilter auf Clientseite entscheidend. Diese wird unmittelbar bedingt durch die Geschwindigkeit der verwendeten Hashfunktion. Abbildung 79 zeigt die über 10 Durchläufe gemittelte Geschwindigkeit der verschiedenen Hashfunktionen als Box-Whisker Plot. Als Parameter wurden $k = 5$ und $m = 1000$ gewählt. Das Ergebnis fällt wie erwartet aus. Die beiden LCG Algorithmen gewinnen deutlich. Auch Murmur ist sehr performant und übertrifft sogar die Prüfsummen. Die kryptographischen Hashfunktionen sind etwas langsamer. Die Carter-Wegman Funktion ist aufgrund der Ganzzahl-Arithmetik auf sehr großen Werten ausgesprochen unperformant.

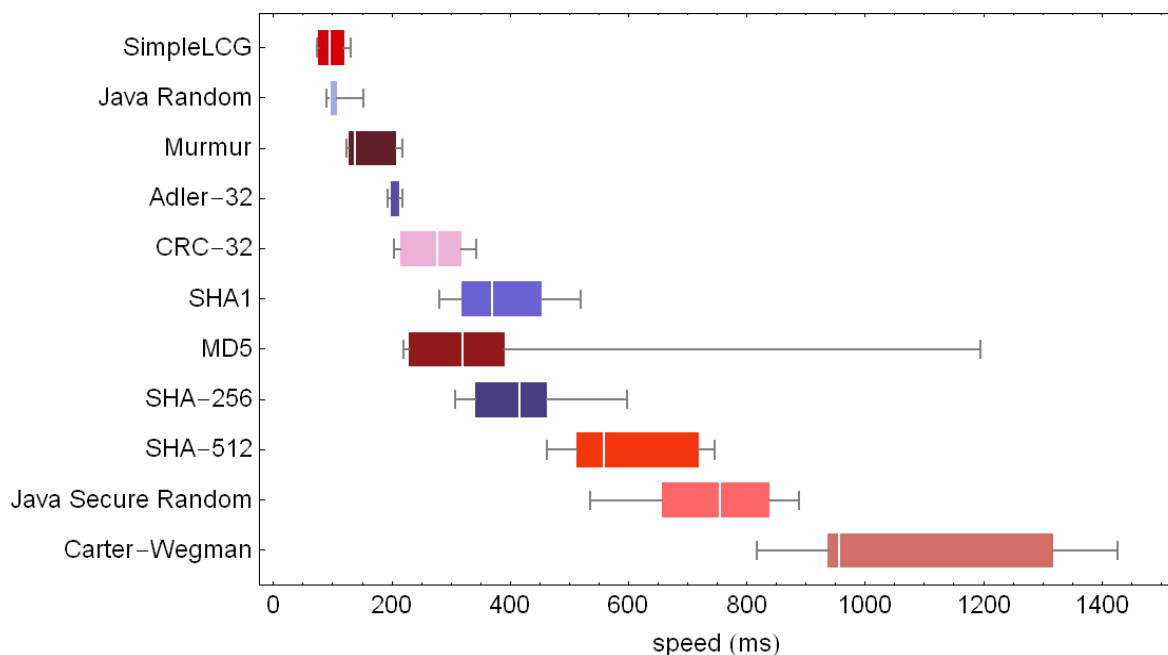


Abbildung 79 Geschwindigkeit der unterschiedlichen Hashfunktionen für die Berechnung von 1.000.000 Hashwerten mit $k=5$ und $m=1000$, gemittelt über 10 Durchläufe.

Zusammenfassend ergibt die Überprüfung der unterschiedlichen Hashfunktionen:

- LCGs und Murmur bieten einen sehr guten Kompromiss aus hoher Geschwindigkeit und einer gleichmäßigen Verteilung.
- Kryptographische Hashfunktionen mit Zerlegung, Modifikation und Rejection Sampling bieten für alle Typen von Eingaben zuverlässig eine gleichmäßige Verteilung. Der Median des p-Wertes fällt sehr zuverlässig auf 0,5 – dem erwarteten Ergebnis einer perfekten Gleichverteilung.

Für den praktischen Einsatz sind LCGs und Murmur deshalb eine gute Wahl, wenn ORESTES aus unterschiedlichsten Programmiersprachen genutzt wird. Für eine besonders hohe Zuverlässigkeit der Hashwerte können die kryptographischen Hashfunktionen genutzt werden. Voraussetzung ist die Nutzung des ORESTES-Servers von ausreichend leistungsfähigen Cli-



ents mit Persistenz-APIs in Programmiersprachen, deren Standardbibliotheken kryptographische Hashfunktionen unterstützen.

3.8 Bloomfilter als komprimiertes Write-Log Fenster

Die Nutzung unserer Bloomfilter Implementierungen in ORESTES kann auf zwei Arten geschehen:

1. Existiert nur ein ORESTES-Server, nutzt dieser einen Counting Bloomfilter, um bei jedem ändernden Request die Objekt-ID des geschriebenen Objekts in den Bloomfilter einzufügen. Nach Ablauf der Expirationsdauer entfernt er das Objekt aus dem Counting Bloomfilter.
2. Existieren mehrere Bloomfilter, nutzen sie einen gemeinsamen Redis Counting Bloomfilter, in den jeder ORESTES-Server die Objekt-IDs geänderter Objekte schreibt. Nach Ablauf der Expirationsdauer entfernt jeder Server die Objekte, die er selbst in den Bloomfilter geschrieben hat.

Der Aufbau für Fall 2 ist in Abbildung 80 gezeigt. Jeder ORESTES-Server pflegt eine eigene Priority-Queue, in die eingehende Writes eingefügt werden. Die Sortierung der Priority-Queue korrespondiert mit den Timestamps der Writes, sodass die ältesten Objekte zuerst entnommen werden. Die Priority Queue erlaubt es, Writes für Objekte mit voneinander abweichenden Expirationsdauern aufzunehmen. Ist die Expirationsdauer für jedes Objekt identisch, wird anstelle einer Priority-Queue lediglich eine herkömmliche Queue benötigt. Der verteilte Aufbau unterscheidet sich von dem zentralisierten Aufbau nur geringfügig – dort wird statt des Redis Bloomfilters der Java Counting Bloomfilter verwendet, der nicht geteilt werden muss. Die Anbindungen an die bisher implementierten Backends von ORESTES beruhen noch auf dem zentralisierten Ansatz. Für kommende Anbindungen, vor allem an verteilte Datenbanken, werden wir jedoch eine verteilte Architektur nutzen, in der mehrere ORESTES-Server eine Datenbank kapseln. Ein ORESTES-Server kann auf diese Weise nicht zum Engpass werden, da alle Anfragen an einen beliebigen ORESTES-Server gerichtet werden können. Etwaiger Overhead, z.B. das Parsen von HTTP und JSON, wird auf diese Weise horizontal skaliert und parallelisiert.

Der Lebenszyklus von Objekten im Bloomfilter ist folgender:

1. Wenn eine Client ein Objekt schreibt, wird der Request von einem ORESTES-Server angenommen.
2. Dieser vermerkt die entsprechende Objekt-ID im Redis Counting Bloomfilter, der mit einer Redis Instanz im gleichen Netzwerk verbunden ist. Gleichzeitig wird die geschriebene Objekt-ID auch in der lokalen Priority Queue des ORESTES-Servers vermerkt, um das Objekt nach Ablauf seiner Expirationsdauer wieder zu entfernen.
3. Da das Objekt geschrieben wurde, sind alle gecachten Kopien des Objekts veraltet. Clients, die eine Transaktion starten, vermeiden das Lesen dieser veralteten Objekte, indem sie initial den Bloomfilter laden. Der Bloomfilter ist bereits materialisiert und

kann ohne Rechenaufwand ausgeliefert werden. Die Übertragung findet dabei im Normalfall über JSON statt. Das JSON Objekt enthält neben den Metainformationen die Bits des Bloomfilters als *Base64* codierten String. Jeglicher Overhead, der durch Base64 entstehen könnte (Base64 nutzt nur 64 von 94 zur Verfügung stehenden Zeichen), wird nivelliert durch die Transportkompression, die zuvor bereits automatisch zwischen Client und Server auf HTTP Ebene ausgehandelt wurde.

4. Sobald die Expirationsdauer des geschriebenen Objektes abgelaufen ist, d.h. garantiert ist, dass jeder Web-Cache eine etwaige Kopie verworfen hat, entnimmt der ORESTES-Server den Eintrag aus dem Counting Bloomfilter.

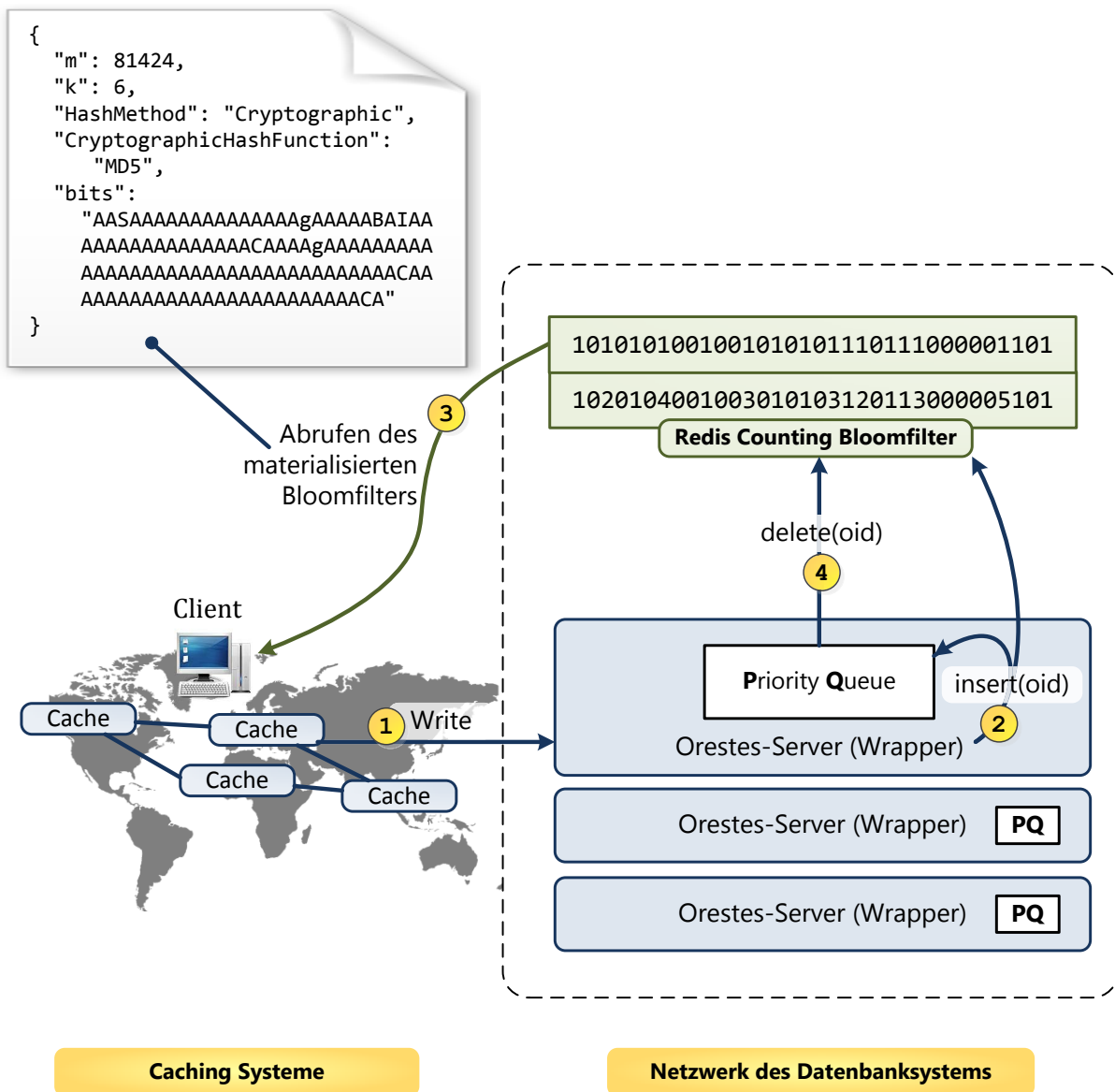


Abbildung 80 Die Nutzung der Bloomfilter in ORESTES.

Auf Clientseite ist die Nutzung des Bloomfilters sehr einfach. Unsere generische Java-Implementierung, die von beliebigen Persistenz-APIs genutzt werden kann, implementiert bereits die gesamte Logik zur Handhabung des Bloomfilters. Dies ist in Abbildung 81 gezeigt:



1. Zum Start einer Transaktion ruft die Persistenz-API (z.B. JDO oder JPA) an der ORESTES Client-Schicht die *beginTransaction* Methode auf. Dieser Aufruf wird in einen POST Request gegen die Transaktionsressource verwandelt.
2. Der Server antwortet u.a. mit der URL der gestarteten Transaktion und dem Bloomfilter.
3. Die generische Client-Schicht nimmt den Bloomfilter entgegen und erzeugt daraus einen einfachen Java Bloomfilter.
4. Wenn der Client ein Objekt lädt, ruft er die *load* Methode an der ORESTES-Schnittstelle auf.
5. Die Implementierung der Methode prüft zuerst mit einem *contains* Aufruf, ob die angefragte OID im Bloomfilter gespeichert ist. Ist dies der Fall, wird dem GET Request für das Objekt eine *Cache-Control: max-age=0* Klausel angehängt. Ansonsten wird der Request unverändert gestellt und darf aus Web-Caches beantwortet werden.

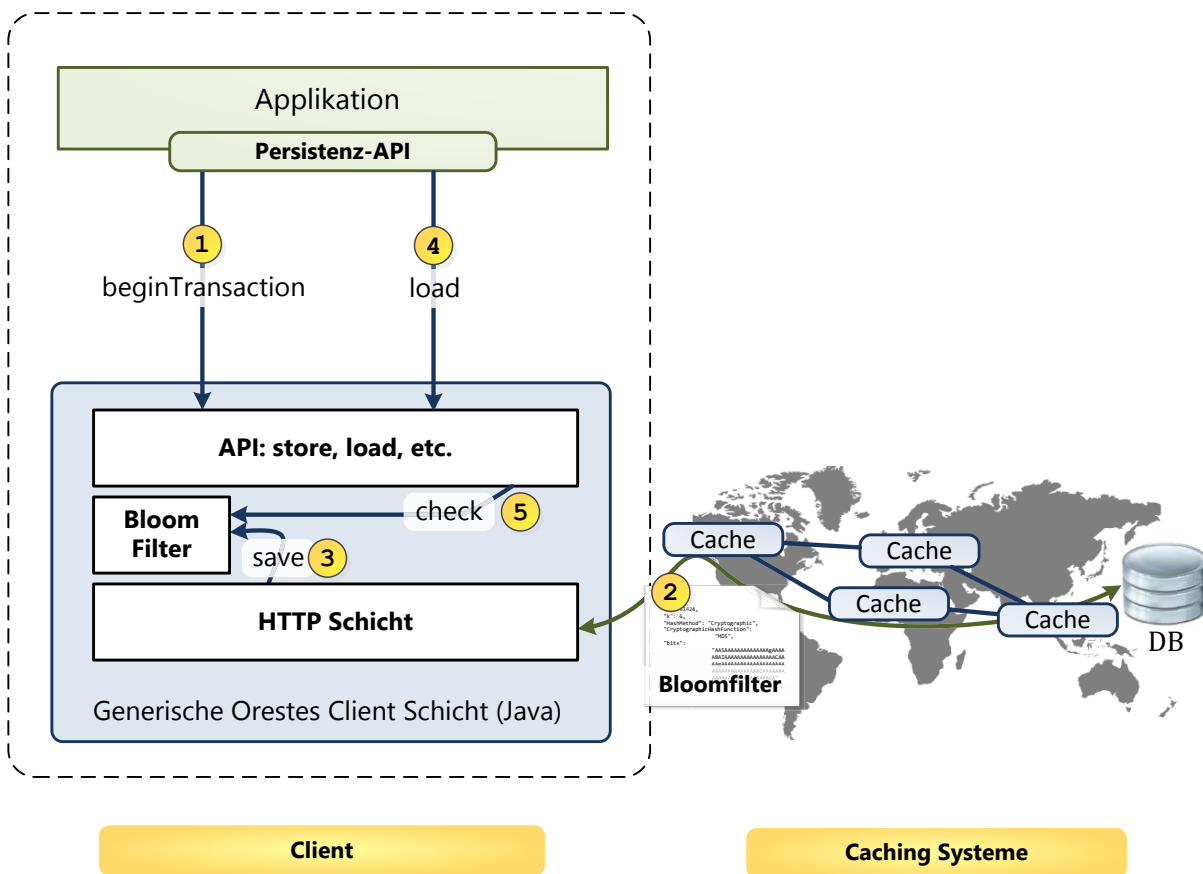


Abbildung 81 Die Benutzung des Bloomfilters auf Clientseite.

Wie unsere Benchmarks gezeigt haben, ist es nahezu ausgeschlossen, dass der gemeinsame Bloomfilter auf Serverseite zu einem Engpass wird. Ein solcher Effekt würde erst auftreten, wenn Objekte deutlich mehr als 100.000 mal pro Sekunde geschrieben werden. Für derartige Workloads ist ORESTES mit seiner optimistischen Nebenläufigkeitskontrolle ohnehin keine geeignete Wahl. Änderungen, die von den ORESTES-Servern an den Bloomfilter geleitet werden, ließen sich überdies problemlos bündeln, da die Operationen *remove* und *add* assoziativ

sind [195]. Auf Clientseite ist der Bloomfilter ebenfalls keine Belastung – wie unsere Benchmarks gezeigt haben, können leicht 120.000 *contains* Anfragen pro Sekunde gegen den Bloomfilter gerichtet werden, sogar bei Verwendung einer kryptographischen Hashfunktion.

3.9 Evaluation

Wir wollen den praktischen Effekt der bloomfilterbasierten Cache Kohärenz an einem einfachen Aufbau demonstrieren, der in Abbildung 82 gezeigt ist. Eine Testanwendung greift über JDO Schnittstelle auf den generischen ORESTES Client zu. Dieser kommuniziert entweder direkt mit dem ORESTES-Server (2) oder über einen Web-Cache (1). Der Web-Cache, der Reverse-Proxy Cache *Varnish*, wird in einer Linux VM auf dem gleichen System ausgeführt. Der Web-Cache ist als Surrogate für den ORESTES-Server eingerichtet, d.h. er leitet Anfragen bei einem Cache-Miss an diesen weiter.

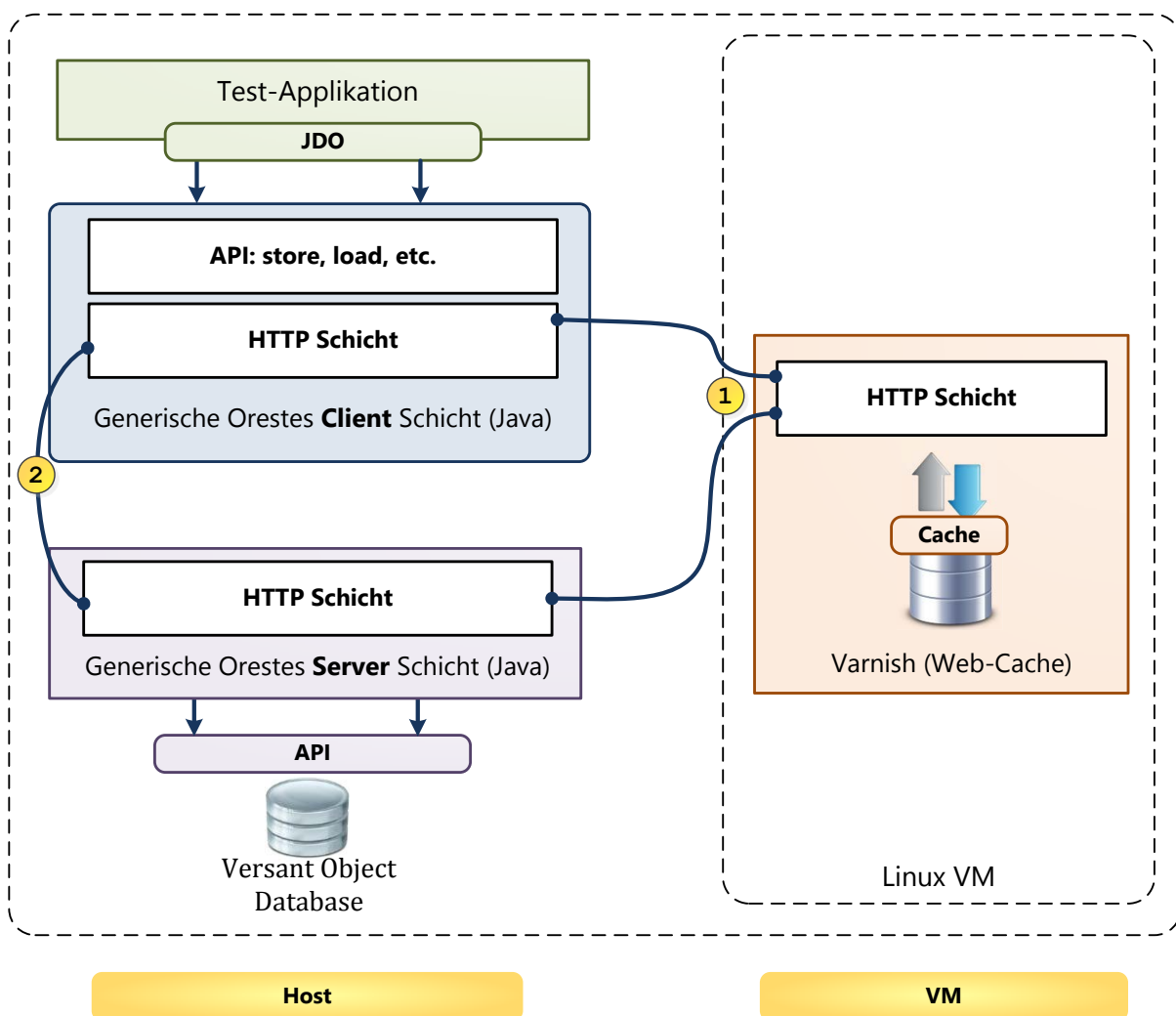


Abbildung 82 Der Aufbau für den praktischen Test der Bloomfilter.

Wir haben diverse Tests für diesen Aufbau erprobt, zwei davon wollen wir hier zeigen. Der erste Test ist in Abbildung 83 gezeigt. Die Klasse `BloomClient` stellt eine Verbindung zur



Datenbank her, die entweder den Cache nutzt oder sich direkt verbindet. Die Klasse hat außerdem zwei Methoden, um Objekte zu manipulieren:

- `read(id)` liest das Objekt mit der logischen ID `id`.
- `write(id)` liest das Objekt mit der logischen ID `id` und schreibt es anschließend.

Der Ablauf des Beispiels ist wie folgt:

1. Der Client, der sich über den Cache verbindet, liest Objekt 1 mit der Version 1.
2. Der zweite Client der direkt verbunden ist, schreibt das Objekt 1, das anschließend Version 2 hat.
3. Der erste Client fragt das Objekt erneut an und ändert es.

Der Ausgang dieses Tests ist abhängig davon, ob die Verwendung von Bloomfilter aktiviert wurde oder nicht. Wurden Bloomfilter nicht aktiviert, liefert der Cache in Schritt 3 das Objekt mit der Version 1 aus. Der Client versucht dieses Objekt zu ändern und die optimistische Validierung schlägt fehl, so dass die Transaktion mit einer `OptimisticVerificationException` scheitert. Sind die Bloomfilter aktiviert, werden sie beim Start der Transaktion in Schritt 3 übertragen. Beim Request für Objekt 1 stellt die ORESTES Client-Schicht fest, dass sich das Objekt im Bloomfilter befindet und fragt es mit einer Revalidierung an. Der Cache liefert auf diese Weise das aktuelle Objekt aus und die Transaktion kann erfolgreich commiten.

```
//Ein Client kommuniziert über den Cache
BloomClient cache = new BloomClient();
//Der andere nutzt den ORESTES-Server direkt
BloomClient nocache = new BloomClient(false);

//Über den Cache in einer Transaktion ein Objekt laden
cache.begin();
Spiel old = cache.read(1);
System.out.println("Version " + JDOHelper.getVersion(old)); // -> Version 1
cache.commit();

//In einer Transaktion mit dem anderen Client dasselbe Objekt ändern
nocache.begin();
nocache.write(1);
nocache.commit(); //Objekt wird auf Version 2 aktualisiert

//Über den Cache eine neue Transaktion starten dasselbe Objekt erneut laden
//und ändern
cache.begin();
try {
    cache.write(1);
    cache.commit();
} catch (JDOException e) {
    System.out.println("Stale Read...");
}
```

Abbildung 83 Beispiel für ein Testszenario, das mit und ohne Bloomfilter ein anderes Ergebnis zeigt.

Ein zweiter Test untersucht Transaktionsabbrüche beim zufälligen Lesen und Schreiben. Der Ablauf ist folgender:

1. Eingabe: Objekte insgesamt (n), Reads (r), Writes (w), Clients mit Cache (c_{on}), Clients ohne Cache (c_{off})
2. Wiederhole für alle Clients in zufälliger Reihenfolge:
Wähle in zufälliger Reihenfolge $r + w$ der n Objekte aus und führe auf ihnen in einer Transaktion r Reads und w Writes aus. Protokolliere jeden Transaktionsabbruch.

Führt man den beschriebenen Test für eine Beispielkonfiguration mit $n = 10$, $r = 6$, $w = 2$, $c_{on} = 6$, $c_{off} = 6$ ohne Bloomfilter aus, kommt es zu etwa 3-6 Transaktionsabbrüchen:

```
Transaction aborts: 5
Execution time for operations on 10 total objects with 6 / 6 clients using /
not using the cache, each performing 6 / 2 read / write operations:
5970.0 ms
```

Derartige Transaktionen müssten in der Praxis ihren Applikationszustand wiederherstellen und die gescheiterte Transaktion ein weiteres Mal versuchen.

Werden die Bloomfilter aktiviert, gelingt jede der Transaktionen und der gesamte Durchlauf terminiert in wesentlich kürzerer Zeit. Auch die völlig zufällige Reihenfolge, mit der Reads und Writes ohne und mit Cache ausgeführt werden, hat keinerlei Auswirkungen auf die Cache Kohärenz, die durchgehend erhalten bleibt.

3.10 Zusammenfassung

Zu Beginn dieses Kapitel haben wir die Anforderungen des Clients und Servers für eine Datenstruktur zur Erhaltung der Cache-Kohärenz formuliert. Mit Bloomfiltern haben wir eine ideale Datenstruktur sowohl für den Server (siehe Abbildung 84) als auch für den Client (siehe Abbildung 85) gefunden.

Effizienz

- Die Counting Bloomfilter für den Server sind hochgradig effizient, sowohl für ein zentralisiertes, als auch ein verteiltes Szenario.

Shared Datastructure

- Die Redis Bloomfilter können von beliebig vielen Orestes-Servern parallel genutzt werden.

Schnelle Generierung

- Die Counting Bloomfilter enthalten einen materialisierten Bloomfilter. Die Generierung wird also bereits vor der Anfrage implizit beim Einfügen und Löschen vorgenommen.

Garbage Collection

- Jeder Orestes-Server kann geschriebene Objekte mithilfe einer Priority Queue direkt bei Ablauf der Expirationdauer wieder aus dem Bloomfilter entfernen. Alternativ kann der CBloomFilterRedis benutzt werden, der Objekte durch Counter-Expiration automatisch entfernt.

Abbildung 84 Erfüllung der serverseitigen Anforderungen durch den Bloomfilter.



Effizienz

- Der Bloomfilter benötigt auf Clientseite lediglich einen $O(1)$ Hash-Lookup. Wie wir gezeigt haben lassen sich mehrere Hunderttausend solcher Lookups pro Sekunde realisieren.

Kompakte Darstellung

- Die Bloomfilter kommen auf konstante 44% dem mathematisch optimalen Platzbedarf nahe.

Abbildung 85 Erfüllung der clientseitigen Anforderungen durch den Bloomfilter.

Für die Cache-Kohärenz und Transaktionen gilt die genauso einfache wie mächtige Garantie der Bloomfilter:

Clients sehen nur Objektversionen, die mindestens so aktuell sind wie der Bloomfilter selbst.

Aus Sicht des Nutzers erfordern Bloomfilter die Konfiguration dreier Verfahrensparameter:

- *E.* Die Expirationsdauer von Objekten.
- *Wps.* Die Anzahl an Writes pro Sekunde.
- *f.* Die maximale False-Positive Rate, die für den Bloomfilter noch akzeptabel ist.

Diese Parameter bestimmen also die Größe des Bloomfilters und die Dauer, für die Objekte gecacht werden. Eine automatische Bestimmung dieser Parameter, z.B. auf Basis von Machine-Learning Techniken, ist ein interessanter Punkt für weitere Forschung.

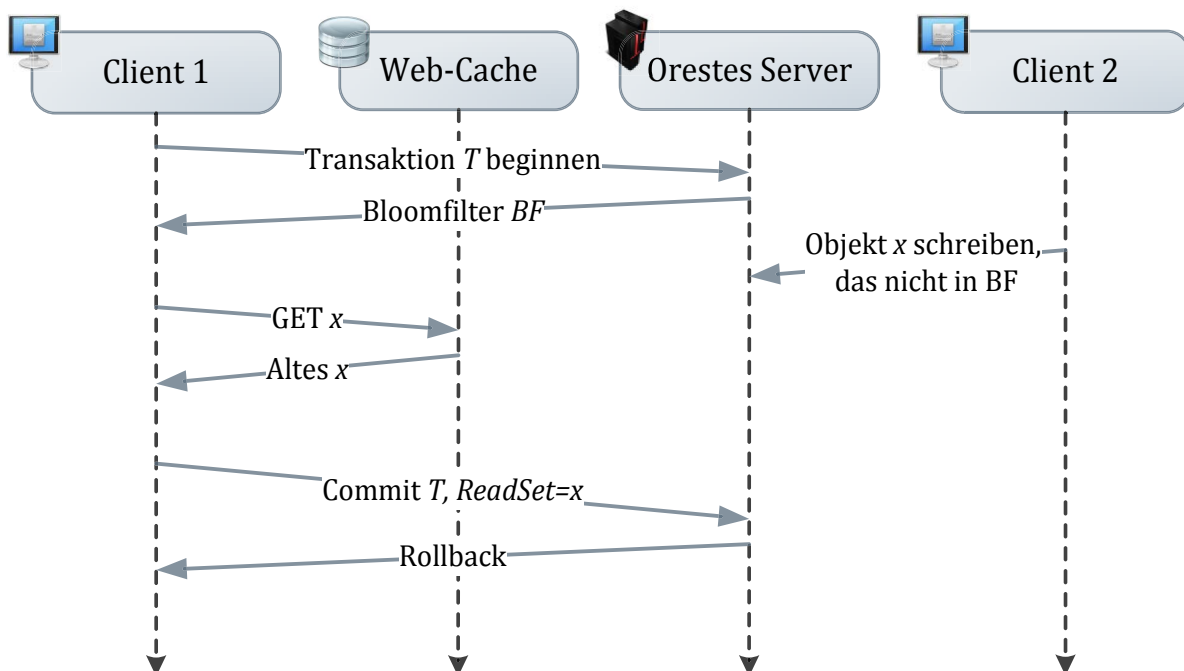


Abbildung 86 Das Szenario, das trotz Bloomfiltern zu einem Transaktionsabbruch führt.

Durch das Bestimmen der Abruffrequenz der Bloomfilter lässt sich ein frei wählbares Maß an Cache Kohärenz zusichern. Da die Bloomfilter zu Beginn jeder Transaktion abgerufen werden, existieren nur wenige und unwahrscheinliche Arten, auf die ein Stale Read dennoch auftreten kann. Damit ein Stale Read für eine Transaktion T auftritt, muss ein Objekt nach dem Abrufen des Bloomfilters über einen anderen Web-Cache geschrieben werden als es anschließend von T gelesen wird. Auch kann es kein häufig geändertes Objekt sein, da es sonst ohnehin bereits im Bloomfilter vermerkt ist. Und der Schreibzugriff muss noch während der Laufzeit von T erfolgreich sein. Diese Konstellation ist in Abbildung 86 gezeigt.

