

Mehrbereichsabfragen in verteilten Hash-Tabellen II auf Basis von CAN

Benjamin Schnaidt
Seminar Innovative Internet-Technologien, WS 2004/05
Lehrstuhl Rechnernetze und Internet
Universität Tübingen
Benjamin@Schnaidt.org

KURZFASSUNG

Aktuelle Peer-to-Peer Systeme wie CAN realisieren eine verteilte Hash-Tabelle. Diese Tabelle besteht aus (*Key*, *Value*) Paaren und ermöglicht das Nachschlagen von jeweils einem *Value* zu einem *Key*. Während CAN sehr gut skalierbar ist, sich robust gegenüber einzelnen Ausfällen zeigt und das Nachschlagen von *Keys* sehr effizient möglich ist, so sind keine Mehrbereichsabfragen oder sonstige komplexe Abfragen möglich. In dieser Ausarbeitung werden zwei Erweiterungen vorgestellt, welche Mehrbereichsabfragen ermöglichen und gleichzeitig die genannten Vorteile von CAN weiterhin erhalten.

Die erste Technik verwendet so genannte *Hilbertfunktionen*, mit denen sich Abfragen nach ganzen Intervallen von *Attributwerten* realisieren lassen. Außerdem wird das Problem von regelmäßigen Updates von *Attributwerten* betrachtet und mit Hilfe von einem *Expressway* und Caches gelöst. Diese ermöglichen in den meisten Fällen den Versand von Updates direkt per IP-Routing, welches schneller als das CAN-Routing ist.

Bei der zweiten Technik werden drei verschiedene Arten der Suche unterstützt: (1) Die Suche nach einzelnen Wörtern, (2) Die Suche nach einem Bereich von *Attributwerten* mit Hilfe von *Bereichsuchbäumen*, (3) Die Suche nach dem *längsten passenden Präfix* zu einem gegebenen Ausdruck mit der *binären Suche*. Außerdem ermöglichen *Bloom-Filter* eine effiziente Repräsentation von Ergebnissen einzelner *Unterabfragen*. Hiermit können diese Ergebnisse über Mengenoperationen zum Endergebnis kombiniert werden, ohne viele unnötige Informationen über das Netzwerk zu übertragen.

Die erste Technik richtet sich speziell an der Verwaltung von *Computational Grids*, während die zweite Technik am *File Sharing* orientiert ist.

SCHLÜSSELWORTE

Distributed hash tables, CAN, range queries, hilbert function, computational grids, range search trees, longest prefix matching

1. EINFÜHRUNG

CAN ist ein System zur Verwaltung einer Tabelle aus (*Key*, *Value*) Paaren, deren einzelne Einträge über ein großes Netzwerk von Rechnern verteilt sind. Abschnitt 2 behandelt die Grundlagen von CAN, wie das Einfügen von neuen (*Key*, *Value*) Paaren in diese Tabelle und das Nachschlagen eines *Keys* zu einem *Value*.

In Abschnitt 3 folgt eine Einführung zur Problematik der Mehrbereichsabfragen, die von CAN nicht ohne Weiteres unterstützt werden. Die Abschnitte 4 und 5 beschreiben zwei Techniken, die CAN um Mehrbereichsabfragen erweitern und auch andere komplexere Abfragetypen zulassen. Im letzten Abschnitt 6 folgt ein Vergleich bzw. eine Zusammenfassung dieser beiden Techniken.

2. CAN

CAN (*Content-Addressable Network*) [1] basiert auf einem d-dimensionalen Raum. Im einfachsten Fall ist dieser Raum eine 2-dimensionale Fläche, wie sie in **Abbildung 1** dargestellt wird.



Abbildung 1: 2-dimensionales CAN mit sechs Knoten

Jeder Knoten (Rechner) im Netzwerk erhält einen *Bereich*¹ (engl. *Zone*) des Raumes. Im Beispiel wird der Raum $[0,1] \times [0,1]$ zwischen sechs Knoten aufgeteilt und jeder Knoten deckt als *Bereich* ein Rechteck in diesem Raum ab - im d-dimensionalen Fall wird dies *Hyperquader* genannt. In der Mitte der Rechtecke in **Abbildung 1** ist kreisförmig jeweils der zugehörige Knoten eingezeichnet. Im realen Fall kann sich dieser an einem beliebigen Punkt im *Bereich* befinden. Die *Bereiche* sind dabei allerdings immer so klein, dass ein *Bereich* nur genau einen Knoten enthält, wie dies auch im Beispiel sichergestellt ist.



Abbildung 2: 2D CAN als Torus dargestellt

Koordinaten außerhalb vom Intervall $[0;1]$ werden ringförmig zurück auf $[0;1]$ abgebildet. Beispielsweise wird 1,2 zu 0,2. Dies gilt für jede einzelne Dimension des Raumes. Ein 2-dimensionales CAN lässt sich daher auch als Torus darstellen, bei dem jeweils die

¹ Wichtige Schlüsselwörter der Verfahren werden *kursiv* dargestellt.

gegenüberliegenden Seiten miteinander ringförmig verbunden wurden (**Abbildung 2**). Zur Veranschaulichung wird im Folgenden aber weiterhin die Darstellung als einfache Fläche verwendet.

Die zentrale Aufgabe eines CAN ist die Verwaltung einer so genannten Hash-Tabelle. Eine Hash-Tabelle ist im Grunde eine normale zweispaltige Tabelle, bei der in jeder Zeile ein *Key* (dt. Schlüssel) auf einen *Value* (dt. Wert) abgebildet wird. Die Tabelle wird dabei aber nicht auf einem zentralen Server gespeichert, sondern mit Hilfe des CAN über alle Knoten verteilt. Der *Key* dient dabei zur "Adressierung" der passenden Tabellenzeile, woraus sich die Bezeichnung *Content-Addressable Network* (dt. inhaltsadressiertes Netzwerk) ableitet.

Die drei typischen Aufgaben des CAN sind: (1) Das Einfügen von (*Key*, *Value*) Paaren, (2) Das Nachschlagen eines *Value*s zu einem *Key*, (3) Das Löschen von Paaren. Oft werden diese mit *Insert*, *Lookup*, *Delete* bezeichnet.

2.1 Einfügen (Insert)

Um ein (*Key*, *Value*) Paar in die Tabelle bzw. das CAN einzufügen, wird ihm ein fester Punkt im Raum zugewiesen. Dazu wird mit einer Hash-Funktion der *Key* auf einen *Punkt* abgebildet. Da der gesamte Raum von den Knoten des Netzwerks abgedeckt ist, gibt es damit einen zugehörigen Knoten, in dessen *Bereich* dieser *Punkt* liegt. **Abbildung 3** zeigt hierfür ein Beispiel, bei dem der zufällig ausgewählte *Punkt* im *Bereich* des Knotens rechts oben liegt. Dieser Knoten speichert nun das (*Key*, *Value*) Paar. Im Grunde ist er also für einen Teilausschnitt der Hash-Tabelle zuständig.

Damit die *Punkte* gleichmäßig über den gesamten Raum verteilt werden, wird eine uniforme Hash-Funktion verwendet. Dadurch soll vermieden werden, dass ein einzelner Knoten sehr viele (*Key*, *Value*) Paare verwalten muss.

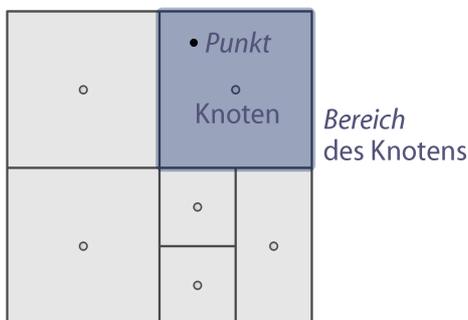


Abbildung 3: Ein Punkt im Bereich des markierten Knotens

2.2 Nachschlagen (Lookup)

Die nächste Aufgabe ist das Nachschlagen eines *Value* zu einem gegebenen *Key*. Dazu wendet der Knoten, der die Anfrage stellen möchte, zuerst die Hash-Funktion auf den *Key* an und erhält damit den passenden *Punkt*. Nun wird eine Nachricht geschickt, die diesen *Punkt* als Zieladresse enthält. Der Knoten, dessen *Bereich* diesen *Punkt* abdeckt, soll die Nachricht schließlich empfangen.

Um wie in einem normalen Netzwerk ein Routing zu ermöglichen, speichert jeder Knoten die IP-Adressen seiner Nachbarn im CAN Raum (**Abbildung 4**). Physikalisch dagegen kann der Knoten von seinen Nachbarn weit entfernt sein. Daher wird hier, im Gegensatz zum physikalischen Netzwerk, von einem *Overlay Netzwerk* gesprochen.

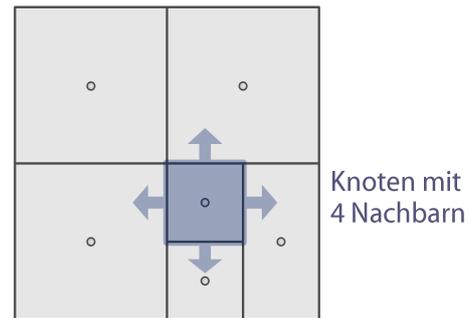


Abbildung 4: Ein Knoten mit vier Nachbarn

Im allgemeinen d -dimensionalen Fall sind dabei zwei Knoten bzw. zwei *Bereiche* benachbart, wenn Sie eine $(d-1)$ -dimensionale Hyperebene gemeinsam haben.

Das Routing folgt dann einem sehr einfachen Prinzip: Die Nachricht wird jeweils an den Knoten weitergeleitet, der dem Zielpunkt am nächsten liegt. Dadurch wird prinzipiell eine gerade Linie vom Start- zum Zielknoten angenähert. **Abbildung 5** enthält ein Beispiel für solch eine Route (Durch den ringförmigen Aufbau des CAN würde hier eigentlich eine Route gewählt, die über den Rand hinweg verläuft - die in der Abbildung gezeigte Fläche soll aber nur einen Teilausschnitt eines deutlich größeren CANs darstellen).

Schließlich erhält der Zielknoten die Nachricht und kann mit dem passenden *Value* zum *Key* antworten (mit einer direkten Nachricht an die IP-Adresse des Startknotens).

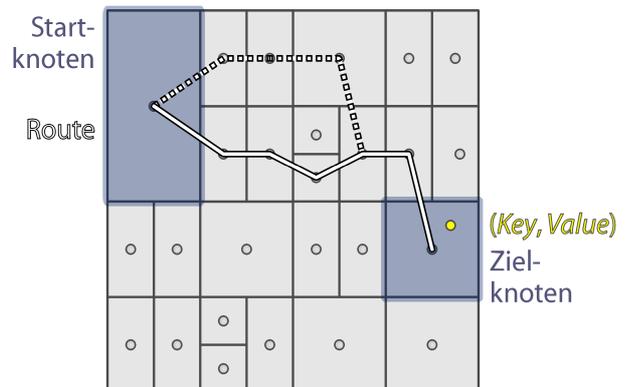


Abbildung 5: Routing vom Start- zum Zielknoten

Diese Methode ist dabei nicht nur einfach, sondern sorgt auch gleichzeitig für die Sicherheit des Netzwerks: Kann der Startknoten den Nachbarn nicht erreichen, so kann er die Nachricht an einen anderen geeigneten Nachbarn weitergeben, da letztendlich viele verschiedene Pfade zum Zielknoten existieren - wie die gestrichelte Route in **Abbildung 5**.

2.3 Kosten

CAN kann als Grundlage für viele verschiedene Dienste dienen. Besonders geeignet ist es für *Peer-to-Peer File Sharing* (dt. Tauschbörsen), wo oft eine sehr große Anzahl an Rechnern im Netzwerk aktiv sind und dementsprechend auch viele (*Key*, *Value*) Paare eingefügt und nachgeschlagen werden. Besonders wichtig ist daher die Skalierbarkeit des Systems. CAN hat folgende Eigenschaften (Dabei wird von einer völlig gleichmäßigen Verteilung der *Bereiche* ausgegangen):

Speicheraufwand: Bei d Dimensionen müssen pro Knoten die IP-Adressen von $2d$ Nachbarn abrufbar sein. Damit ergibt sich ein *Speicheraufwand* von $O(d)$.

Suchaufwand: Das oben beschriebene Routing führt zu einer durchschnittlichen Pfadlänge von $(d/4) \cdot n^{1/d} = O(n^{1/d})$ vom Start- zum Zielknoten, wenn sich n Knoten im Netzwerk befinden.

Allgemein werden Systeme, die wie *CAN* eine über das Netzwerk verteilte Hash-Tabelle realisieren, als *DHTs (Distributed Hash Tables)* bezeichnet. Viele *DHTs* haben einen *Speicher-* und gleichzeitig *Suchaufwand* von $O(\log n)$, beispielsweise *Chord*. Wird $d = (\log_2 n)/2$ gesetzt, so kann auch *CAN* einen *Speicher-* und *Suchaufwand* von $O(\log n)$ erreichen. Allerdings wird empfohlen, d auf einen konstanten Wert zu setzen, so dass die Anzahl der Nachbarn von der Anzahl der Knoten unabhängig ist.

2.4 Erweiterungen

Eine Reihe von Erweiterungen können das Routing beschleunigen und gleichzeitig für eine größere Ausfallsicherheit im Netzwerk sorgen (sowohl bei zufälligen Ausfällen als auch bei Attacken).

