

Algorithmische Grundlagen verteilter Speichersysteme

Friedhelm Meyer auf der Heide
Christian Scheideler

Algorithmische Herausforderungen beim Entwurf verteilter Speichersysteme

Ein verteiltes Speichersystem in seiner einfachsten Art – statisch und uniform – besteht aus einer Anzahl n von gleich großen Speichern. Auf diesen Speichern ist eine Menge von m Datenobjekten so zu verteilen, dass sie die Speicher möglichst gleichmäßig auslasten. Dabei ist von zentraler Bedeutung, die maximale Last möglichst nah bei der durchschnittlichen Last von m/n Daten pro Speicher zu halten. Zudem müssen sich die Anfragen möglichst gleichmäßig auf die Speicher verteilen, und es muss für ein gegebenes Datenobjekt effizient zu berechnen sein, in welchem Speicher es abgelegt ist. Im folgenden Abschnitt werden wir uns mit randomisierten Datenverteilungen befassen, die auf universellem Hashing aufbauen. Randomisierte Verteilungsstrategien haben sich hier als überlegen gegenüber deterministischen Strategien gezeigt, da überlicherweise keinerlei Koordinierungsaufwand zwischen den Speichermedien notwendig ist, um Daten und Anfragen gleichmäßig zu verteilen. Deterministische Strategien hingegen können mit ungünstigen Daten- oder Anfrageverteilungen konfrontiert werden, die Anpassungen der Datenplatzierung im laufenden Betrieb erfordern. Wie wir sehen werden, können randomisierte Strategien die maximale Last mit hoher Wahrscheinlichkeit (oder formal ausgedrückt mit Wahrscheinlichkeit mindestens $1 - 1/n^c$ für eine beliebige Konstante $c > 1$) sehr nah bei m/n halten, und zwar deutlich besser, als eine einfache zufällige Verteilung der Daten dazu in der Lage wäre.

In den darauf folgenden Abschnitten untersuchen wir verteilte Speichersysteme, die sich durch

folgende zwei Eigenschaften von den oben angesprochenen statischen, uniformen Systemen unterscheiden:

- Sie sind *dynamisch*, d. h. zur Laufzeit können neue Speicher hinzugefügt oder vorhandene entfernt werden.
- Sie sind *nichtuniform (heterogen)*, d. h. die Speicher dürfen unterschiedliche Kapazität haben.

In diesen Speichersystemen steht zwar immer noch im Vordergrund, die Daten möglichst fair auf die Speicher zu verteilen, aber von zentraler Bedeutung ist jetzt die Adaptivität der Speichermethoden, d. h. ihre Anpassungsfähigkeit an Veränderungen im Speichersystem. Dazu werden wir genaue Qualitätsmaße in diesem Abschnitt einführen. Danach werden wir zunächst ein Verfahren, bekannt als *konsistentes Hashing*, vorstellen, das für uniforme Speichersysteme anwendbar ist, und darauf aufbauend ein Verfahren namens *SHARE* vorstellen, das auch für beliebige nichtuniforme Speichersysteme funktioniert. Wir werden auch kurz auf alternative Verfahren eingehen, die in diesem Kontext entwickelt worden sind. Im Abschnitt „Vernetzung dynamischer Speichersysteme“ schließlich

DOI 10.1007/s00287-010-0470-2
© Springer-Verlag 2010

Friedhelm Meyer auf der Heide
Heinz Nixdorf Institut und Institut für Informatik,
Universität Paderborn,
Fürstenallee 11, 33102 Paderborn
E-Mail: fmadh@upb.de

Christian Scheideler
Institut für Informatik, Universität Paderborn,
Fürstenallee 11, 33102 Paderborn
E-Mail: scheideler@upb.de

Zusammenfassung

Die Verwaltung von und der effiziente Zugriff auf Daten aus einer riesigen Datenmenge führen klassische Speichersysteme wegen ihrer beschränkten Kapazität und I/O-Fähigkeit an ihre Grenzen. Einen Ausweg zeigen *verteilte* Speichersysteme wie z. B. Storage Area Networks (SANs) auf. Solche Systeme bestehen aus vielen, durchaus auch unterschiedlichen, über ein Netzwerk verbundenen Speichersystemen und sind bei wachsenden Datenmengen erweiterbar. Bei der Entwicklung solcher Systeme stellen sich interessante algorithmische Fragen: Wie werden die Daten im verteilten Speichersystem so verteilt, dass die Speicher gleichmäßig ausgelastet werden, und zwar sowohl bezüglich der Menge der zu verwaltenden Daten als auch hinsichtlich der Anfragen? Wie sehen derartige Verteilungen bei Speichern sehr unterschiedlicher Kapazität aus? Wie kann sich die Datenverteilung an Veränderungen des Systems, etwa das Einfügen oder Entfernen von Speichern, anpassen? Wie kann sich das Netzwerk an die Veränderung des Systems anpassen? In diesem Artikel geben wir einen Überblick über einige Aspekte der algorithmischen Forschung über verteilte Speichersysteme.

diskutieren wir skalierbare Methoden, um die Speichersysteme miteinander zu vernetzen und diese Vernetzung unter Dynamik aufrechtzuerhalten.