Damit der beschriebene Fall eintreten kann, müssen mehrere unwahrscheinliche Faktoren zusammenkommen:

1. Ein Objekt x wird genau dann erfolgreich geschrieben, wenn Transaktion T bereits läuft. Diese Situation führt bei BOCC+ und Optimistic Locking ohnehin meist zu einem Transaktionsabbruch, falls nämlich T das Objekt x gelesen hat, bevor x von dem zweiten Client geschrieben wurde.
2. Das Objekt muss ein selten geändertes Objekt sein, da es andernfalls bereits im Bloomfilter aufgeführt wäre.
3. Die Änderung darf nicht über den Web-Cache erfolgen, aus dem T später liest, da sonst durch *Invalidation by Side-Effect* ohnehin eine Invalidierung erfolgt wäre.

Die Gefahr, dass alle drei Faktoren zusammenkommen, kann als sehr gering angesehen werden: es muss zu einem Read-Write Konflikt auf einem selten geänderten Objekt über einen nicht-geteilten Web-Cache kommen, der mit hoher zeitlicher Lokalität auftritt.

Im Zuge der NoSQL Bewegung ist es populär geworden, Systeme nach dem *CAP Theorem* von Eric Brewer zu klassifizieren [1], [196]. Es besagt, dass ein verteiltes Datenbanksystem entweder Konsistenz (Consistency) oder Verfügbarkeit (Availability) aufgeben muss, wenn eine Netzwerkpartition (Partition) auftritt. ORESTES hat einen wählbaren Effekt auf die Konsistenz, die das zugrundeliegende Datenbanksystem gewährt: wenn der Client bereit ist, potentiell veraltete Objekte zu erhalten, garantiert ORESTES mit der verteilten Web-Caching Hierarchie hohe Verfügbarkeit und Partitionstoleranz, während die Konsistenz herabgesetzt wird. Konsistenz im Sinne des CAP Theorem ist jedoch definiert in Bezug auf ein nichttransaktionales, atomares Read/Write Register und weicht damit von der Definition der Konsistenz im Sinne der ACID Eigenschaften ab [196]. Die ACID Garantien werden in ORESTES unter allen Umständen durch optimistische Verifikation gewährt. Wenn der Client Kohärenz wünscht und deshalb Bloomfilter nutzt, ist die Klassifikation des Gesamtsystems bestimmt durch das zugrundeliegende Datenbanksystem, beispielweise Konsistenz und Verfügbarkeit (CA) für ein Single-Node objektorientiertes oder relationales Datenbanksystem und Konsistenz und Partitionstoleranz (CP) für eine verteilte Dokumentendatenbank wie MongoDB [10].



Als wichtigstes Merkmal der bloomfilterbasierten Cache-Kohärenz sehen wir, dass, obwohl der Mechanismus selbst über eine gewisse Komplexität verfügt, seine Garantie so einfach ist: eine Transaktion die bloomfilterbasierte Cache-Kohärenz nutzt, sieht nur Objekte, die mindestens so aktuell wie die Transaktion selbst sind. Ähnlich wie beim Transaktionskonzept wird hier ein komplexer Mechanismus hinter einer einfachen Abstraktion versteckt.

4 Cache Invalidierungen

Das vorangehende Kapitel hat erklärt, wie mit Hilfe von Bloomfiltern Web-Caches von der Clientseite aus invalidiert und somit Änderungen von Objekten für ein Client sofort sichtbar werden.

Für die Web-Caches, die unter der Kontrolle des Orestes-Servers liegen, die sogenannten Reverse-Proxy-Caches und CDNs, ergibt sich aber noch eine weitere Möglichkeit, diese nach einer Änderung zu invalidieren. Hierfür bieten die Reverse-Proxy-Caches und die meisten CDNs proprietäre Protokolle an, um serverseitig eine Invalidierung anzustoßen. Um zu verstehen wie eine solche Invalidierung ausgelöst werden kann, wird beispielhaft der Reverse-Proxy-Cache Varnish und das Content Delivery Network (CDN) CloudFront von Amazon untersucht. Zunächst werden ihre Funktionsweisen erläutert und ihre Invalidierungsschnittstellen genauer betrachtet. Anschließend wird der Invalidierungsservice in Orestes vorgestellt, der den Anforderungen der verschiedenen proprietären Schnittstellen durch Erweiterbarkeit gerecht wird. Abschließend wird die Kombinierbarkeit von CDNs und klassischen Web-Caches untersucht und einen Ausblick über weitere Cachevarianten gegeben.

4.1 Reverse-Proxy-Cache: Varnish

Reverse-Proxy-Caches sind eine besondere Art von Web-Caches, die direkt vor dem eigentlichen Web-Server aufgestellt sind. Sie fungieren im großen Teil selbst wie der Origin-Server und werden von den Clients direkt angesprochen, als ob sie der echte Web-Server wären. Dies funktioniert in der Regel so, indem die Server-Adresse nicht auf den Origin-Server, sondern auf den Reverse-Proxy-Cache zeigt. Dieser verarbeitet alle Request des Clients und kann einige von ihnen selbstständig und andere nur mit Hilfe des Origin-Servers beantworten. Er erfüllt dabei die Aufgabe, Inhalte des Origin-Servers zu cachen, also zwischen zu speichern und anschließende Request auf dieselbe Ressource selbstständig und ohne erneute Kontaktaufnahme zum Origin-Server zu beantworten. Er kann somit die Antwortzeit beschleunigen und den Origin-Server entlasten.

Unter den verbreiterten Reverse-Proxy-Cache Lösungen befinden sich der Apache Traffic Server, der Squid und der Varnish. Der Apache Traffic Server [197] ist eine Web-Cache Lösung, die ursprünglich von Inktomi als kommerzielles Produkt entwickelt und vertrieben wurde. Yahoo! Inc. hat den Web-Cache später aufgekauft und in ihrer gesamten Web-Infrastruktur sowohl als Forward-, Interception- und Reverse-Proxy eingesetzt. 2009 hat Yahoo! Inc. das Projekt der Apache Software Foundation gespendet und es wurde Open Source. Der Squid [197], [198] ist ebenfalls ein Web-Cache, der vor alledem als Forward-Proxy-Cache entwickelt wird, er kann aber auch als Reverse-Proxy eingesetzt werden. Er entsprang dem von ARPA gegründeten Harvest Projekt und zeichnet sich durch seine zahlreichen Erweiterungen und Treiber neuer Web-Technologien aus, da er seit Gründung als Open Source Projekt entwickelt wird. Wir haben Squid bereits in unserem Master-Projekt als Forward-Proxy-Cache eingesetzt, um die Zugriffszeiten auf Objekte zu verringern.



Varnish [197], [199] ist eine reine Reverse-Proxy-Cache Lösung, die von Varnish Software als Open Source Software entwickelt und vermarktet wird. Sie bezeichnen Varnish selbst als „Web-Application-Accelerator“. Varnish liegt mittlerweile in Version 3 vor und ist für moderne Server-Systeme entwickelt worden. Er verwendet, im Gegensatz zu den meisten Web-Caches, keine eigene Speicherlösung, sondern einzig den virtuellen Hauptspeicher des Betriebssystems für den Cache, wodurch er sehr hohe Performanz erzielt. So zählt Facebook zu den größten Nutzern von Varnish, die vor allem ihren statischen Content, wie Bilder, durch Varnish cachen und ausliefern. Varnish ist zudem so beliebt, weil es eine eigene Konfigurationssprache mitbringt, die Varnish Configuration Language (VCL), mit deren Hilfe die Requestverarbeitung des Varnish sehr stark erweitert und konfiguriert werden kann. Die Konfiguration erfolgt dabei in einem VCL-Skript, in dem die einzelnen Verarbeitungsschritte programmatisch manipuliert werden. Das VCL-Skript wird dann von Varnish zur Laufzeit in ein C-Programm umgewandelt und anschließend kompiliert. Das kompilierte Objekt wird dann in den laufenden Server verlinkt und für alle zukünftig eingehenden Request ausgeführt.

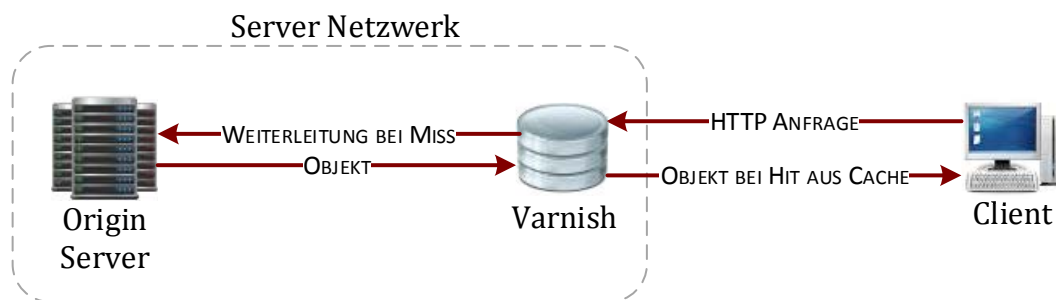


Abbildung 87 Ein einfacher Netzwerkaufbau unter Verwendung von Varnish.

Varnish wird in der Regel einzeln oder als Cluster direkt vor dem Origin HTTP Server im gleichen Netzwerk aufgesetzt und verarbeitet den gesamten HTTP Verkehr zum Origin-Server. In der Abbildung 87 wird diese Netzwerkkonfiguration dargestellt. Damit der Varnish den gesamten Netzwerkverkehr verarbeiten kann, zeigt die Adresse, die zum Verbindungsaufbau verwendet wird, nicht direkt auf den Origin-Server, sondern auf den Varnish. Reverse-Proxy-Caches dürfen dabei einzig lesende Anfragen selbst verarbeiten und müssen alle anderen Anfragen immer an den Origin-Server weiterleiten. Sie verhalten sich somit vollkommen transparent für den Client (*semantische Transparenz*). Als lesende Anfragen gelten einzig GET und HEAD Request. Bei diesen beiden Anfragen darf Varnish die Antwort des Origin-Servers für künftige Request cachen und diese dann direkt ausliefern. Der Origin-Server bestimmt dabei mit einer Lebensdauer, wie lange Varnish die Objekte cachen darf.

Stellt ein Client eine Anfrage an den Server, erreicht der Request immer zunächst den Varnish. Auf diesem wird dann das VCL Skript [200] evaluiert, das den Varnish anweist, wie der Request verarbeitet werden soll. Dieses VCL Skript bietet dabei mehrere Einstiegspunkte an, um das Standardverhalten von Varnish zu manipulieren. Die Verarbeitung des VCL-Skripts basiert dabei auf den Zustandsautomaten aus der Abbildung 88, wobei die Zustände konfiguriert werden können. Der Startzustand ist *vcl_recv*. In diesem Zustand wird entschie-

den, ob das Objekt überhaupt cachbar ist. In der Standardkonfiguration werden nur GET und HEAD Request gecached und somit nur dann die Transition *lookup* ausgeführt. In allen anderen Fällen wird die Transition *pass* gewählt.

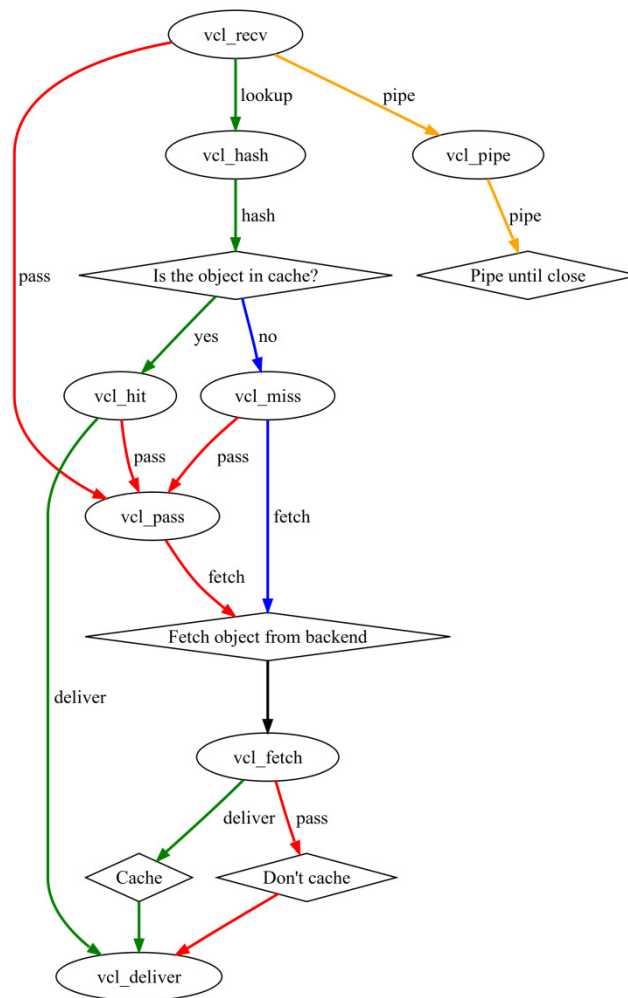


Abbildung 88 Der Varnish Zustandsautomat, nachdem das VCL Skript evaluiert wird.

Quelle: https://www.varnish-software.com/static/book/VCL_Basics.html#the-vcl-state-engine.

Nach dem *vcl_hash* Zustand wird der eigentliche Cache-Lookup durchgeführt, aus dem dann eine Transition zu *vcl_hit* oder *vcl_miss* resultiert, je nachdem ob das Objekt bereits im Cache liegt oder nicht. Wird die Weiterverarbeitung im *vcl_hit* Zustand nicht abgebrochen, so erfolgt ein direkter Übergang in den *vcl_deliver* Zustand. Aus allen anderen Zuständen resultiert eine Weiterleitung des Requests zum Origin-Server. Nachdem dieser den Response zurückgeschickt hat, geht der Zustandsautomat in den *vcl_fetch* Zustand über. In diesem wird nun entschieden, ob der Response für zukünftige Request zwischengespeichert werden soll. In der Regel werden alle Responses, die auf einen GET oder HEAD Request erfolgten und als cachebar markiert sind, gespeichert. Anschließend endet diese Transition auch in den *vcl_deliver* Zustand. Dieser erlaubt die abschließende Verarbeitung des Responses, bevor dieser zum Client geschickt wird.



Der Varnish unterstützt in der Standardkonfiguration keine serverseitige Invalidierung. Um dies zu erreichen, muss der Varnish in mehreren Zuständen mit einem VCL-Skript manipuliert werden. Die gängigste Vorgehensweise hierfür ist, dem Varnish eine neue HTTP Methode PURGE beizubringen, die der Server immer dann benutzt, wenn er eine Ressource invalidieren möchte. Hierfür sendet der Origin Server ein HTTP Request mit der PURGE Methode an den Varnish. In Abbildung 89 wird ein solches VCL-Skript präsentiert, das dem Varnish die PURGE Methode beibringt. Durch diese Konfiguration kann der Varnish von dem Orestes-Server angesprochen werden, um geänderte Objekte aus dem Cache zu verdrängen.

```
# Der Orestes Origin Server
backend default {
    .host = "192.168.1.6";
    .port = "80";
}

# Zugriffsschutz für PURGE Requests
acl purgers { "192.168.1.6"; }

# Konfiguration für den receive Zustand
sub vcl_recv {
    if (req.request == "PURGE") {
        if (!client.ip ~ purgers) {
            error 405 "Not allowed.";
        }
        return (lookup);
    }
}

# Konfiguration für den hit Zustand
sub vcl_hit {
    if (req.request == "PURGE") {
        purge;
        error 200 "Purged.";
    }
}

# Konfiguration für den miss Zustand (auch hier muss der Purge folgen)
sub vcl_miss {
    if (req.request == "PURGE") {
        purge;
        error 200 "Purged.";
    }
}
```

Abbildung 89 VCL-Skript für eine PURGE Methode in Varnish.

Ein VCL-Skript fängt immer mit einer *backend* Definition an, die den Varnish anweist, wie er den Origin-Server erreichen kann. Durch die IP-Adresse *192.168.1.6* und dem zugehörigen TCP-Port *80*, wird in diesem Beispiel, der Orestes-Server dem Varnish bekanntgegeben. Das heißt, dass er alle eingehenden Requests, die er nicht aus dem Cache beantworten kann, an diese Adresse weiterleitet. Als nächstes wird eine Zugriffsregel definiert, die später dafür verwendet wird, die PURGE Requests nur von dem Orestes-Server zu erlauben. Dementsprechend weist sie dieselbe IP-Adresse wie die *backend* Definition auf. Als nächstes wird der

vcl_recv Zustand konfiguriert. Wichtig dabei zu beachten ist, dass die Konfigurationen immer das Standardverhalten von Zuständen erweitern und nie ersetzen. Wird kein expliziter Zustandsübergang in der Konfiguration angestoßen, wird immer das Standardverhalten des Zustandes ausgeführt. Da die PURGE Methode keine Standard HTTP Methode ist und ihre Semantik somit nicht definiert ist, würde der Varnish diese normalerweise nicht interpretieren und einfach an den Origin-Server weiter leiten. Damit der Varnish stattdessen aber ein Cacheeintrag invalidiert, muss diese Semantik implementiert werden. Hierfür wird also zunächst geprüft, ob es sich bei der HTTP Methode um ein PURGE handelt. Ist dies der Fall, wird überprüft, ob der Absender des Requests ein erlaubter Client ist. Falls die Client-IP nicht in der *purgers* Liste enthalten ist, also nicht der Orestes-Server ist, wird die Weiterverarbeitung mit einem 405 Method Not Allowed Status-Code abgebrochen. Andernfalls wird die *lookup* Transition angestoßen, die in einem Cache-Lookup resultiert, damit anschließend ein eventuell gefundener Cacheeintrag invalidiert werden kann. Um dies zu bewirken, müssen die beiden Zustände *vcl_hit* und *vcl_miss* ebenfalls angepasst werden. Der *vcl_hit* Zustand wird immer dann erreicht, wenn ein entsprechender Cacheeintrag existiert. Da dieser Zustand beispielsweise auch durch einen Lesezugriff (GET Request) erreicht werden kann, muss also zunächst wieder überprüft werden, ob es sich um ein PURGE Request handelt. Ist dies der Fall, führt er die eigentliche Invalidierung aus, da er ja einen Cacheeintrag gefunden hat. Dies geht in Varnish 3 mit dem *purge;* Befehl. Anschließend terminieren wir die weitere Verarbeitung mit einem 200 OK Status-Code. Im Falle eines Cache-MISS gelangen wir in den *vcl_miss* Zustand. Auch hier müssen wir zunächst prüfen, ob es sich um ein PURGE Request handelt und wenn dem so ist, die Weiterverarbeitung ebenfalls durch ein 200 OK Status-Code zu terminieren. Würden wir die Weiterverarbeitung an dieser Stelle nicht terminieren, würde der PURGE Request zum Origin-Server weiter geleitet und der entsprechende Content ausgeliefert werden. Mit dieser Erweiterung ist es nun möglich, mit einem PURGE Request vom Origin-Server aus gezielt bestimmte Ressourcen aus dem Cache zu löschen.

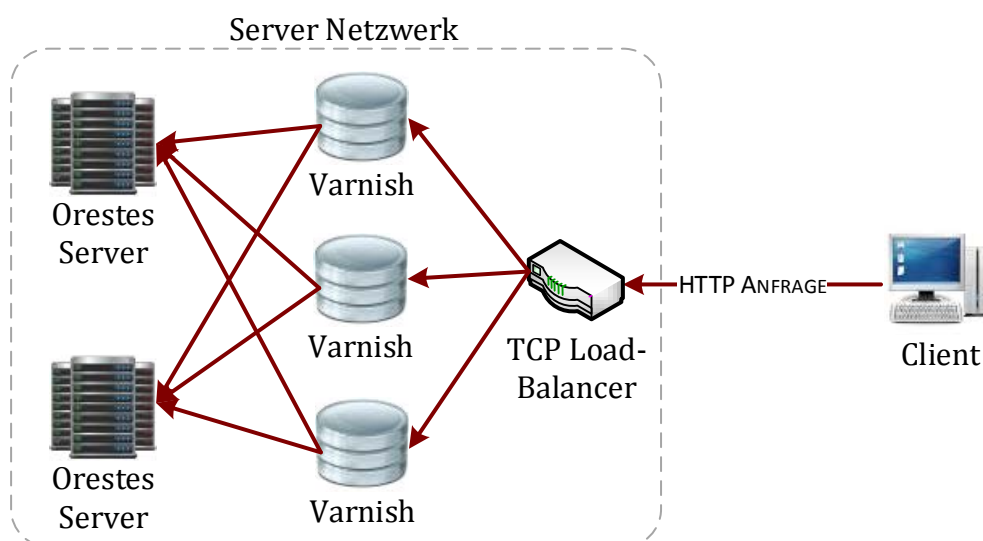


Abbildung 90 Drei Varnish Instanzen, die die HTTP Requests auf zwei Orestes-Server aufteilen.



Varnish kommt von Haus aus mit einem weiteren interessanten Feature für Orestes. Varnish kann neben der Funktionalität als Cache auch Last aufteilen. Dabei können Requests von Varnish an unterschiedliche Origin-Server weitergeleitet werden. Dies kann die Ausfallsicherheit des Systems massiv erhöhen, indem die Orestes-Server redundant aufgestellt werden. In der Abbildung 90 ist ein solches Szenario abgebildet, bei dem ein Client über einen TCP Load-Balancer zufällig auf eine Varnish Instanz weitergeleitet wird. Der jeweilige Varnish überprüft wiederum zunächst, ob der eingehende HTTP Request direkt aus dem Cache beantwortet werden kann. Ist dies nicht der Fall, leitet der Varnish den Request zufällig oder per Round-Robin an einen Orestes-Server weiter. Dadurch, dass die Varnish Caches und die Orestes-Server redundant aufgestellt sind, kann eine wesentlich höhere Ausfallsicherheit gewährleistet werden. Das System ist bereits bei einen aktiven Varnish und einem Orestes-Server vollständig funktionsfähig und kann durch weitere Orestes-Server und Varnish Instanzen beliebig horizontal skaliert werden.

4.2 Content-Delivery-Networks

Content-Delivery-Networks (CDN) [22] haben eine ähnliche Funktionsweise wie Reverse-Proxy-Caches. Sie fungieren ebenfalls als Ersatz für den Origin-Server und werden dementsprechend direkt von den Clients angesprochen. Der Unterschied zu den Reverse-Proxy-Caches besteht darin, dass ein CDN nicht nah beim Server positioniert ist, sondern durch räumliche Verteilung der Caches möglichst nah an den potentiellen Clients positioniert ist. Ein CDN ist somit nicht ein einzelner Server, sondern ein ganzes Netzwerk von Reverse-Proxy-Caches, die stark dezentralisiert sind, aber stets eine gute Anbindung zum Origin-Server haben. Dadurch wird die Netzwerk-Latenz zwischen Client und dem Cache stark reduziert. Die Kommunikation zwischen Client und CDN wird immer über den Cache abgewickelt, der dem Client am nächsten ist. Solche Caches werden in einem CDN als Edge-Knoten bezeichnet. Ein CDN erfüllt bei der Request Verarbeitung gleich mehrere Aufgaben:

- Request-Routing über das CDN zum Origin Server
- Cachen von bereits ausgelieferten Ressourcen
- Auswahl von Ersatzknoten im Falle eines Ausfalls
- Räumliche Last-Verteilung
- Monitoring der eingehenden Request, für die automatische Lastanpassung und Kostenberechnung.

CDNs bestehen aus sehr vielen unterschiedlichen Komponenten, die miteinander interagieren. Dabei ist das CDN, je nach Anforderung und Anbieter, anders aufgebaut, es werden unterschiedliche Netzwerktopologien, Kommunikationsmechanismen und Caching-Strategien eingesetzt. Um die genaue Funktionsweise eines CDNs besser verstehen und analysieren zu können, werden diese Komponenten im Folgenden detaillierter vorgestellt.

4.2.1 CDN Aufbau

Beim Aufbau des Netzwerks können Router und Switches passiv sein, das heißt, die Router und Switches sind nur für das Routing verantwortlich. Dies ist die meist gewählte Lösung,

da sie eine einfache Verwaltung und Integration weiterer Server ermöglicht. Die Router und Switches können aber auch eine aktive Rolle übernehmen, indem sie den Netzwerkverkehr analysieren und je nach Anfrage, die Pakete an spezialisierte Server weiterleiten. Dies ähnelt dem Setup eines Interception-Proxys, bei dem der Router nur den HTTP-Verkehr abfängt und diesen an den Web-Cache weiterleitet.

4.2.2 Server

Es gibt in einem CDN zwei grundsätzliche Arten von Servern, zum einen die Origin-Server, die den eigentlichen Inhalt bereitstellen und die Replikations-Server, die den Inhalt replizieren, wie z.B. Web-Caches, um die Zugriffe auf die Origin-Server zu reduzieren.

4.2.3 Struktur

Aufgrund der Vielfältigkeit und der Anwendungsgebiete gibt es sehr viele verschiedene Arten, wie die einzelnen Server im Netzwerk miteinander interagieren können. Die einzelnen Setups sind in der Abbildung 91 zu sehen. Es gibt die Replikations-Server, die vollkommen transparent, stellvertretend für einen Origin-Server, Request von Clients direkt verarbeiten oder als Proxy fungieren, sofern sie die Request nichts selbst verarbeiten können. Dann gibt es Caches, die nur auf bestimmte Inhalte spezialisiert sind und mit Hilfe eines aktiven Routers den Request an den entsprechenden Cache geroutet wird. Beispiele für solche spezialisierten Caches sind Streaming-Server oder File-Server, die auf ein verteiltes Dateisystem zugreifen. Zudem gibt es noch zwei Verbundarten von Caches, die aus der Inter-Cache-Kommunikation resultieren. Die sogenannten Cache-Proxy-Arrays, bei denen es ein Master-Knoten gibt, der die gesamte Kommunikation mit den anderen Caches abwickelt. Außerdem gibt es Proxy-Meshes, die einem Peer-to-Peer Netzwerk ähneln, bei dem jeder Cache jeder Zeit mit jedem anderen Cache kommunizieren kann. Bei beiden Varianten kommuniziert der Client jeweils mit irgendeinem Cache aus dem Verbund, der den Request mit Hilfe seines eigenen Caches, des Verbundes oder des Origin-Server beantwortet.

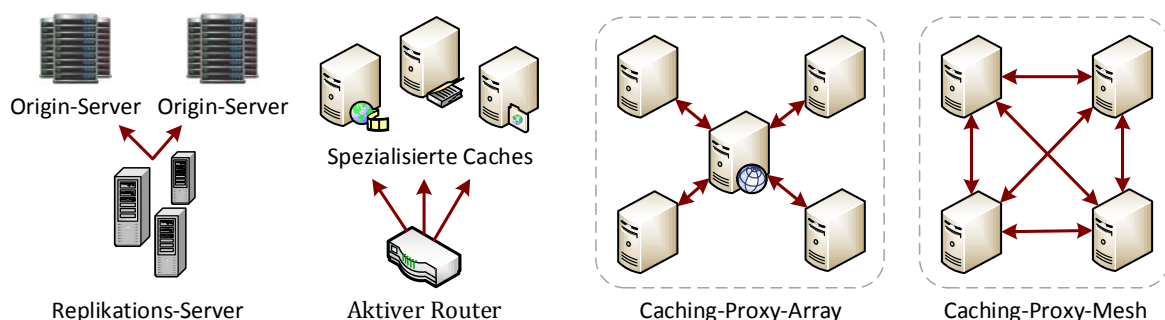


Abbildung 91 Verschiedene Netzwerkstrukturen innerhalb eines CDNs.

4.2.4 Interaktions Protokolle

Für Web-Caches gibt es eine reiche Anzahl an Inter-Cache-Protokollen, die ebenfalls in einem CDN zum Einsatz kommen können. Dies ermöglicht Inter-Cache-Kommunikation, indem ein Cache einen Nachbar-Cache nach einem Eintrag fragen kann, den er selbst nicht besitzt. Dadurch können zusätzliche Anfragen an den Origin-Server vermieden werden,



wenn bereits ein benachbarter Cache den Eintrag besitzt. Zudem gibt es Protokolle, mit denen aktive Netzwerkknoten konfiguriert werden können.

- *Das Network Element Control Protocol (NECP)* [201] ist ein Working Draft der IETF und standardisiert die Netzwerkkommunikation zwischen Servern und Netzwerkelementen wie Switches und Routern. Es ist vor allem für die Steuerung des Load-Balancings innerhalb eines Netzwerkes gedacht. Ein Server kann sich über das Protokoll an einem Load-Balancer, an- oder abmelden oder sich über seine Verfügbarkeit informieren. Hierfür wird eine TCP-Verbindung vom Server zum Netzwerkknoten etabliert und aufrechterhalten. Fällt einer der beiden Teilnehmer aus, kann der jeweils andere Kommunikationspartner handeln. So könnte sich der Server bei einem anderen Load-Balancer anmelden oder im Falle eines Server Ausfalls der Load-Balancer keinen weiteren Netzwerkverkehr an diesen weiterleiten.
- *Web Cache Coordination Protocol (WCCP)* [202] ist ein von Cisco entwickeltes und in ihren Routern implementiertes IP basiertes Protokoll, um Interception-Proxy an einem Router zu registrieren. Dieses Protokoll ist in einer ganzen Reihe von Web-Caches implementiert und eingesetzt. Damit ein Interception-Proxy Requests von einem Router vermittelt bekommt, muss dieser alle 10 Sekunden eine Alive-Nachricht an den Router schicken. Dieser schickt dann über das gleiche Protokoll abgefangene Nachrichten an die registrierten Web-Caches. Werden mehrere Web-Caches an einem Router registriert, kann ein ausgewählter Web-Cache eine Hashtable konfigurieren, anhand dessen der Router entscheidet, an welchen Web-Cache er ein Request weiterleitet.
- *Cache Array Routing Protocol (CARP)* [43] ist ein Protokoll um Cache-Proxy-Arrays zu verwalten. Hierbei wird eine Hashtable zwischen den einzelnen Caches ausgetauscht, die angibt, welche Caches für welchen Hash-Bereich zuständig sind. Stellt ein Client ein Request an einen Cache aus dem Cache-Verbund, so berechnet dieser aus der URL einen Hashwert und leitet den Request an den zuständigen Cache weiter. Dies vermeidet unnötige Duplikate von Cache-Einträgen zwischen den verschiedenen Caches.
- *Internet Cache Protocol (ICP)* [44] ist in RFC 2186 spezifiziert und ist eine UDP-basiertes Protokoll, das vor allem für mehrstufige Proxy-Meshes ausgelegt ist. Stellt ein Client eine Anfrage an einen Cache, fragt dieser zunächst seine Nachbar-Caches nach dem entsprechenden Eintrag, sofern er ihn nicht selber hat. Haben diese den Eintrag ebenfalls nicht, wird der Eltern-Cache befragt usw., bis die Anfrage schlussendlich den Origin-Server erreicht. Alle Caches, die auf dem Weg passiert wurden, cachen den Response ebenfalls für zukünftige Anfragen. Die Idee dabei ist, dass Caches, die sich innerhalb eines Meshes befinden, lokalisiert sind und es somit billiger ist, zunächst seine Nachbarchaches zu befragen, bevor der weiter entfernte Eltern-Cache befragt wird. Das Protokoll beruht allerdings auf HTTP/0.9, das keine HTTP Header vorsieht und somit keine Caching Meta-Informationen des Origin-Servers beachtet.

- *Hyper Text Caching Protocol (HTCP)* [45] ist in RFC 2756 spezifiziert und soll ICP ablösen. Es erfüllt die gleiche Funktionalität wie ICP, ist aber für HTTP/1.0 definiert und kann somit mit Meta-Informationen des Origin-Servers umgehen. Zudem erlaubt es auch mehrere Repräsentationen des Inhaltes zu cachen, was erst ab dieser HTTP Version möglich ist. Es basiert ebenfalls auf UDP (optional auch TCP) und ist ebenfalls für hierarchische Cache-Meshes ausgelegt.
- *Cache Digest* [168] ist ein Cacheeintrag-Austausch-Protokoll. Ein Problem, das ICP und HTCP gemein haben ist, dass sie im Falle von Cache Misses eine große Menge an Netzwerkverkehr erzeugen, da sie alle Ihre Nachbarn zur gleichen Zeit kontaktieren. Dies soll durch Cache Digests vermieden werden, indem regelmäßig zwischen den Cache-Nachbarn ein Bloomfilter ausgetauscht wird. Mit Hilfe des Bloomfilters, können die Caches ermitteln, welcher Nachbar potentiell ein Cacheeintrag zu einem Request haben könnte, um diesen dann gezielt zu kontaktieren. So wird das Netzwerk-aufkommen zwischen den Caches stark reduziert.