In [1] werden insgesamt acht verschiedene Erweiterungen diskutiert und außerdem in Simulationen getestet. Fast alle Erweiterungen erhöhen zwar den *Speicheraufwand* pro Knoten, sorgen dafür aber für einen deutlich reduzierten *Suchaufwand*. Eine Simulation mit $n = 260\,000$ ergab eine Latenzzeit beim Routing, die weniger als doppelt so hoch wie die Latenzzeit des darunter liegenden IP-Netzwerkes war.

Da die Erweiterungen aber die grundlegende Funktionsweise des *CAN* nicht verändern und die Mehrbereichsabfragen auch unabhängig davon sind, wird hier nur beispielhaft eine dieser Erweiterungen aufgeführt:

- **Mehr Dimensionen:** Eine Erhöhung der Dimensionalität (**Abbildung 6**) verringert die durchschnittliche Pfadlänge bzw. den *Suchaufwand*. Außerdem stehen im Fehlerfall durch die größere Anzahl von Nachbarn mehr alternative Pfade zur Verfügung, was die Sicherheit erhöht.

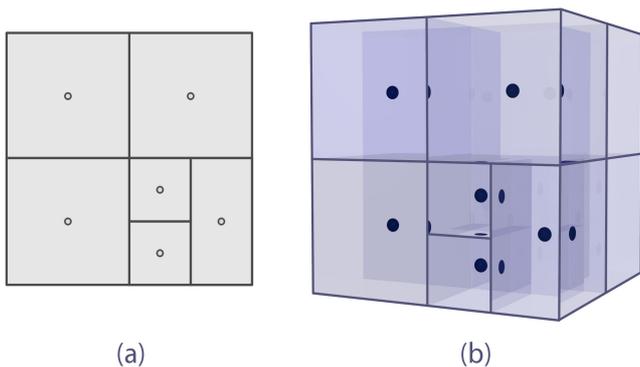


Abbildung 6: *CAN* mit (a) zwei und (b) drei Dimensionen

3. MEHRBEREICHSABFRAGEN

Wie bereits erwähnt wurde, ist das *File Sharing* eine typische Anwendung von *DHTs* wie *CAN*. Hiermit lässt sich auch leicht die Problematik bei Mehrbereichsabfragen demonstrieren.

Bisher wurde *Hashing* an zwei Stellen angewendet: (1) *CAN* selbst realisierte eine verteilte Hash-Tabelle, die (*Key*, *Value*) Paare sammelt. (2) Ein *Key* wird mit einer Hash-Funktion auf einen *Punkt* im Raum abgebildet.

Beim *File Sharing* kommt nun eine dritte Anwendung hinzu: Eine Hash-Funktion bildet zuerst den *Dateinamen* auf einen *Key* ab. In der Hash-Tabelle können dann Paare (*Key*, *IP-Adresse*) gespeichert werden, wobei die *IP-Adresse* den Speicherort der Datei angibt².

Beispielsweise könnte der *Dateiname* *Opera723.exe* auf den *Key* *1342* abgebildet werden. (*1342*, "*217.81.152.218*") wird als Paar in die Hash-Tabelle aufgenommen, falls der Rechner mit der gegebenen *IP-Adresse* die Datei gespeichert hat.

Bereichsabfragen suchen statt nach einem speziellen *Dateinamen* nach einem ganzen Reihe an *Dateinamen*, die einem bestimmten Muster entsprechen - zum Beispiel alle verfügbaren Versionen vom Opera Web Browser 7.x im Netzwerk (**Abbildung 7**). Da *CAN* aber nur die Suche nach einem bestimmten *Key* zulässt, kann solch eine Abfrage nicht ohne Weiteres realisiert werden.

Eine Suche nach *Opera7?? .exe* ist grundsätzlich noch möglich. *?* steht für ein einzelnes beliebiges Zeichen und kann in diesem Fall auf Zahlen *0-9* eingeschränkt werden. Dadurch ergeben sich 100 verschiedene *Dateinamen* und die entsprechenden *Keys* dazu können im Netzwerk nachgeschlagen werden. Während dies bereits nicht sehr effizient ist, sind Abfragen wie *Opera7** dagegen nicht mehr möglich. *** sucht nach einem beliebigen String, von denen es unendlich viele gibt.

Abbildung 7 zeigt, dass die *Keys* zu den entsprechenden *Dateinamen* über das ganze Netzwerk verstreut sein können. Während dies bisher ein Vorteil war, da sich durch das uniforme Hashing die Last über das gesamte *CAN* verteilt, wird dies bei Bereichsabfragen zum Problem.

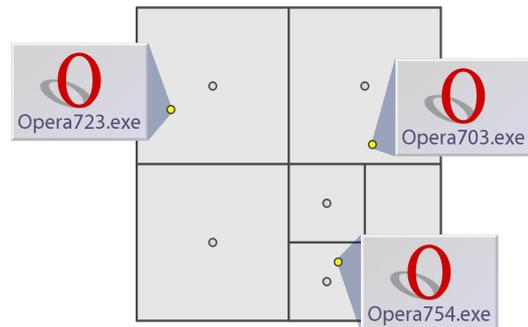


Abbildung 7: Verschiedene Versionen von Opera 7 im *CAN*

Im Folgenden werden zwei verschiedene Ansätze besprochen, die *CAN* um solche (Mehr-)Bereichsabfragen erweitern und dabei auch dieses Problem näher betrachten.

4. BEREICHSABFRAGEN MIT HILBERTFUNKTIONEN

Die erste Technik [2] hat statt dem *File Sharing* so genannte *Computational Grids* im zentralen Blickpunkt. Diese verbinden eine große Anzahl von Servern (oder auch normalen PCs), um deren Rechenleistung oder Festplattenkapazität zu teilen. Auf diese Weise können auch ohne teure Supercomputer beträchtliche Rechenleistungen erzielt werden.

Prinzipiell können so Tausende oder sogar Millionen von Servern in vielen einzelnen lokalen oder regionalen Netzwerken verbunden werden, die wiederum global miteinander verknüpft sind.

² *Dateiname* und *IP-Adresse* dienen hier lediglich als anschauliches Beispiel, es sind viele andere Kodierungen möglich.

Solch eine große Anzahl von Servern gehört dabei normalerweise nicht einer einzelnen Person oder Organisation und kann außerdem geographisch weit verstreut sein. Um die Verfügbarkeit von Ressourcen zu verwalten, wird daher ein *Informationsdienst* (engl. *Grid Information Service*) benötigt, der Informationen sammelt und gleichzeitig eine Suche nach bestimmten Ressourcen ermöglicht.

Ein einzelner Server wird dabei normalerweise durch eine Reihe von *Attributen* beschrieben. Typische *Attribute* sind die Architektur (Prozessor), das Betriebssystem, die aktuelle CPU-Auslastung und der aktuell freie Speicher (RAM). **Abbildung 8** ist ein Beispiel für ein Netzwerk mit insgesamt neun Servern, wobei die *Attributwerte* von zwei Servern aufgelistet sind.

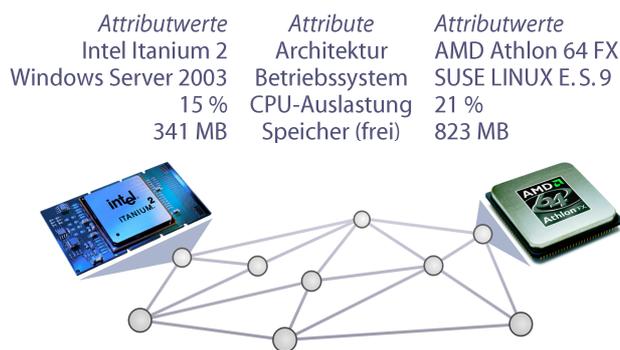


Abbildung 8: Zwei Server eines *Computational Grids*

Ein typischer Informationsdienst für die Verwaltung von solchen *Attributen* ist *MDS* (*Monitoring and Discovery Service*) [3]. *MDS* besteht dabei aus zwei wesentlichen Komponenten: Ein Dienst zum Bereitstellen von Informationen und ein Dienst zum Sammeln dieser Informationen. Die gesammelten Informationen können dabei auf einem zentralen Server oder einer Hierarchie von mehreren Servern gespeichert werden. Beide Möglichkeiten haben aber eine beschränkte Skalierbarkeit [4].

Die wichtige zweite Komponente zum Sammeln der Informationen wird auch als *Indexdienst* (engl. *Grid Index Information Service*) bezeichnet. Denn wie bei Datenbanken wird aus den Informationen ein Index erstellt, der die schnelle Suche auf einer großen Menge an Daten ermöglicht.

4.1 CAN als Indexdienst

Auch *DHTs* wie *CAN* sind in der Lage, einen solchen *Indexdienst* bereitzustellen. Das Sammeln von Informationen entspricht dem Einfügen von *Keys* und das Suchen dem Nachschlagen von *Keys*.

Die Vorteile solcher Peer-to-Peer Systeme gegenüber *MDS* sind dabei:

- **Selbstorganisierendes Netzwerk:** Es werden keinerlei zentrale Server benötigt (Abgesehen von eventuellen *CAN Bootstrap* Knoten, die neu hinzukommenden Servern einen schnellen Einstieg ins bestehende *CAN* ermöglichen).
- **Robust gegenüber Fehlern:** Fallen einzelne Server aus oder werden einzelne Server attackiert, so ist der Schaden lokal begrenzt und kann durch benachbarte Server kompensiert werden. Außerdem ist *CAN* *symmetrisch* bezogen auf das Nachschlagen von *Keys*, das heißt, alle Knoten sind grundsätzlich bei der Suche gleich wichtig [5]. Dadurch ist ein gezielter Angriff zentraler Knotenpunkte nicht möglich.

- **Sehr gute Skalierbarkeit:** Wie bereits im Abschnitt 2.3 gezeigt, ermöglicht *CAN* die Verwaltung von Millionen von beteiligten Servern und vermeidet dabei das Entstehen von Flaschenhälsen.

Gegenüber anderen bekannten P2P Systemen wie *Gnutella* oder *Freenet* hat *CAN* außerdem den Vorteil, dass ein bestimmtes (*Key*, *Value*) Paar auf jeden Fall gefunden wird, wenn es im Netzwerk vorhanden ist. Dies wird auch als *strukturiertes* oder *deterministisches* Nachschlagen bezeichnet [5].

Auf der anderen Seite bestehen aber auch zwei Probleme bzw. Nachteile:

- **Keine Mehrbereichsabfragen**
- **Schnell ändernde Attribute:** *CAN* ist vor allem auf das *File Sharing* ausgerichtet, bei dem sich nur selten ein *Dateiname* bzw. ein *Key* einer Datei ändert. In *Computational Grids* ändern sich dagegen *Attribute* wie die CPU-Auslastung ständig. Updates nach mehreren Stunden oder Tagen sind hier nicht ausreichend, aber häufige Updates wurden im Design von *CAN* bisher nicht berücksichtigt.

Beide Probleme können effizient gelöst werden. Zuerst wird aber die allgemeine Vorgehensweise bei einer Abfrage betrachtet.

Im Gegensatz zum *Nachschlagen* eines einzelnen *Keys* wird der Begriff *Abfrage* im Datenbankbereich für eine Auswahl mit komplexeren Suchkriterien verwendet. Außerdem kann in einer Abfrage nach einer Kombination von mehreren *Attributen* gesucht werden. **Abbildung 9** enthält ein Beispiel für solch eine Abfrage.

Es wird nun davon ausgegangen, dass jedes einzelne *Attribut* mit einer eigenen Indexstruktur (sprich einem eigenen *CAN*) verwaltet wird. Außerdem soll der Datentyp jedes *Attributs* global bekannt sein - beispielsweise *Enumeration*, *Integer*, *Float*, *String* (dt. Aufzählung, Ganzzahl, Gleitkommzahl, Zeichenkette).

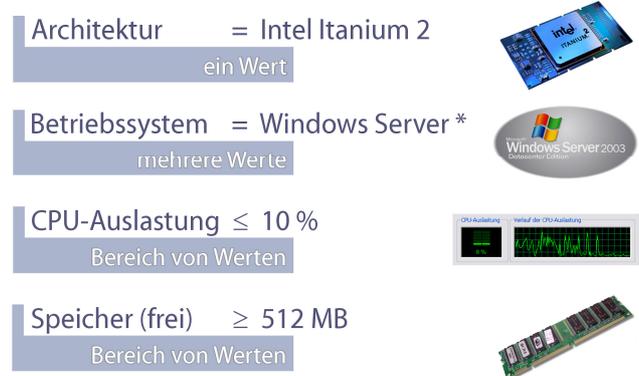


Abbildung 9: Beispiel für eine Abfrage

Das erste *Attribut* **Architektur** ist dabei vom Typ *Enumeration* und deckt alle bekannten Architektur- bzw. Prozessortypen ab. Die Suche nach **Intel Itanium 2** ist problemlos von einem normalen *CAN* durchführbar. Auch das zweite *Attribut* **Betriebssystem** ist vom Typ *Enumeration*, wodurch selbst das Suchmuster **Windows Server *** für ein normales *CAN* kein Problem darstellt. Es werden alle passenden Betriebssystemsnamen einzeln nachgeschlagen, beispielsweise **Windows Server 2003**.

Die *Attribute* **CPU-Auslastung** und **Speicher** sind dagegen vom Typ *Float* oder *Integer*. Die Suche nach einem ganzen Bereich von Werten wie $\leq 10 \%$ ist hier entweder gar nicht oder nicht effizient möglich. Im folgenden Abschnitt wird daher ein um Bereichsabfragen erweitertes *CAN* definiert.

Da jedes *Attribut* eine getrennte Indexstruktur hat, werden nun Abfragen nur noch für einzelne *Attribute* betrachtet. Abfragen mit mehreren *Attributen* werden erst einzeln ausgeführt und dann mit einem so genannten *Join* verbunden (Dies ist der bei Datenbanken übliche Begriffe zur Verknüpfung der Abfrageergebnisse mehrerer *Attribute*).

4.2 Die Hilbertfunktion

Neben der Einschränkung auf ein einzelnes *Attribut* wird nun zusätzlich davon ausgegangen, dass das *Attribut* auf dem *Intervall* $[0;1]$ definiert ist. Für die Datentypen *Float* und *Integer* bedeutet dies keine Einschränkung. *Strings* werden hierbei nicht unterstützt, wobei aber eine begrenzte Implementierung mit Hilfe der *arithmetischen Kodierung* denkbar wäre (Diese konvertiert *Strings* in Gleitkommazahlen zwischen 0 und 1 [6]).

CAN speichert nun (*Key, Value*) Paare in der Form (*Attributwert, IP-Adresse*). Ein typisches Attribut ist die CPU-Auslastung in Prozent: ("0,15", "217.81.152.218") ist gleichbedeutend zu 15 % CPU-Auslastung auf dem Server mit der gegebenen IP.

Wie auch bisher ist dabei jeder Server im Netzwerk für einen Teil der Paare der Hash-Tabelle verantwortlich. Während dieser Teil bisher aber nur durch den rechteckigen *Bereich* im CAN bestimmt wurde, wird nun außerdem ein *Teilintervall* von $[0;1]$ definiert, für das der Server zuständig ist.

In **Abbildung 10** ist ein Ausschnitt der gesamten Hash-Tabelle für den Server dargestellt, welcher das *Teilintervall* $[0,25;0,5]$ verwaltet. Er speichert alle Paare, deren *Attributwerte* in dieses Intervall fallen (In [2] wird er daher als *Interval Keeper* bezeichnet). Andere Server mit einer CPU-Auslastung zwischen 25 % und 50 % benachrichtigen diesen Server mit ihrem aktuellen Wert.

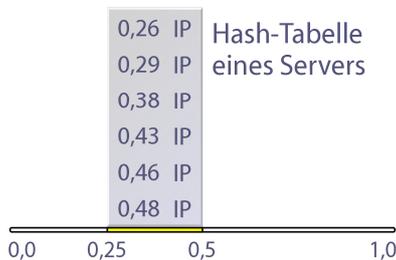


Abbildung 10: Hash-Tabelle eines einzelnen Servers

Da die in Abschnitt 1 definierten Prinzipien von CAN weiterhin gelten müssen, ist die Abbildung dieser *Teilintervalle* auf *Bereiche* im CAN entscheidend für die Bereichsabfragen. Hierfür wird die raumfüllende *Hilbertfunktion* (engl. *Space-Filling Curve*) verwendet.

Die *Hilbertfunktion* ist rekursiv definiert und beginnt mit einem Punkt bei *Level 1*. **Abbildung 11** zeigt *Level 2* und *3*. Dabei entspricht das *Intervall* $[0;1]$ der weißen Linie und am Rand sind die zugehörigen *Teilintervalle* des *Intervalls* angegeben.

Grundidee ist hierbei, dass das 1-dimensionale *Intervall* $[0;1]$ so auf das 2-dimensionale CAN abgebildet wird, dass alle *Bereiche* gleichmäßig abgedeckt werden. Bei *Level 2* entsprechen den vier *Bereichen* die vier gleich großen *Teilintervalle* $[0,0;0,25]$, $[0,25;0,5]$, $[0,5;0,75]$ und $[0,75;1,0]$. Dasselbe gilt für die d-dimensionalen Erweiterungen der *Hilbertfunktion*.

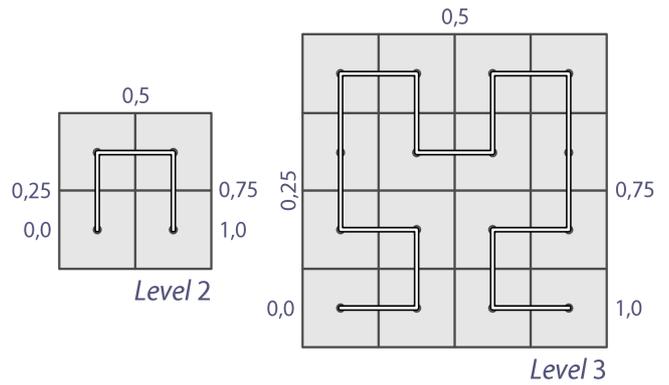


Abbildung 11: Level 2 und 3 der Hilbertfunktion

Weiterhin ist entscheidend, dass bei einer Aufteilung eines *Teilintervalls* in zum Beispiel zwei *Teilintervalle* sich auch der zugehörige *Bereich* in zwei *Bereiche* aufteilen lässt. Ansonsten wäre keine Anpassung möglich, wenn neue Server hinzugefügt werden. Auch dies ist durch die *Hilbertfunktion* gewährleistet. In **Abbildung 11** wird jeder *Bereich* in vier gleich große neue *Bereiche* aufgeteilt und dasselbe gilt für jedes *Teilintervall*. Wichtig dabei anzumerken ist, dass im Gegensatz zum *File Sharing* nicht alle Rechner im *Computational Grid* auch gleichzeitig einen *Bereich* im CAN einnehmen. Nur ein Teil der Server des *Computational Grids* ist für den *Indexdienst* nötig. Eine Aufteilung wie oben muss daher nicht sofort erfolgen, wenn ein Server dem *Computational Grid* beiträgt.

Wie bereits erwähnt, ist die *Hilbertfunktion* rekursiv definiert und folgt dabei dem in **Abbildung 12** definierten Schema. Das Muster "P" von *Level 2* wird in *Level 3* viermal wieder verwendet. Statt 4 *Bereichen* ergeben sich somit 16 *Bereiche* und ebenfalls 16 *Teilintervalle*. Das Muster "P" wird dabei gedreht und an den markierten Punkten werden benachbarte Muster verbunden (Diese geben den Anfang und das Ende des *Intervalls* $[0;1]$ bei *Level 2* an). Allgemein folgt der Schritt von *Level l* zu *Level l+1* genau demselben Prinzip.

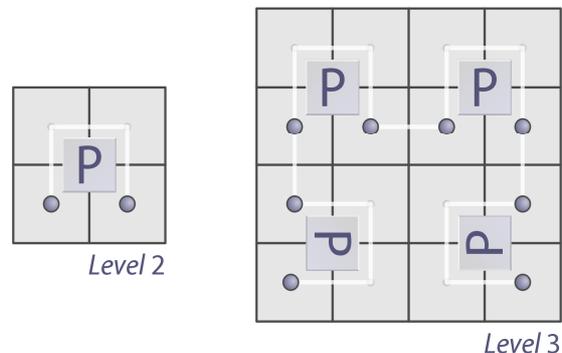


Abbildung 12: Konstruktion der Hilbertfunktion

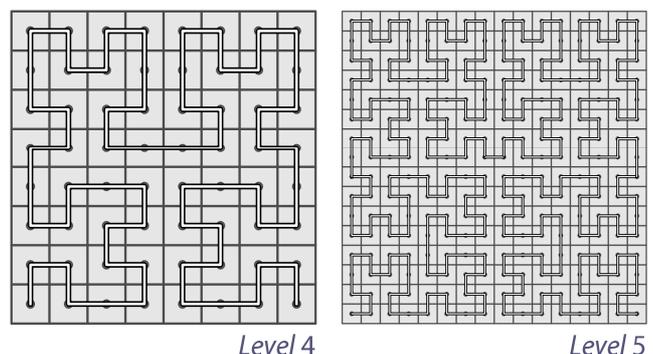


Abbildung 13: Level 4 und 5 der Hilbertfunktion

Beispielsweise wird beim Schritt von *Level 3* zu *4* wiederum *Level 3* als Muster "P" verwendet und viermal eingesetzt. Das endgültige Ergebnis für *Level 4* und *5* zeigt **Abbildung 13**.

Es wird nun davon ausgegangen, dass die Dimension *d* des CAN und das *Level l* der *Hilbertfunktion* global bekannt sind. Ist eine Bereichsabfrage vorgegeben, wird zuerst das Rechteck (genauer der *Hyperquader*) zum *Intervall der Abfrage* berechnet. Für beispielsweise $[0,375;0,75]$ ergibt sich in **Abbildung 14** der dunkel markierte Bereich. Dann wird eine Nachricht an alle diese Knoten verschickt, welche dann mit den IP-Adressen der Server antworten, deren CPU-Auslastung zwischen 37,5 % und 75 % liegt.

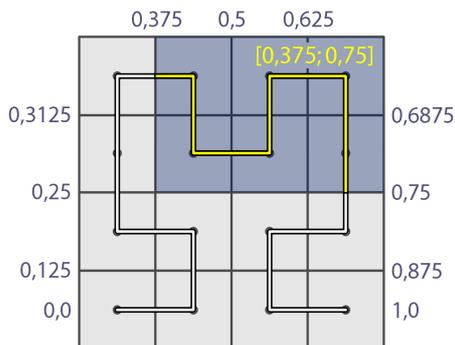


Abbildung 14: Bereichsabfrage $[0,375;0,75]$

Im allgemeinen Fall muss das *Intervall der Abfrage* aber nicht exakt den *Bereichen* im CAN entsprechen. Dies folgt detaillierter im Abschnitt 4.3.

Auch wenn die *Teilintervalle* uniform bzw. gleichmäßig im CAN Raum verteilt sind, werden ähnliche *Attributwerte* nun nicht mehr zufällig über den Raum verstreut, sondern liegen auch im CAN Raum nahe beieinander. Während dies für die Bereichsabfragen sehr wichtig ist, sorgt dies aber auch für eine ungleichmäßige Auslastung von einigen Servern:

- **Dichte Verteilung von Werten:** In einigen *Teilintervallen* können sich besonders viele *Attributwerte* anhäufen (Beispielsweise eine hohe oder niedrige CPU-Auslastung). Dadurch erhält der Server eines solchen *Teilintervalls* mehr Updates.
- **Häufige Updates von Werten:** Ebenso können einzelne *Attributwerte* besonders häufige Updates erfordern (Beispielsweise eine niedrige CPU-Auslastung, die durch Hintergrundprozesse nie 0 % ist). Auch dann muss der Server des entsprechenden *Teilintervalls* mehr Updates verwalten.

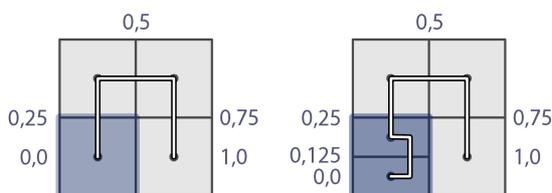


Abbildung 15: Dynamische Anpassung

Bei einer Überlastung eines Servers ist eine Aufteilung auf zwei Server die effektivste Lösung. Hierzu wird das betreffende *Teilintervall*

in zwei *Teilintervalle* aufgespalten und genauso der *Bereich*. (**Abbildung 15**).

Die Implementierung aus [2] führt dabei solch eine dynamische Anpassung durch, falls die Anzahl der Server, die dem stark belasteten Server regelmäßig Updates zukommen lassen, einen bestimmten Wert übersteigt (*Split Threshold* [2]). Dadurch können die ankommenden Updates auf zwei oder mehr Server verteilt werden. Dies löst nicht das zweite Problem von besonders häufigen Updates, allerdings geschehen in dieser Implementierung alle Updates mit derselben Frequenz, wodurch dieser Fall erst gar nicht eintreten kann.

Lokale Anpassungen führen aber leider dazu, dass das *Level* der *Hilbertfunktion* nicht mehr für den ganzen Raum angegeben werden kann. Grundsätzlich muss das *Level* der *Hilbertfunktion* nicht global bekannt sein, wie oben vorausgesetzt wurde. Da ein *Bereich* immer nur in kleinere *Bereiche* aufgeteilt wird, kann von einem sehr groben *Level* der *Hilbertfunktion* (beispielsweise 2) ausgegangen werden. Solange jeder Server seine Nachbarn im CAN und deren wirkliches *Level* bzw. *Bereich* kennt, können die Bereichsabfragen dennoch zuverlässig abgearbeitet werden. Allerdings kann solch eine grobe Sicht auf das Netzwerk die Effizienz der in 4.3 folgenden Strategien beeinflussen (Es können zu große Bereiche ausgewählt werden und auch ein *Punkt-zu-Punkt* Routing wird erschwert, da beispielsweise der *Attributwert* 0,64 an verschiedenen *Punkten* im Raum liegen kann). Daher kann zumindest das minimale oder maximale *Level* der *Hilbertfunktion* global festgehalten werden. Leider wird dieses Detail der Implementierung in [2] nicht angesprochen - vermutlich wird hier das Minimum verwendet. Die Beschreibung in Abschnitt 4.3 geht von einer völlig gleichmäßigen Verteilung und einem global bekannten *Level* aus.