4.2.5 Content

Die etablierten CDNs sind meist auf einen der drei folgenden Inhaltstypen optimiert. So gibt es CDNs, die vor allem auf das Cachen und Ausliefern von statischen Web-Content spezialisiert sind, z.B. statische HTML-Seiten, Cascading-Style-Sheets (CSS), JavaScript Dateien oder Bilder. Dann gibt es eine ganze Reihe von CDNs, die auf das Ausliefern von Streaming-Content, wie Audio und Video-Dateien ausgelegt sind. Diese CDNs haben meist sehr optimierte proprietäre Protokolle, um den Streaming-Content zwischen den einzelnen Netzwerkknoten zu synchronisieren. Eine besondere Herausforderung ist das Bereitstellen von Live-Streams. Bei diesen wird der Live-Stream zunächst auf die Knoten des CDNs verteilt und die Clients verbinden sich dann zu den CDN Knoten, die den Stream dann an diese weiterleiten. Zu guter Letzt gibt es CDNs, die sich darauf spezialisiert haben, Services bereitzustellen, um vollwertige Web-Anwendungen zu unterstützen. So gibt es CDNs, die das Clientseitige hochladen von Medieninhalten übernehmen können und so ebenfalls den Origin-Server entlasten.

4.2.6 Verteilung

Einer der wichtigsten Aspekte, der die Qualität eines CDNs ausmacht, ist der Verteilungsgrad der Knoten eines CDNs. Dabei gibt es zwei Ansätze, Single-ISP (Internet Service Provider) und Multi-ISP. Bei Single-ISP betreibt der Anbieter des CDNs ein so großes Netzwerk, dass er in jeder größeren Stadt einen Knotenpunkt positioniert, um Inhalte auszuliefern. Bei einem Multi-ISP Anbieter hat der CDN-Anbieter Verträge mit den unterschiedlichen ISPs, um in deren Netzwerken eigene Knotenpunkte aufstellen zu dürfen. So gehört Akamai [203], [204] zu den größten Multi-ISP Anbietern, indem sie über 50000 Knoten auf der ganzen Welt bei unterschiedlichen ISPs positioniert haben.



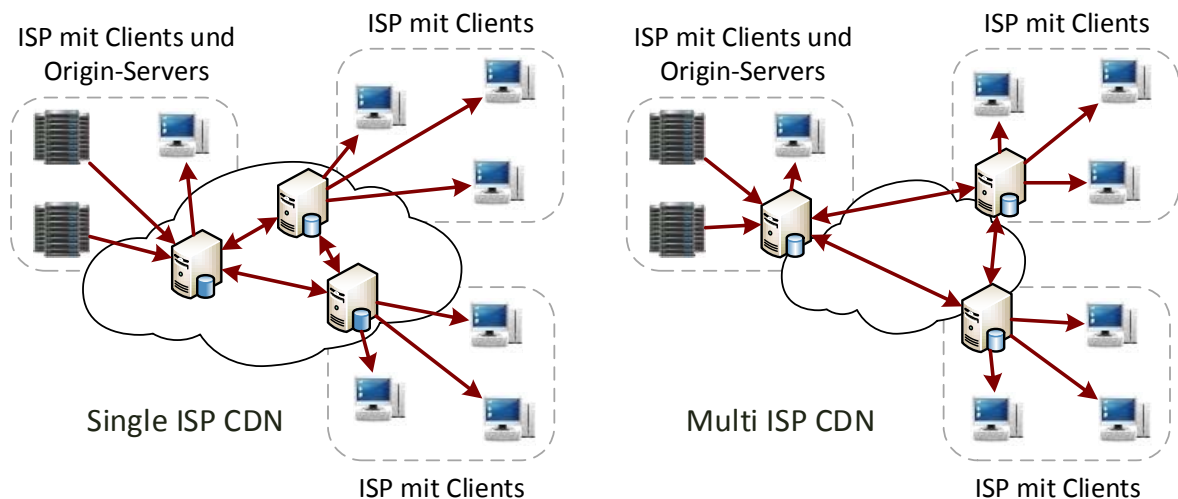


Abbildung 92 Die beiden Verteilungsarten von CDNs.

4.2.7 Auswahl des Contents

Es gibt grundsätzlich zwei Lösungen, welcher Inhalt von einem CDN gecacht wird. Zum einen kann der gesamte Inhalt der Web-Seite gecacht werden. Hierbei übernimmt das CDN im Prinzip die Rolle eines Reverse-Proxy-Caches. In der Regel wird bei dieser Lösung der DNS Eintrag der Web-Seite durch das CDN aufgelöst, indem der nächstgelegene Knoten zum Client alle Anfragen beantwortet und ändernde Anfragen zum Origin-Server weiterleitet. Die andere Möglichkeit ist, dass der Inhalt nur zum Teil von dem CDN ausgeliefert wird. Dies ist vor allem für dynamische Seiten sinnvoll, so wird nur eingebetteter Inhalt von dem CDN ausgeliefert, wie statische Skript-Dateien oder Bilder. Hierfür wird ein separater DNS Eintrag angelegt, mit dem der statische Content dann selektiv verlinkt werden kann. Der eigentliche DNS-Eintrag für die Web-Seite zeigt aber direkt auf den Origin-Server.

4.2.8 Aktualisierung Strategie

Eine weitere wichtige Eigenschaft für CDNs ist die Fähigkeit, Änderungen von Inhalten des Origin-Servers zu ermitteln. Generell kann der Origin-Server seinem Content eine Lebensdauer geben, die bestimmt, wie lange dieser im CDN ohne Revalidierung ausgeliefert werden darf. Daraus resultiert auch die erste Aktualisierungs-Strategie:

- *Periodische Updates*, die zu sehr viel unnötigen Netzwerkverkehr innerhalb des CDNs führen können, der daraus resultiert, aktuellen Content immer wieder zu revalidieren.
- Bei *Änderungspropagierung* wird eine Änderungen vom Inhalt am Origin-Server aktiv an das CDN weitergegeben, wodurch der bestehende Inhalt durch den aktualisierten Content ersetzt wird.
- *Änderung auf Anfrage* kann ein Client auslösen, indem er beispielsweise ein Revalidierungs-Header seinem Request beifügt, der besagt, dass er keinen Content aus dem

Cache erhalten möchte. Dies könnte aber zu einem Sicherheitsrisiko führen, da ein Client so Denial-of-Service (DoS) Attacken gegen das CDN durchführen könnte.

- Durch *Serverseitige Invalidierung* kann der Server dem CDN eine Änderung von Content mitteilen, das daraufhin die entsprechenden Cache-Einträge entfernt. Wird der Content dann erneut angefragt, wird er frisch vom Server geladen und wieder gecacht.

4.3 Netzwerklatenz verschiedener CDNs

Aus den zahlreichen Anforderungen, die an ein CDN gestellt und den Möglichkeiten, wie diese implementiert werden können, ergibt sich eine ganze Reihe von CDN Anbietern, die sich auf ganz bestimmte Anforderungen spezialisiert haben. Einer der wichtigsten Aspekte für Orestes ist eine geringe Netzwerk-Latenz, da über das Orestes-Protokoll in der Regel viele kleine Objekte abgerufen werden, wobei die Latenz ein maßgeblicher Faktor für die Performanz des Protokolls ist.

Um einen Überblick über die bestehenden CDNs und ihre Netzwerk-Latenzen zu bekommen, gibt es von CloudHarmony [69] eine Web-Applikation, mit dessen Hilfe man in erster Linie Cloud-Anbieter vergleichen kann. Sie bieten aber auch einen Dienst an, mit dessen Hilfe CDNs miteinander verglichen werden können. Der Test lädt dabei im eigenen Browser eine immer gleichgroße Datei von einem für das jeweilige CDN spezifischen Web-Server und misst die Gesamtdauer der Anfrage. In Tabelle 10 sind die Testergebnisse, sortiert nach der durchschnittlichen Anfragedauer, zu sehen.

Service	Time (secs)	# of Samples	Min (ms)	Max (ms)	Std Dev	Median (ms)	Avg (ms)
Akamai CDN	0.37	6	36	38	2.65%	38	37.17
Rackspace Cloud CDN	0.36	7	33	41	6.44%	38	37.71
HP Cloud CDN	0.37	7	30	42	10.76%	40	38.86
Edgecast CDN	0.44	7	43	50	5.31%	49	48.29
Highwinds CDN	0.51	7	49	51	1.59%	49	49.57
Edgecast CDN - Small Objects	0.45	7	48	51	1.91%	50	49.71
Limelight CDN	0.48	7	49	52	2.21%	50	50.29
AgileCAST CDN	0.47	7	48	56	5.93%	49	50.43
Amazon CloudFront	0.54	7	51	59	5.09%	55	54.86
CDNvideo	0.57	6	52	67	8.78%	56	57.5
MaxCDN	0.9	5	54	77	16.38%	56	59.6
Internap CDN	0.51	6	61	64	1.94%	63	62.33
Cotendo CDN	0.57	7	62	63	0.78%	63	62.71
BitGravity	0.53	6	61	67	3.38%	63	63.17
CacheFly CDN	0.55	6	53	69	10.22%	69	63.83
Windows Azure CDN	0.56	6	64	67	1.84%	66	65.67



CDN77	0.67	6	66	83	9.51%	67	69.67
Duodecad CDN	0.61	6	70	74	2.08%	73	72.33
CDNetworks CDN	0.74	6	51	138	52.08%	52	76.67
Level 3 CDN	0.76	4	76	123	25.51%	79	89
NGENIX CDN	0.23	2	98	133	21.43%	133	115.5
KT ucloud CDN	1.05	3	347	349	0.33%	349	348.33
NetDNA	1.16	3	111	541	61.84%	504	385.33
SwiftServe CDN	0.79	2	394	395	0.18%	395	394.5
CloudFlare							Test Failed
Dynomesh CDN							Test Failed

Tabelle 10 CloudHarmony CDN Network Latency Tests.

In dem Test ist in der ersten Spalte der jeweilige CDN-Anbieter aufgeführt, gefolgt von der Dauer des Tests für den jeweiligen Anbieter in Sekunden. Der Test stellt dabei maximal 7 gleiche Anfragen an das CDN, die in der darauffolgenden Spalte aufgeführt sind. Anschließend folgt die Minimal- und Maximaldauer einer Anfrage. Aus diesen Ergebnissen wird dann eine Standardabweichung, der Median und die durchschnittliche Anfragedauer bestimmt. Es ist bei diesen Testergebnissen zu beachten, dass diese nicht repräsentativ, sondern nur die Latenzzeiten von dem getesteten Internet-Anschluss aus Hamburg wiedergeben.

4.4 Amazon CloudFront

CloudFront ist ein Service der Amazon Web Services (AWS) und übernimmt die Aufgaben eines CDNs innerhalb dieser Services und lässt sich besonders gut mit den anderen Amazon Web Services kombinieren. Aus dem CloudHarmony™ Inc. CDN Test wird ersichtlich, dass CloudFront zu einem der schnellen Anbieter gehört. Im Folgenden werden wir die relevanten Funktionen von CloudFront vorstellen und erläutern, inwiefern diese mit dem Orestes-Protokoll harmonieren.

4.4.1 Infrastruktur

CloudFront ist nach dem Prinzip von Origin-Server und Replikations-Server aufgebaut. Als Origin-Server kommt vor allem der Storage Service Amazon S3 zum Einsatz. Allerdings kann als Origin-Server auch jeder beliebige andere HTTP Web-Server eingesetzt werden. Der Orestes-Server kann somit z.B. in der Amazon Cloud EC2 aufgesetzt werden und als Origin-Server für das CDN dienen.

CloudFront ist eine Single-ISP Anbieter, der an großen Knotenpunkten größere Rechenzentren unterhält, um den Netzwerkverkehr zu verarbeiten. CloudFront hat vor allem dort Knotenpunkte, an denen auch EC2 Instanzen gemietet werden können. Es gibt aber zusätzliche Knotenpunkte nur für CloudFront. Die folgende Tabelle gibt eine Übersicht über alle aktuellen Knotenpunkte von CloudFront:

USA	Europa	Asien	Südamerika
Ashburn, VA (2)	Amsterdam (2)	Hongkong	São Paulo
Dallas/Fort Worth, TX (2)	Dublin	Osaka	
Jacksonville, FL	Frankfurt (2)	Singapur (2)	
Los Angeles, CA (2)	London (2)	Sydney	
Miami, FL	Madrid	Tokio	
New York, NY (2)	Mailand		
Newark, NJ	Paris (2)		
Palo Alto, CA	Stockholm		
San Jose, CA			
Seattle, WA			
South Bend, IN			
St. Louis, MO			

Tabelle 11 Knotenpunkte von Amazons CDN CloudFront.

4.4.2 Unterstützte Content-Typen

CloudFront unterstützt nur das Cachen von eingebettetem Inhalt, da es nur die beiden HTTP-Methoden HEAD und GET unterstützt. Es kann also nicht als Proxy für eine vollwertige Web-Applikation fungieren, da es keine ändernden Operationen zulässt. Dies stellt zunächst ein Problem für das Orestes-Protokoll dar, da es auch ändernde Operationen vorsieht und auch benötigt, sofern Objekte geändert werden sollen. Um mit dieser Situation umzugehen, kann in Orestes eine separate CDN-URI konfiguriert werden, die dann ausschließlich für lesenden Zugriff auf die Datenbank verwendet wird. So könnte die CDN-URI für einen Orestes-Server, der unter `http://try.orestes.info/` erreichbar ist, `http://cdn.orestes.info/` lauten. Sobald ein Client eine nicht lesende Operation durchführen möchte, verwendet er die alternative URI. Diese wird dann nicht von dem CDN beantwortet, sondern direkt von dem Origin-Server.

In CloudFront können sogenannte Distributions angelegt werden [205]. Es gibt zum einen Download Distributions und Streaming Distributions. Hiermit unterstützt CloudFront zwei der drei möglichen Konzepte für die Behandlung von Content. Mit den Download Distributions kann statischer Inhalt gecacht und ausgeliefert werden. Mit den Streaming Distributions können Medieninhalte, wie Audio- und Video-Dateien über das CDN ausgeliefert werden. Für Orestes sind aber nur die Download Distributions von Interesse, da einzig statische Objekte über das Orestes-Protokoll persistiert und ausgeliefert werden können.

4.4.3 Cache-Verhalten und Aktualisierung

CloudFront interpretiert standardmäßig die Cache-Informationen von Objekten, die vom Origin-Server mit den HTTP-Metadaten bereitgestellt werden, um die Objekte in dem Cache vorzuhalten. So kann der Origin-Server mit dem Cache-Control-Header das CDN anweisen, wie lange ein Objekt gecacht werden darf. Allerdings kann auch eine feste *Time to Live (TTL)*



für alle Objekte für das CDN konfiguriert werden, damit das CDN mit einem anderen TTL-Wert konfiguriert werden kann, als die übrigen Web-Caches. Stellt ein Client eine lesende Anfrage an das CDN und der verarbeitende Knotenpunkt hat das Objekt noch nicht in seinem Cache, so lädt er es vom Origin-Server und hält es dann für die konfigurierte TTL vor.

CloudFront interpretiert auch die Versionsinformationen, wie den ETag- und LastModified-Header und nutzt diese, um Cacheinträge nach dem HTTP-Standardverfahren zu revalidieren. Wird ein veraltetes Objekt von einem Client angefragt, stellt der entsprechende Knotenpunkt ein Conditional-HTTP Request an den Origin-Server. Dieser kann die Aktualität des Objektes bestätigen oder eine neuere Version dem Knotenpunkt zurückschicken. D.h. Aktualisierungen werden spätestens nach Ablauf der TTL im CDN propagiert. Dies kann für den statischen Web-Content ausreichen, aber für Orestes reicht dieser Mechanismus alleine jedoch nicht. Denn wenn ein Objekt geändert wird, soll diese Änderung so schnell wie möglich sichtbar werden. Da das CDN nur für Lesezugriffe verwendet und für alle weiteren Operationen der Origin-Server direkt angesprochen wird, könnte das CDN auch für alle gerade geänderten Objekte umgangen werden. Diese Umgehung müsste dann aber für die gesamte TTL Zeit durchgeführt werden, da Objekte erst nach Ablauf dieser Zeitspanne wieder revalidiert werden. Dies kann die Performanz des Gesamtsystems aber stark reduzieren, da dadurch viele Objekte nicht aus dem CDN-Cache kommen können.

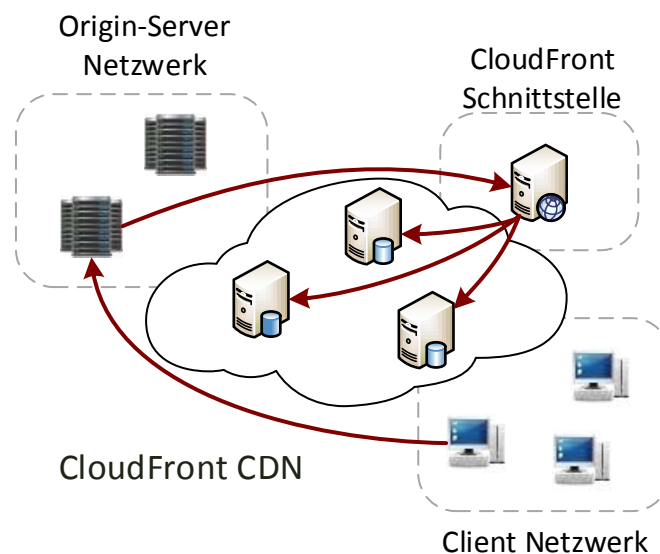


Abbildung 93 Ablauf einer Invalidation in CloudFront.

Dieses Problem kann vermieden werden, indem Objekte vorzeitig aus dem Cache entfernt werden. Hierfür unterstützt CloudFront das Konzept der serverseitigen Invalidation. CloudFront stellt eine eigene Schnittstelle zur Verfügung, über die der Server, der eine Änderung erfährt, einen Invalidation-Request absetzen kann. Dieser Request wird an eine von CloudFront bereitgestellte REST-API gestellt, der dann die Invalidation der Knotenpunkte initiiert. Dies ist in der Abbildung 93 gezeigt. Eine solch angestoßene Invalidation benötigt unter Umständen einige Minuten, bis diese alle Knotenpunkte erreicht hat. Das System verhält sich somit *eventually consistent*, es erreicht irgendwann einen konsistenten Zu-

stand. Dieses Verfahren ist vor allem durch neuere Datenbanksysteme wie DynamoDB [73] oder Casandra [63] geprägt ist.

4.4.4 Invalidation-API

Invalidierungsaufträge werden in CloudFront als Batch-Anweisungen in Auftrag gegeben. Dabei muss jedes zu invalidierende Objekt explizit aufgelistet werden. In CloudFront können maximal drei Aufträge gleichzeitig ausgeführt werden und pro Auftrag ist die Anzahl der Invalidationen auf 1000 limitiert. Um einen Invalidationauftrag aufzugeben, wird eine HTTP POST-Request an die CloudFront-Domain *cloudfront.amazonaws.com* geschickt. In Abbildung 95 ist der Inhalt des Invalidation-Requests beispielhaft aufgezeigt. Die REST-API, mit deren Hilfe CloudFront verwaltet wird, ist komplett XML-basiert und erwartet somit auch den Invalidationauftrag als XML-Dokument. Mit dem einleitenden XML-Tag *InvalidationBatch* wird jeder Invalidationauftrag eingeleitet. Dieser enthält ein *Path*- und ein *CallerReference*-Tag. Unter dem *Path*-Tag wird zunächst mit *Quantity* die Anzahl der zu invalidierenden Objekte spezifiziert, in diesem Fall also drei. Darunter folgen in einem *Items*-Tag, die Referenzen der zu invalidierenden Objekte in einem *Path*-Tag. In diesem Beispiel wird ein Invalidationauftrag für die drei Coffee-Objekte erzeugt, die zuvor durch einen Client am Origin-Server geändert wurden. Somit werden hier die drei Referenzen von diesen Objekten aufgelistet. Mit Hilfe von *CallerReference* wird dem Request ein eindeutiger Identifier zugeordnet, damit fälschlicherweise doppelt gesendete Requests ignoriert werden können (Idempotenz).

```
<InvalidationBatch xmlns="http://cloudfront.amazonaws.com/doc/2012-07-01/">
  <Paths>
    <Quantity>3</Quantity>
    <Items>
      <Path>/db/test.persistent/Coffee/1125899906847742</Path>
      <Path>/db/test.persistent/Coffee/1125899906847739</Path>
      <Path>/db/test.persistent/Coffee/1125899906847757</Path>
    </Items>
  </Paths>
  <CallerReference>1353248717084</CallerReference>
</InvalidationBatch>
```

Abbildung 94 Ein Beispiel-Invalidationauftrag an CloudFront

CloudFront teilt dem Request eine eindeutige Id zu und initiiert die Invalidation. Der Request wird mit einem Response bestätigt, der die Id, das Erstellungsdatum, den Status der Invalidation und die zu invalidierenden Referenzen enthält. Mit Hilfe der Id kann an der REST-API jederzeit der Status der Invalidation abgerufen und verfolgt werden.

4.4.5 Evaluation

Um zu verstehen, wie ein konkretes CDN in das Orestes-Protokoll eingebunden werden kann, ist CloudFront ein ziemlich gut geeignetes CDN, da es eine leichtgewichtige API zur Konfiguration und alle wichtigen Eigenschaften, die für das Orestes-Protokoll benötigt wer-



den, aufweist. So versteht CloudFront alle Caching-Metainformation, die Orestes für die gespeicherten Objekte bereit stellt und ermöglicht die vorzeitige Invalidierung von Objekten.

CloudFront bringt aber für den produktiven Einsatz einige Nachteile mit sich. Invalidierungs-Requests sind in CloudFront aus zweierlei Gründen nicht für Anwendungen mit hochfrequenten Änderungen geeignet. Denn für jedes zu invalidierende Objekt muss eine Gebühr von \$0,005 [206] gezahlt werden und die Anzahl der gleichzeitig durchführbarer Invalidierungen ist limitiert. Daraus folgt somit maßgeblich die maximale Anzahl der Änderungsoperationen, die über das Orestes-Protokoll abgewickelt werden können. Ein weiteres Problem ist, dass CloudFront nicht als vollwertiger Proxy interagieren kann, da es keine ändernden HTTP-Methoden zulässt. Dies zwingt den Client dazu, immer mit zwei unterschiedlichen Domains gleichzeitig arbeiten zu müssen, eine für lesende und eine weitere für ändernde Operationen. Zudem müssen Lese-Anfragen von gerade geänderten Objekten ebenfalls über die alternative URI abgewickelt werden, was insgesamt sehr viel Logik im Client bedarf, um alle diese Fälle abzudecken. Da das Ziel des Orestes-Protokolls ist, eine möglichst leichtgewichtige API zu bereitzustellen, würde dies diesem Ansatz widersprechen. Für das Orestes-Protokoll ist also ein CDN, das wie ein vollwertiger Proxy interagiert und zudem schnelle Invalidierungen von Objekten erlaubt, eine wesentlich bessere Wahl.

4.5 Der Orestes Invalidation-Service

In den vorherigen Kapiteln wurden die beiden Invalidierungsmechanismen von dem Reverse-Proxy-Cache Varnish und dem CDN CloudFront vorgestellt. Diese sind zwar jeweils nur ein repräsentatives Beispiel für die Funktionsweise, verdeutlichen aber schon, dass es für die Invalidierung eines Caches keinen etablierten Standard gibt. So musste im Varnish die Funktionalität erst konfiguriert/implementiert werden, der Apache Traffic Server unterstützt fest die PURGE-Methode zum Invalidieren [207] und CloudFront hat, wie die meisten CDNs, eine anbieterspezifische Schnittstelle, um Invalidierungsaufträge aufzugeben. Um diesen Anforderungen gerecht zu werden, haben wir den Invalidation-Service in Orestes eingeführt. Diese erlaubt es, über eine Schnittstelle, beliebig viele *Invalidator* Implementierungen zu registrieren. In diesen kann dann die Cache-Schnittstellenspezifische Logik implementiert werden. Der Aufbau des Invalidation-Service ist in der Abbildung 95 zu sehen. Diese zeigt die einzelnen Invalidatoren, die am Invalidation-Service registriert sind. Dabei gibt es für jede Cache-Schnittstelle, die vom Server angesprochen werden kann, einen verantwortlichen Invalidator, der alle Invalidierungen in die Schnittstellen spezifischen Aufrufe umwandelt.

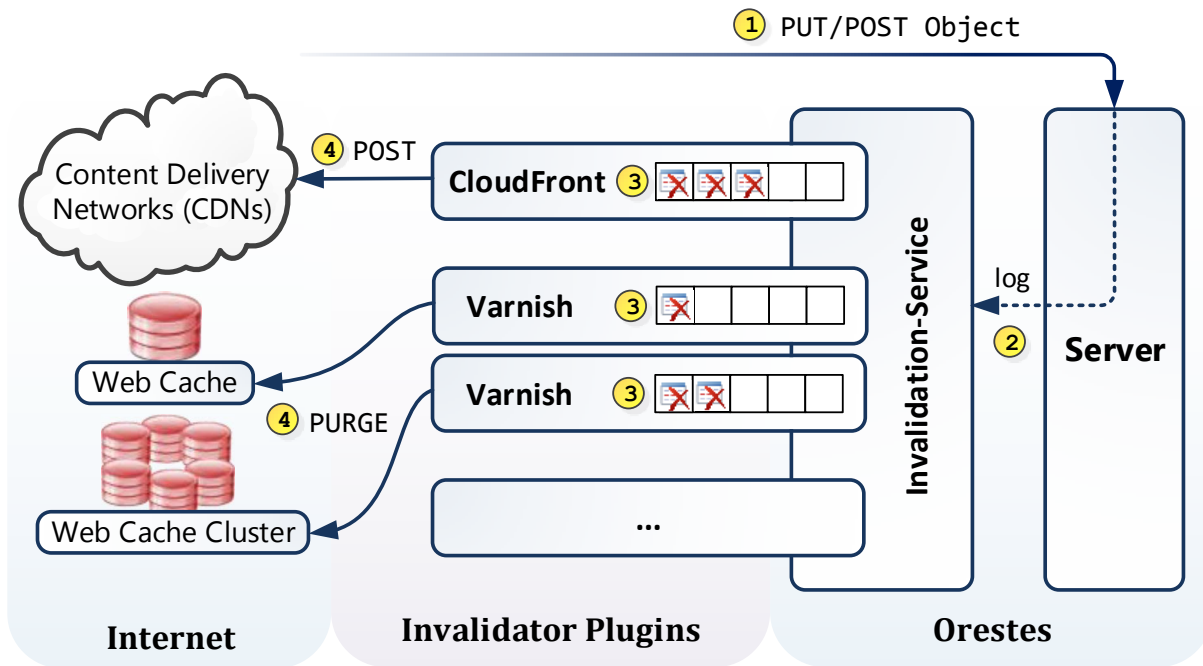


Abbildung 95 Funktionsweise des Invalidation-Service in Orestes.

Der Invalidation-Service durchläuft für eine Invalidation folgende Schritte:

1. Erfährt der Orestes-Server eine Änderung durch das Schreiben eines Objektes, so informiert er den Invalidation-Service zunächst über die durchgeführte Änderung.
2. Dieser erzeugt dann für jeden registrierten Invalidator einen entsprechenden Invalidation-Auftrag und hinterlegt diesen in einer Warteschlange, in der alle ausstehenden Invalidation-Aufträge, für den entsprechenden Invalidator, gesammelt werden.
3. Der jeweilige Invalidator entnimmt den Invalidation-Auftrag der Warteschlange und wandelt diesen in den entsprechenden Schnittstellen-Aufruf um.
4. Der Aufruf erreicht die Cache-Schnittstelle und eine entsprechende Invalidation im Cache-System wird angestoßen oder das geänderte Objekt wird direkt aus dem Cache entfernt.

Der gesamte Prozess verläuft somit komplett asynchron und entkoppelt von der eigentlichen Verarbeitungslogik des Orestes-Servers. Dies sorgt dafür, dass die eigentliche Verarbeitung nicht durch den Invalidation-Service beeinflusst werden kann und somit beispielsweise ein Cacheausfall nicht das ganze System blockiert. Zudem ermöglicht es die Auslagerung des Invalidation-Services auf einen anderen Server, falls die Invalidationen zu einem Performanceengpass werden.

4.6 Web-Caching und CDNs

Wir haben gezeigt, dass Orestes sowohl mit Forward- und Reverse-Proxy-Caches, als auch mit CDNs harmonieren kann. Die Frage, die noch nicht beantwortet wurde ist, ob sich CDNs mit den anderen Web-Caches in einem Orestes-Setup vertragen können und ob sich weitere



Vorteile daraus erzielen lassen. In der Abbildung 96 ist ein mögliches aber sehr vollständiges Orestes-Setup aufgezeigt. Auf der linken Seite ist ein Client-Netzwerk zusehen, bei dem alle Clients über ein Forward-Proxy-Cache, in diesem Fall Squid, an das Internet angebunden sind. Dies ist eine übliche Aufstellung, wie sie in größeren Netzwerken oft vorzufinden ist. Hinter dem Squid befindet sich das CDN, das für den Orestes-Server Anfragen entgegennimmt und sie im besten Fall direkt aus seinen lokalen Caches beantworten kann. Auf der anderen Seite befindet sich das Origin-Server Netzwerk, in dem der Orestes-Server steht. Clients können aber mit diesem nicht direkt kommunizieren, sondern sie kommunizieren immer mit einem Reverse-Proxy-Cache, der die Anfragen für den Orestes-Server stellvertretend entgegennimmt. In diesem Fall übernehmen dies zwei Varnish Instanzen.

Will ein Client nun ein Objekt laden, so ruft er zunächst den BloomFilter (Abk. BF. in den Abbildungen) vom Server ab. Dieser enthält aber keine Elemente und so kann er einfach eine lesende Anfrage an den Server schicken. Diese wird zunächst von dem Squid abgefangen, der zunächst überprüft, ob er das Objekt bereits im Cache vorliegen hat. In diesem Fall hat er keinen entsprechenden Eintrag in seinem lokalen Cache gefunden und leitet die Anfrage weiter. Der nächstgelegene Knoten des CDNs fängt die Anfrage wieder ab und überprüft ebenfalls, ob er diese direkt aus dem Cache beantworten kann, doch dies ist wieder nicht der Fall und er muss die Anfrage weiter an den Server leiten. Nun kommt die Anfrage im Server Netzwerk an, wo diese von einem der Reverse-Proxy-Caches entgegen genommen wird. Auch dieser überprüft zunächst sein Cache, muss die Anfrage aber ebenfalls weitergeben. Sie erreicht letztendlich den Orestes-Server. Dieser lädt das angefragte Objekt aus der Datenbank und sendet es an den Client zurück. Alle beteiligten Caches, die auf dem Weg zum Server passiert wurden, cachen nun das angefragte Objekt. Würde ein zweiter Client nun das gleiche Objekt erneut anfragen würde er dieses direkt aus dem Squid Cache bekommen. Ein Client, der aus derselben Region kommt, würde es dementsprechend von dem nächstgelegenen CDN-Knoten erhalten.