4.3 Bereichsabfragen

Die allgemeine Vorgehensweise bei der Auswertung einer Bereichsabfrage ist folgende: Nun wird davon ausgegangen, dass das *Intervall der Abfrage* vorgegeben ist (Besteht die Abfrage aus mehreren einzelnen Intervallen, werden diese nacheinander abgearbeitet). Zuerst wird die Mitte des *Intervalls der Abfrage* berechnet. Im Fall $[0,39;0,89]$ entspricht dies dem *Attributwert* 0,64 (**Abbildung 16**). Dann wird der *Punkt* des *Attributwerts* im CAN Raum berechnet und über das CAN Routing eine Nachricht zum Server mit dem passenden *Bereich* geschickt.

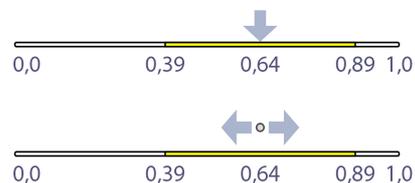


Abbildung 16: Mitte des Intervalls und Weiterleitung

Vom mittleren *Bereich* aus wird die Nachricht an alle Server weitergeleitet, deren *Teilintervalle* zur Abfrage $[0,39;0,89]$ gehören. Da dies viele Server sein können, wird die Weiterleitung als *Fluten* (engl. *Flooding*) bezeichnet. Hierfür werden drei verschiedene Strategien definiert.

4.3.1 Strategie 1: Normales Fluten

Die erste Strategie wurde bereits grob beschrieben. Es wird das kleinste Rechteck (der kleinste *Hyperquader*) berechnet, welches das *Intervall der Abfrage* komplett enthält. Genauer muss das Rechteck etwas größer sein, um alle *Bereiche* im CAN abzudecken, die von der Abfrage betroffen sind (**Abbildung 17**). Dann wird die Nachricht vom mittleren *Bereich* (heller Pfeil) aus über dieses Rechteck hinweg geflutet.

Das Fluten geschieht dabei ähnlich zur *Breitensuche* (*Breadth-First-Search*, *BFS*). Bei der Breitensuche werden erst alle Nachbarn eines Knotens in einem Graph abgearbeitet, bevor deren Nachbarn (bei Bäumen Kindknoten) betrachtet werden [7]. Da aber das Fluten im Netzwerk parallel geschieht, entsteht hierdurch keine streng festgelegte Abarbeitungsreihenfolge wie bei Graphen.

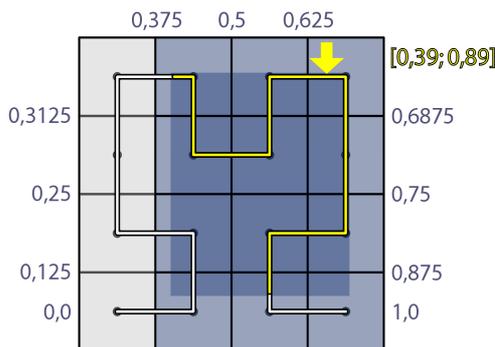


Abbildung 17: Normales Fluten

Der Nachteil dieser Methode ist, dass einige benachrichtigte Server eigentlich nicht zum *Intervall der Abfrage* gehören (Links und rechts unten in **Abbildung 17**).

4.3.2 Strategie 2: Kontrolliertes Fluten

Die zweite Strategie arbeitet ähnlich zur ersten. Die Nachricht wird vom aktuellen Knoten aus an alle Nachbarn weitergeleitet, allerdings nur wenn der Nachbar wirklich zum *Intervall der Abfrage* gehört (Das heißt, wenn sein *Teilintervall* das *Intervall der Abfrage* schneidet).

In **Abbildung 18** sind die betroffenen Server und die dabei entstehenden Nachrichten (kleine Pfeile) gezeigt. Generell kann und sollte natürlich verhindert werden, dass eine Nachricht an den Nachbarn zurückgeschickt wird, von dem die Nachricht stammt. Aber auch hier sind durch die parallele Abarbeitung und verschiedene Latenzzeiten mehrere Abarbeitungsreihenfolgen möglich.

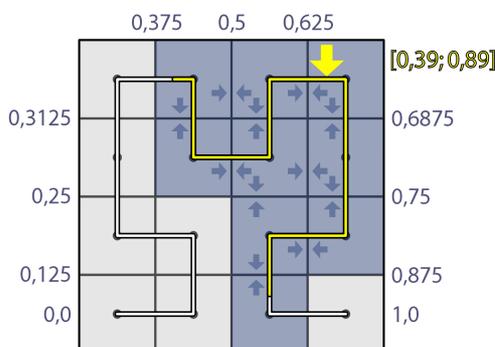


Abbildung 18: Kontrolliertes Fluten

Während nun keine überflüssigen Server mehr benachrichtigt werden, hat auch dieser Ansatz zwei Probleme: (1) Ein Server erhält eventuell mehrere Nachrichten zur selben Abfrage. (2) Die Abarbeitung kann weniger parallel als bei Strategie 1 sein (Im Beispiel ist dies aber unwahrscheinlich).

4.3.3 Strategie 3: Gerichtetes kontrolliertes Fluten

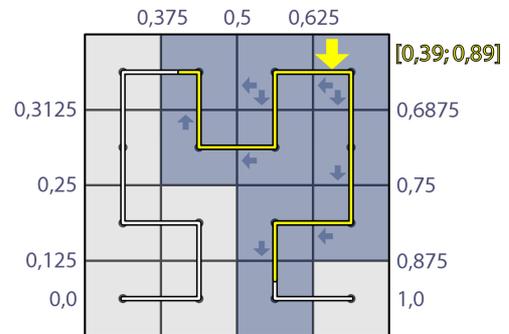


Abbildung 19: Gerichtetes kontrolliertes Fluten

Beim gerichteten kontrollierten Fluten wird die Nachricht in zwei Schritten weitergeleitet:

- **Erster Schritt:** Die Nachricht wird nur an Nachbarn weitergeleitet, deren *Teilintervall* das *Intervall der Abfrage* schneidet und gleichzeitig höher ist als das *Teilintervall* des aktuellen Knotens.
- **Zweiter Schritt:** Analog zum ersten Schritt werden nur niedrigere *Teilintervalle* betrachtet.

Beide Schritte können natürlich parallel abgearbeitet werden.

In **Abbildung 19** werden wesentlich weniger Nachrichten benötigt als bei Strategie 2. Im *Bereich* links von 0,625 wird dennoch eine Nachricht dupliziert, was in begrenztem Maße aber auch sinnvoll ist, da ganz ohne Parallelität die *Hilbertkurve* linear abgearbeitet werden müsste (Insgesamt bleibt die Parallelität auch geringer als bei Strategie 1).

4.4 Updates

Jeder Server sendet seinen aktuellen *Attributwert* in regelmäßigen Abständen zum Server mit dem passenden *Teilintervall* bzw. *Bereich* - beispielsweise in Abständen von fünf Minuten. Um Verwechslungen zu vermeiden, wird nun der Server mit dem aktuellen *Attributwert* als *Sender* bezeichnet und der Server mit dem dazu passenden *Teilintervall* als *Empfänger* (des *Attributwerts*).

Erhält der *Empfänger* in der darauf folgenden Updaterunde kein Update, entfernt er den Eintrag (*Attributwert*, *IP des Senders*) automatisch aus seinem Teil der Hash-Tabelle. Dieses sehr einfache Schema hat gleichzeitig viele Vorteile:

- Es werden keine zusätzlichen Nachrichten benötigt, wenn sich ein *Attributwert* so stark ändert, dass er von einem zum *Empfänger* benachbarten *Teilintervall* verwaltet wird.
- Der *Sender* muss nicht darüber informiert werden, wenn sich der *Bereich* des *Empfängers* ändert (Beispielsweise bei einer dynamischen Anpassung).
- Fällt der *Empfänger* aus, gehen alle Einträge seiner Hash-Tabelle verloren. Bei CAN wird der *Bereich* aber umgehend von einem anderen Server übernommen, welcher in

der nächsten Updaterunde wieder alle Einträge zur Verfügung hat.

Insgesamt ergibt sich also eine hohe Fehlertoleranz und weiterhin ein völlig selbstorganisierendes Netzwerk. Allerdings wird bei jedem Update das CAN-Routing verwendet, welches ohne spezielle Erweiterungen deutlich höhere Latenzzeiten als das IP-Routing haben kann. Die Verwendung eines Cache und eines so genannten *Expressways* kann dieses Problem aber weitestgehend beseitigen.

4.4.1 Cache

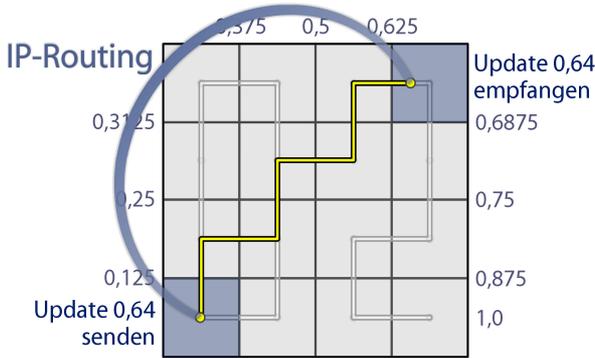


Abbildung 20: IP-Routing mit Cache

Zuerst benutzt der *Sender* das normale CAN-Routing. Nachdem der *Empfänger* das Update erhalten hat, sendet er seine IP-Adresse und sein momentanes *Teilintervall* zurück. Der *Sender* bewahrt nun den Eintrag (*Teilintervall*, *IP-Adresse*) in einem lokale Cache auf. Steht ein neues Update an, überprüft er zuerst den Cache auf ein passendes *Teilintervall* und versucht ein direktes IP-Routing (Abbildung 20).

Das CAN-Routing ist nur noch nötig, wenn kein passender Cache-Eintrag gefunden wird (engl. *Cache Miss*) oder der *Empfänger* keine positive Bestätigung zurücksendet. Letzteres könnte der Fall sein, wenn das *Teilintervall* des *Empfängers* nicht mehr diesen *Attributwert* abdeckt.

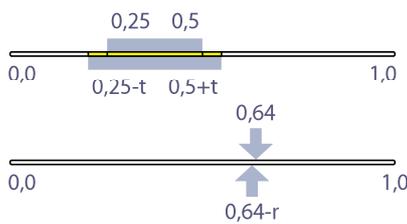


Abbildung 21: Toleranz- und Randomisierungslevel

Abfragen mit *Attributen* wie der CPU-Auslastung müssen meist keine exakten Resultate zurückliefern. Ein Server mit 11 % Auslastung ist genauso für weitere Berechnungen verwendbar wie ein Server mit 10 % Auslastung. Dies kann für zwei weitere Techniken ausgenutzt werden:

- **Toleranzlevel t:** Bei einer großen Anzahl von Servern werden die *Teilintervalle* der einzelnen Server entsprechend klein, was die Effizienz des Caches negativ beeinflusst und den Anteil des IP-Routing wieder verringert. Daher wird

zusätzlich ein *Toleranzlevel t* eingeführt, welches das *Teilintervall* eines Cacheeintrags an beiden Seiten vergrößert (Abbildung 21 oben).

- **Randomisierungslevel r:** Anstatt den aktuellen *Attributwert* unverändert an den *Empfänger* weiterzugeben, randomisiert ihn der *Sender* um $[-r;r]$ (Abbildung 21 unten). Dies verteilt die Last auf Server, deren *Teilintervalle* an den *Empfänger* angrenzen.

4.4.2 Expressway

Die Grundidee des *Expressway-Routing* [8] ist es, auf der einen Seite das *Overlay Netzwerk* von CAN homogen strukturiert zu belassen und auf der anderen einen *Expressway* zu erzeugen, der an das physikalische Netzwerk angepasst ist und ein beschleunigtes Routing zulässt (Abbildung 22).

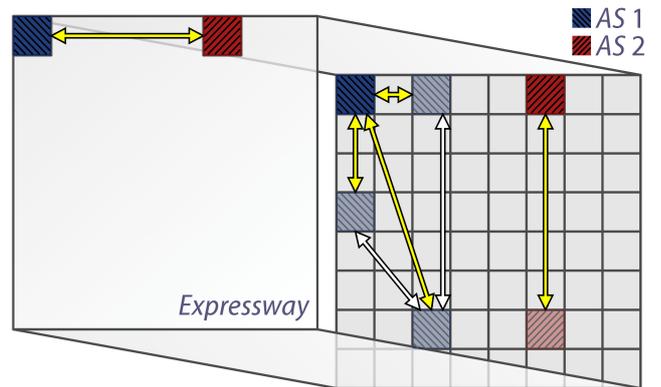


Abbildung 22: Expressway

Die *Expressway Knoten* (dunkel schraffiert) werden dabei in der Nähe von Routern und Gateways gewählt und sollen außerdem hochverfügbare Server mit entsprechenden Weiterleitungskapazitäten sein. Sie sind untereinander verbunden und formen zusammen den *Expressway* (Abbildung 22 links).