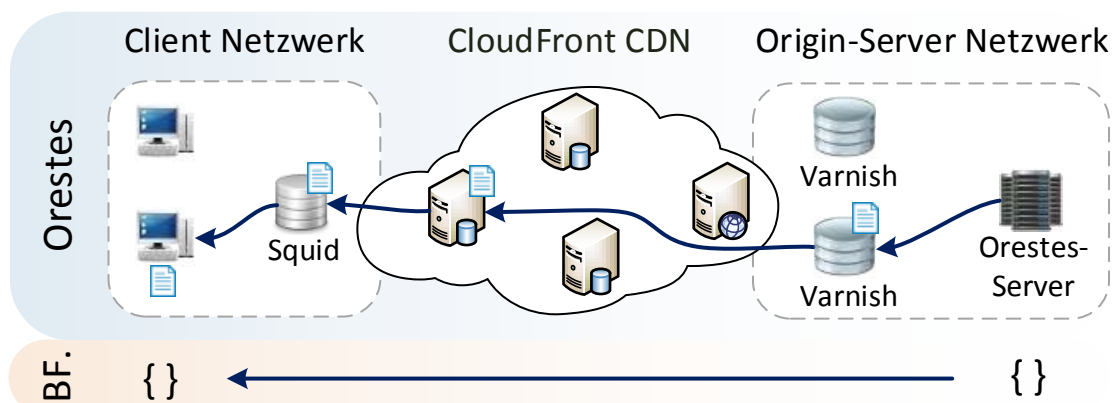


Abbildung 96 Eine lesende Anfrage eines noch nicht gecachten Objektes.

Der Client führt nun eine Änderung an dem Objekt aus und will dies zurück in die Datenbank schreiben. Die Situation ist in der Abbildung 97 gezeigt. Da in diesem Beispiel CloudFront als CDN eingesetzt wird, das keine schreibenden Operationen zulässt, muss der Client

seine Änderung an eine andere URI schicken, um das CDN zu umgehen. Dies bedeutet aber für den Squid, dass er den Schreibvorgang zwar sieht, seinen Cacheeintrag des Objektes aber nicht invalidiert, da der Schreibvorgang auf einer anderen URI erfolgt. Der Schreibvorgang passiert das weitere Netzwerk ohne, dass CloudFront, aus den bereits genannten Gründen, involviert ist. Im Servernetzwerk nimmt wieder ein Varnish die Anfrage entgegen, dieser kann so konfiguriert werden, dass er die beiden unterschiedlichen URIs als identisch ansieht, indem die eine Domain als Alias für die andere definiert wird. Er kann somit seinen Cacheeintrag, wie zuvor invalidieren und leitet anschließend den Request an den Orestes-Server weiter. Der Orestes-Server nimmt die Schreibenanfrage entgegen, validiert die Version und führt den Schreibvorgang durch. Der Schreibvorgang wird also genauso behandelt, als ob nur ein CDN ohne weitere Web-Caches eingesetzt würde. Dies führt aber dazu, dass Forward-Proxy-Caches bei einem Schreibzugriff ihre Cacheeinträge nicht selbständig invalidieren können.

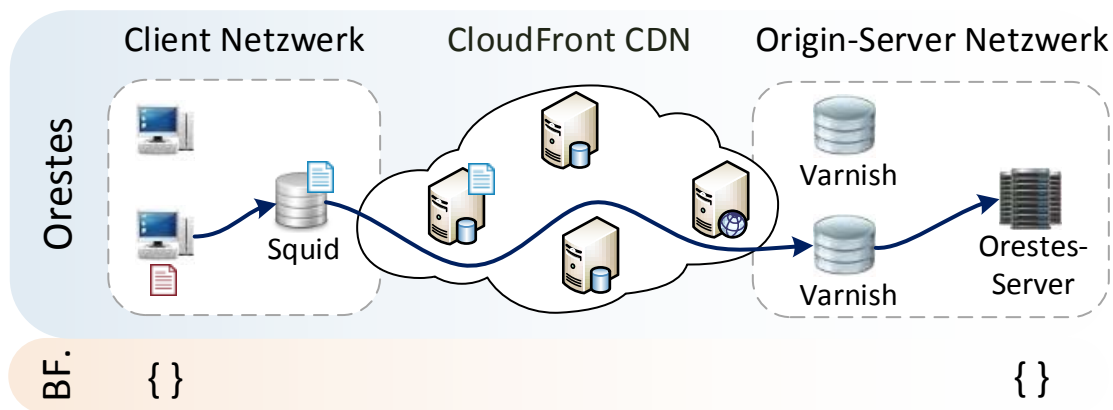


Abbildung 97 Ein Schreibvorgang eines Objektes.

Der Server stößt nach dem erfolgreichen Ausschreiben des Objektes eine Invalidierung an. Diese ist in der Abbildung 98 dargestellt. Der Orestes-Server schickt eine Invalidierung an die beiden Varnish-Instanzen, die daraufhin ihren jeweiligen Cacheeintrag entfernen und sich somit bei der nächsten Anfrage wieder eine frische Version des Objektes vom Server holen müssen. Das CDN bekommt ebenfalls eine Invalidierungs-Aufforderung. Da die Knoten des CDNs verteilt sind und die Invalidierungs-Anfragen immer gebündelt gestellt werden müssen, braucht es jedoch einige Zeit bis die Invalidierung in den einzelnen Knoten tatsächlich durchgeführt wird. Würde ein Client nun erneut das Objekt lesen, so würde es die veraltete Version aus dem Squid oder CDN bekommen. Um dies zu verhindern, pflegt der Orestes-Server den BloomFilter, der von den Clients vor den eigentlichen Lesezugriffen und zu Beginn der Transaktion abgerufen wird. In diesem wird ebenfalls vom Server das geänderte Objekt vermerkt.



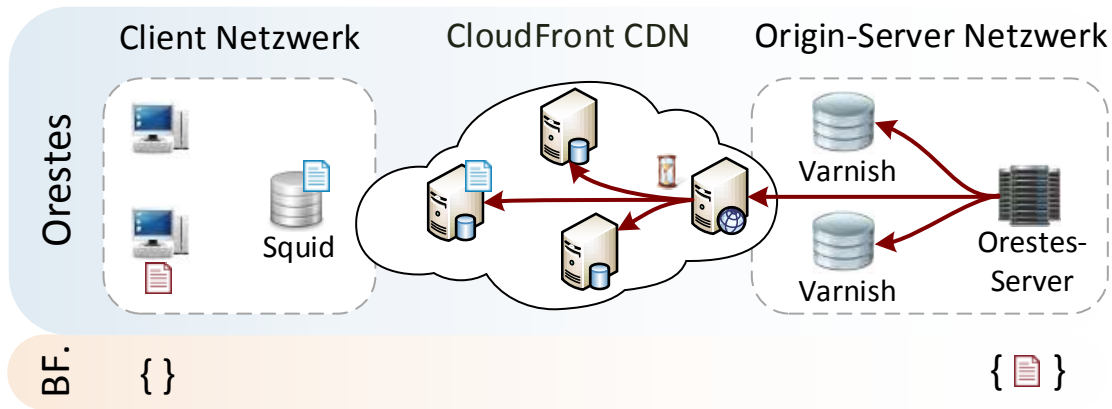


Abbildung 98 Ablauf einer Invalidierung nach dem ein Objekt geschrieben wurde

Will ein Client nun das geänderte Objekt lesen, so ruft er zunächst, wie in der Abbildung 99 zu sehen ist, wieder den BloomFilter ab. Dieser enthält diesmal aber das geänderte Objekt. Somit weiß der Client, dass das Objekt vor kurzem geändert wurde und die Invalidierung u.U. noch nicht abgeschlossen ist. Er umgeht somit das CDN, indem er die alternative URI auch für die lesende Anfrage benutzt. Der Squid besitzt für die alternative URI keinen Cacheeintrag und gibt die Anfrage deshalb weiter. Die Anfrage passiert das weitere Netzwerk ungehindert und erreicht das Origin-Server Netzwerk. Hier nimmt einer der Varnish-Caches die Anfrage entgegen. Dieser hat aber seit der Invalidierung noch keine neue Version abgerufen und gibt deswegen die Anfrage an den Server weiter. Dieser gibt das aktuelle Objekt zurück, das vom Varnish wieder gecacht wird. Das CDN bekommt von dem neuen Objekt nichts mit, da es über die alternative URI angefragt wurde, und cacht dieses somit auch nicht. Der Squid cacht das geladene Objekt ebenfalls nicht, da das Objekt für die Zeit der Invalidierung als nicht cachebar ausgeliefert wird. Der alte Cacheeintrag wurde aber noch nicht entfernt und würde bei der nächsten normalen Anfrage wieder ausgeliefert werden. Hier kommt dann die konfigurierte Lebenszeit für die Objekte ins Spiel, die dafür sorgt, dass das Objekt, nach Ablauf dieser Zeit, aus dem Cache verdrängt wird.

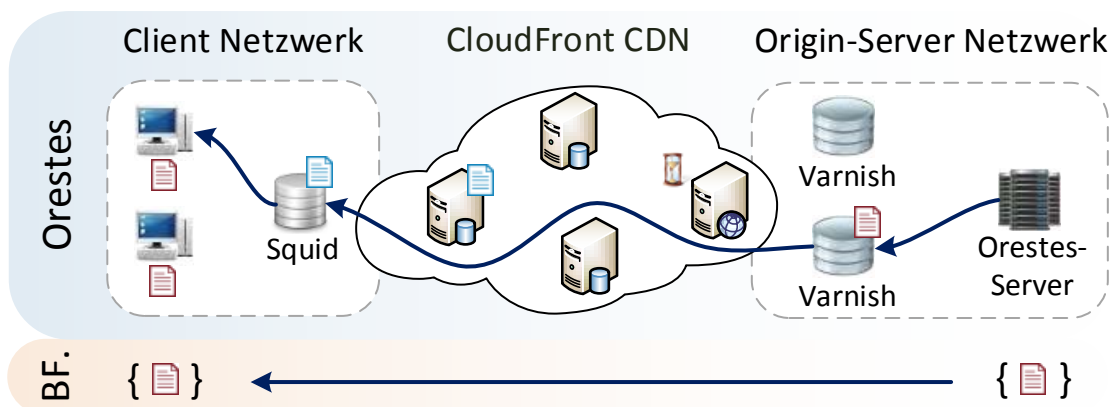


Abbildung 99 Ablauf einer lesenden Anfrage, wenn das CDN veraltete Daten enthält

Auch hier ist wieder zu sehen, dass zusätzliche Schwierigkeiten auftreten können, weil CloudFront nicht als vollwertige Proxy interagiert. Lokale Caches bekommen von einer Än-

derung nichts mit und invalidieren dadurch ihre Cache-Einträge nicht automatisch. Dies würde aber nicht der Fall sein, wenn das eingesetzte CDN als vollwertiger Proxy interagieren würde und Schreibvorgänge somit nicht über eine alternative URI abgewickelt werden müssten.

Es ist aber dennoch zu sehen, dass der Einsatz von weiteren Web-Caches, neben dem eigentlichen CDN durchaus praktikabel ist. Allerdings ergeben sich aus der aktuellen Konfiguration keinerlei Vorteile, zusätzliche Web-Caches einzusetzen. Dies ändert sich jedoch, wenn für die unterschiedlichen Web-Caches unterschiedliche Lebenszeiten für die Objekte, definiert werden.

- Die Reverse-Proxy-Caches können so konfiguriert werden, dass sie alle Anfragen, auch die, die eine explizite Revalidierung verlangen, direkt beantworten. Zudem kann die Lebenszeit für Objekte für diese auf unbegrenzte Zeit gesetzt werden, denn die Varnish Caches werden von dem Orestes-Server in dem Moment über die Änderung informiert, in dem diese passieren. Das heißt, dass Stale-Reads auf der Ebene nahezu ausgeschlossen sind und der Orestes-Server entlastet wird, da er keine Revalidierungs-Requests mehr beantworten muss.
- Das CDN kann sehr lange Lebenszeiten für die Objekte erhalten, da dieses ebenfalls von dem Orestes-Server invalidiert werden kann. Da eine Invalidierung Zeit beansprucht, wird diese Zeit mit Hilfe der BloomFilter überbrückt.
- Die Forward-Proxy-Caches sollten eher eine kurze Lebensdauer für Objekte erhalten, da diese nicht vom Server invalidiert werden können. Dies könnte sonst zu vielen Transaktionsabbrüchen aufgrund von Stale-Reads führen. Da mit Bloomfiltern die Einträge für geänderte Objekte so lange im BloomFilter verbleiben, wie die Lebensdauer der Objekte, werden Stale-Reads bei Einsatz von Bloomfiltern vollständig vermieden.

Somit ergibt sich die optimale Konfiguration für die Lebenszeit von den Objekten für Forward-Proxy-Caches genau aus der maximalen Zeit, die das CDN zum Invalidieren braucht. Kann das CDN hingegen die Cacheeinträge sofort invalidieren, kann eine normale minimale Lebenszeit für die Forward-Proxy-Caches definiert werden. Denn genau für diese Zeit existieren entsprechende Einträge im BloomFilter, die das Revalidieren für Forward-Proxy-Caches erzwingen. Für diesen Zeitraum übernehmen die Reverse-Proxy-Caches die Verarbeitung aller Leseanfragen. Ist die Invalidierung des CDNs vollzogen, kann dieses künftige Leseanfragen wieder konsistent beantworten. Die Forward-Proxy-Caches haben aufgrund der kurzen Lebensdauer spätestens dann ihre Cache-Einträge mindestens einmal revalidiert. Das System verhält sich insgesamt zu jedem Zeitpunkt konsistent.

4.7 Evaluation

Wir haben zwei Verfahren implementiert und vorgestellt, die es erlauben, veraltete Objekte aus den Web-Caches zu verdrängen und eine Invalidierung zu erzwingen. So können die



Forward-Proxy-Caches mit Hilfe der BloomFilter und die Reverse-Proxy-Caches durch serverseitige Invalidierungen aktuell gehalten werden. Um die Implementierungen zu testen und zu evaluieren, haben wir einen Test geschrieben, der sich des bereits bekannten Szenarios des sozialen Netzwerkes aus unserem Master-Projekt bedient. Dabei werden in dem Test 25 Clients initiiert, die den News-Feed ihres besten Freundes beobachten, indem sie das Profil des Freundes alle 5 Sekunden laden und es auf neue Post-Einträge prüfen. Entdecken sie dabei einen neuen Post-Eintrag, so veröffentlichen sie diesen in ihrem eigenen News-Feed. Die Clients sind durch die Freundschaften so verbunden, dass sie eine Kette bilden. Der erste Client veröffentlicht jede Minute einen neuen Beitrag, der von den weiteren Clients immer weitergegeben wird. Wir messen nun, wie lange die einzelnen Nachrichten brauchen, bis sie den nächsten Client erreichen, die Verbreitungszeiten, wie lange eine Nachricht insgesamt unterwegs ist, die Gesamtdauer und wie viele Transaktionsabbrüche durch das Lesen von veralteten Objekten aufgetreten sind. Da die einzelnen Profile sehr oft abgerufen werden, landen diese sofort im Web-Cache, der diese für zwei Minuten cachen darf. Liest ein Client eine veraltete Version des Profils, kann er dies zum Ende der Transaktion validieren lassen, was in dem Fall zum Transaktionsabbruch führt.

Zum Einsatz für den Test kam der Reverse-Proxy-Cache Varnish als Amazon-EC2 Instanz in der Region Irland. Der Orestes-Server und das Datenbankbackend VOD, sind ebenfalls als EC2 Instanz in derselben Cloud aufgesetzt. Die Instanzen sind jeweils mit einer ECU und 1,7 GB Speicher ausgestattet. Die 25 Clients liefen auf einem Rechner, der aus Hamburg auf den Varnish zugegriffen hat. Dieser hat die Requests, entweder direkt oder mit Hilfe des Orestes-Servers beantwortet. Bei dem Test wurden folgende drei Strategien eingesetzt und getestet, um den Cache zu invalidieren:

1. Ein intelligenter Client liest die entsprechenden Objekte und lässt diese zum Transaktionsabschluss vom Server validieren. War eines der gelesenen Objekte veraltet, so wird die Transaktion zurückgesetzt und der Client erhält die Informationen der veralteten Objekte. Er führt die Transaktion anschließend sofort noch einmal aus, allerdings fordert er eine Revalidierung der Cacheeinträge der veralteten Objekte. Der Varnish erlaubt in diesem Szenario clientseitige Invalidierungen und verhält sich somit wie ein Forward-Proxy-Cache.
2. Der Client lädt sich zu Beginn jeder Transaktion einen Bloomfilter vom Server, der die kürzlich geänderten Objekte enthält. Der Client fragt somit von vorn herein die geänderten Objekte mit einer Revalidierungsauforderung von dem Varnish an. Auch hier erlaubt der Varnish die clientseitige Invalidierung.
3. Der Client hat keine besondere Funktionalität implementiert, sondern der Server invalidiert bei jeder eingehenden Änderung den entsprechenden Cacheeintrag im Varnish. Der Varnish erlaubt in diesem Fall nur die serverseitige Invalidierung.

Die Clients fragen immer, zeitversetzt um 2,5 Sekunden zu ihrem vorherigen Client, die Pinnwand ab. Somit braucht eine veröffentlichte Nachricht in der Regel 2,5 Sekunden, bis

der nächste Client diese veröffentlicht hat. Eine Nachricht muss insgesamt 25 Clients passieren, somit ergibt sich eine erwartete Gesamtdauer von:

$$\text{Gesamtdauer} = 24 * 2,5s = 60s$$

Allerdings kommt noch die Verarbeitungs- und Übertragungszeit hinzu. In der Abbildung 100 werden die drei Strategien durch einen Box-Whisker-Plot miteinander verglichen. Die Box deckt dabei die mittleren 50% aller Verbreitungszeiten ab. Der Whisker deckt die weiteren 1.5fachen Verbreitungszeiten außerhalb der Box ab. Darüber hinausgehende Verbreitungszeiten gelten als Ausreißer. Unten in der Darstellung in Blau sind die einzelnen Verbreitungszeiten der Nachricht von einem, zum nächsten Client mit der ersten Strategie aufgezeigt (in Millisekunden). Darüber folgt in Grün die mittlere Strategie mit BloomFiltern und in Orange oben die Zeiten der serverseitigen Invalidierungsstrategie. Mit der grauen Linie ist die Zeit angedeutet, nach der ein Client anfangen sollte, eine Nachricht weiter zu geben (2,5 Sekunden).

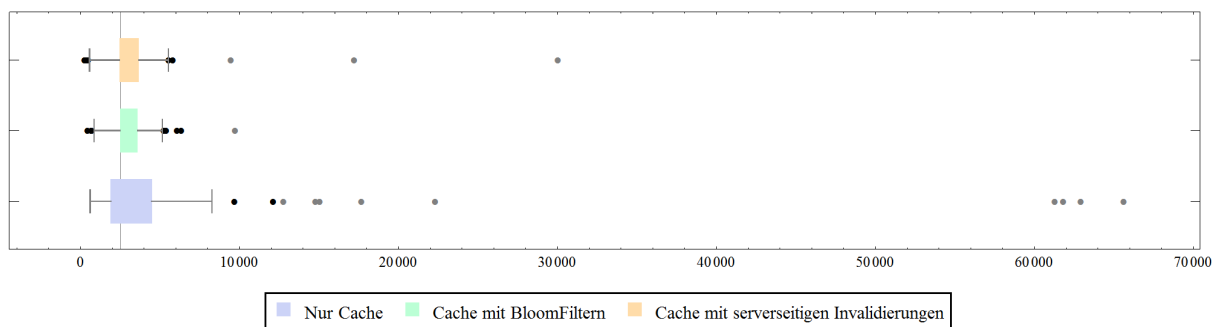


Abbildung 100 Ein Vergleich der drei Invalidierungsstrategien, der die Ausbreitungsdauer der jeweils verschickten Nachrichten miteinander vergleicht.

Es ist deutlich zu sehen, dass der Großteil der Nachrichten bei der zweiten und dritten Strategie auch tatsächlich nach 2,5 Sekunden verbreitet wird. Bei der ersten Strategie hingegen dauert die Verbreitung der Nachricht im Durchschnitt doppelt so lange, zudem gibt es einen vergleichsweise hohen Anteil an Ausreißern. Da der intelligente Client immer erst zur Commitzeit Stale-Reads entdeckt und somit seine Transaktion zurückrollen muss. Dies wird noch einmal deutlicher, wenn man die Tabelle 12 betrachtet. Ebenso verhält es sich für die durchschnittliche Gesamtdauer, die eine Nachricht insgesamt für die Verbreitung braucht. Zudem weist der Client eine massive Anzahl an Ausreißern auf. Diese resultieren aus der optimistischen Transaktionsbehandlung. Da der intelligente Client, im Gegensatz zu den anderen beiden Strategien, die Stale-Reads vom Server aufdecken lassen muss, sind seine Lesezugriffe Teil der Validierung. In der Validierung werden von VOD alle Objekte zunächst mit einem Read-Lock versehen, damit dann anschließend atomar die gelesenen Objektversionen validiert und die Transaktion commitet werden kann (Optimistic Locking). Da der Client aber seine Änderungen schon geschrieben hat, sind diese mit einem Write-Lock versehen. Da nun 25 Clients zur gleichen Zeit auf dem gleichen Datenbestand arbeiten, führt dies zu teilweise sehr langen Wartezeiten aufgrund von Sperrkonflikten. Somit sind bei dieser Strategie teilweise sehr lange Transaktionszeiten und somit lange Verbreitungszeiten zu



beobachten. Bei den anderen beiden Strategien können Stale-Reads vorzeitig erkannt werden oder sie treten gar nicht erst auf. Somit müssen in diesen beiden Strategien, in unserem Szenario, keine Lesezugriffe serverseitig validiert werden. Zudem ist zu sehen, dass die Gesamtdauer von 60 Sekunden zwar nicht erreicht wird, durch den Einsatz der beiden Invalidierungsstrategien, die Caches aber schnell genug invalidiert werden. Deshalb kommt zu keiner massiven Zeitverzögerung, bis Änderungen sichtbar werden.

Strategie	Min [s]	Avg [s]	Max [s]	Avg Gesamtdauer [s]	Transaktionen	Rollbacks
Intelligenter Client	0,59	6,69	182,02	160,68	2405	243
BloomFilter	0,48	3,07	9,74	73,56	2654	0
Serverseitige Inval.	0,30	3,11	30,07	74,75	2601	0

Tabelle 12 Vergleichswerte der drei getesteten Invalidierungsstrategien.

Das wesentlich interessantere Ergebnis ist, dass diese Tests zeigen, wie unsere Invalidierungstechniken dafür sorgen, dass wir bei den über 2600 Transaktionen keinen einzigen Transaktionsabbruch zu verzeichnen haben. Das heißt, die BloomFilter konnten vollständig Stale-Reads aus den Forward-Proxy-Caches verhindern und der Invalidierungs-Service konnte immer rechtzeitig den Reverse-Proxy-Cache invalidieren. Hingegen muss der intelligente Client immer erst die Objektversion vom Server validieren lassen. In diesem Szenario werden insgesamt 250 Änderungen durchgeführt, die anschließend von demselben oder einem anderen Client gelesen werden. Somit müssten insgesamt 250 Transaktionsabbrüche zu beobachten sein, tatsächlich sind 243 Abbrüche aufgetreten, was mit der erwarteten Anzahl gut übereinstimmt. Durch die teilweise sehr langen Transaktionen, kann es aber gut passiert sein, dass ein Client gleich zwei Änderungen auf einmal invalidiert hat. Daraus erklärt sich die minimale Diskrepanz zwischen der erwarteten und tatsächlichen Anzahl an Abbrüchen.

4.8 Ausblick

Die meisten modernen Persistenz-API Standards, wie Java Data Objects (JDO) [27] oder Java Persistence API (JPA), verlangen von der Implementierung einen lokalen Cache für bereits geladene Objekte. Dies ist der sogenannte *2nd Level Cache*. Im Gegensatz zu dem 1st Level Cache, der als Objekt-Puffer für Transaktionen dient, wird der 2nd Level Cache im Client global geteilt. Dieser kann die Ladezeit von bereits geladenen Objekten erheblich verkürzen, da keinerlei Netzwerkzugriff beim erneuten Laden nötig ist. Dies erweist sich auch dann als besonders sinnvoll, wenn mehrere Applikationen auf dem gleichen Rechner laufen, wie es in konsoliderten Server-Umgebungen häufig der Fall ist.

Der 2nd Level Cache wird immer zusammen mit der Persistenz-Schnittstellenimplementierung entwickelt und ist somit meist anbieterspezifisch. Mit dem

Orestes-Protokoll ergibt sich eine neue Möglichkeit, einen solchen 2nd level cache zu implementieren. Die Implementierung kann direkt auf dem Protokoll aufbauen und im Prinzip wie ein lokaler Web-Cache interagieren. Dies entkoppelt die 2nd Level Cache Implementierung vollständig von der Persistenz-Schnittstellenimplementierung und ist somit austauschbar. Da ein Web-Cache im Prinzip nur ein Key-Value-Store ist, der HTTP versteht, bietet es sich an, einen bestehenden Key-Value-Store für eine solche Implementierung einzusetzen. Hierfür könnte Redis [131] oder Memcache [86] in Frage kommen.

In der Abbildung 101 ist der Orestes Aufbau um den lokalen Cache erweitert worden. Dieser wird als lokaler Proxy vor die eigentliche HTTP-Schnittstelle ins Netzwerk gesetzt. Alle lesenden Zugriffe werden von dem Proxy abgefangen und aus dem Key-Value-Store beantwortet, sofern es einen entsprechenden Cacheeintrag gibt. Ansonsten wird der Aufruf, wie auch alle anderen einfach weitergeleitet. Der Vorteil, der sich aus diesem Aufbau gegenüber einem normalen Forward-Proxy-Cache ergibt ist, dass jeder lesende Zugriff von dem lokalen Cache direkt beantwortet werden kann und somit der zusätzliche Netzwerk-Hop eingespart wird. Außerdem können sich verschiedene Client-APIs denselben 2nd Level Cache teilen, da dieser nicht mehr implementierungsabhängig ist. So könnten ein Node.JS-Server und ein Applikation-Server, der JPA verwendet den gleichen 2nd Level Cache benutzen.

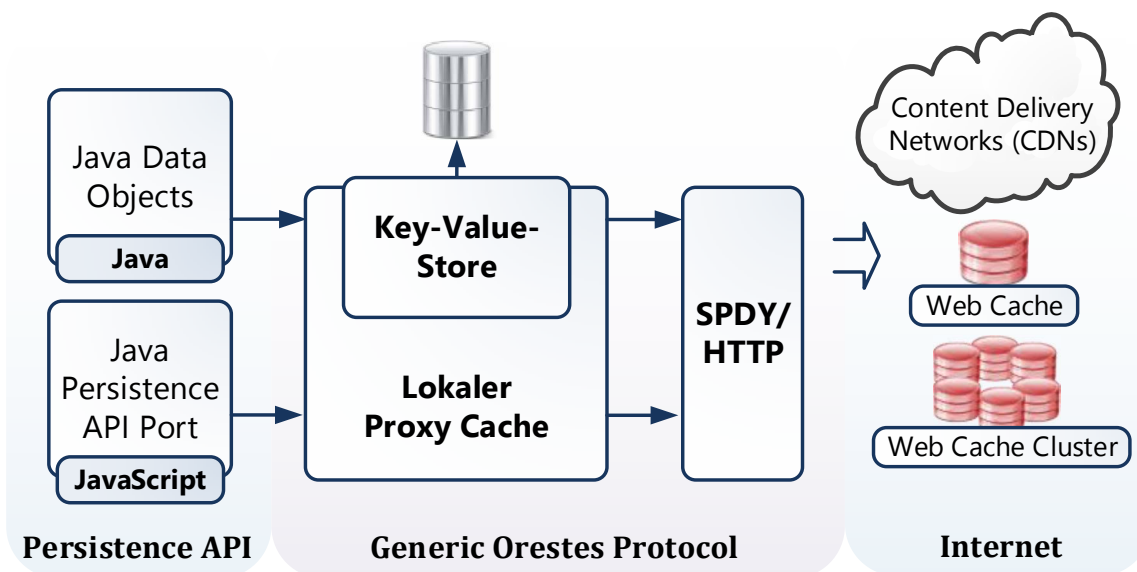


Abbildung 101 Lokaler Cache auf Basis eines Key-Value-Stores

Zwei Erweiterungen, die für das Orestes-Protokoll hierfür eine wichtige Rolle spielen, sind zum einen der Einsatz von SPDY [208], anstelle von HTTP und zum anderen Protocol Buffers [209], die die JSON-Repräsentation der Objekte ersetzen können. SPDY ist eine Erfindung von Google und wird als Anwärter auf den HTTP 2.0 Standard gesehen. Es bringt im Wesentlichen drei Vorteile gegenüber der klassischen HTTP Kommunikation:

- HTTP Header werden komprimiert oder entfernt, sofern sie bei einem vorherigen Request oder Response bereits verschickt wurden, was die Bandbreite der Verbindung besser ausnutzt.



- Über SPDY können mehrere HTTP-Verbindungen über eine TCP-Verbindung abgewickelt werden, was Round-Trips spart und niedrig priorisierte Requests davon abhält, höher priorisierte zu blockieren (Multiplexing).
- Der Server kann dem Client aktiv Ressourcen zuschicken, wenn er weiß, dass diese von dem Client benötigt werden (Server Push). Dies spart Netzwerkverkehr, da der Client weniger Anfragen stellen muss.

Vor alledem der zweite Punkt ist wichtig für eine lokale Cache Implementierung, da HTTP Verbindungen keinen schnellen Cachezugriff auf mehrere Objekte gleichzeitig ermöglichen können. Der dritte Punkt könnte die Geschwindigkeit des Protokolls zudem massiv verbessern, indem Referenzen eines Objektes, das gerade geladen wurde, vom Server über SPDY aktiv mitgeschickt werden und diese somit auf Clientseite sofort aufgelöst werden können.

Protocol Buffers sind ebenfalls eine Technologie von Google und erlauben die Übertragung von fest spezifizierten Nachrichten auf sehr effiziente Weise:

- Die Daten werden binär übertragen, sie nehmen somit wesentlich weniger Platz in Anspruch.
- Das Format hat eine sehr simple Struktur, wodurch die Nachrichten sehr schnell geparkt werden können.
- Die Daten sind typisiert und strukturiert, das trotz des Binärformates eine sichere Verarbeitung der Daten ermöglicht.

Durch Einsatz der Protocol Buffers wird also im Wesentlichen der Platzbedarf für die Kommunikation und das Speichern der Objekte im Cache reduziert.

Ein weiterer Cache, den wir in die serverseitige Orestes-Implementierung integrieren wollen, ist ein In-Memory-Cache, der direkt vor dem eigentlichen Datenbanksystem positioniert ist und von den verschiedenen Orestes-Servern geteilt wird. Dies macht vor alledem bei Anwendungen Sinn, die im gleichen Netzwerk, wie das Datenbanksystem, liegen, also in üblichen Umgebungen des Anwendungsservers. Bei solchen Netzwerkkonfigurationen wäre ein Web-Cache ein zu großer Overhead für das Cachen von Objekten. Stattdessen kommunizieren die Anwendungsserver über SPDY direkt mit den Orestes-Servern. Um dennoch das Datenbanksystem zu entlasten, wird ein Cache direkt zwischen dem Orestes-Server und dem Datenbanksystem eingesetzt, der dann lesende Zugriffe direkt beantworten kann. Zudem liegt der Cache komplett unter der Kontrolle des Servers, wodurch dieser den Cache bei einem Schreibzugriff sofort invalidieren kann. In der Abbildung 102 ist eine solche Netzwerkkonfiguration gezeigt. Die Anwendungsserver bedienen die Anfragen der Clients und kommunizieren dabei mit der Datenbank über das Orestes-Protokoll. Die Orestes-Server schlagen zunächst das angefragte Objekt in dem Cache nach und liefern dieses direkt aus, wenn es einen Cacheeintrag gibt. Andernfalls wird die eigentliche Datenbank kontaktiert und das Objekt anschließend für zukünftige Anfragen in den Cache gelegt. Erfährt einer der Orestes-Server eine Änderung, so entfernt er einfach das Objekt aus dem Cache.