Normale Knoten (hell schraffiert) verbinden sich direkt mit dem lokalen *Expressway Knoten* ihres *autonomen Systems* (AS)³. Sie können über die *Expressway Knoten* Verbindungen zu Knoten außerhalb des AS aufbauen und gleichzeitig auch Verbindungen zu anderen *normalen Knoten* innerhalb des AS (Nachdem diese etabliert wurden, können *lokale Knoten* direkt miteinander kommunizieren, wie die weißen Pfeile zeigen).

Insgesamt kann mit dem *Expressway* fast ein optimales Routing ermöglicht werden. Das Verhältnis der Latenzzeit vom *Expressway-Routing* und *Shortest-Path-Routing* variierte in Simulationen von 1,04 bis 1,16.

4.5 Simulation

Die in [2] durchgeführten Simulationen basieren auf Protokollen (engl. *Traces*) eines Rechenzentrums. Als *Attribut* diente dabei die CPU-Auslastung. Zur Erstellung des Protokolls wurden 41 Server in 5 Minuten Intervallen für insgesamt 34 Tage überwacht.

4.5.1 Bereichsabfragen

Zuerst soll die Effizienz der Bereichsabfragen betrachtet werden. Diese besteht aus (1) dem Routing zum mittleren *Bereich* und (2) dem Fluten. Der erste Teil entspricht weitestgehend dem normalen

³ Das Internet ist aus einer Vielzahl von Teilnetzen (*autonomen Systemen*) aufgebaut, die untereinander verbunden sind [14].

CAN-Routing und lässt sich Hilfe des *Expressway* effizient durchführen. Bezüglich des Flutens wurden drei verschiedene Messungen durchgeführt (**Abbildung 23**). In allen Fällen wurde ein CAN mit 4 Dimensionen und ein Netzwerk mit 1000 Servern im CAN verwendet. Dazu begann die Simulation bei nur einem Server im CAN. Durch Updates und die daraus resultierende dynamische Anpassung wurde der Raum immer weiter verfeinert, bis 1000 Server erreicht wurden (Auch hier unterscheidet sich die Anzahl der Server im *Computational Grid* und im CAN). Zu den drei Messungen:

- **Abbildung 23 oben:** Zuerst wurde die durchschnittliche Pfadlänge gemessen, die nötig war, um alle Server beim Fluten zu erreichen. Im Schaubild ist diese Pfadlänge in Abhängigkeit von der Größe des *Intervalls der Abfrage* abgetragen - von 0,002 bis 0,6 mit 10 zufälligen Versuchen bzw. Abfragen pro Intervallgröße. Wie auch in den folgenden beiden Messungen wurden die Werte auf die Gesamtzahl der Server normiert, welche vom Fluten betroffen waren.
- **Abbildung 23 unten:** Zusätzlich wurde die Anzahl der gesendeten Nachrichten und der besuchten Knoten während des Flutens abgetragen. Auch hier liefert das kontrollierte Fluten deutlich bessere Ergebnisse und damit weniger Overhead an Nachrichten. Die beiden kontrollierten Strategien liefern aber weitestgehend identische Ergebnisse.

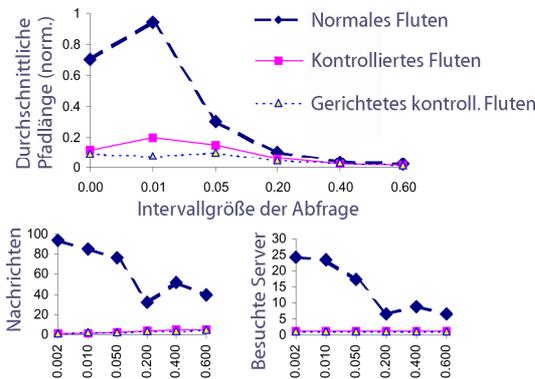


Abbildung 23: Auswertung für Bereichsabfragen

4.5.2 Updates

Für die Simulation der Updates wurden diese in 5 Minuten Intervallen über 24 Stunden hinweg durchgeführt. Insgesamt liefen die Updates von 33 verschiedenen Tagen parallel, um mehr Updates pro Updaterunde zu erhalten. Es waren am Ende etwa 110 bis 160 Server im CAN (ebenfalls durch dynamische Anpassung):

- **Abbildung 24 oben:** Hier ist der Anteil des IP-Routing in Abhängigkeit von der Cachegröße abgetragen. Der Cache beginnt etwa ab einer Größe von 10 Einträgen zu sättigen (7-10 % der Server). Mit einer Erhöhung des *Toleranzlevel* t kann der Anteil noch deutlich verbessert werden.
- **Abbildung 24 unten:** Das Schaubild zeigt die Verteilung der Server (*Empfänger*) über alle *Attributwerte* von 0 bis 1 - genauer über jeweils kleine Intervalle von *Attributwerten*. Ein höheres *Randomisierungslevel* r lässt eine gleichmäßigere Verteilung vermuten, der Effekt ist aber auf Peaks begrenzt.

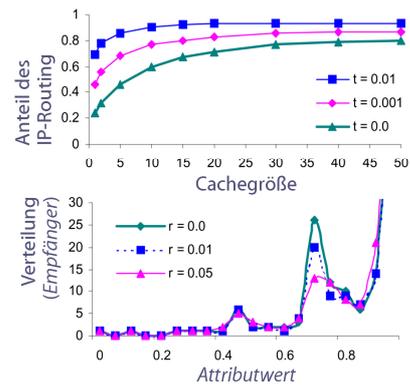


Abbildung 24: Auswertung für Updates

4.6 Vor- und Nachteile

Zusammengefasst haben die *Bereichsabfragen mit Hilbertfunktionen* folgende Vorteile:

- Die Suche nach einem ganzen Intervall von *Attributwerten* wird ermöglicht.
- Regelmäßige Updates werden effizient realisiert.

Dagegen bleibt aber auch ein offenes Problem bzw. ein Nachteil:

- Ist das *Intervall der Abfrage* sehr groß, müssen sehr viele Server abgefragt werden. Im Extremfall $[0;1]$ ist ein Fluten des gesamten CAN nötig, welches aus sehr vielen Servern bestehen kann, selbst wenn nur ein Teil der Server des *Computational Grids* für den *Indexdienst* verwendet wird.

5. CANDY

Die zweite Technik [9] führt von *Computational Grids* zurück zum *File Sharing* als Anwendung. Statt Hilfsmitteln wie der *Hilbertfunktion* werden hier zwei getrennte CANs eingesetzt. Die Erweiterung nennt sich *CANDy (Content-Addressable Network DirectorY)*.

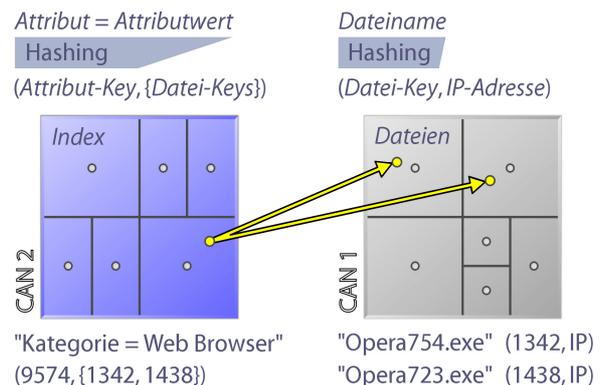


Abbildung 25: Erstes und zweites CAN

Erstes CAN (Abbildung 25 rechts): Wie in Abschnitt 3 wird ein *Dateiname* über eine Hash-Funktion auf einen *Datei-Key* abgebildet. Auch hier können im CAN Paare aus (*Datei-Key*, *IP-Adresse*) gespeichert werden, die den Speicherort der Datei liefern. In der Abbildung zeigen die *Datei-Keys* 1342 und 1438 auf zwei verschiedenen Versionen von Opera.

Dieses CAN ist damit beschränkt auf das reine *Nachschlagen* von gegebenen *Datei-Keys*.

Zweites CAN (Abbildung 25 links): Neben einem *Dateiname* kann eine Datei durch eine ganze Reihe an *Attributen* beschrieben werden (*Dateiname*, *Dateigröße*, *Kategorie*, ...). Im Gegensatz zum einfachen *Nachschlagen* zielen *Abfragen* auf alle diese *Attribute* ab bzw. auf eine Kombination bestimmter *Attributwerte* - mit eventuell komplexeren Suchkriterien. Um dies zu ermöglichen, wird ein zweites CAN benötigt, welches ähnlich wie in Büchern als *Index* dient. Das *Attribut* und der *Attributwert* werden mit einer Hash-Funktion gemeinsam auf einen *Attribut-Key* abgebildet (Beispielsweise "*Kategorie = Web Browser*" auf *9574*). Der *Attribut-Key* dient nun zum Nachschlagen im *Index*, in dem Paare aus (*Attribut-Key*, {*Datei-Keys*}) gespeichert sind. Die Menge von *Datei-Keys* verweist auf das erste CAN und enthält alle Dateien, deren *Attributwert* zum *Attribut-Key* passt. Im Beispiel trifft dies auf die beiden *Datei-Keys* *1342* und *1438* zu, da Opera zur *Kategorie Web Browser* gehört.

Auch im Folgenden die beiden CANs logisch getrennt sind, können sie problemlos mit Hilfe eines CAN implementiert werden (Beispielsweise durch Unterscheidung mit einem Bit im Key).

Für jedes *Attribut* wird zusätzlich noch Verwaltungsinformation benötigt. Dies übernimmt der so genannte *Deskriptor* des *Attributs*, welcher ebenfalls im *Index* gespeichert wird. Eine Hash-Funktion bildet dazu das *Attribut* (den Attributnamen) auf den *Deskriptor-Key* ab. **Abbildung 26** enthält die *Deskriptoren* von drei verschiedenen *Attributen*.



| Attribut | Datentyp | Suche | Beispielwert |
|------------|----------|-----------|--------------|
| Dateiname | String | Präfix | Opera754.exe |
| Dateigröße | Integer | Bereich | 3,67 MB |
| Kategorie | String | Wortmenge | Web Browser |

Abbildung 26: Deskriptoren der Attribute

Der *Deskriptor* bestimmt den *Datentyp* des *Attributs* und außerdem die Art der Suche nach gegebenen *Attributwerten*. Beispielsweise wird die *Kategorie* als **String** abgespeichert und die *Suche* ist über **Wortmengen** organisiert (Zum Beispiel die Wörter **web** und **browser**). Die *Dateigröße* dagegen ist ein **Integer** und ermöglicht eine Suche auf einem **Bereich** von Werten. Der *Dateiname* ist ein **String** und hier kann (zu einem gegebenen *Dateinamen*) nach dem längsten passenden **Präfix** gesucht werden. In Abschnitt 5.1 werden diese drei Suchmethoden im Detail betrachtet.

Die Art der Suche bestimmt außerdem die Art der Speicherung im CAN - ähnlich zu den *Hilbertfunktionen*. Je nach Art der Suche kann dadurch noch zusätzliche Verwaltungsinformation benötigt werden, die ebenfalls im *Deskriptor* abgelegt wird.

5.1 Abfragen

Eine *Abfrage* wird nun insgesamt in drei Schritten abgearbeitet (**Abbildung 27**).

Im ersten Schritt wird die *Abfrage* analysiert. Das heißt, die beteiligten *Attribute* werden ermittelt und deren *Deskriptor-Keys* berechnet. Hiermit kann im *Index* der *Datentyp* und die Art der Suche bzw. Speicherung nachgeschlagen werden.

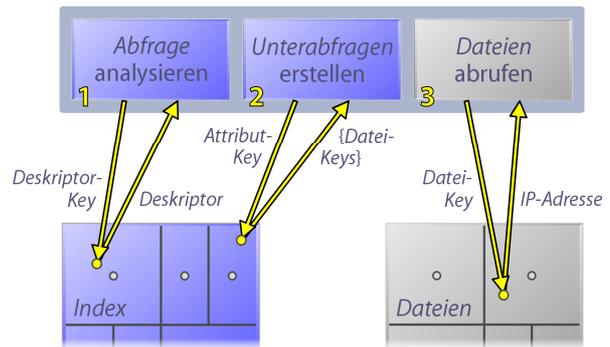


Abbildung 27: Vorgehensweise bei Abfragen

Danach wird im zweiten Schritt die *Abfrage* in mehrere *Unterabfragen* aufgeteilt. Dies geschieht so, dass jede *Unterabfrage* gerade einem *Attribut-Key* entspricht, das heißt einen Zugriff auf den *Index* benötigt. Der *Index* liefert zu jedem *Attribut-Key* eine Menge von *Datei-Keys* zurück. In einer *Abfrage* werden diese typischerweise über Mengenoperationen verknüpft. Beispielsweise wird der *Schnitt* aller Mengen gebildet, wenn der *Attributwert* alle in der *Abfrage* gegebenen Suchwörter enthalten soll.

Nachdem nun die passenden *Datei-Keys* ermittelt wurden, können im dritten und letzten Schritt diese im CAN für *Dateien* wie gewohnt nachgeschlagen werden (**Abbildung 27** rechts). Zuvor kann der Benutzer natürlich noch eine manuelle Auswahl der Dateien durchführen.

Abbildung 27 zeigt, dass in allen drei Schritten ein oder mehrere *Keys* nachgeschlagen werden. Beim ersten und dritten Schritt kann dies bei mehreren *Keys* problemlos nacheinander geschehen. Beim zweiten Schritt würde dies bedeuten, dass zuerst alle Mengen von *Datei-Keys* zum Knoten, der die *Abfrage* gestellt hat, zurückgeschickt werden und dieser dann die Verknüpfung durchführt. Beispielsweise beim *Schnitt* können die einzelnen Mengen im Vergleich zur Ergebnismenge aber sehr groß sein, wodurch das Netz unnötig stark belastet wird.

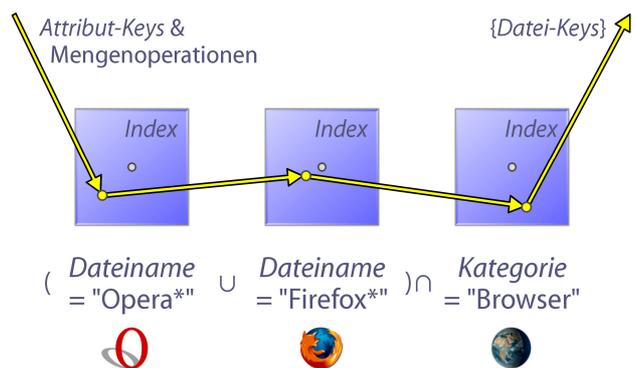


Abbildung 28: Abfrage mit drei Unterabfragen

Da jede *Unterabfrage* einem *Attribut-Key* entspricht, wird jede *Unterabfrage* von genau einem Knoten im CAN bearbeitet. Daher ist folgende alternative Abarbeitungsstrategie möglich (**Abbildung 28**): Die *Unterabfragen* werden von Knoten zu Knoten weitergeleitet. Der aktuelle Knoten verknüpft seine Menge mit dem bisherigen Ergebnis und schickt das neue Ergebnis weiter. Die Weiterleitung geschieht dabei über das CAN-Routing. Im Beispiel werden zuerst alle zum *Präfix Opera** passenden *Datei-Keys* nachgeschlagen. Die Menge an *Datei-Keys* wird weitergeleitet und um die *Datei-Keys* zum *Präfix Firefox** erweitert (*Opera** und *Firefox** könnten die *längsten passenden Präfixe* zu den *Dateinamen*

Opera754.exe und Firefox10.exe sein). Schließlich findet im dritten Knoten ein Schnitt mit den Kategorien statt, die das Wort (die Wortmenge) Browser enthalten. Nun wird das Ergebnis zum Knoten zurückgeschickt, der die Anfrage gestellt hat.

Das Endergebnis ist wahrscheinlich deutlich kleiner als die Einzelmengen und die Gesamtlast der Abfrage wurde auf alle beteiligten Knoten verteilt, wodurch weniger Engpässe im Netzwerk entstehen.

Beim Erstellen der Unterabfragen werden die nötigen Attribut-Keys und die verknüpfenden Mengenoperationen ermittelt. Die Mengenoperationen können dabei beispielsweise wie in einem Stack kodiert werden, das heißt in umgekehrter polnischer Notation ($AB\cap$ statt $A\cap B$ [10]).

Im obigen Beispiel werden zwischen dem ersten und zweiten Knoten und dem zweiten und dritten Knoten immer noch viele Datei-Keys übertragen. Die Menge an übertragenen Datei-Keys kann durch mathematische Umformungen und Bloom-Filter (Abschnitt 5.3) stark verringert werden. Im obigen Beispiel können durch das Auflösen der Klammern mit dem Distributivgesetz $(a\cup b)\cap c = (a\cap c)\cup(b\cap c)$ und durch die effiziente Berechnung der beiden Schnitte mit Bloom-Filtern die übertragenen Datei-Keys fast auf die des wirklichen Endergebnisses beschränkt werden.

Die verfügbaren Mengenoperationen bei CANDy sind:

- Vereinigung: $A \cup B$
- Schnitt: $A \cap B$
- Differenz: $A \setminus B$
- Komplement: $\bar{A} = C \setminus A$

Der Komplement-Operator macht dabei nur in Verbindung mit einer Basismenge C Sinn und entspricht dann der Differenz. Oft ist diese Basismenge C sehr groß, wie beispielsweise alle Zeichenketten in einem String. In diesem Fall kann alternativ auch die Menge A mit einer Komplement-Markierung übertragen werden. Wird beispielsweise $\bar{A}\cap B$ berechnet, dann ist das Aufzählen von \bar{A} nicht nötig - es genügt die passenden Elemente aus B zu filtern.

Einige Anfragen können trotz Optimierungen zu einer großen Anzahl an übertragenen Datei-Keys führen. Falls die Implementierung des CAN nur eine begrenzte Nachrichtengröße zulässt, kann eine einzelne Nachricht in viele durchnummerierte kleine Nachrichten aufgeteilt werden. Alternativ lässt sich eine Verbindung per CAN-Routing aufbauen und die eigentliche Nachricht per IP-Routing übertragen.

Zusätzlich ist ein Streaming möglich, wenn alle Datei-Keys nach einem bestimmten Kriterium geordnet sind. Das heißt ein Knoten kann bereits die Mengenoperation anwenden, bevor er alle Datei-Keys des vorherigen Knotens empfangen hat.

5.2 Suche

Wie bereits erwähnt, werden drei verschiedene Arten der Suche unterstützt: Die Suche auf Wortmengen, die Bereichssuche und die Präfixsuche.

5.2.1 Suche auf Wortmengen

Bei der Suche auf Wortmengen werden Attributwerte in einzelne Wörter aufgeteilt und jedes Wort erhält einen eigenen Attribut-Key.

Im Beispiel in **Abbildung 29** ist der Attributwert PDF Konverter (Freeware) als Kategorie gegeben. Statt einen Attribut-Key aus "Kategorie = PDF Konverter (Freeware)" zu berechnen, werden drei getrennte Attribut-Keys für jedes der Einzelwörter ermittelt. Unter jedem können nun die Datei-Keys der zum vollen Attributwert passenden Software gefunden werden, beispielsweise Ghostscript⁴.

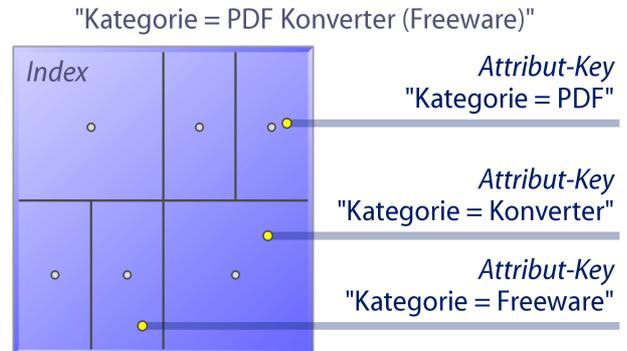


Abbildung 29: Beispiel für die Suche auf Wortmengen

Der Vorteil dieser Vorgehensweise ist natürlich, dass nun eine Suche nach Einzelwörtern möglich ist. Außerdem zeigt **Abbildung 29** aber auch, dass die drei Attribut-Keys verschiedenen Knoten zugeordnet sind. Insgesamt werden so alle Attributwerte weiter über das Netzwerk verteilt, was zu einer stärker verteilten Abarbeitung von Unterabfragen führt (Ein Vorteil bezüglich der Ausfallsicherheit und einzelnen Engpässen im Netzwerk).

Erweiterungen sind denkbar, die das Suchen von Wörtern in einer bestimmten Reihenfolge ermöglichen (PDF Konverter statt Konverter PDF). Ebenfalls kann die Suche auf so genannte n-Gramme erweitert werden [12]. Dabei wird ein Wort in alle möglichen Teilwörter mit jeweils n Buchstaben aufgeteilt. In beiden Fällen ist aber zusätzlicher Verwaltungsaufwand nötig, während oben mehrere Wörter einfach über den Schnitt der Datei-Key Mengen verbunden werden können.

5.2.2 Bereichssuche

Die Bereichssuche ist auf Zahlen vom Typ Float oder Integer ausgelegt. Im Folgenden wird davon ausgegangen, dass ein Intervall mit diskreten Werten vorgegeben ist, beispielsweise $[0;1;2;...;7]$ (Im Falle von Floats können statt diskreten Werten kleine Intervalle verwendet werden).

Ein RST (Range Search Tree, dt. Bereichssuchbaum) ermöglicht eine effiziente Suche auf solch einem Intervall. **Abbildung 30** enthält ein Beispiel für einen RST, der obiges Intervall abdeckt. Die Blätter bestehen dabei aus den einzelnen diskreten Werten, welche in der darüber liegenden Stufe jeweils zu einem Teilintervall zusammengefasst werden. Zum Beispiel die Blätter 0 und 1 werden zum Teilintervall $[0;1]$ bzw. $0-1$. Die Wurzel des Baumes deckt das gesamte Intervall ab. Im CAN wird jedem Knoten im Baum ein Attribut-Key im Index zugewiesen, welcher die zum Teilintervall passenden Datei-Keys enthält (In **Abbildung 30** rechts fehlen aus Platzgründen die Attribut-Keys für die Blätter).

Bei einer Suche werden die größten passenden Teilintervalle zum Intervall der Abfrage gesucht. Das heißt, das Intervall der Abfrage soll durch möglichst wenige Knoten im Baum abgedeckt werden.

⁴ Genau genommen ist Ghostscript keine Freeware, sondern steht unter der GPL (GNU General Public License) oder AFPL (Aladdin Free Public License) [11].

Eine Abfrage 1-5 führt beispielsweise zu den Unterabfragen bzw. Teilintervallen 1, 2-3 und 4-5. Ohne einen RST würde diese Abfrage fünf Unterabfragen benötigen. Bei großen Bäumen wird dieser Unterschied noch wesentlich deutlicher.

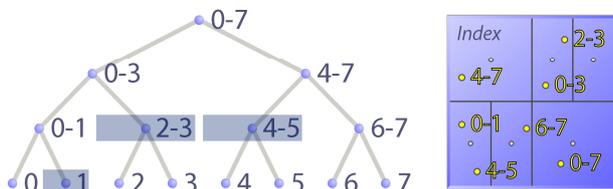


Abbildung 30: Beispiel für die Bereichssuche

Der Deskriptor des zugehörigen Attributs speichert zusätzliche Informationen, wie den Knotengrad k und die Tiefe des Baumes.

Allgemein ist der Speicheraufwand für den Baum und der Suchaufwand bei einer Abfrage logarithmisch. Genauer: Bei vorgegebenem Knotengrad k und bei w diskreten Werten (im Beispiel $k=2$ und $w=8$) ergibt sich $O(\log_k w)$ für den Speicher- und Suchaufwand. Der Speicheraufwand gibt dabei an, wie oft ein Datei-Key auf anderen Ebenen des Baumes dupliziert werden muss, als Vergleich zur einmaligen Speicherung jedes Datei-Keys ohne RST (wenn nur die Blätter in den Index aufgenommen werden).

5.2.3 Präfixsuche

Bei der Präfixsuche wird zu einem gegebenen Attributwert das längste passende Präfix ermittelt. Normalerweise wird diese Technik eingesetzt, um einen String oder eine binäre Zahl einer Klasse zuzuordnen, wobei jede Klasse durch ein Präfix beschrieben ist.

Ein typisches Beispiel ist das CIDR (Classless Inter-Domain Routing) [13]. Um ein IP-Paket weiterzuleiten, müssen Router in ihren Routing-Tabellen das längste passende Präfix zur gegebenen IP-Adresse ermitteln. Die gefundene Klasse entspricht hierbei dem Zielnetzwerk für die Weiterleitung. Zur binären IP-Adresse 11000010.00011000.00010001.00000100 passt beispielsweise das Präfix 11000010.00011000.0001* [14].

Um weiterhin beim File Sharing zu bleiben, wird als Beispiel die Präfixsuche auf Dateinamen angewendet. Zu einem gegebenen Dateinamen (z. B. Opera754.exe) kann so das längste passende Präfix (z. B. Opera7*) gefunden werden. Das Nachschlagen des Attribut-Keys zum Präfix ergibt alle Datei-Keys, deren Dateiname mit Opera7* beginnt und damit zu Opera754.exe ähnliche Dateinamen (Auch wenn das Prinzip sinnvoll scheint, so sind Dateinamen keine typische Anwendung der Präfixsuche, da hier meist zu einem gegebenen Präfix (z. B. Opera*) alle passenden Dateinamen ermittelt werden sollen, was gerade der umgekehrten Vorgehensweise entspricht).

Zuerst wird jedes gegebene Präfix als Attribut-Key im Index gespeichert - in der Form "Dateiname = Präfix". Da die Suche nach der Präfixlänge organisiert ist, wird dabei eine maximale Präfixlänge festgelegt, hier 7 Zeichen. Im Beispiel sind die Präfixe Firefo*, Opera6*, Opera7* und Opera* vorgegeben (Dunkle Felder in Abbildung 31). Der Index enthält zu jedem Präfix die passenden Datei-Keys.