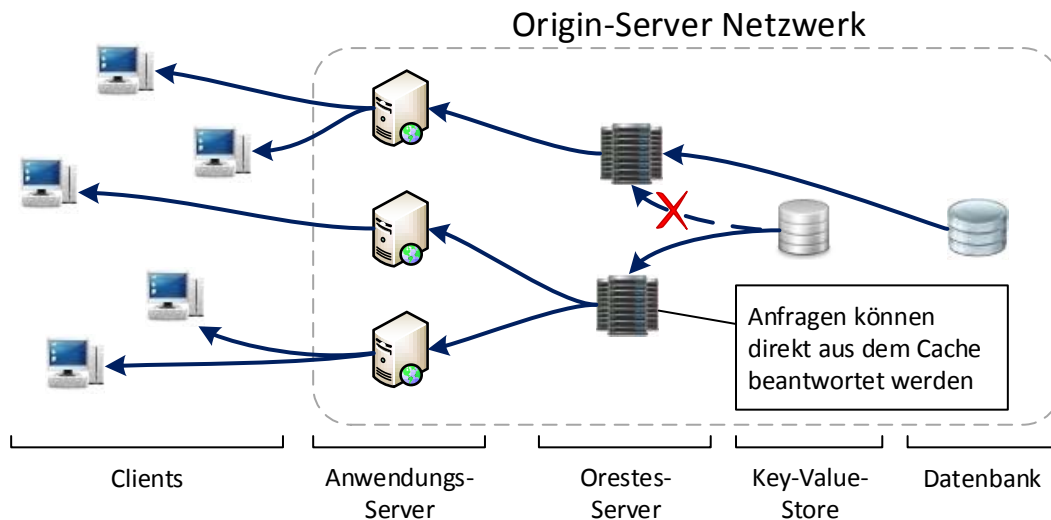


Abbildung 102 Ein Key-Value-Store basierter In-Memory-Cache für die Orestes-Server

Die Vorteile eines solchen Caches, gegenüber eines Reverse-Proxy-Caches, liegen klar auf der Hand, die Orestes-Server teilen sich alle den gleichen Cache, wodurch dieser stets den Zustand der Datenbank repräsentiert, das heißt, dass Stale-Reads ausgeschlossen sind. Außerdem kann ein solcher Cache leicht auf Ebene der Objekt-IDs partitioniert werden, was die horizontale Skalierbarkeit gewährleistet. Das Datenbanksystem wird wie bei einem Reverse-Proxy-Cache entlastet, ohne jedoch den Overhead von HTTP auf Kommunikations- und Cachingebene in Kauf nehmen zu müssen. Die tiefgreifenden Operationen, wie die Transaktionsverwaltung und das Ausführen von Querys, werden aber weiterhin vom Datenbanksystem übernommen.

Die Implementierung könnte dabei auf Protokollebene passieren, ganz ähnlich der Implementierung für den 2nd Level Cache auf der Clientseite. Dies würde den Cache von dem Orestes-Server entkoppeln. Dies hätte aber den Nachteil, dass das komplette Orestes-Protokoll von dem Cache ebenfalls implementiert werden müsste. Die andere und einfachere Möglichkeit wäre, den Cache direkt in den Orestes-Server zu integrieren, der dann die entsprechenden Aufrufe an den Cache übergibt. Beide Varianten wären auf jeden Fall transparent für den Datenbank-Wrapper, welches den Vorteil bietet, dass der Cache für jede Datenbankimplementierung eingesetzt werden kann. Als Cacheimplementierung könnte hier auch wieder Redis oder Memcache zum Einsatz kommen.



5 Polyglot Persistence

Polyglot Persistence entspringt einer denkbar einfachen Idee: kein Datenbanksystem ist in der Lage, alle Anforderungen gleichgut zu erfüllen. Betrachtet man die Fähigkeiten eines Datenbanksystems als Spektrum, gilt es, für einen gegebenen Use-Case die Fähigkeiten von unterschiedlichen Datenbanken zu kombinieren, die von der Applikation benötigt werden. Das können funktionale Eigenschaften wie z.B. eine hohe Query-Mächtigkeit oder Constraints zur Integritäts-erhaltung sein. Aber dank der NoSQL Bewegung geraten zunehmend auch nicht-funktionale Eigenschaften wie horizontale Skalierbarkeit, Latenz und Elastizität in den Vordergrund. Sowohl die funktionalen als auch nichtfunktionalen Eigenschaften eines einzigen Systems sind durch fundamentale „Impossibility Results“ limitiert, z.B. das CAP Theorem, das Fischer-Lynch-Patterson Ergebnis, die NP-Vollständigkeit der Serialisierbarkeitprüfung in VSR und viele weitere [58], [210]–[212]. Das zunehmende Bewusstsein für die Einschränkungen von Datenbanken hat zu einer erhöhten Bereitschaft geführt, Systeme mit sehr speziellen Trade-offs einzusetzen. Wir denken, dass auf Protokollebene ein gemeinsamer Nenner für diese Systeme erzielt werden kann. Deshalb wollen wir das Protokoll und die Architektur von Orestes so gestalten, dass es das gesamte Spektrum an Fähigkeiten gestattet, aber dem spezifischen Datenbank-Backend und der Persistenz-API freistellt, welche dieser Fähigkeiten tatsächlich bereitgestellt werden.

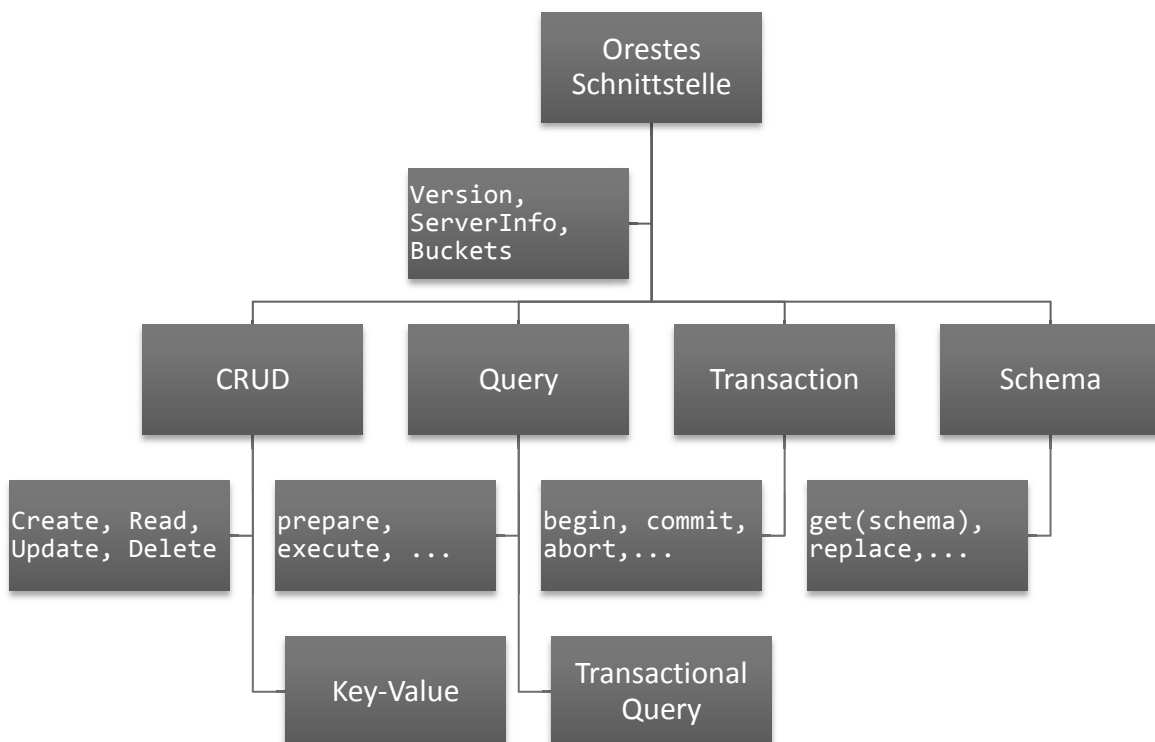


Abbildung 103 Die zerlegte Orestes-Schnittstelle.

Auf Ebene des Orestes Frameworks wird dazu die Datenbankschnittstelle, wie Abbildung 103 gezeigt, zerlegt. Die Schnittstelle und ihre Datentypen wurden dazu so überarbeitet, dass Gemeinsamkeiten auf sinnvolle Weise ausgenutzt werden können. So korrespondiert bei-

spielsweise ein *Namespace mit Klasse* eines Klassenmodells mit dem *Bucket* eines Key-Value Stores und in Zukunft auch mit der *Collection* einer Dokumentendatenbank. Diese Klassen besitzen deshalb eine gemeinsame Basisklasse, so dass beispielsweise die CRUD Methoden unabhängig davon sind, ob Objekte oder Key-Value Paare gespeichert werden. Auch die Transaktionsschnittstelle unterscheidet weder auf Protokoll- noch auf Frameworkebene zwischen objektorientierten Datenbanken, Key-Value Stores oder anderen Datenmodellen. D.h. die Synergie, die für unterschiedliche Datenbanksysteme entsteht, ist die Nutzung gemeinsamer, bereits bestehender Funktionalität (Web-Interface, Bloomfilter, etc.) und die lose Kopplung von Persistenz-API und Datenbank durch eine einheitliche Schnittstelle. Wir demonstrieren den Effekt im Folgenden an einer Implementierung des objektorientierten Modells für Redis.

5.1 Umsetzung von Polyglot Persistence: Redis

Die Nutzung von Redis für objektorientierte Persistenz mag auf den ersten Blick wie der klassische Fehler der Prä-NoSQL Ära erscheinen: ein Datenbanksystem wird für ein Datenmodell benutzt, für das dieses nicht geeignet ist (*Impedence Mismatch*). Tatsächlich jedoch sind die Basis-Operationen eines objektorientierten Datenbanksystems (OODBMS) nahezu identisch mit denen eines Key-Value Stores. Das Abrufen von Objekten geschieht fast immer über einen Schlüssel, die Objekt-ID, da beim navigierenden Zugriff meist komplexe Netzwerke aus Objekten durch das Verfolgen von Referenzen traversiert werden.

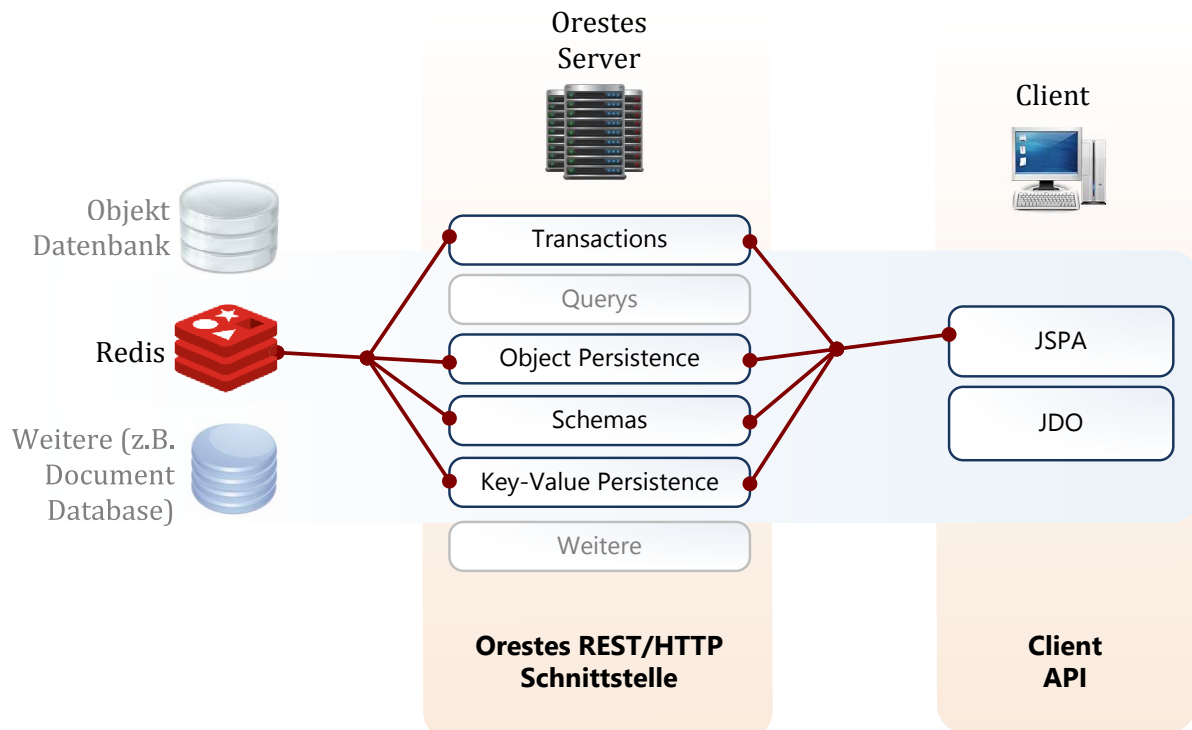


Abbildung 104 Umsetzung von Polyglot Persistence für Redis - Auswahl geeigneter Eigenschaften aus der Orestes Schnittstelle.



Der Punkt, in dem sich die Key-Value Persistenz sehr klar von klassischer objektorientierter Persistenz unterscheidet, sind Querys und Transaktionen. Ein Key-Value Store verhält sich agnostisch bezüglich der unter einem Key gespeicherten Daten und verfügt deshalb nicht über die Möglichkeit, Anfragen auf den Werten auszuführen. Auch Transaktionen werden von Key-Value Stores in der Regel nicht unterstützt, die Batch Transaktionen von Redis bilden eine interessante Ausnahme. Durch diesen Trade-off zu einem simplen Datenmodell können Key-Value Stores eine Skalierbarkeit und Geschwindigkeit erreichen, die objektorientierter Persistenz über ein OODBMS oder RDBMS mit Mapping verwehrt bleibt. Hier sehen wir die große Option für Polyglot Persistence. Wenn für die gespeicherten objektorientierten Daten zwar hohe Anforderungen an nichtfunktionale Eigenschaften wie den Durchsatz und Latenz gestellt werden und dafür auf die Query-Mächtigkeit verzichtet werden kann, ist ein Key-Value Store eine ideale Ergänzung zu einem mächtigeren System. Entsprechende Use-Cases sind nicht schwer zu konstruieren. Für ein soziales Netzwerk könnte es beispielweise nötig sein, mit sehr hoher Geschwindigkeit neue Einträge in einen News-Feed schreiben zu können, während das Zugriffsmuster stets darin besteht, über einen Schlüsselzugriff die 10 neusten Einträge für den Newsfeed eines Users zu laden. Eine Polyglot Persistence Lösung könnte so aussehen, dass die Newsfeed Daten - gekapselt durch die gleiche objektorientierte API – in einem Key-Value Store gespeichert werden, während die übrigen Daten auf klassischem Weg gespeichert werden. Tatsächlich setzen sehr große Social Media Plattformen wie Twitter, Pinterest und Quora Redis für genau diesen Newsfeed Use-Case ein [213].

Die Architektur der objektorientierten Persistenz durch Redis und Orestes ist in Abbildung 104 gezeigt. Das Redis Datenbankbackend für Orestes implementiert die geeigneten Teile der Schnittstelle, d.h. Schemaverwaltung, Transaktionen, Objektpersistenz und Key-Value Persistenz. Die Query API wird explizit nicht implementiert. Mit unserer Implementierung der objektorientierten Persistenz durch Redis können wir zeigen, dass nahezu die gesamte Funktionalität aus bereits bestehenden Fragmenten zusammengesetzt werden kann. Die Größe der Implementierung beläuft sich auf knapp 400 Zeilen Java Code, was gut demonstriert, wie dünn die Schicht sein kann, die das Orestes Protokoll und sein Framework von der zugrundeliegenden Datenbank trennt. So nutzt die Redis-Implementierung zur Schemaverwaltung beispielsweise ausschließlich die Java In-Memory Schemaverwaltung, die bereits nutzbarer Teil des Orestes Frameworks ist.

Der Aufwand zur Verwaltung der Objekte ist sehr gering (siehe Abbildung 105). Im Wesentlichen werden außer Objekten nur wenige Metadaten gehalten:

- **Objekte.** Key-Value Paare speichern die Objekte. Der Key korrespondiert mit der Objekt-ID, während der Value das JSON Objekt enthält.
- **Change-Sets** und **Delete-Sets.** Die Änderungen von laufenden Transaktionen werden in einem Change-Set und Delete-Set gehalten. Beim Commit der Transaktionen werden diese Mengen entweder atomar in die Datenbank eingebracht oder verworfen.

- **Buckets und AllIDs.** Für jede Klasse (Bucket) wird eine Liste angelegt, in der die Extension gespeichert ist, d.h. die Objekt-IDs der Instanzen. Analog wird in einer weiteren Liste die gesamte Extension der Datenbank gepflegt (AllIDs). Die Listen werden konstruiert, da das Orestes Protokoll vorschreibt, dass alle Objekte eines Buckets bzw. der gesamten Datenbank als Liste von URL-Referenzen auf die Objekte abgerufen werden können (unter der URL `db/{namespace}.{class}/all_objects`).

Die Menge aller laufenden Transaktionen und das Schemamanagement werden durch das Orestes-Framework verwaltet.

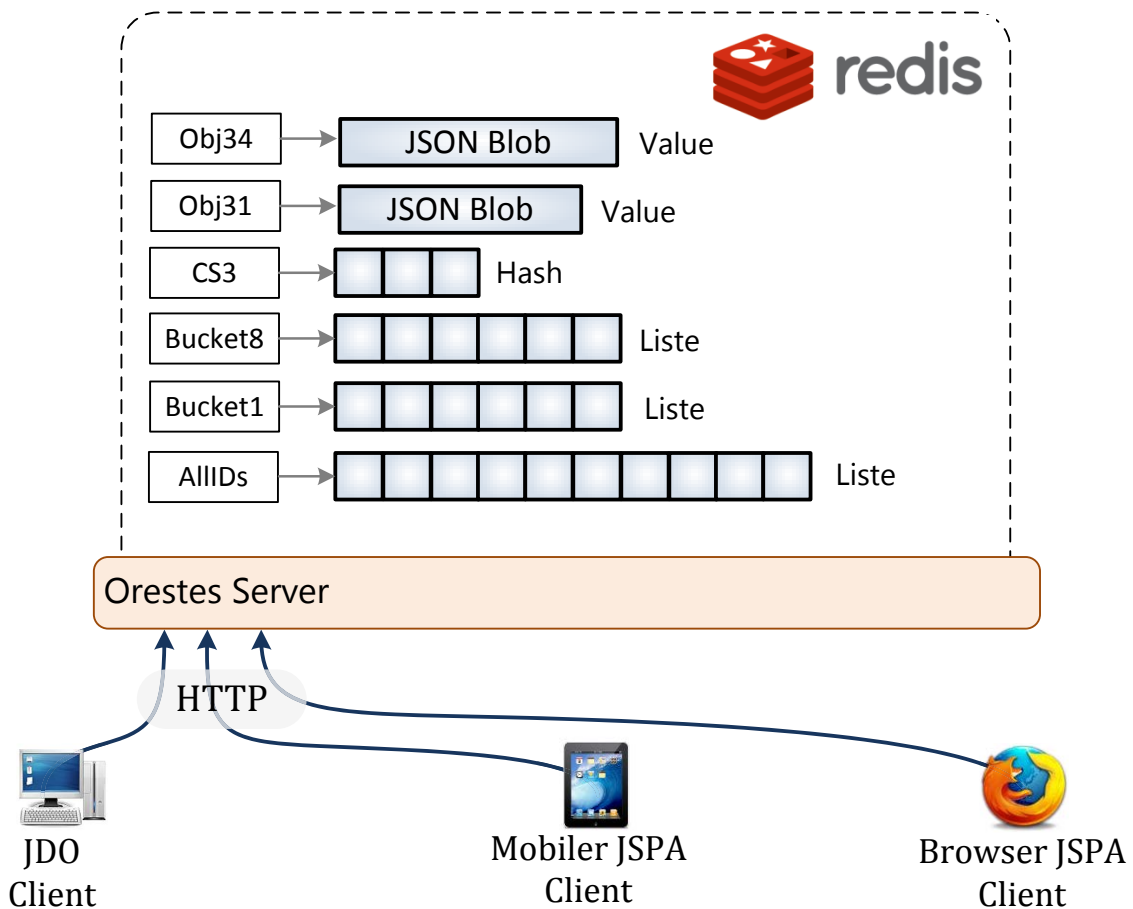


Abbildung 105 Speicherung von Objekten in Redis.

Die Implementierung der Operationen ist sehr einfach. Als Beispiel diene `store(object)` aus dem CRUD Interface:

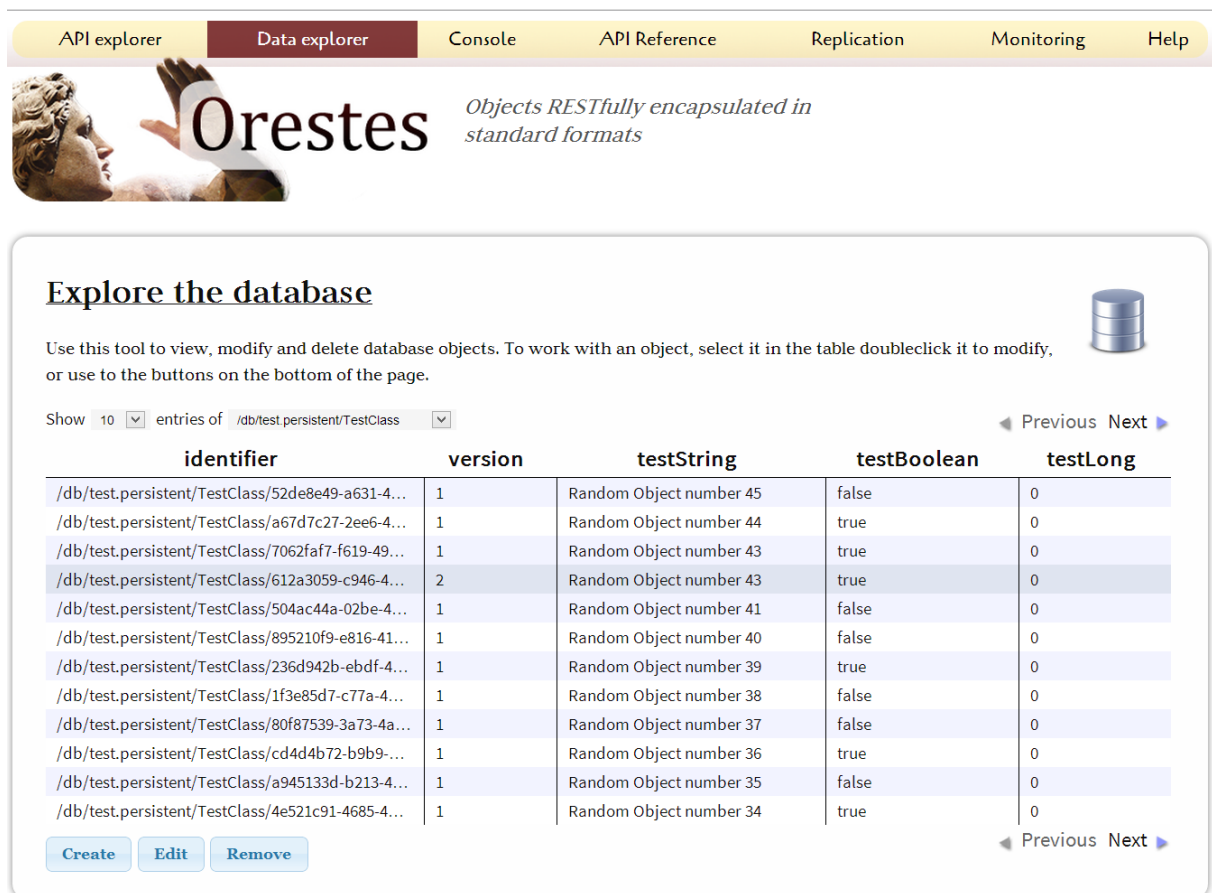
1. Serialisiere *object* zu einem JSON String. Wenn das Objekt nicht existiert und vom Client keine Object-IDs allokiert wurden, erzeuge eine UUID als Objekt-ID.
2. Starte eine *MULTI* Transaktion mit Pipelining.
3. Wenn das Objekt in einer Transaktion geschrieben wurde:
 - Füge das Objekt zum Change-Set hinzu (ein Hash) und notiere die ID und die Version jeweils in einer dazugehörigen Liste.
4. Sonst (wenn das Objekt nicht-transaktional geschrieben wurde):



Schreibe das Objekt als Key-Value-Paar und hänge die Objekt-ID an die Liste des Buckets und die Liste der Datenbankextension an.

5. Führe die Batch-Transaktion aus und leere die Pipeline.

Der *store* Aufruf wird also in einem einzigen Roundtrip und nur drei Operationen ausgeführt, sowohl mit, als auch ohne eine aktive Transaktion. Abbildung 106 zeigt die tabellarische Ansicht einiger Objekte, die über diese Methode in einem Test geschrieben wurden. Der Orestes-Server wird beim Schreiben von etwa 2.000-3.000 Objekten pro Sekunde durch das Parsing und Netzwerk I/O zum Bottleneck für einen einzelnen Client. Das Orestes Web-Interface muss nicht angepasst werden, da es direkt auf dem Orestes Protokoll aufsetzt und kann durch die Redis Implementierung durch einen Aufruf gestartet werden.



Explore the database

Use this tool to view, modify and delete database objects. To work with an object, select it in the table doubleclick it to modify, or use to the buttons on the bottom of the page.

Show 10 entries of /db/test.persistent/TestClass

identifier	version	testString	testBoolean	testLong
/db/test.persistent/TestClass/52de8e49-a631-4...	1	Random Object number 45	false	0
/db/test.persistent/TestClass/a67d7c27-2ee6-4...	1	Random Object number 44	true	0
/db/test.persistent/TestClass/7062faf7-f619-49...	1	Random Object number 43	true	0
/db/test.persistent/TestClass/612a3059-c946-4...	2	Random Object number 43	true	0
/db/test.persistent/TestClass/504ac44a-02be-4...	1	Random Object number 41	false	0
/db/test.persistent/TestClass/895210f9-e816-41...	1	Random Object number 40	false	0
/db/test.persistent/TestClass/236d942b-ebdf-4...	1	Random Object number 39	true	0
/db/test.persistent/TestClass/1f3e85d7-c77a-4...	1	Random Object number 38	false	0
/db/test.persistent/TestClass/80f87539-3a73-4a...	1	Random Object number 37	false	0
/db/test.persistent/TestClass/cd4d4b72-b9b9-...	1	Random Object number 36	true	0
/db/test.persistent/TestClass/a945133d-b213-4...	1	Random Object number 35	false	0
/db/test.persistent/TestClass/4e521c91-4685-4...	1	Random Object number 34	true	0

Create Edit Remove

Abbildung 106 Der Datenbank-Explorer der Web-Oberfläche des Orestes-Servers mit Redis als Backend.

Um einen Commit für eine Transaktion durchzuführen, müssen die geänderten Objekte aus dem Change-Set atomar eingebracht werden und gleichzeitig alle gelöschten Objekte tatsächlich gelöscht werden. Das Change-Set ist als *Hash* Datenstruktur modelliert, um bei einer transaktionalen *load* Anfrage in $O(1)$ das Objekt aus dem Change-Set ausliefern zu können. Gleichzeitig sind alle Objekt-IDs des Change-Sets und alle zugehörigen Versionsnummern in einer sortierten Liste abgelegt, sodass der Commit effizient durch Optimistic Locking durchgeführt werden kann ohne, dass die im Chang-Set gespeicherten Objekte gelesen werden müssen:

1. Lade alle Versionen und Objekt-IDs des Change-Sets.
2. Aktiviere Optimistic Locking durch ein *WATCH* auf den Objekt-IDs (die Keys sind).
3. Benutze die Objekt-IDs, um alle aktuellen Versionen des Objekts zu laden (*MGET*).
4. Vergleiche die Versionen der geladenen Objekte mit den zuvor geladenen Versionen des Change-Sets. Schlägt die Validierung fehl, fahre mit 7. fort.
5. Starte ein *MULTI* Transaktion und füge alle Objekte aus dem Change-Set in die Datenbank ein (*MSET*) und lösche die gelöschten Objekte endgültig.
6. Führe die Bulk Transaktion aus. Wenn sie erfolgreich ausgeführt werden konnte (alle Keys unverändert geblieben sind), verlasse die Commit Methode mit Erfolg. Andernfalls fahre mit 7. fort.
7. Führe einen Rollback durch, indem alle transaktionsbezogenen Metadaten gelöscht werden. Benachrichtige den Client über den Rollback und übertrage die neuen Versionsnummern der veralteten Objekte.

Mithilfe der beschriebenen Commit-Prozedur, die das Optimistic Locking wie in Kapitel 3 beschrieben umsetzt, erhält die Datenbank vollwertige ACID Transaktionen. Die Applikationslogik ist nicht länger gezwungen, Datenbankalgorithmen in der Applikation zu implementieren (cf. [9]). Die Mächtigkeit der Transaktionen geht weit über *MULTI* Transaktionen hinaus – es können in einer ACID Transaktion Lese- und Schreiboperationen beliebig aufeinander aufbauen.

Redis bietet als Backend von Orestes also das volle objektorientierte Datenmodell und ACID Transaktion, wobei eine sehr hohe Geschwindigkeit und niedrige Datenbank-Latenz durch Einschränkungen in der Dauerhaftigkeit der Änderungen und der Query-Mächtigkeit erkauft werden. Eine Polyglot Persistence Anwendung kann Objekte mit derselben Schnittstelle in zwei Systemen mit völlig unterschiedlichen nichtfunktionalen Eigenschaften verwalten.

5.2 Umsetzung neuer Persistenz APIs: JSPA

JavaScript erlebt zurzeit eine Art „dritte Revolution“. JavaScript wurde 1995 von Netscape [214] unter diesen Namen als Skriptsprache für den Netscape Navigator veröffentlicht und begann das Internet zum ersten Mal zu revolutionieren. JavaScript ließ es erstmalig zu, dass Inhalte dynamisch auf der Clientseite verarbeitet werden konnten. Anfangs konnten lediglich Webformulare manipuliert werden, wenig später konnte die Skript-Sprache mit eingebetteten Java-Applets interagieren, daher auch der Name JavaScript. Der Hype flaute dann aber erstmals ab, da die Sprache in dem Browser gefangen war und nur Interaktionen auf der Webseite zuließ. Dies änderte sich 2004 schlagartig als JavaScript zum zweiten Mal das Internet mit **Asynchronous JavaScript and XML (AJAX)** revolutionierte [215]. Dies erlaubt den Zugriff aus JavaScript auf andere Web-Ressourcen. So kann Inhalt dynamisch nachgeladen werden oder eine externe Schnittstelle eines Drittanbieters direkt aus dem Browser heraus angesprochen werden. Die Sprache erlebt seither eine immer größere Beliebtheit und die Standardisierung machte große Fortschritte, die verschiedenen Ausprägungen der Sprache zu vereinheitlichen. So liegt der ECMA-262 Standard [216] mittlerweile in 5ter Version der



Sprache zugrunde. Mit der steigenden Beliebtheit wurde es zunehmend verbreiteter JavaScript außerhalb des Browsers einzusetzen. Wie in der Einleitung bereits kurz eingeführt, gelang mit Node.JS [60], dessen Entwicklung 2009 begann, endgültig der Durchbruch JavaScript auch auf der Serverseite einzusetzen. Durch seine komplett asynchrone Architektur erlaubt Node.JS, hochperformante Netzwerk-Anwendungen zu schreiben, die sich zugleich sehr ressourcenschonend verhalten.

5.2.1 Java Persistence API (JPA)

Die Java Persistence API (JPA) [152] reiht sich in die Java EE Technologien ein und bietet eine standardisierte Möglichkeit, Objekte auf Relationen zu mappen. Die Spezifikation für JPA liegt mittlerweile in zweiter Version vor [28]. Diese beschreibt nur eine Schnittstelle, wie die Metadaten für die Datenbank von Objekten definiert werden und wie die Objekte (auch Entities genannt) persistiert, geladen und manipuliert werden können. Zudem beschreibt sie das grundlegende Verhalten aller Operationen und den Lifecycle von Entities. Die Implementierung dieser Schnittstelle müssen die verschiedenen Datenbankanbieter für ihre jeweilige Datenbank aber selber bereitstellen. So bietet z.B. Oracle TopLink [217] als eigene Implementierung der JPA Spezifikation an. Es gibt aber auch Frameworks wie Hibernate [218] oder EclipseLink [219], die die JPA implementieren und die Aufrufe dann über beispielsweise JDBC an die spezifische Datenbank weiter geben können. Diese Implementierungen unterstützen dadurch eine ganze Reihe an relationalen Datenbanken. Zudem unterstützt EclipseLink bereits MongoDB und in Hibernate wird an einer Integration von MongoDB, Infinispan [220] und Ehcache [221] gearbeitet.