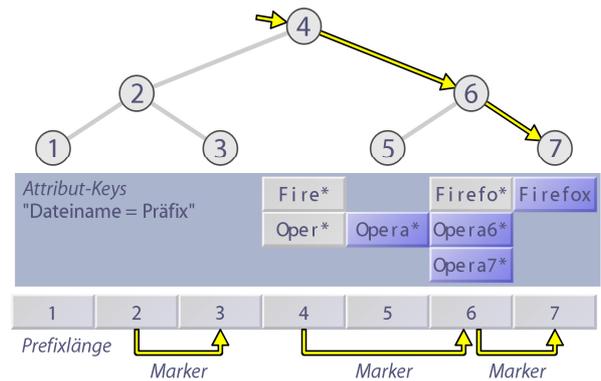


Abbildung 31: Beispiel für die Präfixsuche

Um die Suche nach dem längsten passenden Präfix effizient zu gestalten, wird die binäre Suche verwendet. Die binäre Suche ist dabei nach der Präfixlänge organisiert. Der Baum in Abbildung 31 zeigt das Prinzip: Zuerst wird die Mitte 4 des Intervalls 1-7 gewählt. Dann wird entschieden, ob die Suche im linken Teilbaum 1-3 oder rechten Teilbaum 5-7 fortgeführt werden soll und dies rekursiv fortgesetzt. Normalerweise wird die binäre Suche auf eine sortierte Liste von Zahlen angewendet, in der eine gegebene Zahl gefunden werden soll. Das mittlere Element der Liste wird dazu mit der gegebenen Zahl verglichen und abhängig vom Ergebnis des Vergleichs ($<$, $>$, $=$) der linke oder rechte Teilbaum gewählt (Im Fall $=$ ist das Element gefunden) [7]. Das Nachschlagen eines Attribut-Keys dagegen liefert lediglich, ob das Element gefunden oder nicht gefunden wurde. Dies alleine reicht für die binäre Suche nicht aus.

Daher wird zuerst folgende Annahme gemacht: Wird ein passendes Präfix der Länge 4 gefunden, dann wird dies als Hinweis dafür gewertet, dass ein noch längeres passendes Präfix existieren könnte und die Suche im rechten Teilbaum fortgesetzt. Gleichzeitig wird aber das bisher gefundene Präfix gespeichert, falls die weitere Suche erfolglos bleibt. Wird kein Präfix gefunden, wird dagegen die Suche im linken Teilbaum fortgesetzt.

Damit dieses Schema funktioniert, müssen so genannte Marker eingefügt werden (Helle Felder in Abbildung 31). Beispielsweise für das Präfix Firefox wird der Marker Fire* als Attribut-Key eingefügt⁵, der bei der Suche nach einem Präfix der Länge 4 als Hinweis dient, dass ein längeres passendes Präfix existiert (Falls das Präfix Fire* bereits vorhanden ist, wird natürlich kein Marker benötigt). Die Marker liefern daher die Richtungsinformation, die bisher bei der binären Suche fehlte. Durch das Schema der binären Suche (Es wird rekursiv jeweils von der Mitte ausgehend der linke oder rechte Teilbaum gewählt) müssen die Marker nur an den dazu passenden Stellen eingesetzt werden. Die gelben Pfeile unten in Abbildung 31 geben an, wo Marker nötig sind und auf welchen weiteren Marker oder Präfix sie verweisen. Zum Beispiel führt der Marker Fire* zum Marker Firefo* und dieser schließlich zum Präfix Firefox.

Auch dieser Ansatz hat noch ein Problem: Die Suche nach einem Präfix zum Dateiname OperSys.zip wird durch den Marker Oper* in den rechten Teilbaum weitergeleitet und würde dort aber nicht weiterkommen. In diesem Fall muss die Suche zum Marker Oper* zurückkehren und den linken Teilbaum weiterfolgen (Im Beispiel bleibt auch dies erfolglos, da kein passendes Präfix vorhanden ist, aber prinzipiell könnte dort ein Präfix zu finden sein,

⁵ Markern und Präfixen wird ein Attribut-Key zugewiesen, sie werden aber logisch unterschieden, was beispielsweise durch einen speziellen Code im Value zum Paar (Key, Value) möglich ist.

wie beispielsweise Op^*). Um Zeit zu sparen, kann zu jedem *Marker* das zu ihm *längste passende Präfix* vorberechnet werden, was die Suche im linken Teilbaum ersetzt (Im Beispiel wäre dieses gespeicherte *Präfix* leer).

Werden all diese Verbesserungen implementiert, liefert die *binäre Suche* einen logarithmischen *Suchaufwand* in Abhängigkeit von der *Präfixlänge*.

5.3 Bloom-Filter

Ein *Bloom-Filter* [15] ermöglicht es, eine Menge sehr kompakt darzustellen. Auf diese Weise kann eine Menge an *Datei-Keys* durch nur wenige Bits dargestellt werden. Dennoch bleibt der *Element-Test*, ob ein bestimmter *Datei-Key* Teil der Menge ist, weiterhin möglich (mit gewissen Einschränkungen). Die Idee hierbei ist, weniger Daten über das Netzwerk transferieren zu müssen.

Abbildung 32 zeigt einen *Bloom-Filter* mit $m = 12$ Bit. Um den *Bloom-Filter* zu erstellen, wird dieser zuerst mit 0-Bits initialisiert. Nun werden schrittweise alle *Datei-Keys* der Menge eingefügt. Für jeden *Datei-Key* wird dabei folgendermaßen vorgegangen: Auf den *Datei-Key* wird eine Hash-Funktion angewendet, die eine Bitposition zwischen 1 und m liefert. An dieser Stelle wird ein 1-Bit gesetzt. Dies wird mit einer Reihe anderer Hash-Funktionen wiederholt, im Beispiel drei verschiedenen. Der *Datei-Key* wird durch diese 1-Bits repräsentiert. Bei den übrigen *Datei-Keys* ist die Vorgehensweise dieselbe, wobei gesetzte 1-Bits unverändert bleiben, wie das erste Bit von *Datei-Key 2* in **Abbildung 32**.

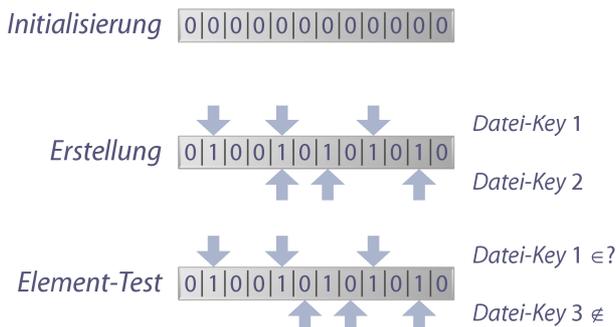


Abbildung 32: Beispiel für eine Bloom-Filter

Beim *Element-Test* werden die passenden Bit-Positionen zum gegebenen *Datei-Key* berechnet und auf 1-Bits überprüft. In **Abbildung 32** unten liegt der *Datei-Key 1* in der Menge, während der *Datei-Key 3* durch die beiden 0-Bits auf jeden Fall nicht enthalten sein kann. Leider liefert der Test für *Datei-Key 1* kein sicheres Ergebnis, denn die 1-Bits könnten auch von anderen Elementen stammen. Dadurch kommen zur bisherigen Menge neue *falsche Elemente* (*False Positives*) hinzu.

Bloom-Filter haben insgesamt folgende Nachteile:

- **Falsche Elemente (False Positives):** Die Wahrscheinlichkeit, dass der *Element-Test* für ein Element außerhalb der Menge erfolgreich ist ("fälschlicherweise positiv"), kann abgeschätzt werden. Auch wenn *falsche Elemente* nicht vermieden werden können, ermöglichen ein hinreichend großer *Bloom-Filter* und geeignete Hash-Funktion das Beschränken dieser Wahrscheinlichkeit auf ein geringes Maß.
- **Aufzählbarkeit:** Die Elemente der Menge können nicht mehr aufgezählt werden, falls kein beschränktes Intervall angegeben werden kann, aus dem die *Datei-Keys* stammen können (Auch wenn in der Praxis die Bitlänge der *Datei-*

Keys und damit deren Anzahl beschränkt ist, wären im Normalfall zu viele *Element-Tests* nötig).

Dagegen gibt es aber auch eine Reihe sehr nützlicher Eigenschaften bzw. möglicher Operationen:

- **Erstellung:** Das Erstellen eines *Bloom-Filters* aus einer gegebenen Menge A erfolgt wie oben beschrieben.

$$A \rightarrow \text{Bloom}(A)$$

- **Schnitt:** Zwei *Bloom-Filter* derselben Größe und mit denselben Hash-Funktionen können durch ein einfaches bitweises *AND* miteinander geschnitten werden. Die Operation ist *kommutativ* und *assoziativ*.

$$\begin{aligned} \text{Bloom}(A \cap B) &= \text{Bloom}(A) \cap \text{Bloom}(B) \\ &= \text{Bloom}(A) \text{ AND } \text{Bloom}(B) \end{aligned}$$

- **Vereinigung:** Die *Vereinigung* ist analog zum *Schnitt* mit einem bitweisen *OR* möglich.

$$\begin{aligned} \text{Bloom}(A \cup B) &= \text{Bloom}(A) \cup \text{Bloom}(B) \\ &= \text{Bloom}(A) \text{ OR } \text{Bloom}(B) \end{aligned}$$

- **Element-Test:** Wie bereits erwähnt, führt der *Element-Test* zu *falschen Elementen*. Daher kann abstrakt die Menge $\text{Bloom}(A)$ als die Menge A plus eine zusätzliche Menge von *falschen Elementen* $\epsilon(A)$ betrachtet werden.

$$\text{Bloom}(A) = A \cup \epsilon(A)$$

Auch wenn die Menge $\epsilon(A)$ groß sein kann (mathematisch betrachtet sogar unendlich), kann sie für die folgenden Überlegungen als eine im Vergleich zu A kleine Menge veranschaulicht werden. Dies liegt daran, dass die *falschen Elemente* über alle möglichen *Datei-Keys* verstreut sind und der *Schnitt* $(A \cup \epsilon(A)) \cap B$ mit einer aufzählbaren Menge B (per *Element-Test*) wahrscheinlich nur wenige *falsche Elemente* aus $\epsilon(A)$ enthält. Dies wird im Folgenden ausgenutzt.

Ziel ist es nun, mit *Bloom-Filtern* die Menge an Daten zu reduzieren, die bei der Abarbeitung von *Unterfragen* zwischen den Knoten ausgetauscht werden muss. Dabei sollen möglichst nur die *Datei-Keys* übertragen werden, die auch tatsächlich Teil des Endergebnisses sind.

Dazu werden die Mengenoperationen *Schnitt*, *Differenz* und *Vereinigung* erneut betrachtet und optimiert. Wichtig anzumerken ist, dass hier das Endergebnis weiterhin korrekt bleiben soll und keine *falschen Elemente* enthalten darf.

5.3.1 Schnitt

Es wird angenommen, dass eine *Abfrage* bestehend aus zwei *Unterabfragen* gegeben ist, die über einen *Schnitt* verbunden werden sollen. Die Menge mit *Datei-Keys* der ersten *Unterabfrage* wird nun mit A und der zweiten *Unterabfrage* mit B bezeichnet.

Abbildung 33 enthält die optimierte Berechnung von $A \cap B$ mit Hilfe von *Bloom-Filtern*. Zuerst wird im linken Knoten der *Bloom-Filter* $\text{Bloom}(A)$ erstellt und zum rechten Knoten weitergeleitet. Dort wird per *Element-Test* $\text{Bloom}(A) \cap B$ berechnet (Die Menge B von *Datei-Keys* ist natürlich aufzählbar). Dies ist bereits eine gute Näherung des Endergebnisses. Es wird nun zum linken zurückgeschickt und dieser bildet $\text{Bloom}(A) \cap B \cap A$, was dem exakten Endergebnis $A \cap B$ entspricht, denn alle falschen Elemente werden durch die beiden Schnitte mit den Originalmengen herausgefiltert.

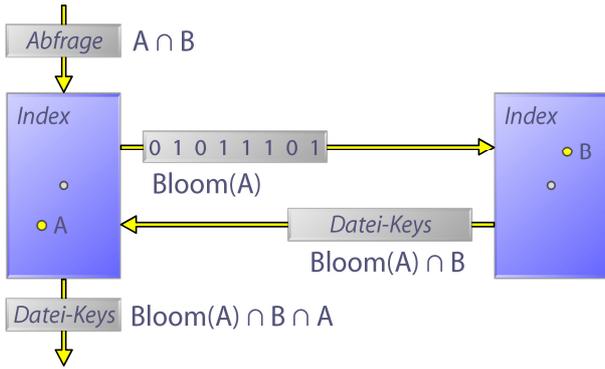


Abbildung 33: Schnitt mit Bloom-Filtern

A und B können im Vergleich zu $A \cap B$ sehr große Mengen sein. Das neue Verfahren überträgt aber lediglich den kompakten Bloom-Filter von A und eine gute Näherung des Endergebnisses, wodurch die großen Mengen nur lokal in den Knoten verarbeitet werden.

Falls A und B beinahe dieselben Elemente enthalten, hilft das obige Verfahren nicht, aber auch dann werden im Wesentlichen nur die Datei-Keys übertragen, die das Endergebnis wirklich enthält.