In JPA können Klassen, in Form von Java-Beans mit Annotationen versehen werden, die angeben, wie die entsprechende Klasse auf eine Relation in der Datenbank gemappt werden soll. Über ein Java-Agent wird zur Laufzeit oder mit einem Bytecode-Enhancer zur Kompilierzeit die entsprechenden Klassen mit Metainformationen angereichert und mit Tracking-mechanismen versehen, um Lese- und Schreibzugriffe auf Objektfeldern abzufangen. Hierdurch kann bei einem Lesezugriff ein referenziertes Objekt aus der Datenbank nachgeladen werden und bei einem Schreibzugriff das Objekt als geändert markiert werden, damit bei der nächsten Synchronisierung mit der Datenbank die Änderung zurückgeschrieben wird. Die Zugriffe werden aber nur abgefangen, wenn das Objekt mit einem persistenten Kontext verknüpft ist. Jeder *EntityManager* verwaltet in JPA einen persistenten Kontext und ein Objekt kann maximal einem Kontext zurzeit angehören. Wie der Abbildung 107 zu entnehmen ist, können neu erstellte Objekte, die initial *transient* sind, mit einem *persist()* Aufruf, dem *EntityManager* bekannt gemacht werden. Das Objekt ist dann Teil des persistenten Kontextes und somit *persistent*. Alle Änderungen werden nun an dem Objekt verfolgt und mit der Datenbank durch ein *flush()* oder einem Transaktionsende synchronisiert. Ein persistentes Objekt kann aus dem Kontext durch einen *remove()* Aufruf entfernt werden. Das Objekt gilt dann als *removed*. Weitere Änderungen werden somit nicht mehr verfolgt und das Objekt wird bei der nächsten Synchronisierung aus der Datenbank entfernt. Mit einem *persist()* kann das Objekt dem Kontext und dementsprechend auch der Datenbank wieder hinzugefügt

werden. Objekte können aber auch den Kontext verlassen, ohne aus der Datenbank entfernt zu werden. Dies wird mit einem `detach()` Aufruf erreicht. Das Objekt ist dann *detached* und kann lokal manipuliert werden, ohne dass diese Änderungen persistent gemacht werden. Sollen die Änderungen aber zurückgeschrieben werden, so muss das Objekt zunächst einem persistenten Kontext wieder hinzugefügt werden. Hierfür wird die `merge()` Methode verwendet. Das Objekt muss aber nicht zwangsläufig dem gleichen persistenten Kontext wieder hinzugefügt werden aus dem es ursprünglich stammt. Objekte können somit offline geändert und zu einem späteren Zeitpunkt in die Datenbank zurückgeschrieben werden.

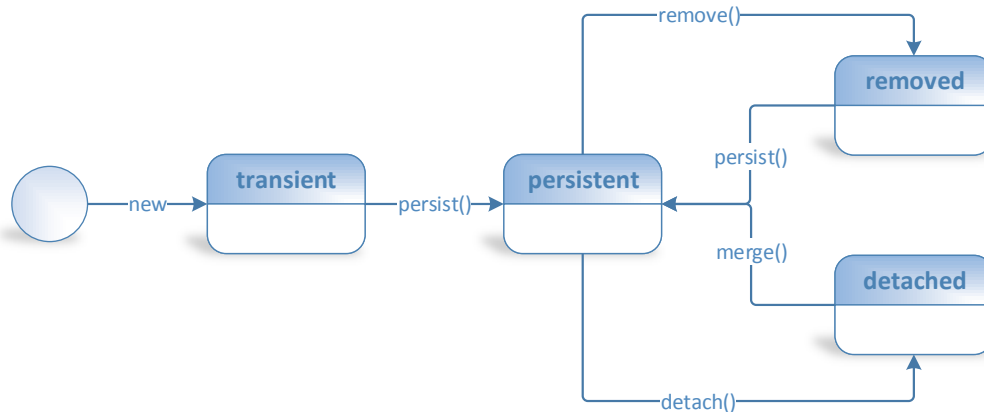


Abbildung 107 Der Object-Lifecycle in JPA.

5.2.2 Einführung in JSPA

Aufgrund der Beliebtheit von JavaScript und der Möglichkeit JavaScript auch serverseitig einsetzen zu können, entstand die Idee, eine Persistenz-API für JavaScript zu entwickeln. Die Unterstützung für Datenbankzugriffe aus JavaScript heraus ist zurzeit noch sehr mangelhaft. Zudem gibt es in Browsern keine Möglichkeit, direkt eine TCP-Verbindung zu öffnen und somit die meisten Datenbankschnittstellen gar nicht angesprochen werden können, da diese meist auf binärem TCP beruhen. Jedoch kann jeder Browser HTTP sprechen und erlaubt, dies auch aus JavaScript heraus (AJAX). Somit ist das Orestes-Protokoll ideal um einen Zugang aus JavaScript zu Datenbanken zu etablieren. Da es in Java bereits sehr gute und zudem einfache Schnittstellen für die Datenbankkommunikation gibt, haben wir uns zur Orientierung JPA [152] ausgesucht, um eine Persistenz-Schnittstelle für JavaScript zu entwickeln. Diese API haben wir JavaScript Persistence API (JSPA) getauft, in der Analogie zu JPA, auf der die API zum größten Teil beruht. JSPA folgt dabei zum großen Teil der JPA-Spezifikation, jedoch konnten wir JPA nicht eins zu eins nach JavaScript übersetzen, da JavaScript von rein asynchroner Natur ist. JPA hingegen ist eine blockierende API, das heißt, alle Aufrufe an der API blockieren solange, bis die angefragte Operation durchgeführt wurde. Da hinter jeder Operation an der API ein Datenbankaufruf steht und daraus mindestens ein HTTP-Aufruf resultiert, können die Operationen in JavaScript nicht synchron ausgeführt werden. Somit musste die API den neuen Anforderungen angepasst werden.

Betrachten wir zunächst noch einmal den JSPA Query-Aufruf aus der Einleitung. Hier wird zunächst ein Query-Objekt mit Hilfe des EntityManagers erzeugt. Dieser Aufruf gleicht dem



EntityManager in Java, unter der Verwendung von JPA. Im nächsten Schritt soll das Ergebnis des erzeugten Querys abgerufen werden, dies bedeutet also einen Datenbankzugriff. In JPA würde der Aufruf den laufenden Thread solange blockieren, bis das Ergebnis vollständig abgerufen ist. In JavaScript gibt es aber nur einen einzigen Thread. Würde dieser blockiert, wäre das gesamte Programm, samt User-Interface, komplett blockiert, bis das Ergebnis vorliegt. Um dies zu vermeiden, wird ein Callback als Event-Handler dem Aufruf übergeben, der dann gerufen wird, wenn das Ergebnis vollständig geladen ist. Der Callback erhält also das Event-Objekt und das Ergebnis als Argumente. In diesem Callback kann dann die Logik folgen, die das Query-Ergebnis verarbeiten soll.

```

var factory = new jspa.EntityManagerFactory();
var pu = factory.persistenceUnitUtil;
var em = factory.createEntityManager();

//create a new extent query
var query = em.createQuery(null, Coffee);
query.maxResults = 3;
query.firstResult = 0;

//fetch the result list
query.getResultList(function(e, result) {
    log('First 3 coffees:');
    for (var i = 0, coffee; coffee = result[i]; ++i) {
        log('ID: ' + pu.getIdentifier(coffee) + ': ' + coffee.name);
    }
});

```

Abbildung 108 Erstellen und abrufen eines Querys in JSPA

Betrachten wir noch ein weiteres Beispiel, das zeigt, wie Aufrufe prinzipiell an der API getätigt und in welcher Reihenfolge sie ausgeführt werden. In Abbildung wird nach der Erzeugung des EntityManagers zunächst ein Coffee-Objekt erstellt und anschließend eine Transaktion gestartet. Auch dieser Aufruf bedeutet ein Datenbankaufruf und kann somit ebenfalls nur asynchron ausgeführt werden. Als nächstes kommt der *persist* Aufruf, der das neue Objekt an dem EntityManager registriert. Da die Transaktion aber noch nicht gestartet wurde, da dies ja ein asynchroner Aufruf ist, dürfte der *persist* Aufruf also noch nicht Bestandteil der Transaktion sein. Anschließend folgt der *commit* Aufruf der Transaktion, die jedoch immer noch nicht gestartet wurde. Das heißt, es gibt eigentlich noch gar keine Transaktion, die committed werden kann. Tatsächlich wird der *persist* Aufruf aber Teil der Transaktion, das Objekt wird also transaktional geschrieben und der *commit* Aufruf wird die bestehende Transaktion abschließen.

```

//create a new coffe object
var coffee = new Coffee("Arabica Jamaican Finest", 15, true, null);

// start a transaction
em.transaction.begin();
// register the new object on the persistent context
// The object becomes managed

```

```
em.persist(coffee);

// sync the changes to the database and commit the transaction
// wait until the entity manager is prepared and the transaction
// is completed
em.transaction.commit(function() {
    // get the identifier of the object
    // i.e. the generated oid
    id = pu.getIdentifier(coffee);
});
```

Abbildung 109 Eine Transaktion in JSPA

Um zu verstehen, wie die einzelnen Aufrufe an der API gehandhabt werden und wann welche Operation tatsächlich ausgeführt wird, müssen wir zunächst die interne Architektur von JSPA genauer betrachten.

5.2.3 Architektur

In JSPA beginnt alles mit der *EntityManagerFactory*. Eine *EntityManagerFactory* ist für die Verbindung zu einem logischen Orestes-Server verantwortlich. Sie hält intern, wie der Abbildung 110 zu entnehmen ist, ein *Connection*-Objekt, das die gesamte Kommunikation zwischen Client und Server abwickelt. Zudem hält es noch eine Referenz auf das *Metamodel*, das die vorhandenen Schemata der Datenbank repräsentiert. Dies ist anders als in JPA, denn dort repräsentiert das Metamodel in erster Linie die vorhandenen Entity-Klassen der Anwendung. Dies ist in JSPA aber nicht möglich, da Klassen in JavaScript nicht so wie in Java existieren und diese auch keine Typinformationen enthalten. Da das Datenbankschema aber typisiert ist, benötigt der JSPA Client diese Typinformationen auch und generiert sie somit aus den Schemainformationen des Servers. Da JavaScript eine dynamische Programmiersprache ist, bringt dieser Weg auch einige Vorteile, so können Klassen, die auf der Clientseite nicht existieren, einfach aus dem Datenbankschema generiert oder fehlende Felder an bestehenden Klassen ergänzt werden. Die Klassen in JavaScript passen sich somit immer dem Schema der Datenbank an.

Die nächste wichtige Komponente ist genauso wie in JPA der *EntityManager*, welcher von der *EntityManagerFactory* erzeugt wird. Ein *EntityManager* verwaltet immer eine Menge von Entitäten, die mit Hilfe des *EntityManagers* aus der Datenbank geladen, geändert und gelöscht werden können. In diesen Punkten verhält sich JSPA so wie es die JPA-Spezifikation verlangt. Mit der Ausnahme, dass alle Operationen nicht synchron, sondern asynchron ausgeführt werden. Hierfür hält jeder *EntityManager* eine *Queue*, über die alle Operationen synchronisiert werden. Wenn also eine Operation durch die entsprechende Methode an der API ausgelöst wird, so wird diese nicht sofort ausgeführt, sondern zunächst in eine Queue gelegt. Der Aufrufer erhält ein *Result*-Objekt, an dem er ein oder mehrere Event-Handler registrieren kann, die nach Abschluss der Operation gerufen werden sollen. Die Queue wird dann Schritt für Schritt abgearbeitet. Folgt aus einer Operation ein Netzwerkzugriff, so wird die Abarbeitung pausiert, bis das entsprechende Ergebnis vorliegt. Die Netzwerkaufrufe werden



dabei in einzelne HTTP-Nachrichten verpackt, die dem Orestes-Protokoll genügen. Diese werden dann von dem Connector an den Server geschickt. Nachdem der Response von dem Connector durch das Auslösen eines Events zurückgegeben wird, kann die aktuelle Operation abgeschlossen und die entsprechenden Result-Handler gerufen werden. Anschließend wird die nächste anstehende Operation aus der Queue verarbeitet.

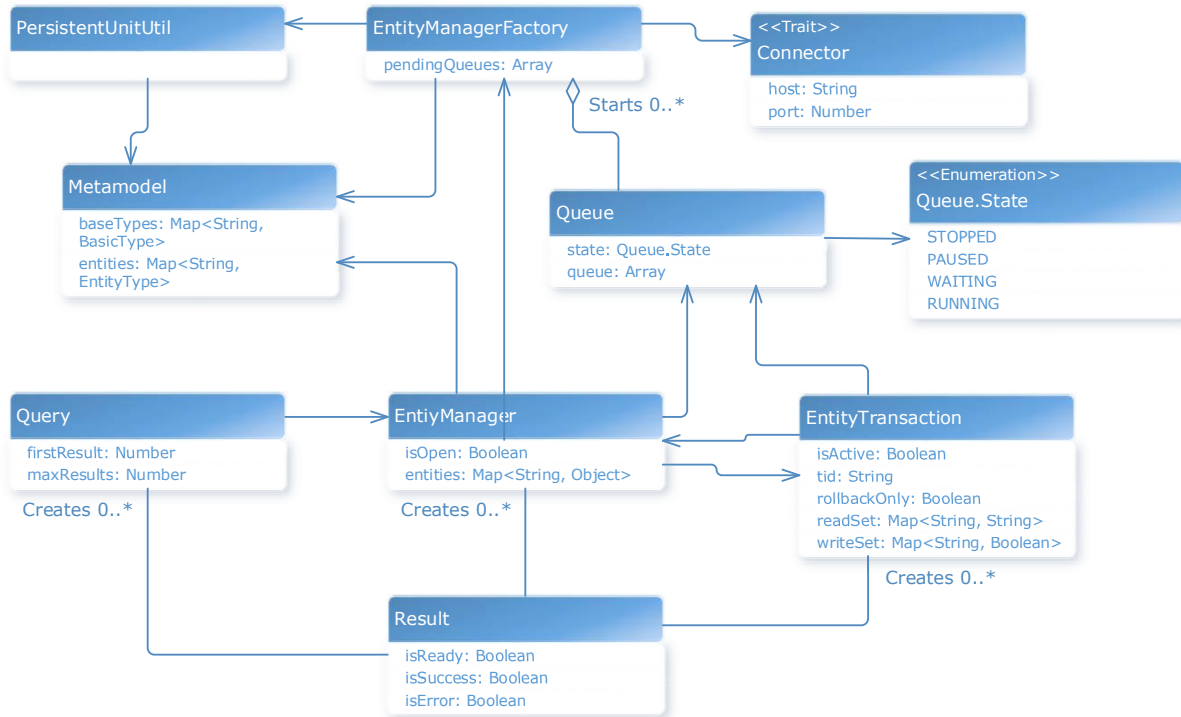


Abbildung 110 Vereinfachtes JSPA Architekturdiagramm

In dem Einführungsbeispiel aus Abbildung 108 wird zudem am EntityManager ein Query-Objekt erzeugt. Mit diesem Objekt können Querys an die Datenbank geschickt und ausgeführt werden. JSPA unterstützt zurzeit aber nur die Ausführung von Extent-Querys, wie es in dem Beispiel zu sehen ist. Wird das Query-Ergebnis abgerufen, so muss auch hier ein Eventhandler für den Erhalt des Ergebnisses registriert werden. Dieser wird dann aufgerufen, wenn die entsprechende Operation ausgeführt worden ist und das Queryergebnis vorliegt. Die Operation wird ebenfalls über die Queue des EntityManagers, der das Query-Objekt ursprünglich erzeugt hat, synchronisiert.

In Abbildung wurde zudem eine Transaktion benutzt. Hierfür hat jeder EntityManager ein EntityTransaction-Objekt, welches für die Verwaltung von Transaktionen zuständig ist. An einem EntityManager kann maximal eine aktive Transaktion gleichzeitig ausgeführt werden. Wenn mehrere Transaktionen parallel ausgeführt werden sollen, muss für jede Transaktion ein eigener EntityManager verwendet werden. Auch das Starten und Beenden von Transaktionen wird als Operation aufgefasst und diese deshalb ebenfalls in die Queue des EntityManagers gelegt. Auch hier werden wieder Result-Objekte zurückgegeben, mit deren Hilfe der Status der Transaktion geprüft werden kann.

5.2.4 Synchronisierung der Operationen über die Queue

Wie den beiden vorangehenden Beispielen bereits zu entnehmen ist, wird die Ausführung der eigentlichen Operationen, die durch einen Methodenaufruf ausgelöst wird, verzögert und intern über eine Queue synchronisiert. Hierdurch werden die Operationen tatsächlich in der Reihenfolge ausgeführt, wie sie an der API angestoßen werden. In der Regel reicht diese Synchronisierung alleine aber noch nicht aus. Wenn z.B. einer Datenbankoperation, wie dem *flush*, aus dem Beispiel in Abbildung 111, weitere Operationen folgen sollen, diese aber von der vorhergehenden Operation abhängen, so können die entsprechenden Aufrufe erst nach Abschluss der vorherigen Operation durchgeführt werden. So kann die zugewiesene Id des Objektes erst nachdem dieses mit der Datenbank synchronisiert wurde, abgerufen werden. Würden die Operationen einfach nur nach der Aufrufzeit synchronisiert werden, so würde die *detach* Operation, die das Objekt aus dem persistenten Kontext entfernt, nach dem *clear* aufgerufen werden. Dies würde aber zu einem Fehler führen, da das Objekt durch das *clear* bereits aus dem Kontext entfernt wurde. Es wäre also nicht möglich, auf das Ergebnis einer Operation zu warten und daraufhin weitere Operationen anzustoßen.

```
// create a new coffee object
var coffee = new Coffee("Arabica Jamaican Finest", 15, true, null);

// register the new object on the persistent context
// The object becomes managed
em.persist(coffee);

// sync the changes to the database
// and wait until the sync is completed
em.flush(function() {
    // get the identifier of the object
    // i.e. the generated oid
    id = pu.getIdentifier(coffee);

    // detach the object from the persistence provider
    // the object will not be managed further
    em.detach(coffee);

    // retrieve a new managed coffee object
    em.find(Coffee, id, function(e, loadedCoffee) {
        // the fetched object isn't the same object we used before
        coffee.name = "Robusta Brazilian Selection";

        // merge the changes form the detached object back into the
        // persistent context
        em.merge(coffee, function(e, mergedCoffee) {
            // sync the local changes to the database
            em.flush();
        });
    });
});

// detach all objects from the persistent context
em.clear();
```

Abbildung 111 Ein Beispiel, dass die Queue basierte Synchronisation demonstriert



Um dies aber zu ermöglichen, werden Event-Handler, die auf das Ergebnis einer Operation warten, bevorzugt, indem die Operationen, die sie wiederum anstoßen, vor allen weiteren ausstehenden Operationen ausgeführt werden. Dies wird in Abbildung 112 illustriert. Wenn an der Schnittstelle eine Methode aufgerufen wird, so wird die entsprechende Operation als Task in die Queue gelegt. Eine Operation wird dabei meist in mehrere Tasks zerlegt. Die Queue arbeitet die einzelnen Tasks ab, indem sie ein Element nach dem anderen aus der Queue entfernt und den Task aufruft. Ein Task ist dabei einfach ein hinterlegter Callback. Wenn ein Task mit der Datenbank kommunizieren muss, so schickt er ein oder mehrere asynchrone HTTP Nachrichten an den Server. Bevor der Task beendet wird, wird ein neuer Task in der Queue hinterlegt, der die Antworten der versandten HTTP Nachrichten verarbeitet. Da nun auf die Antworten gewartet werden muss, wird die Queue in den Wartezustand versetzt, bis alle HTTP Nachrichten empfangen wurden. Die Queue setzt dann die Abarbeitung fort, indem sie den nächsten Task ausführt. Dieser ist der zuvor abgelegte Task, der die Antworten der HTTP Nachrichten verarbeitet. Dies liegt daran, dass wenn immer ein Task aufgerufen wird und dieser wieder neue Tasks in die Queue legt, diese immer direkt hinter dem aktuellen Task abgelegt werden. Deswegen werden auch die Operationen, die in einem Eventhandler angestoßen werden, immer vor allen anderen Operationen ausgeführt, da die Eventhandler immer von dem letzten ausgeführten Task aufgerufen werden.

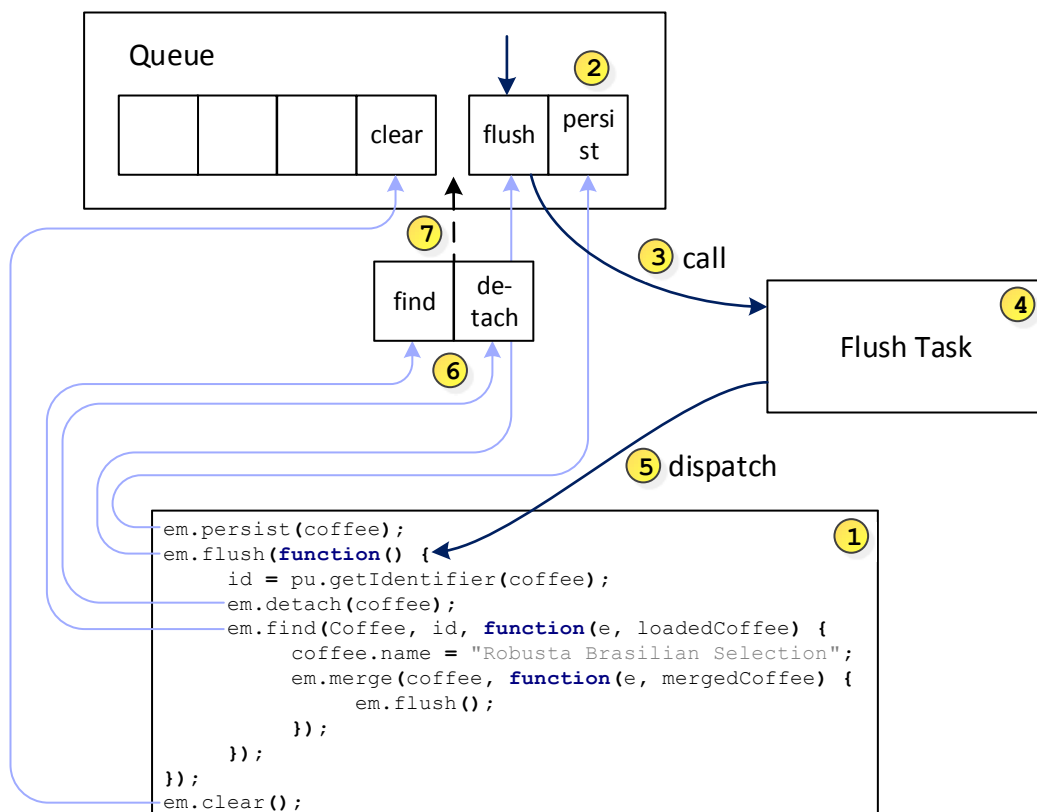


Abbildung 112 Ablauf des Programms aus Abbildung 111.

Der Ablauf des Programms ist somit folgender:

1. Der *persist*, *flush* und *clear* Aufruf des Programms legen drei entsprechende Tasks in der Queue an.

2. Die Queue arbeitet nun zunächst den *persist* Task ab, der das Objekt dem persistenten Kontext bekannt macht.
3. Die Queue arbeitet den nächsten Task ab und ruft somit den *flush* Task auf.
4. Der *flush* Task schickt das Objekt zur Datenbank, dieses wird somit persistent und erhält eine Id.
5. Nachdem die Datenbankoperation ausgeführt wurde, werden die Eventhandler gerufen.
6. Der Eventhandler ruft die *detach* und *find* Methode auf, die wiederum zwei entsprechende Tasks in der Queue hinterlegen. Diese werden allerdings direkt hinter dem *flush* Task eingefügt.
7. Die Queue führt nun den zuvor eingefügten *detach* Task aus, usw.

JSPA stellt also eine asynchrone Portierung der Persistenz-API JPA nach JavaScript dar, die es erlaubt, aus Browsern und Plattformen wie Node.JS über das Orestes-Protokoll auf Datenbanken zuzugreifen.



6 Future Work

Es gibt zahlreiche Aspekte, die wir durch weitere Forschung vertiefen wollen.

Bloomfilter. Für die Bloomfilter ist die richtige Wahl der Parameter entscheidend. Die Expirationsdauer, die False-Positive Rate und Größe des Bloomfilters stehen in einem engen Zusammenhang und sind vor allem durch den Workload des Systems bestimmt. Es wäre deshalb wünschenswert, wenn das System automatisch im laufenden Betrieb die optimalen Parameter lernt. Ein unnötig großer Bloomfilter kann z.B. den Start einer Transaktion verzögern, während eine zu geringe Expirationsdauer die Cache-Hit Ratio verschlechtert. Die Anwendung von Machine-Learning Techniken könnte hier eine Lösung bieten, um die ideale Konfiguration automatisch zu ermitteln. Auch die Ermittlung von Hot-Spot Objekten könnte so vorgenommen werden. Für diese könnte eine besonders geringe Expirationsdauer festgelegt werden oder die Vorschrift, vor der Änderung eines Hot-Spot Objekts eine Sperre anzufordern. Für die Darstellung von Hot-Spot Objekten könnte untersucht werden, ob sich die Techniken des Bloomier Filters, Counting Bloomfilters und spektralen Bloomfilters in einer Datenstruktur vereinigen lassen. Auch ein Bloomfilter, der auf Serverseite durch Faltung seine Größe für den Client an der Anzahl geschriebener Objekte ausrichtet, wäre interessant.

Transaktionen. Für die Transaktionsverwaltung gibt es viele offene Fragen. Besonders interessant erscheint uns die Untersuchung und Implementierung eines hybriden Transaktionsmodells, das für die Mehrzahl der Objekte optimistische Nebenläufigkeitskontrolle einsetzt und für Hot-Spot Objekte Sperren. Vorteilhaft wäre auch ein datenbankagnostischer Transaktionslayer im Orestes-Framework, der automatisch die Write-Sets von laufenden Transaktionen trackt und eine BOCC+ Validierung beim Commit vornimmt. Ein sehr mächtiges Konzept, das bisher keinen Einzug in die NoSQL Bewegung gefunden hat, sind verteilte Transaktionen, z.B. auf Basis des Paxos Commit Protokolls oder 2PC. Gerade in Bezug auf Polyglot Persistence wäre es sehr vorteilhaft, Transaktionen auszuführen, die mehrere Datenbanken umfassen. Eine Erweiterung des Orestes Protokolls um verteilte Transaktionen würde es z.B. erlauben, in einer Transaktion mit der gleichen API 10.000 Objekte aus einem Key-Value Store zu lesen und anschließend wenige Objekte in ein OODBMS zu schreiben.

DBaaS. In Bezug auf DBaaS wird der Quality-of-Service Aspekt wie für viele Cloud-Dienste eine zunehmende Rolle spielen. Es wäre deshalb interessant, für Orestes zu untersuchen, welche Service Level Objectives sich durch Web-Caching zusichern lassen, z.B. ein garantiertes 99% Quantil für die Latenz von Reads und einen zugesicherten Durchsatz von Reads pro Sekunde. Für diesen Zweck ist vor allem die praktische Integration von Orestes mit einem CDN wichtig, um dann ggf. ein Mapping der CDN-SLAs auf DBaaS-SLAs vorzunehmen. Auf gleiche Weise ließe sich untersuchen, welche zusätzlichen SLOs Orestes gewähren könnte, wenn sein Backend bereits ein DBaaS ist (z.B. DynamoDB). Auch elastische Ressourcenallokation durch Autoscaling ließe sich in Orestes vornehmen. So könnten neue Web-Cache Instanzen gestartet werden, wenn die Verletzung von SLOs (z.B. Durchsatz) droht.

Eine dynamische Rekonfiguration eines Load-Balancers würde genügen, um eine neue Web-Cache Instanz einzubinden. Auch die Untersuchung geeigneter Pay-as-you-go Geschäftsmodelle, die mit dem Web-Caching kompatibel sind, wäre aufschlussreich.

Smart Clients. Für die Persistenz von Smart-Clients haben wir ein Konzept entworfen, das es erlauben soll, mit einer API sowohl lokale Persistenz (Applikationszustand) als auch eine entfernte Datenbank zu nutzen. Viele Mobile Apps benutzen eingebettete Datenbanken wie SQLite für lokale Persistenz und eine gänzlich andere Schnittstelle, um mit einem zentralen Datenbestand zu arbeiten. Wir sehen hier die Möglichkeit, beide Wege zu vereinen. Dazu könnte die Orestes-Schnittstelle für einen Smart Client ohne Konnektivität transparent auf die lokale Speicherung durch HTML5 Standards wie LocalStorage und IndexedDB abgebildet werden (*Graceful Degradation*). Ist der Smart-Client wieder verbunden, könnte transparent ein Synchronisationsprozess gestartet werden, der die lokalen Änderungen mit dem Zustand der entfernten Datenbanken abgleicht und ggf. eine Konfliktauflösung fordert. Eine andere interessante Perspektive sind 2-Tier Architekturen, in denen ein Smart-Client ohne einen Application Server direkt mit verschiedenen REST Services kommuniziert, von denen die Datenbank der wichtigste Service ist. Für Orestes könnte dieses Modell umgesetzt werden, indem der Orestes-Server statische Ressourcen (z.B. HTML, CSS, JavaScript) ausliefert, die eine App darstellen. Die Applikationslogik in JavaScript würde mit dem Orestes-Server kommunizieren, um beispielsweise die Daten zur Darstellung einer Produktseite zu erhalten. Ein zu lösendes Problem solcher Single-Page-Apps ist u.a. die Search-Engine-Optimierung, die jedoch für HTML5 Mobile Apps entfallen kann.

Protokolle. Um mit Orestes auch für Szenarien, in denen Applikation und Datenbank im selben Netzwerk sind, optimale Performance zu erreichen, wäre es interessant, SPDY als Session-Layer Protokoll für HTTP zu nutzen und neben JSON optimierte Formate wie ProtocolBuffers zu unterstützen. SPDY enthält interessante Konzepte wie Server-Push, die der Orestes-Server nutzen könnte, um häufig gemeinsam geladene Objekt proaktiv zum Client zu übertragen. Dabei könnte er entweder auf empirische Heuristiken setzen oder explizite *Fetch-Graphs* [152].

Security. Die Untersuchung eines Rechtemodells ist wichtig sowohl für Smart-Clients, als auch für DBaaS. Ein einfaches Modell könnte Usern Rollen zuweisen. Diesen Rollen wären Lese- und Schreibrechte auf konkreten Objekten, Klassen oder Namespaces entweder erlaubt oder versagt. Auch die Authentifizierung eines Nutzers ist wichtig. Orestes unterstützt bisher HTTP-Authorization. Protokolle wie OAuth scheinen sich jedoch eher durchzusetzen. Die Untersuchung von Verschlüsselung ist ebenfalls wichtig. Als einfachste Lösung könnte einen Reverse-Proxy wie Nginx als sogenannte SSL-Terminierung benutzt werden. Durch die Verschlüsselung wäre das Caching beschränkt auf den Client Cache und Reverse-Proxy Caches. Die Latenz wäre dadurch weiterhin ein Engpass. Einige CDNs unterstützen ebenfalls SSL-Terminierung. Eine größere Herausforderung wäre die Verschlüsselung auf Repräsentationsebene, z.B. das Ausliefern von verschlüsselten, voll cachbaren JSON Objekten.



Stored Procedures. Als ein sehr mächtiges Pattern hat sich in vielen NoSQL Datenbanken serverseitiges JavaScript herausgestellt (z.B. Riak, CouchDB und MongoDB [9], [13]). Will ein Client beispielsweise die ganze Extension einer Klasse manipulieren, so könnte es durch große Anzahl erforderlicher Roundtrips sehr lange dauern, bis die Objekte alle abgerufen und wieder gespeichert sind. Außerdem ist die Wahrscheinlichkeit dann besonders hoch, dass die Transaktion durch die optimistische Validierung fehlschlägt. Mit einer Stored Procedure könnte die Änderung sehr effizient direkt auf der Serverseite erfolgen. Zudem könnten mit Stored Procedures Änderungsoperationen für Clients implementiert werden, die aus Sicherheitsgründen nicht direkt den Datenbestand ändern dürfen. Diese würden dann, um ein Objekt zu ändern, eine Stored Procedure aufrufen, in der die Eingaben des Clients validiert und weiterverarbeitet werden können. Eine Möglichkeit dies zu realisieren wären Stored Procedures in JSPA. Diese könnten als JavaScript Funktion an den Server gesendet werden, wo sie anschließend als Ressource der REST-API aufgerufen werden könnten.

Hooks und Events. In Systemen wie CouchDB kann ein Eventhandler auf die Änderung eines Dokumentes oder Objekts registriert werden. Dieses Feature erlaubt es, Echtzeitanwendungen zu implementieren. Ein Client, der sich auf die Änderung eines Objektes registriert, wird in dem Moment über die Änderung informiert, in dem sie passiert. Diese Möglichkeit wollen wir auch für Orestes realisieren, sodass ein Client sich auf die Änderung eines einzelnen Objektes oder eine ganze Gruppe von Objekten registrieren kann. Dabei wären Eventhandler wie *onCommit()* oder *onWrite(Namespace, Class)* denkbar. Die Implementierung derartiger Event-Handler in existierenden Systemen ist meist zu grobgranular, um nützlich zu sein [3]. Wir wollen deshalb untersuchen, wie Klassen von Events ausgedrückt werden können, z.B. durch eine Domain Specific Language. Desweiteren wollen wir ermitteln, mit welchen Mechanismen und Algorithmen feingranulare Eventhandler realisiert werden können, ohne dass die Skalierbarkeit des Orestes-Servers verletzt wird. Auch die Ausfallsicherheit muss dabei berücksichtigt werden. Ein Client, der sich auf eine bestimmte Änderung registriert hat, muss auch dann über Änderungen informiert werden, wenn er zum Zeitpunkt der Änderung vorübergehend nicht erreichbar war.

Evaluation. Wir haben in dieser Arbeit keinen Test mit einem CDN durchgeführt, da sich während der Vorbereitung herausstellte, dass Cloudfronts Invalidierungs-Schnittstelle für massive Schreibvorgänge nicht geeignet ist. In Zukunft werden wir deshalb CDNs ermitteln, die Invalidierungen in dem Maße unterstützen, das ein schreiblastiger Test verursacht. Unsere bisherigen Tests haben bereits gezeigt, dass Invalidierungen ihren Zweck erfüllen und Transaktionsabbrüche verhindern. Ein komplexes Test-Szenario, bei dem sowohl mehrere Forward-Proxy-Caches, als auch ein CDN und mehrere Reverse-Proxy-Caches zum Einsatz kommen, wäre ideal, um das Zusammenspiel aller Techniken zu untersuchen. Dieser Test sollte mit mehreren Clients durchgeführt werden, die durch unterschiedliche Forward-Proxy-Caches mit dem Sever kommunizieren und auf einer zufälligen Objektmenge arbeiten. Dies würde das Zusammenspiel von CDNs, Bloomfiltern und serverseitigen Invalidierungen testen und neue Einblicke in ihre Funktionalität als Gesamtsystem gestatten.

Architektur. Um zu verhindern, dass ein Orestes-Server zu einem Performance-Bottleneck wird, ist es nötig, mehrere Orestes-Server für eine Datenbank zu nutzen. Eine wichtige Grundvoraussetzung dafür ist, dass der Zustand der Orestes-Server, insbesondere die Verwaltung der Transaktionen, verteilt werden kann. Einen wichtigen Schritt in diese Richtung haben wir bereits unternommen, indem wir den Zustand auf das Nötigste reduziert und durch eine Schnittstelle abstrahiert haben. Wir planen, den verbleibenden Zustand zukünftig in einem Cluster aus Instanzen des In-Memory-Caches Redis zu speichern. Die Orestes-Server sind dann zustandslos, so dass jeder Orestes-Server jeden beliebigen Request bearbeiten kann. Ein Load-Balancer kann Requests dann über alle Orestes-Server verteilen. Dies ist der wichtigste Schritt, um auch den Orestes-Server selbst horizontal skalierbar zu machen.

Polyglot Persistence. Wir haben in dieser Arbeit mit dem Redis-Wrapper bereits gezeigt, dass Orestes nicht auf Objektdatenbanken beschränkt ist. Wir wollen deshalb untersuchen, wie die Schnittstelle erweitert werden kann, sodass im Orestes Protokoll auch Dokumentendatenbanken, relationale Datenbanken, Wide Column Stores und ggf. Graphendatenbanken unterstützt werden. Dabei sollen alle gemeinsamen Konzepte generisch gelöst werden, z.B. Transaktionen und Schemaverwaltung. So konnten wir bei der Implementierung bereits Gemeinsamkeiten zwischen dem Key-Value-Store Redis und den Objektdatenbanken identifizieren, die Spezifika abstrahieren und neue Funktionalität (Transaktionen und Schemaverwaltung) hinzufügen. Um Orestes mit den übrigen Datenmodellen zu verbinden, ist zunächst eine eingehende Untersuchung der Datenmodelle nötig, um die gemeinsamen Konzepte im Protokoll identisch abzubilden, z.B. Tupel, Key-Value Paare und Objekte.

Neue Anbindungen und APIs.: Zahlreiche neue Persistenz-APIs und Datenbankbackends sind für Orestes denkbar. Wir wollen untersuchen, welche geeignet und vorteilhaft sind. Zu den Persistenz-APIs, die wir als nächstes implementieren wollen, zählen die Java Persistence API und JCache (ein Key-Value API). Unter den zahlreichen neueren Datenbanksystemen erscheint uns vor allem eine Anbindung an MongoDB und Cassandra interessant.



7 Schluss

Es ist bemerkenswert, dass Eric Brewer, dessen CAP Theorem gewaltigen Einfluss auf die NoSQL Bewegung hat, in der Key-Note Präsentation „Advancing Distributed Systems“ der RICON Konferenz 2012 *Consistent Caching* als eines der wichtigsten fehlenden Features heutiger NoSQL Systeme einordnet [222]. Er spricht in seinem Vortrag explizit einen Invalidierungsmechanismus durch *Commit Hooks* an, der dem Ansatz gleicht, den wir als serverseitige Cache-Invalidierungen untersucht haben. Wir denken, dass die bloomfilterbasierte Cache Kohärenz ein ebenso mächtiger und überdies generischer Ansatz für Consistent Caching ist.

Betrachten wir zusammenfassend Abbildung 113, wo die untersuchten Techniken und ihr Zusammenspiel zur Erhaltung der Cache Kohärenz demonstriert ist. Potentielle Clients sind beliebige Endgeräte. So könnte eine mobile App aus dem Mobilfunknetz, eine Desktopanwendung aus einem Heimnetzwerk, eine Businessanwendung aus einem Firmennetzwerk oder ein Application Server aus einer Cloudumgebung den Orestes-Server ansprechen. Da das Orestes-Protokoll auf HTTP und einer einfachen REST Architektur beruht, kann es praktisch aus jeder Programmiersprache und auf jedem System genutzt werden. Clients, die aus einem Firmennetzwerk HTTP kommunizieren, sind meist nicht direkt mit dem Server verbunden, sondern durch einen Forward-Proxy-Cache. Diese können, genau wie Browser-Caches und Interception Proxys von ISPs, durch den Orestes-Server nicht kontrolliert werden und bei Objekt-Änderungen nicht invalidiert werden. Viele Web-Anwendungen liefern aufgrund dieser Problematik ihre Inhalte für solche Caches als nicht cachebar aus. Wir haben jedoch gezeigt, dass mit Bloomfiltern eine serverseitige Invalidierung gar nicht notwendig ist, sondern diese Aufgabe an den Client verlagert werden kann (in der Abbildung durch grüne Pfeile gekennzeichnet). Wir konnten zeigen, dass ein Client, der Bloomfilter verwendet, nur Objektversionen sieht, die mindestens so aktuell sind, wie der geladene Bloomfilter selbst. Die drei Parameter, die es für den Orestes-Server zu wählen gilt, sind:

- Die Cachingdauer von Objekten.
- Die erwartete Anzahl an Änderungen pro Sekunde.
- Die maximale False-Positive Rate, die für den Bloomfilter noch akzeptabel ist.

Durch die geeignete Wahl der Parameter kann eine Kombination aus einer hohen Cache-Hit Ratio durch langes Cachen von Objekten und einem kompakten Bloomfilter gefunden werden. Wir haben gezeigt, dass unsere Umsetzung der Bloomfilter existierenden Implementierungen überlegen ist und auch für Cluster aus Orestes-Servern genutzt werden kann.

In dieser Arbeit konnten wir außerdem zeigen, dass das Orestes-Protokoll mit den Web-Cache-Technologien Content-Delivery-Networks (CDNs) und Reverse-Proxy-Caches gewinnbringend kombinierbar ist. Ein neuer Invalidierungs-Service, der um beliebig viele Invalidierungs-Protokolle für CDNs und Reverse-Proxys erweitert werden kann, erlaubt es, Änderungen sofort an Caches zu propagieren (durch rote Pfeile gekennzeichnet). Diese Caches können dadurch selbst dann kohärent Anfragen beantworten, wenn ein Client eine Revalidierung gefordert hat. Durch die Kombination von Bloomfiltern und dem Invalidierungs-Service können wir überdies die Zeit überbrücken, die ein CDN eventuell für eine

Invalidierung benötigt (lila Pfeil). Wir erhalten also die Cache-Kohärenz für serverseitig kontrollierbare Caches durch den Invalidierungs-Service und für alle übrigen Caches durch Bloomfilter. Orestes wird dadurch zu einem idealen Protokoll für DBaaS- und Cloud-Deployment-Modelle. Durch das kohärente Web-Caching können Clients stets mit geringer Netzwerklatenz auf die Datenbank zugreifen, unabhängig von ihrer Position und ohne erhöhte Transaktionsabbruchraten durch Stale-Reads.

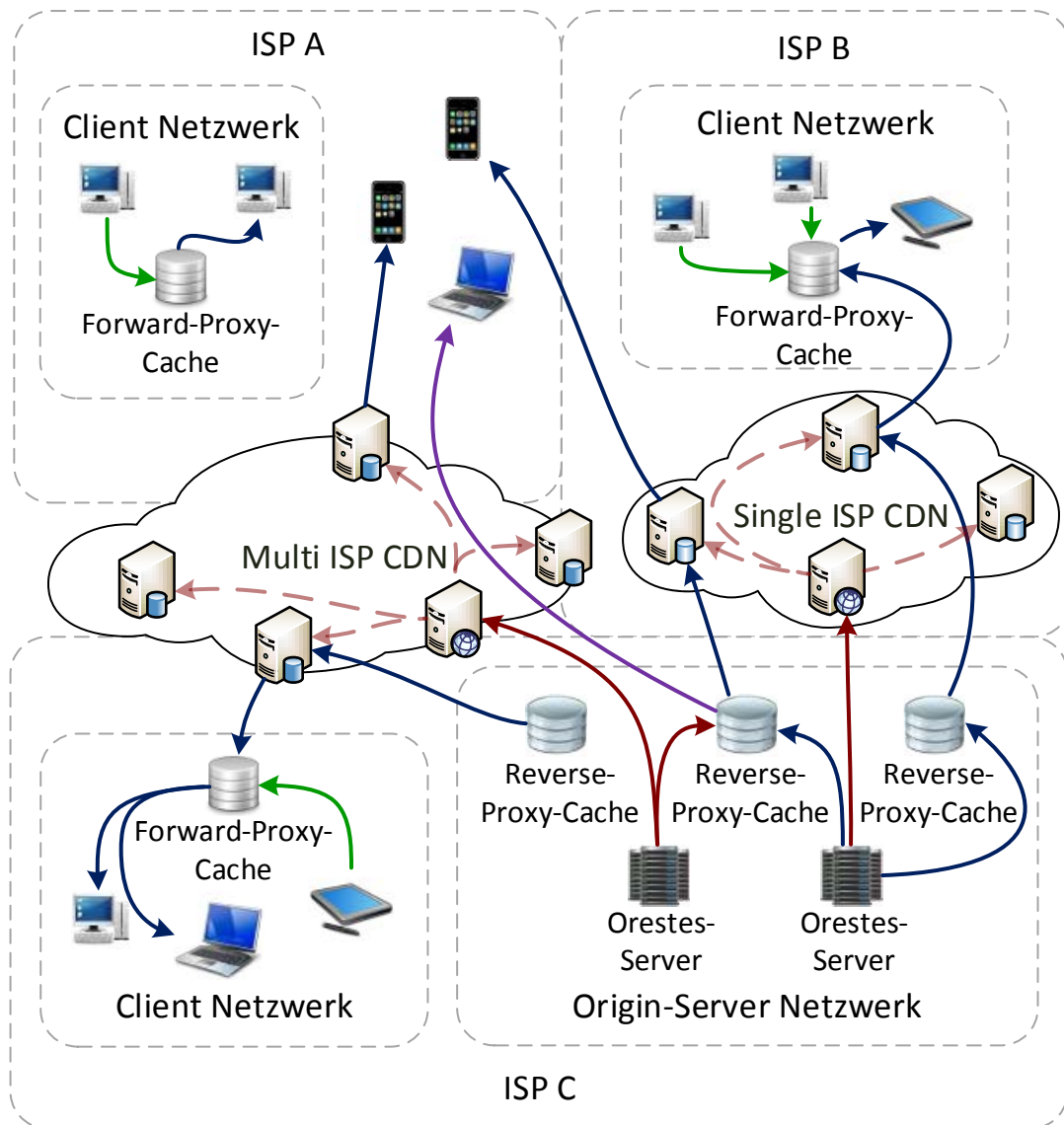


Abbildung 113 Zusammenspiel der in dieser Arbeit untersuchten Techniken.

Dass die Ergebnisse dieser Arbeit übertragbar sind auf andere Datenmodelle, Persistenz-APIs und Datenbanksysteme haben wir durch einen neuen Wrapper für den Key-Value Store Redis praktisch demonstriert. Außerdem haben wir die JavaScript Persistenz-API JSPA vorgestellt, die beispielsweise aus HTML5 Apps und Browsern genutzt werden kann. Wir sind deshalb überzeugt, dass mit Orestes und seinem kohärenten Web-Caching die Basis für ein erweiterbares Polyglot Persistence Protokoll gelegt ist.



A Abbildungsverzeichnis

Abbildung 1 Anforderungen. Diese sieben Forderungen an ein modernes Datenbanksystem sollen durch die Ergebnisse dieser Arbeit erfüllt werden.....	3
Abbildung 2 Bloomfilterbasierte Cache Kohärenz.....	4
Abbildung 3 Nutzung serverkontrollierter Cache Verbunde.....	6
Abbildung 4 Polyglot Persistence. Das ORESTES Protokoll erhält eine neue Abstraktion zur Speicherung von Key-Value Paaren. Die Implementierung erhält eine Anbindung an einen Key-Value Store, der einerseits Objektpersistenz (ohne Querys) übernehmen kann und andererseits Key-Value Operationen.....	7
Abbildung 5 Die Anforderungen an ein Persistenzprotokoll.....	10
Abbildung 6 Das Skalierungsmodell von ORESTES. Der Scale-out basiert auf dem Fan-out von HTTP Requests durch Load Balancer und dem Abfangen von Leseanfragen durch Web-Caches.....	11
Abbildung 7 Die REST/HTTP Schnittstelle von ORESTES. Die Abbildung zeigt einen Ausschnitt der Ressourcenstruktur <i>root > db > namespace > class > object</i> . Die Abhängigkeiten von Ressourcen (Hypermedia) sind in rot markiert.....	12
Abbildung 8 Die fünf verschiedenen Arten von Web-Caches. Man kann zwischen eingebetteten (1, 5) und serverbasierten (2,3,4) Web Caches unterscheiden.....	13
Abbildung 9 Skalierungsmechanismen von Web-Caches.....	15
Abbildung 10 Arbeitsweise eines Web-Caches.....	18
Abbildung 11 Ein einfaches Beispiel, das mit JSPA alle Objekte einer Klasse lädt.....	18
Abbildung 12 Ausführung eines einfachen Querys über das ORESTES Web-Interface.....	19
Abbildung 13 Abfrage der Extension der Klasse „Coffee“.....	20
Abbildung 14 Abfrage einer Instanz der Klasse „Coffee“.....	21
Abbildung 15 Übersicht beteiligter Systeme bei einer konditionalen Leseanfrage für ein Objekt.....	22
Abbildung 16 Klassische und moderne Datenbankkommunikation.....	25
Abbildung 17 Protokolle von 123 Datenbank-APIs (aus [62]).....	26
Abbildung 18 Die vier Szenarien für die Beziehung zwischen Anwendung und Datenbank...28	28
Abbildung 19 Typische Schritte zur Nutzung eines DBaaS Angebots am Beispiel von MongoHQ [66].....	29
Abbildung 20 Einordnung von DBaaS gemäß der NIST Definition [70].....	31
Abbildung 21 Konsolidierungsstrategien für Multitenancy Datenbanken (cf. [77], [78]).....	32
Abbildung 22 UML Klassendiagramm der persistenten Objekte des Benchmarks.....	35
Abbildung 23 Testbed für die Evaluierung von ORESTES unter massiv paralleler Anfragelast...36	36
Abbildung 24 Ergebnisse für die Ausführungszeit von 50 parallelen Clients. Die statistische Standardabweichung ist durch die schwarzen Balken gekennzeichnet.....	37
Abbildung 25 Gemittelter Anteil verschiedener Arten von Operationen an der Ausführungszeit (Read/Write Verhältnis 10/1, 30.000 Datenbankobjekte).....	38
Abbildung 26 Die Problematik der Stale Reads.....	39
Abbildung 27 Taxonomie für Caches.....	43

Abbildung 28 Änderungsdatenstruktur. Schreibvorgänge können von beliebigen ORESTES-Servern bedient werden, die Änderungen an die Änderungsdatenstruktur propagieren.....	46
Abbildung 29 Ablauf einer ORESTES Transaktion auf Protokollebene.....	47
Abbildung 30 Grundlegender Commit Algorithmus des ORESTES Datenbankwrappers	48
Abbildung 31 Ausgeführte Instruktionen eines relationalen Datenbanksystem unter der Last eines Standardbenchmarks (TPC-C). Die gestrichelte Linie gibt den Anteil nützlicher Operationen an, ermittelt durch die Ausführung auf einem modifizierten Overhead-freien Kernel mit nur einer Transaktion. Ergebnisse und Grafik aus [148].	49
Abbildung 32 Die explizite Offenlegung der Nebenläufigkeitskontrolle in JPA.....	51
Abbildung 33 Algorithmen zur Nebenläufigkeitskontrolle in Datenbanksystemen.	51
Abbildung 34 Der SGT Algorithmus.....	52
Abbildung 35 Die Unvereinbarkeit von SGT und Web-Caching.....	53
Abbildung 36 Die drei Phasen in optimistischen Synchronisationsverfahren.....	55
Abbildung 37 Die Validierung im BOCC und FOCC Algorithmus.	55
Abbildung 38 Die Gefahr von Nonrepeatable-Reads bei fehlender Validierung des Read-Sets.	56
Abbildung 39 Beispiel für die Arbeitsweise von BOCC in Kombination mit Web-Caching.	57
Abbildung 40 Die Problematik unechter Konflikte im mengenbasierten BOCC.	58
Abbildung 41 Die ursprüngliche Form des Optimistic Locking in expliziter Form (cf. [8])......	58
Abbildung 42 Optimistic Locking in objektorientierten Persistenz-APIs am Beispiel von JPA.	59
Abbildung 43 Balancierte Suchbäume für die Änderungsdatenstruktur bei n eingefügten Elementen.....	63
Abbildung 44 Nutzung einer Hashtabelle für die Änderungsdatenstruktur.....	64
Abbildung 45 Das Prinzip eines Bloomfilters.....	66
Abbildung 46 Die <i>Insert</i> und <i>Contains</i> Methode eines Bloomfilters.....	67
Abbildung 47 Beispiel für Einfügen und Abfragen auf einem Bloomfilter.	68
Abbildung 48 Die False-Positive Rate f für unterschiedliche Lasten, bei $m=345062$ Bit. Die erwartete Last von $WPS=10$ ist durch die rote Linie markiert. Die x-Achse (logarithmisch) gibt an, um welchen Faktor die tatsächliche Last von diesem Wert abweicht.....	70
Abbildung 49 Implementierung von Vereinigungs- und Schnittmengen durch bitweise Operatoren.....	73
Abbildung 50 Die Funktionsweise von Summary Cache.....	75
Abbildung 51 Die Sorted-String Table Datenstruktur.	76
Abbildung 52 Die Verwendung von Bloomfiltern zur Vermeidung eines Festplattenzugriffs..	77
Abbildung 53 Die <i>Insert</i> , <i>Contains</i> und <i>Delete</i> Methode eines Counting Bloomfilters.....	79
Abbildung 54 Beispiel für die Funktionsweise eines Counting Bloomfilters.....	80
Abbildung 55 Der Materialized Counting Bloomfilter.....	81
Abbildung 56 Der Bitwise Bloomfilter.	82
Abbildung 57 Ein stark vereinfachtes Klassendiagramm der Bloomfilter.	89
Abbildung 58 Benchmarkergebnisse für die Default Einstellung unserer Bloomfilters bei der Ausführung des Benchmarks von Skjogstad [177].....	90
Abbildung 59 Das Prinzip der Redis Bloomfilter.	92



Abbildung 60 Mehrstufige asynchrone Replikation in Redis.....	93
Abbildung 61 Vereinfachte Architektur des Redis Bloomfilters.....	94
Abbildung 62 Die <i>Insert</i> , <i>Contains</i> und <i>Delete</i> Methode eines Counting Bloomfilters.....	95
Abbildung 63 Die Problematik nebenläufiger Änderungen des Bit-Arrays (Leserichtung: von links nach rechts und von oben nach unten).....	96
Abbildung 64 Das Einfügen in den Counting Bloomfilter als serverseitiges Lua Script.	98
Abbildung 65 Das Einfügen in den Counting Bloomfilter als serverseitiges Lua Script.	99
Abbildung 66 Ergebnisse für einen exemplarischen Durchlauf des einfachen Benchmarks. .	101
Abbildung 67 Die hashCode Implementierung von String in Java (cf. [187]).....	103
Abbildung 68 Das Problem der ungleichmäßigen Verteilung bei der Verwendung der Modulo Reduktion.	104
Abbildung 69 Lösung der ungleichmäßigen Verteilung durch Rejection Sampling.....	105
Abbildung 70 Nutzung kryptographischer Hashfunktionen für Bloomfilter.	108
Abbildung 71 Die Chi-Quadrat Verteilung für drei verschiedene Freiheitsgrade.....	112
Abbildung 72 Ergebnisse für den Chi-Quadrat Test bei Random Strings.	114
Abbildung 73 Die p-Werte für den Chi-Quadrat Test bei Random Strings.	115
Abbildung 74 Die deutlich ungleichmäßige Verteilung von Adler-32.....	115
Abbildung 75 Ergebnis für Increasing Integers.....	116
Abbildung 76 Test-Ergebnisse mit und ohne Rejection Sampling.....	117
Abbildung 77 Finale False-Positive Rate beim Einfügen in Bloomfilter $m = 300000$ und $n = 30000$	118
Abbildung 78 Qualität von Hashwerten auf künstlichen Objekt-IDs anhand des Mittelwerts der Chi-Quadrat-Statistik.	118
Abbildung 79 Geschwindigkeit der unterschiedlichen Hashfunktionen für die Berechnung von 1.000.000 Hashwerten mit $k=5$ und $m=1000$, gemittelt über 10 Durchläufe.....	119
Abbildung 80 Die Nutzung der Bloomfilter in ORESTES.....	121
Abbildung 81 Die Benutzung des Bloomfilters auf Clientseite.	122
Abbildung 82 Der Aufbau für den praktischen Test der Bloomfilter.....	123
Abbildung 83 Beispiel für ein Testszenario, das mit und ohne Bloomfilter eine anderes Ergebnis zeigt.....	124
Abbildung 84 Erfüllung der serverseitigen Anforderungen durch den Bloomfilter.....	125
Abbildung 85 Erfüllung der clientseitigen Anforderungen durch den Bloomfilter.....	126
Abbildung 86 Das Szenario, das trotz Bloomfiltern zu einem Transaktionsabbruch führt.	126
Abbildung 87 Ein einfacher Netzwerkaufbau unter Verwendung von Varnish.	130
Abbildung 88 Der Varnish Zustandsautomat, nachdem das VCL Skript evaluiert wird. Quelle: https://www.varnish-software.com/static/book/VCL_Basics.html#the-vcl-state-engine	131
Abbildung 89 VCL-Skript für eine PURGE Methode in Varnish.	132
Abbildung 90 Drei Varnish Instanzen, die die eingehenden HTTP Requests auf zwei Orestes-Server aufteilen.....	133
Abbildung 91 Verschiedene Netzwerkstrukturen innerhalb eines CDNs.....	135
Abbildung 92 Die beiden Verteilungsarten von CDNs.....	138
Abbildung 93 Ablauf einer Invalidierung in CloudFront.....	142

Abbildung 94 Ein Beispiel-Invalidierungsauftrag an CloudFront.....	143
Abbildung 95 Funktionsweise des Invalidation-Service in Orestes.....	145
Abbildung 96 Eine lesende Anfrage eines noch nicht gecachten Objektes.....	146
Abbildung 97 Ein Schreibvorgang eines Objektes.....	147
Abbildung 98 Ablauf einer Invalidierung nach dem ein Objekt geschrieben wurde	148
Abbildung 99 Ablauf einer lesenden Anfrage, wenn das CDN veraltete Daten enthält.....	148
Abbildung 100 Ein Vergleich der drei Invalidierungsstrategien, der die Ausbreitungsdauer der jeweils verschickten Nachrichten miteinander vergleicht.	151
Abbildung 101 Lokaler Cache auf Basis eines Key-Value-Stores	153
Abbildung 102 Ein Key-Value-Store basierter In-Memory-Cache für die Orestes-Server.....	155
Abbildung 103 Die zerlegte Orestes-Schnittstelle.....	156
Abbildung 104 Umsetzung von Polyglot Persistence für Redis - Auswahl geeigneter Eigenschaften aus der Orestes Schnittstelle.....	157
Abbildung 105 Speicherung von Objekten in Redis.	159
Abbildung 106 Der Datenbank-Explorer der Web-Oberfläche des Orestes-Servers mit Redis als Backend.	160
Abbildung 107 Der Object-Lifecycle in JPA.....	163
Abbildung 108 Erstellen und abrufen eines Querys in JSPA.....	164
Abbildung 109 Eine Transaktion in JSPA.....	165
Abbildung 110 Vereinfachtes JSPA Architekturdiagramm	166
Abbildung 111 Ein Beispiel, dass die Queue basierte Synchronisation demonstriert.....	167
Abbildung 112 Ablauf des Programms aus Abbildung 111.....	168
Abbildung 113 Zusammenspiel der in dieser Arbeit untersuchten Techniken.....	175



B Quellen und Referenzen

- [1] R. Cattell, „Scalable sql and nosql data stores“, *ACM SIGMOD Record*, Bd. 39, Nr. 4, S. 12–27, 2011.
- [2] D. Sitaram und G. Manjunath, *Moving To The Cloud: Developing Apps in the New World of Cloud Computing*. Syngress, 2011.
- [3] E. Redmond, J. Wilson, und J. Carter, *Seven databases in seven weeks: a guide to modern databases and the NoSQL movement*. Lewisville, Tex.; Farnham: Pragmatic Bookshelf; O’Reilly [distributor], 2012.
- [4] T. White, *Hadoop: The definitive guide*. Yahoo Press, 2010.
- [5] T. Härder und E. Rahm, *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001.
- [6] F. Gessert und F. Bücklers, „Performanz- und Reaktivitätssteigerung von OODBMS vermittels der Web-Caching-Hierarchie unter Einsatz und Adaption offener Standards“.
- [7] F. Gessert, F. Bücklers, und N. Ritter, „Orestes: a REST protocol for horizontally scalable cloud database access“.
- [8] G. Weikum und G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Pub, 2002.
- [9] P. J. Sadalage, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional.
- [10] S. Sakr, A. Liu, D. M. Batista, und M. Alomari, „A survey of large scale data management approaches in cloud environments“, *Communications Surveys & Tutorials, IEEE*, Bd. 13, Nr. 3, S. 311–336, 2011.
- [11] S. Edlich, *NoSQL Einstieg in die Welt nichtrelationaler Web-2.0-Datenbanken*. München: Hanser, 2010.
- [12] D. Crockford, „The application/json media type for javascript object notation (json)“, 2006.
- [13] J. C. Anderson, J. Lehnardt, N. Slater, und Safari Tech Books Online, *CouchDB the definitive guide*. Sebastopol, Calif.: O’Reilly Media, Inc., 2010.
- [14] J. Dean und S. Ghemawat, „MapReduce: Simplified data processing on large clusters“, *Communications of the ACM*, Bd. 51, Nr. 1, S. 107–113, 2008.
- [15] M. Isard, M. Budiu, Y. Yu, A. Birrell, und D. Fetterly, „Dryad: distributed data-parallel programs from sequential building blocks“, *ACM SIGOPS Operating Systems Review*, Bd. 41, Nr. 3, S. 59–72, 2007.
- [16] „Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real | Apache Hadoop for the Enterprise | Cloudera“. [Online]. Available: <http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/>. [Accessed: 12-Nov-2012].
- [17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, und others, „Spanner: Google’s Globally-Distributed Database“, *To appear in Proceedings of OSDI*, S. 1, 2012.
- [18] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, und G. Czajkowski, „Pregel: a system for large-scale graph processing“, in *Proceedings of the 2010 international conference on Management of data*, 2010, S. 135–146.

- [19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, und T. Vassilakis, „Dremel: interactive analysis of web-scale datasets“, *Proceedings of the VLDB Endowment*, Bd. 3, Nr. 1–2, S. 330–339, 2010.
- [20] J. Gray und A. Reuter, *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.
- [21] M. Stonebraker und R. Cattell, „10 rules for scalable performance in simple operation datastores“, *Communications of the ACM*, Bd. 54, Nr. 6, S. 72–80, 2011.
- [22] M. Pathan und R. Buyya, „A taxonomy of CDNs“, *Content delivery networks*, S. 33–77, 2008.
- [23] P. Membrey, E. Plugge, und D. Hows, *Practical Load Balancing: Ride the Performance Tiger*, 1. Aufl. Apress, 2012.
- [24] S. V. Nagaraj, *Web caching and its applications*, Bd. 772. Springer, 2004.
- [25] F. Gessert, F. Bücklers, und N. Ritter, „Poster - Orestes: a REST protocol for horizontally scalable cloud database access“ . .
- [26] D. J. Abadi, „Data management in the cloud: Limitations and opportunities“, *IEEE Data Engineering Bulletin*, Bd. 32, Nr. 1, S. 3–12, 2009.
- [27] C. Russell, „JSR 12: Java Data Objects (JDO) specification“, *Sun Microsystems*, 2003.
- [28] L. DeMichiel, „JSR 317: Java Persistence API, Version 2.0“, *Java Specification Request, Sun Microsystems*, 2009.
- [29] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, und S. Zdonik, „The object-oriented database system manifesto“, in *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, 1989, Bd. 57.
- [30] C. Strauch, U. L. S. Sites, und W. Kriha, „NoSQL databases“, *Lecture Notes, Stuttgart Media University*, 2011.
- [31] M. Rabinovich und O. Spatscheck, „Web caching and replication“, *SIGMOD Record*, Bd. 32, Nr. 4, S. 107, 2003.
- [32] „Versant Object Database“. [Online]. Available: <http://www.versant.com/>. [Accessed: 28-Juni-2012].
- [33] „db4o Open Source Object Database“. [Online]. Available: <http://www.db4o.com/>. [Accessed: 28-Juni-2012].
- [34] R. T. Fielding, „Architectural styles and the design of network-based software architectures“, University of California, 2000.
- [35] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, und T. Berners-Lee, „RFC 2616: Hypertext Transfer Protocol–HTTP/1.1, 1999“, URL <http://www.rfc.net/rfc2616.html>, 2009.
- [36] Q. Zhang, L. Cheng, und R. Boutaba, „Cloud computing: state-of-the-art and research challenges“, *Journal of Internet Services and Applications*, Bd. 1, Nr. 1, S. 7–18, 2010.
- [37] S. Podlipnig und L. Böszörményi, „A survey of web cache replacement strategies“, *ACM Computing Surveys (CSUR)*, Bd. 35, Nr. 4, S. 374–398, 2003.
- [38] T. Härder, „Caching over the entire user-to-data path in the internet“, *Data Management in a Connected World*, S. 150–170, 2005.