Eine Verallgemeinerung auf $A \cap \dots \cap Z$ ist ebenfalls möglich. Grundsätzlich werden die Schritte aus **Abbildung 33** in mehrere Einzelschritte aufgeteilt:

$$\begin{aligned}
 \mathcal{N}_A &\rightarrow \mathcal{N}_B : \mathcal{B}(A) \\
 \mathcal{N}_B &\rightarrow \mathcal{N}_C : \mathcal{B}(\mathcal{B}(A) \cap B) = \mathcal{B}(A \cap B) \\
 &\vdots \\
 \mathcal{N}_Y &\rightarrow \mathcal{N}_Z : \mathcal{B}(\mathcal{B}(A \cap B \cap \dots \cap X) \cap Y) = \mathcal{B}(A \cap B \cap \dots \cap X \cap Y) \\
 \mathcal{N}_Z &\rightarrow \mathcal{N}_Y : \mathcal{B}(A \cap B \cap \dots \cap Y) \cap Z \\
 \mathcal{N}_Y &\rightarrow \mathcal{N}_X : \mathcal{B}(A \cap B \cap \dots \cap Y) \cap Z \cap Y \\
 &\vdots \\
 \mathcal{N}_B &\rightarrow \mathcal{N}_A : \mathcal{B}(A \cap B \cap \dots \cap Y) \cap Z \cap Y \cap \dots \cap C \cap B \\
 \mathcal{N}_A & : \mathcal{B}(A \cap B \cap \dots \cap Y) \cap Z \cap Y \cap \dots \cap C \cap B \cap A
 \end{aligned}$$

\mathcal{N}_A gibt den Knoten mit der Menge A an und $\mathcal{B}(A) = \text{Bloom}(A)$. Die markierten Ausdrücke werden vom vorherigen Schritt wieder verwendet.

5.3.2 Differenz

Die Differenz $A \setminus B$ scheint schwerer zu berechnen, da nicht das Komplement eines Bloom-Filters gebildet werden kann oder Elemente aus einem Bloom-Filter entfernt werden können (Ein 1-Bit repräsentiert normalerweise mehrere Elemente auf einmal).

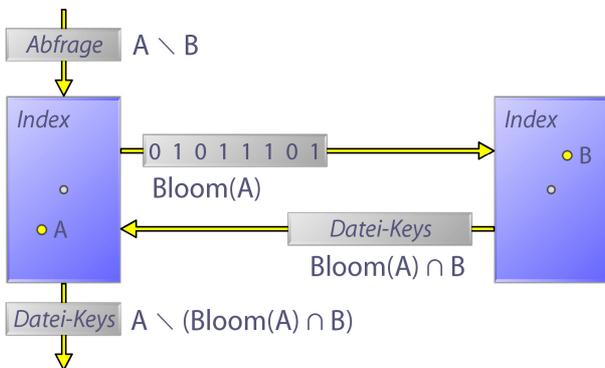


Abbildung 34: Differenz mit Bloom-Filtern

Die Berechnung des Schnitts lässt sich aber so abändern, dass die Differenz berechnet wird (**Abbildung 34**). Die ersten beide Schritte sind identisch. Im letzten Schritt wird nun $A \setminus (\text{Bloom}(A) \cap B)$ berechnet, was zu $A \setminus B$ äquivalent ist (Von A werden nur die Elemente entfernt, die A und B gemeinsam haben).

Problematisch an dieser Lösung ist, dass der Schnitt $A \cap B$ größer als das Differenz $A \setminus B$ sein kann - dennoch wird der Schnitt über das Netzwerk übertragen. Auf Grund der falschen Elemente in Bloom(A) scheint aber keine bessere Lösung machbar.

Die Verallgemeinerung auf $(A \cap \dots \cap M) \setminus (N \cap \dots \cap Z)$ ist auch hier wieder möglich. In diesem Fall muss die Berechnung aber auf zwei Pfade aufgeteilt werden:

$$\begin{aligned}
 \mathcal{N}_A &\rightarrow \mathcal{N}_B : \mathcal{B}(A) \\
 \mathcal{N}_B &\rightarrow \mathcal{N}_C : \mathcal{B}(A \cap B) \\
 &\vdots \\
 \mathcal{N}_M &\rightarrow \mathcal{N}_N : \mathcal{B}(A \cap \dots \cap L) \cap M \\
 &\vdots \\
 \mathcal{N}_Z &\rightarrow \mathcal{N}_Y : \mathcal{B}(A \cap \dots \cap Y) \cap Z \\
 \mathcal{N}_Y &\rightarrow \mathcal{N}_X : \mathcal{B}(A \cap \dots \cap Y) \cap Z \cap Y \\
 &\vdots \\
 \mathcal{N}_O &\rightarrow \mathcal{N}_N : \mathcal{B}(A \cap \dots \cap Y) \cap Z \cap \dots \cap O \\
 \mathcal{N}_N &\rightarrow \mathcal{N}_A : \mathcal{B}(A \cap \dots \cap Y) \cap Z \cap \dots \cap O \cap N = \alpha \\
 \hline
 \mathcal{N}_M &\rightarrow \mathcal{N}_L : \mathcal{B}(A \cap \dots \cap L) \cap M \\
 &\vdots \\
 \mathcal{N}_B &\rightarrow \mathcal{N}_A : \mathcal{B}(A \cap \dots \cap L) \cap M \cap \dots \cap B = \beta \\
 \mathcal{N}_A & : (\beta \cap A) \setminus \alpha
 \end{aligned}$$

Die getrennte Berechnung von $A \cap \dots \cap M$ und $N \cap \dots \cap Z$ dagegen macht keinen Sinn, da die beiden Schnitte jeweils sehr groß sein können (Stattdessen wird der gemeinsame Schnitt aller beteiligten Mengen übertragen, wie dies bei $A \setminus B$ der Fall war).

5.3.3 Vereinigung

Bei der Vereinigung ist die Optimierung schwierig, da alle Elemente von $A \cup B$ zumindest einmal über das Netzwerk geschickt werden müssen.

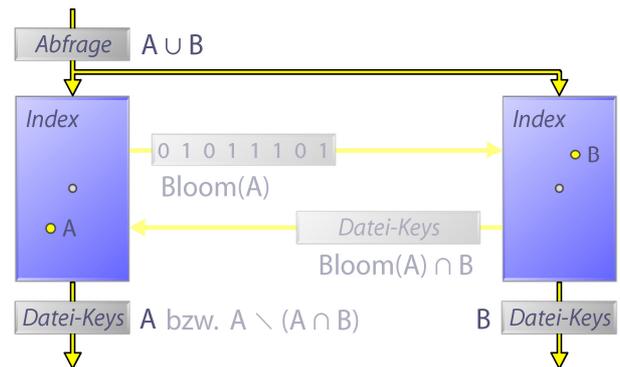


Abbildung 35: Vereinigung mit Bloom-Filtern

Die einfachste Implementierung besteht darin, dass beide Knoten A und B jeweils getrennt zum Empfänger schicken und dieser beide Mengen vereinigt (**Abbildung 35** ohne den transparenten Teil). Dadurch werden aber gemeinsame Elemente beider Mengen doppelt übertragen, was im schlimmsten Fall die Datenmenge verdoppeln kann.

Dieser Ansatz ist aber auch kaum verbesserbar, da die gemeinsamen Elemente zuerst über das Netzwerk transferiert werden müs-

sen, wenn sie in einem der beiden Knoten aussortiert werden sollen. Dies führt wieder zu einer doppelten Übertragung.

Dennoch ist dieses Prinzip nützlich, wenn der Empfänger der Nachricht mit einer geringen Bandbreite an das Netzwerk angebunden ist - beispielsweise über ein Modem (Falls die *Abfrage* aus keinen weiteren Operationen besteht, ist der Empfänger der Nachricht gleichzeitig der Sender der *Abfrage* und stellt diese eventuell von einem Rechner aus, der nicht aktiver Teil des CAN ist). Der transparente Teil von **Abbildung 35** berechnet zuerst normal den *Schnitt* und schickt im linken Knoten A ohne die gemeinsamen Elemente zurück. Auch wenn insgesamt dieselbe Menge an Daten transferiert wurde, ist der Empfänger damit entlastet.

5.4 Vor- und Nachteile

Die Vorteile bzw. Stärken von *CANDy* sind damit die verschiedenen Arten der Suche und die Verknüpfung von *Unterabfragen* mit Mengenoperationen.

Auch hier bleiben einige Nachteile:

- In der oben beschriebenen Form ist die *Bereichssuche* nur begrenzt skalierbar, da Knoten in der Nähe der Wurzel des Baumes sehr viele (*Key, Value*) Paare verwalten müssen.
- Ist das Endergebnis der *Abfrage* sehr groß, so fallen trotz *Bloom-Filtern* große Mengen an zu übertragenden Daten an, die im Netzwerk für Engpässe sorgen können.

6. VERGLEICH UND ZUSAMMENFASSUNG

Abschließend folgt ein kurzer Vergleich bzw. eine Gegenüberstellung der beiden Techniken, die in 4 und 5 beschrieben wurden.

Gemeinsam ist beiden, dass sie Bereichsabfragen bzw. Mehrbereichsabfragen ermöglichen. Dennoch liegen die Schwerpunkte auf unterschiedlichen Merkmalen.

Die erste Technik, *Bereichsabfragen mit Hilbertfunktionen*, hat folgende besondere Eigenschaften:

- *Teilintervalle*, die besonders dicht mit *Attributwerten* belegt sind, können dynamisch angepasst werden. Das heißt, das *Teilintervall* lässt in zwei oder mehr *Teilintervalle* aufteilen.
- Regelmäßige Updates von *Attributwerten* werden unterstützt. Dies schließt auch sehr häufige Updates mit ein, beispielsweise in 5 Minuten Intervallen. Abfragen ohne exakte Ergebnisse lassen sich zur Optimierung einsetzen.

Die zweite Technik, *CANDy*, weist folgende Besonderheiten auf:

- Neben der *Bereichssuche* sind außerdem die *Suche auf Wortmengen* und die *Präfixsuche* möglich. Prinzipiell können dabei auch zwei Suchmethoden für ein *Attribut* verwendet werden, um die *Abfragen* flexibler gestalten zu können.
- *Unterabfragen* lassen sich mit Mengenoperationen verknüpfen. *Bloom-Filter* sorgen dafür, dass dies auf sehr effiziente Weise geschieht.

Beide Techniken stellen real einsetzbare Verfahren zur Verfügung, die CAN um verschiedene Abfragetypen erweitern und gleichzeitig entscheidende Vorteile von CAN beibehalten - wie die Skalierbarkeit und die Ausfallsicherheit. Darüber hinaus können Teile beider Techniken kombiniert werden, wodurch beispielsweise komplexe Mengenoperationen gleichzeitig mit regelmäßigen Updates möglich sind. Allerdings muss dabei beachtet werden, dass die erste

Technik speziell auf *Computational Grids* und die zweite auf das *File Sharing* optimiert ist, wodurch zum Beispiel *Abfragen* ohne exakte Ergebnisse für das *File Sharing* ungeeignet sein können.

In allen Fällen wurde dabei besonders auf Effizienz geachtet, sowohl bei der Speicherung als auch bei der Suche und Weiterleitung von Ergebnissen.

LITERATUR

- [1] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S. A Scalable Content-Addressable Network. ACM Press, New York, NY, USA, 2001.
- [2] Andrzejak, A., Xu, Z. Scalable, Efficient Range Queries for Grid Information Services. HP Laboratories, Palo Alto, CA, USA, 2002.
- [3] Globus: Information Infrastructure in the Globus Toolkit. <http://www.globus.org/toolkit/information-infrastructure.html>.
- [4] Balaton, Z. Grid Information and Monitoring Systems. In Proceedings of Miniszimpózium (2004), 40-41. www.mit.bme.hu/events/minisy2004/papers/ms04_zoltan_BALATON.pdf.
- [5] Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I. Looking Up Data in P2P Systems. Communications of the ACM Vol. 46, No. 2 (2003), 43-48.
- [6] Schwarz, J., Sörmann, G. Kompressionsalgorithmen. Seminararbeit (1995). http://www.ztt.fh-worms.de/de/seminararbeiten/ws95_96/kompressionsalgorithmen/node21.html.
- [7] Duden. Informatik, ein Fachlexikon für Studium und Praxis. Dudenverlag, Mannheim, 2001.
- [8] Xu, Z., Mahalingam, M., Karlsson, M. Turning Heterogeneity into an Advantage in Overlay Routing. HP Laboratories, Palo Alto, CA, USA, 2003.
- [9] Bauer, D., Hurley, P., Pletka, R., Waldvogel, M. Bringing Efficient Advanced Queries to Distributed Hash Tables. In Proceedings of IEEE LCN (2004), 6-14.
- [10] Duden. Rechnen und Mathematik. Dudenverlag, Mannheim, 2000.
- [11] Ghostscript. <http://www.ghostscript.com>.
- [12] Harren, M., Hellerstein, J. M., Huebsch, R., Loo, B. T., Shenker, S., Stoica, I. Complex Queries in DHT-based Peer-to-Peer Networks. Lecture Notes in Computer Science, Vol. 2429, 242-259. Springer Verlag, London, UK, 2002.
- [13] Waldvogel, M., Varghese, G., Turner, J., Plattner, B. Scalable High-Speed Prefix Matching. ACM Press, New York, NY, USA, 2001.
- [14] Tanenbaum, A. S. Computer Networks, Fourth Edition. Prentice Hall, Upper Saddle River, New Jersey, USA, 2003.
- [15] Broder, A., Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. In Proceedings of the 40th Annual Allerton Conference on Communications, Control and Computing (2002), 636-646.