- [39] D. Wessels und others, „Squid web proxy cache“, URL: <http://www.squid-cache.org>, 2001.
- [40] „Microsoft Forefront Threat Management Gateway (TMG)“. [Online]. Available: <http://www.microsoft.com/tmg>. [Accessed: 28-Juni-2012].
- [41] K. Gilly, C. Juiz, und R. Puigjaner, „An up-to-date survey in web load balancing“, *World Wide Web*, Bd. 14, Nr. 2, S. 105–131, 2011.
- [42] J. van Vliet und F. Paganelli, *Programming Amazon EC2*. O'Reilly Media, 2011.
- [43] V. Valloppillil und K. W. Ross, *Cache array routing protocol v1*. 1998.
- [44] D. Wessels und K. Claffy, „Application of internet cache protocol (ICP), version 2“, 1997.
- [45] P. Vixie und D. Wessels, „Hyper Text Caching Protocol (HTCP/0.0)“, RFC 2756, January, 2000.
- [46] A. Rousskov und D. Wessels, „Cache digests“, *Computer Networks and ISDN Systems*, Bd. 30, Nr. 22, S. 2155–2168, 1998.
- [47] S. Ghemawat, H. Gobioff, und S. T. Leung, „The Google file system“, in *ACM SIGOPS Operating Systems Review*, 2003, Bd. 37, S. 29–43.
- [48] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, und R. E. Gruber, „Bigtable: A distributed storage system for structured data“, *ACM Transactions on Computer Systems (TOCS)*, Bd. 26, Nr. 2, S. 4, 2008.
- [49] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J. M. Léon, Y. Li, A. Lloyd, und V. Yushprakh, „Megastore: Providing scalable, highly available storage for interactive services“, in *Proc. of CIDR*, 2011, S. 223–234.
- [50] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, und others, „H-store: a high-performance, distributed main memory transaction processing system“, *Proceedings of the VLDB Endowment*, Bd. 1, Nr. 2, S. 1496–1499, 2008.
- [51] L. George, *HBase: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2011.
- [52] M. T. Özsu und P. Valduriez, *Principles of distributed database systems*. Springer, 2011.
- [53] T. Guidici, *Windows Azure Platform*, New. Apress, 2011.
- [54] R. Brunetti, *Windows Azure Step by Step*, Pap/Psc. Microsoft Press Corp., 2011.
- [55] G. Saake, A. Heuer, und K. U. Sattler, *Datenbanken: Implementierungstechniken*. Hüthig Jehle Rehm, 2005.
- [56] K. Amiri, S. Park, R. Tewari, und S. Padmanabhan, „DBProxy: A dynamic data cache for Web applications“, in *Proceedings of the International Conference on Data Engineering*, 2003, S. 821–831.
- [57] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, und B. Reinwald, „Adaptive database caching with DBCache“, *Data Engineering*, Bd. 27, Nr. 2, S. 11–18, 2004.
- [58] R. Ramakrishnan, „CAP and Cloud Data Management“, *Computer*, Bd. 45, Nr. 2, S. 43–49, 2012.
- [59] D. Flanagan, *JavaScript: the definitive guide*. O'Reilly Media, Incorporated, 2006.
- [60] S. Tilkov und S. Vinoski, „Node.js: Using JavaScript to build high-performance network programs“, *Internet Computing, IEEE*, Bd. 14, Nr. 6, S. 80–83, 2010.

- [61] B. A. Tate, „Seven Languages In Seven Weeks: A Pragmatic Guide To Learning Programming Languages (Pragmatic Programmers) Author: Bruce“, 2010.
- [62] „123 Database APIs: GeoNames, Freebase and Yahoo Query Language“. [Online]. Available: <http://blog.programmableweb.com/2012/05/23/123-database-apis-geonames-freebase-and-yahoo-query-language/>. [Accessed: 14-Nov-2012].
- [63] A. Lakshman und P. Malik, „Cassandra: a decentralized structured storage system“, *ACM SIGOPS Operating Systems Review*, Bd. 44, Nr. 2, S. 35–40, 2010.
- [64] C. Hay und B. H. Prince, *Azure in Action*. Manning, 2010.
- [65] „NoSQL Databases“. [Online]. Available: <http://nosql-database.org/>. [Accessed: 14-Nov-2012].
- [66] „MongoHQ“. [Online]. Available: <https://www.mongohq.com/home>. [Accessed: 14-Nov-2012].
- [67] „Amazon CloudFront“. [Online]. Available: <http://aws.amazon.com/de/cloudfront/>. [Accessed: 28-Juni-2012].
- [68] „Windows Azure CDN“. [Online]. Available: <http://www.windowsazure.com/de-de/home/features/caching/>. [Accessed: 28-Juni-2012].
- [69] „Cloud Speed Test | CloudHarmony“. [Online]. Available: <http://cloudharmony.com/speedtest>. [Accessed: 14-Nov-2012].
- [70] P. Mell und T. Grance, „The NIST definition of cloud computing“, *National Institute of Standards and Technology*, Bd. 53, Nr. 6, S. 50, 2009.
- [71] „MongoLab - MongoDB Hosting | Cloud Hosted MongoDB“. [Online]. Available: <https://mongolab.com/home>. [Accessed: 14-Nov-2012].
- [72] „Cassandra.io“. [Online]. Available: <http://cassandra.io/>. [Accessed: 15-Nov-2012].
- [73] „Amazon DynamoDB“. [Online]. Available: <http://aws.amazon.com/de/dynamodb/>. [Accessed: 14-Nov-2012].
- [74] „Azure Table Service - REST API“. [Online]. Available: <http://msdn.microsoft.com/en-us/library/windowsazure/dd179423.aspx>. [Accessed: 28-Juni-2012].
- [75] D. Sanderson, *Programming Google App Engine*, 2. Auflage. O’Reilly Media, 2012.
- [76] „For Developers | Cloudant“. [Online]. Available: <https://cloudant.com/for-developers/>. [Accessed: 14-Nov-2012].
- [77] M. Seibold und A. Kemper, „Database as a Service“, *Datenbank-Spektrum*, S. 1–4, 2012.
- [78] B. Reinwald, „Database support for multi-tenant applications“, in *IEEE Workshop on Information and Software as Services*, 2010, Bd. 1, S. 2.
- [79] C. A. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. R. Madden, H. Balakrishnan, N. Zeldovich, und others, „Relational cloud: A database-as-a-service for the cloud“, 2011.
- [80] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, und J. Rittinger, „Multi-tenant databases for software as a service: schema-mapping techniques“, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, S. 1195–1206.
- [81] „SAP HANA One on AWS Marketplace“. [Online]. Available: https://aws.amazon.com/marketplace/pp/B009KA3CRY/ref=mkt_blg_jb_SAPHana1. [Accessed: 16-Nov-2012].
- [82] „Redis To Go“. [Online]. Available: <http://redistogo.com/>. [Accessed: 14-Nov-2012].



- [83] „RedisGreen - Reliable Redis hosting“. [Online]. Available: <http://www.redisgreen.net/>. [Accessed: 14-Nov-2012].
- [84] „Redis Cloud“. [Online]. Available: <http://redis-cloud.com/>. [Accessed: 14-Nov-2012].
- [85] „openredis – Redis Hosting Service“. [Online]. Available: <https://openredis.com/>. [Accessed: 14-Nov-2012].
- [86] „Memcache | Heroku Dev Center“. [Online]. Available: <https://devcenter.heroku.com//articles/memcache#local-memcache-setup>. [Accessed: 14-Nov-2012].
- [87] „Iris Couch“. [Online]. Available: <http://www.iriscouch.com/>. [Accessed: 14-Nov-2012].
- [88] „HostedMongo.com The only UK / USA based MongoDB Hosting“. [Online]. Available: <http://hostedmongo.com/>. [Accessed: 15-Nov-2012].
- [89] „ObjectRocket - Industrial Strength MongoDB“. [Online]. Available: <http://www.objectrocket.com/>. [Accessed: 15-Nov-2012].
- [90] „MongoOd.com - Solution d'hébergement pour vos bases mongoDB.“ [Online]. Available: <https://www.mongood.com/>. [Accessed: 15-Nov-2012].
- [91] „RavenHQ - Home“. [Online]. Available: <https://ravenhq.com/>. [Accessed: 14-Nov-2012].
- [92] „CloudBird - RavenDB in the Cloud“. [Online]. Available: <https://www.cloudbird.net/>. [Accessed: 14-Nov-2012].
- [93] „NuvolaBase: cloudize your data - Commercial support, training and services about OrientDB“. [Online]. Available: <http://www.nuvolabase.com/site/>. [Accessed: 14-Nov-2012].
- [94] M. C. Chu-Carroll, *Code in the Cloud: Programming Google AppEngine*. Pragmatic Programmers, 2011.
- [95] „Hadoop-based Cloud Data Warehouse | Treasure Data“. [Online]. Available: <http://www.treasure-data.com/>. [Accessed: 14-Nov-2012].
- [96] P. Chaganti und R. Helms, *Amazon SimpleDB Developer Guide*. Packt Publishing, 2010.
- [97] „Neo4j | Heroku Dev Center“. [Online]. Available: <https://devcenter.heroku.com//articles/neo4j>. [Accessed: 14-Nov-2012].
- [98] „Amazon Relational Database Service (Amazon RDS)“. [Online]. Available: <http://aws.amazon.com/de/rds/>. [Accessed: 14-Nov-2012].
- [99] „Google Cloud SQL — Google Developers“. [Online]. Available: <https://developers.google.com/cloud-sql/>. [Accessed: 14-Nov-2012].
- [100] „Xeround Cloud Database“. [Online]. Available: <http://xeround.com/>. [Accessed: 15-Nov-2012].
- [101] „ScaleDB: NewSQL Database for Public Cloud and Private Cloud“. [Online]. Available: <http://www.scaledb.com/>. [Accessed: 14-Nov-2012].
- [102] „ClearDB - The Ultra Reliable, Globally Distributed Cloud Database For Your MySQL Applications“. [Online]. Available: <http://www.cleardb.com/>. [Accessed: 14-Nov-2012].
- [103] „Database Load Balancing | MySQL High Availability | Scalebase“. [Online]. Available: <http://www.scalebase.com/products/product-architecture/>. [Accessed: 14-Nov-2012].
- [104] R. Jennings, *Cloud Computing with the Windows Azure Platform (Wrox Programmer to Programmer)*. Wrox, 2009.

- [105] „Oracle Database Cloud Service | Oracle Cloud“. [Online]. Available: <https://cloud.oracle.com/mycloud/f?p=service:database:0::::> [Accessed: 14-Nov-2012].
- [106] „Postgres Plus® Cloud Database | EnterpriseDB“. [Online]. Available: <http://www.enterprisedb.com/products-services-training/products-overview/postgres-plus-tools/postgres-plus-cloud-server>. [Accessed: 14-Nov-2012].
- [107] „Heroku Postgres“. [Online]. Available: <https://postgres.heroku.com/>. [Accessed: 15-Nov-2012].
- [108] „Clustrix“. [Online]. Available: <http://www.clustrix.com/>. [Accessed: 14-Nov-2012].
- [109] „Online Database Software: Zoho Creator“. [Online]. Available: <https://creator.zoho.com/>. [Accessed: 14-Nov-2012].
- [110] „Database.com“. [Online]. Available: <http://database.com/>. [Accessed: 15-Nov-2012].
- [111] H. Hacigumus, B. Iyer, und S. Mehrotra, „Providing database as a service“, in *Data Engineering, 2002. Proceedings. 18th International Conference on*, 2002, S. 29–38.
- [112] M. Brantner, D. Florescu, D. Graf, D. Kossmann, und T. Kraska, „Building a database on S3“, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, S. 251–264.
- [113] S. Das, D. Agrawal, und A. El Abbadi, „Elastras: An elastic transactional data store in the cloud“, *USENIX HotCloud*, Bd. 2, 2009.
- [114] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, und W. Vogels, „Dynamo: amazon’s highly available key-value store“, in *ACM SIGOPS Operating Systems Review*, 2007, Bd. 41, S. 205–220.
- [115] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, und R. Yerneni, „PNUTS: Yahoo!’s hosted data serving platform“, *Proceedings of the VLDB Endowment*, Bd. 1, Nr. 2, S. 1277–1288, 2008.
- [116] M. Hui, D. Jiang, G. Li, und Y. Zhou, „Supporting database applications as a service“, in *Data Engineering, 2009. ICDE’09. IEEE 25th International Conference on*, 2009, S. 832–843.
- [117] R. A. Popa, C. Redfield, N. Zeldovich, und H. Balakrishnan, „CryptDB: protecting confidentiality with encrypted query processing“, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, S. 85–100.
- [118] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, und A. Kemper, „Quality of service enabled database applications“, *Service-Oriented Computing–ICSOC 2006*, S. 215–226, 2006.
- [119] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, und H. Hacigümü\cS, „ActiveSLA: A profit-oriented admission control framework for database-as-a-service providers“, in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, S. 15.
- [120] „Open Data Protocol (OData)“. [Online]. Available: <http://www.odata.org/>. [Accessed: 28-Juni-2012].
- [121] „Protocol Reference - Google Data APIs — Google Developers“. [Online]. Available: <https://developers.google.com/gdata/docs/2.0/reference>. [Accessed: 16-Nov-2012].
- [122] T. Haselmann, G. Thies, und G. Vossen, „Looking into a REST-Based Universal API for Database-as-a-Service Systems“, in *Commerce and Enterprise Computing (CEC), 2010 IEEE 12th Conference on*, 2010, S. 17–24.



- [123] J. Wang, „A survey of web caching schemes for the internet“, *ACM SIGCOMM Computer Communication Review*, Bd. 29, Nr. 5, S. 36–46, 1999.
- [124] B. Krishnamurthy und J. Rexford, „Web Protocols and Practice, HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement“, *Recherche*, Bd. 67, S. 02, 2001.
- [125] K. Gilly, C. Juiz, und R. Puigjaner, „An up-to-date survey in web load balancing“, *World Wide Web*, Bd. 14, Nr. 2, S. 105–131, 2011.
- [126] W. Ali, S. M. Shamsuddin, und A. S. Ismail, „A survey of web caching and prefetching“, *Int. J. Advance. Soft Comput. Appl*, Bd. 3, Nr. 1, S. 18–44, 2011.
- [127] D. Wessels, „Squid Proxy Cache“. [Online]. Available: <http://www.squid-cache.org/>. [Accessed: 28-Juni-2012].
- [128] „Bug 3379 – Combination of If-Match and a Cache Hit result in TCP Connection Failure“. [Online]. Available: http://bugs.squid-cache.org/show_bug.cgi?id=3379. [Accessed: 15-Nov-2012].
- [129] „Bug 3398 – persistent server connection closed before PUT/POST/DELETE/etc“. [Online]. Available: http://bugs.squid-cache.org/show_bug.cgi?id=3398. [Accessed: 15-Nov-2012].
- [130] T. Härder und A. Bühmann, „Datenbank-Caching-Eine systematische Analyse möglicher Verfahren“, *Informatik-Forschung und Entwicklung*, Bd. 19, Nr. 1, S. 2–16, 2004.
- [131] T. Macedo und F. Oliveira, *Redis Cookbook*. O'Reilly Media, 2011.
- [132] H. Plattner, „A common database approach for OLTP and OLAP using an in-memory column database“, in *Proceedings of the 35th SIGMOD international conference on Management of data*, 2009, S. 1–2.
- [133] H. Plattner und A. Zeier, *In-memory data management: an inflection point for enterprise applications*. Springer, 2011.
- [134] M. T. Ozsu und P. Valduriez, *Principles of distributed database systems*. Springer-Verlag New York Inc, 2011.
- [135] P. A. Bernstein und E. Newcomer, *Principles of Transaction Processing, Second Edition (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2009.
- [136] B. H. Bloom, „Space/time trade-offs in hash coding with allowable errors“, *Communications of the ACM*, Bd. 13, Nr. 7, S. 422–426, 1970.
- [137] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, und G. Varghese, „An improved construction for counting bloom filters“, *Algorithms-ESA 2006*, S. 684–695, 2006.
- [138] L. Fan, P. Cao, J. Almeida, und A. Z. Broder, „Summary cache: a scalable wide-area web cache sharing protocol“, *IEEE/ACM Transactions on Networking (TON)*, Bd. 8, Nr. 3, S. 281–293, 2000.
- [139] P. S. Almeida, C. Baquero, N. Pregui\cca, und D. Hutchison, „Scalable bloom filters“, *Information Processing Letters*, Bd. 101, Nr. 6, S. 255–261, 2007.
- [140] B. Chazelle, J. Kilian, R. Rubinfeld, und A. Tal, „The Bloomier filter: an efficient data structure for static support lookup tables“, in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004, S. 30–39.
- [141] Y. Koucheryavy, G. Giambene, D. Staehle, F. Barcelo-Arroyo, T. Braun, und V. Siris, *Traffic and QoS management in wireless multimedia networks: COST 290 final report*, Bd. 31. Springer, 2009.

- [142] A. Lall und M. Ogihara, „The Bitwise Bloom Filter“, 2007.
- [143] S. Cohen und Y. Matias, „Spectral bloom filters“, in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, S. 241–252.
- [144] M. Mitzenmacher, „Compressed bloom filters“, *IEEE/ACM Transactions on Networking (TON)*, Bd. 10, Nr. 5, S. 604–612, 2002.
- [145] A. G. Konheim, *Hashing*, 1. Auflage. John Wiley & Sons, 2010.
- [146] D. E. Culler, J. P. Singh, und A. Gupta, *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999.
- [147] A. Thomasian, „Concurrency control: methods, performance, and analysis“, *ACM Computing Surveys (CSUR)*, Bd. 30, Nr. 1, S. 70–119, 1998.
- [148] S. Harizopoulos, D. J. Abadi, S. Madden, und M. Stonebraker, „OLTP through the looking glass, and what we found there“, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, S. 981–992.
- [149] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, und P. Helland, „The end of an architectural era:(it’s time for a complete rewrite)“, in *Proceedings of the 33rd international conference on Very large data bases*, 2007, S. 1150–1160.
- [150] I. B. M. C. R. Division, J. N. Gray, P. Homan, R. L. Obermarck, und H. F. Korth, *A straw man analysis of probability of waiting and deadlock*. 1981.
- [151] P. A. Franaszek, J. T. Robinson, und A. Thomasian, „Concurrency control for high contention environments“, *ACM Transactions on Database Systems (TODS)*, Bd. 17, Nr. 2, S. 304–345, 1992.
- [152] H. Wehr und B. Müller, *Java Persistence API 2: Hibernate, EclipseLink, OpenJPA und Erweiterungen*. Carl Hanser Verlag GmbH & CO. KG, 2012.
- [153] T. Härder, „Observations on optimistic concurrency control schemes“, *Information Systems*, Bd. 9, Nr. 2, S. 111–120, 1984.
- [154] H. T. Kung und J. T. Robinson, „On optimistic methods for concurrency control“, *ACM Transactions on Database Systems (TODS)*, Bd. 6, Nr. 2, S. 213–226, 1981.
- [155] H. Garcia-Molina und K. Salem, *Sagas*, Bd. 16. ACM, 1987.
- [156] E. Rahm, „Optimistische Synchronisationskonzepte in zentralisierten und verteilten Datenbanksystemen“, *Informationstechnik*, Bd. 30, Nr. 1, S. 28–47, 1988.
- [157] A. Troelsen, *Pro C# 2010 and the .NET 4 Platform*. Apress, 2010.
- [158] T. H. Cormen, C. E. Leiserson, R. L. Rivest, und C. Stein, *Introduction to Algorithms*, 3rd edition. Student. Mit Press, 2009.
- [159] T. Parr, *Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages*, 1. Aufl. Pragmatic Programmers, 2010.
- [160] M. Mitzenmacher und E. Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
- [161] S. S. Skiena, *The Algorithm Design Manual*, 2nded. 2008 Aufl. Springer, 2008.
- [162] S. Dasgupta, C. H. Papadimitriou, und U. Vazirani, *Algorithms*. Mcgraw-Hill Higher Education, 2006.
- [163] A. Broder und M. Mitzenmacher, „Network applications of bloom filters: A survey“, *Internet Mathematics*, Bd. 1, Nr. 4, S. 485–509, 2004.



- [164] „Latency: The New Web Performance Bottleneck - igvita.com“. [Online]. Available: <http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/>. [Accessed: 20-Nov-2012].
- [165] A. Pagh, R. Pagh, und S. S. Rao, „An optimal Bloom filter replacement“, in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 2005, S. 823–829.
- [166] F. Putze, P. Sanders, und J. Singler, „Cache-, hash- and space-efficient bloom filters“, *Experimental Algorithms*, S. 108–121, 2007.
- [167] D. Guo, J. Wu, H. Chen, und X. Luo, „Theory and network applications of dynamic bloom filters“, in *Proc. 25th IEEE INFOCOM*, 2006, Bd. 1.
- [168] „SquidFaq/CacheDigests - Squid Web Proxy Wiki“. [Online]. Available: <http://wiki.squid-cache.org/SquidFaq/CacheDigests>. [Accessed: 22-Nov-2012].
- [169] P. O’Neil, E. Cheng, D. Gawlick, und E. O’Neil, „The log-structured merge-tree (LSM-tree)“, *Acta Informatica*, Bd. 33, Nr. 4, S. 351–385, 1996.
- [170] A. Kirsch und M. Mitzenmacher, „Less hashing, same performance: Building a better Bloom filter“, *Algorithms–ESA 2006*, S. 456–467, 2006.
- [171] „Jonathan Ellis’s Programming Blog - Spyced: All you ever wanted to know about writing bloom filters“. [Online]. Available: <http://spyced.blogspot.de/2009/01/all-you-ever-wanted-to-know-about.html>. [Accessed: 22-Nov-2012].
- [172] „Cassandra Bloomfilter Implementierung“. [Online]. Available: <https://github.com/acruise/cassandra-bloom-filter/tree/master/src/main/java/com/facebook/infrastructure/utills>. [Accessed: 22-Nov-2012].
- [173] „HBase Bloomfilter Implementierung“. [Online]. Available: <http://javasourcecode.org/html/open-source/hbase/hbase-0.90.3/org/apache/hadoop/hbase/util/ByteBloomFilter.java.html>. [Accessed: 22-Nov-2012].
- [174] „A Garden Variety of Bloom Filters | Matthias Vallentin“. [Online]. Available: <http://matthias.vallentin.net/blog/2011/06/a-garden-variety-of-bloom-filters/>. [Accessed: 22-Nov-2012].
- [175] F. Deng und D. Rafiei, „Approximately detecting duplicates for streaming data using stable bloom filters“, in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, S. 25–36.
- [176] M. K. Yoon, „Aging bloom filter with two active buffers for dynamic sets“, *Knowledge and Data Engineering, IEEE Transactions on*, Bd. 22, Nr. 1, S. 134–138, 2010.
- [177] „MagnusS/Java-BloomFilter · GitHub“. [Online]. Available: <https://github.com/MagnusS/Java-BloomFilter>. [Accessed: 23-Nov-2012].
- [178] „igrigorik/bloomfilter-rb · GitHub“. [Online]. Available: <https://github.com/igrigorik/bloomfilter-rb>. [Accessed: 23-Nov-2012].
- [179] „Pluto Scarab — Evaluation of CRC32 for Hash Tables“. [Online]. Available: <http://home.comcast.net/~bretm/hash/8.html>. [Accessed: 23-Nov-2012].
- [180] „How fast is Redis? – Redis“. [Online]. Available: <http://redis.io/topics/benchmarks>. [Accessed: 23-Nov-2012].
- [181] „EVAL – Redis“. [Online]. Available: <http://redis.io/commands/eval>. [Accessed: 24-Nov-2012].

- [182] R. Ierusalimschy, *Programming in Lua*, 0002 Aufl. Roberto Ierusalimschy, 2006.
- [183] „xetorthio/jedis · GitHub“. [Online]. Available: <https://github.com/xetorthio/jedis>. [Accessed: 25-Nov-2012].
- [184] D. E. Knuth, *The Art of Computer Programming, Volumes 1-4: 1-4A*, 3rd edition. Addison-Wesley Longman, Amsterdam, 2011.
- [185] M. Datar, N. Immorlica, P. Indyk, und V. S. Mirrokni, „Locality-sensitive hashing scheme based on p-stable distributions“, in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, S. 253–262.
- [186] Y. Lamdan und H. Wolfson, „Geometric hashing: A general and efficient model-based recognition scheme“, 1988.
- [187] J. Bloch, *Effective Java: A Programming Language Guide*, 2nd Revised edition (REV). Addison-Wesley Longman, Amsterdam, 2008.
- [188] „How the Java String hash function works (2)“. [Online]. Available: http://www.javamex.com/tutorials/collections/hash_function_technical_2.shtml. [Accessed: 25-Nov-2012].
- [189] „Collections Framework Enhancements in Java SE 7“. [Online]. Available: <http://docs.oracle.com/javase/7/docs/technotes/guides/collections/changes7.html>. [Accessed: 25-Nov-2012].
- [190] C. M. Bishop, *Pattern Recognition and Machine Learning*, 1st ed. 2006. Corr. 2nd printing. Springer New York, 2007.
- [191] B. Jenkins, „A Hash Function for Hash Table Lookup“. [Online]. Available: <http://www.burtleburtle.net/bob/hash/doobs.html>. [Accessed: 25-Nov-2012].
- [192] A. Webster und S. Tavares, „On the design of S-boxes“, in *Advances in Cryptology—CRYPTO’85 Proceedings*, 1986, S. 523–534.
- [193] C. Henke, C. Schmoll, und T. Zseby, „Empirical evaluation of hash functions for multipoint measurements“, *ACM SIGCOMM Computer Communication Review*, Bd. 38, Nr. 3, S. 39–50, 2008.
- [194] „statistics - murmurhash“. [Online]. Available: <https://sites.google.com/site/murmurhash/statistics>. [Accessed: 25-Nov-2012].
- [195] M. Shapiro, N. Pregui\cca, C. Baquero, und M. Zawirski, „A comprehensive study of convergent and commutative replicated data types“, 2011.
- [196] S. Gilbert und N. Lynch, „Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services“, *ACM SIGACT News*, Bd. 33, Nr. 2, S. 51–59, 2002.
- [197] L. Hedstrom, „Apache Traffic Server, HTTP Proxy Server on the Edge“. .
- [198] „SquidFaq/AboutSquid - Squid Web Proxy Wiki“. [Online]. Available: <http://wiki.squid-cache.org/SquidFaq/AboutSquid>. [Accessed: 28-Nov-2012].
- [199] P.-H. Kamp, „Varnish http accelerator - A 2006 software design“, 2006.
- [200] „VCL — Varnish version 3.0.3 documentation“. [Online]. Available: <https://www.varnish-cache.org/docs/3.0/reference/vcl.html>. [Accessed: 28-Nov-2012].
- [201] A. Cerpa, J. Elson, H. Beheshti, A. Chankhunthod, P. Danzig, R. Jalan, C. Neerdaels, und T. Shroeder, „G. Tomlinson,“ NECP: The Network Element Control Protocol“, *Work in Progress*, 2000.



- [202] M. Cieslak, D. Forster, G. Tiwana, und R. Wilson, „Web Cache Coordination Protocol (WCCP) V2“, 0, Internet-Draft, draft-wilsonwrec-wccp-v2-00. txt, 2000.
- [203] Akamai, „Akamai“. [Online]. Available: http://www.akamai.de/technology_index.html. [Accessed: 28-Nov-2012].
- [204] E. Nygren, R. K. Sitaraman, und J. Sun, „The Akamai Network: A platform for high-performance Internet applications“, *ACM SIGOPS Operating Systems Review*, Bd. 44, Nr. 3, S. 2–19, 2010.
- [205] Amazon Web Services, „Introduction to Amazon CloudFront - Amazon CloudFront“. [Online]. Available: <http://docs.amazonwebservices.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html>. [Accessed: 28-Nov-2012].
- [206] Amazon Web Services, „Amazon CloudFront Pricing“. [Online]. Available: <http://aws.amazon.com/de/cloudfront/pricing/>. [Accessed: 28-Nov-2012].
- [207] Apache Software Foundation, „Apache Traffic Server Administrator’s Guide“. [Online]. Available: <http://trafficserver.apache.org/docs/v2/admin/>. [Accessed: 28-Nov-2012].
- [208] M. Belshe und R. Peon, „SPDY Protocol“, 2012.
- [209] Google Developers, „Developer Guide - Protocol Buffers“. [Online]. Available: <https://developers.google.com/protocol-buffers/docs/overview>. [Accessed: 28-Nov-2012].
- [210] D. Abadi, „Consistency tradeoffs in modern distributed database system design: CAP is only part of the story“, *Computer*, Bd. 45, Nr. 2, S. 37–42, 2012.
- [211] E. Brewer, „Pushing the CAP: Strategies for consistency and availability“, *Computer*, Bd. 45, Nr. 2, S. 23–29, 2012.
- [212] S. S. Y. Shim, „The CAP Theorem’s Growing Impact“, *Computer*, S. 21–22, 2012.
- [213] „Facebook News Feed: Social Data at Scale“. [Online]. Available: <http://www.infoq.com/presentations/Facebook-News-Feed>. [Accessed: 27-Nov-2012].
- [214] Netscape Communications, „NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET“. [Online]. Available: <http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html>. [Accessed: 28-Nov-2012].
- [215] J. J. Garrett, „Ajax: A new approach to web applications“, 2005.
- [216] S. Ecma, *ECMA-262 ECMAScript Language Specification*. 2009.
- [217] Oracle Inc., „Oracle Fusion Middleware Developer’s Guide for Oracle TopLink 11g Release 1 (11.1.1)“. [Online]. Available: http://docs.oracle.com/cd/E14571_01/web.1111/b32441/toc.htm. [Accessed: 28-Nov-2012].
- [218] C. Bauer und G. King, „Hibernate in action“, 2005.
- [219] Eclipse Foundation, „EclipseLink JPA“. [Online]. Available: <http://www.eclipse.org/eclipselink/jpa.php>. [Accessed: 28-Nov-2012].
- [220] F. Marchioni und M. Surtani, *Infinispan Data Grid Platform*. Packt Publishing, Limited, 2012.

- [221] Terracotta, Inc, „Ehcache“. [Online]. Available: <http://ehcache.org/>. [Accessed: 28-Nov-2012].
- [222] „A Distributed Systems Conference for Developers - RICON“. [Online]. Available: <http://basho.com/community/ricon2012/>. [Accessed: 26-Nov-2012].



C Arbeitsaufteilung

Die Verfassung der vorliegenden, schriftlichen Ausarbeitung haben sich die Autoren wie folgt aufgeteilt:

	Florian Bücklers	Felix Gessert
Kapitel 1 Einführung		alles
Kapitel 2 Related Work		alles
Kapitel 3 Kohärentes Web-Caching		alles
Kapitel 4 Cache Invalidierungen	alles	
Kapitel 5 Polyglot Persistence	5.2	5.1
Kapitel 6 Future Work	zusammen	zusammen
Kapitel 7 Schluss	zusammen	zusammen

D Erklärung

Hiermit erkläre ich, dass ich, Felix Gessert, abgesehen von der gemeinsamen Planung und Besprechung der Inhalte, meinen Teil der Arbeit selbstständig und ohne fremde Hilfe angefertigt, sowie mich anderer als der im Literatur- und Quellenverzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Bachelorarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Hamburg, den

[Felix Gessert]

Hiermit erkläre ich, dass ich, Florian Bücklers, abgesehen von der gemeinsamen Planung und Besprechung der Inhalte, meinen Teil der Arbeit selbstständig und ohne fremde Hilfe angefertigt, sowie mich anderer als der im Literatur- und Quellenverzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Bachelorarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Hamburg, den

[Florian Bücklers]