Hashing-Verfahren für uniforme, statische Systeme

Die grundlegende Idee beim Einsatz von Hashing-Verfahren in Speichersystemen ist es, eine zeit- und speichereffiziente Funktion, die sogenannte *Hash-Funktion*, zu verwenden, um jedem Datenobjekt einen Speicher zuzuordnen. Wir gehen davon aus, dass unsere Datenobjekte durch Adressen aus einem Adressraum $U = \{1, \dots, M\}$ identifiziert werden. Sei $S \subseteq U$ eine solche Datenmenge, die über n Speicher verteilt werden soll. Diese Speicher seien der Einfachheit halber von 1 bis n durchnummeriert. Natürlich ist es im Prinzip kein Problem, die Datenobjekte in S gleichmäßig auf die Speicher zu verteilen. Diese Orte müssen allerdings in einer geeigneten Suchstruktur vermerkt werden, damit die

Daten schnell wiedergefunden werden können. Speichert man hierbei explizit für jedes Datum seinen Ort, kann das zu einer sehr speicheraufwändigen Suchstruktur führen, die nicht mehr vollständig im RAM der Server gespeichert werden kann. Viel effizienter ist hier die Verwendung kompakter und schnell auswertbarer Hash-Funktionen, die jedem Datenobjekt aus U einen Speicher zuordnen, also Hash-Funktionen der Form $h : U \rightarrow \{1, \dots, n\}$. Allerdings muss einige Sorgfalt in die Wahl einer geeigneten Hash-Funktion gelegt werden. Wir könnten z. B. die Hash-Funktion $h(x) = (x \bmod n) + 1$ nutzen. Diese verteilt zwar die gesamte Menge U gleichmäßig, aber z. B. die Menge aller Vielfachen von n in U extrem ungleichmäßig; alle diese Elemente werden in Speicher 1 abgelegt. Derartige „schlechte Mengen“ lassen sich auch mit anderen Hash-Funktionen nicht vermeiden. Wir können allerdings bei zufälliger Wahl einer Hash-Funktion aus einer geeigneten Klasse von schnell auswertbaren Hash-Funktionen dafür sorgen, dass jede Menge S nur mit sehr kleiner Wahrscheinlichkeit schlecht verteilt wird. Das Konzept des universellen Hashing liefert genau den Kompromiss zwischen „genügend zufällig“ und „auf wenig Platz speicherbar und effizient auswertbar“, den wir hier benötigen. Es wurde von Carter und Wegman [5] vorgestellt. Seitdem sind universelle Hash-Klassen gefunden worden, die bei konstanter Auswertungszeit die Eigenschaft haben, dass eine zufällig aus ihnen gewählte Funktion sich in vielerlei Hinsicht vergleichbar zu einer zufälligen Funktion verhält. Im Folgenden werden wir der Einfachheit halber immer annehmen, dass wir eine echt zufällige Hash-Funktion haben.

Falls $h : U \rightarrow \{1, \dots, n\}$ eine uniform zufällig gewählte Hash-Funktion und $|S| = m$ ist, ist es einfach zu sehen, dass für ein beliebiges, unabhängig von h gewähltes $S \subseteq U$ der Größe m die erwartete Last von Speicher i , d. h. die Anzahl der Daten in Speicher i , genau m/n ist. Wir interessieren uns im Folgenden für die erwartete *maximale* Last. Es ist einfach einzusehen, dass diese $O(m/n)$ ist, sobald $m = \Omega(n \log n)$ gilt. Eine genauere Analyse mittels Chernoff-Schranken [10] ergibt, dass die maximale Last eines Speichers bei Verwendung einer zufälligen Hash-Funktion höchstens

$$\frac{m}{n} + O\left(\sqrt{\frac{m}{n} \log n} + \frac{\log n}{\log \log n}\right)$$

mit hoher Wahrscheinlichkeit ist. Diese Schranke liefert asymptotisch optimale Last $\frac{m}{n} + o(\frac{m}{n})$ für $m = \omega(n \log^2 n)$. Die recht hohe Abweichung von m/n für kleine m kann man durch zwei Methoden deutlich verkleinern. Beide beruhen auf der Idee der Nutzung von Alternativen bei der Datenverteilung (vgl. [1, 9]). Der folgende Trick von Azar et al. [1] liefert asymptotisch optimale Last für den Fall $m = \omega(n \log \log n)$.

Wir nutzen zwei zufällige und unabhängig gewählte Hash-Funktionen h_1 und h_2 . Die Daten werden nacheinander in die Speicher eingefügt. Für jedes Datum x überprüfen wir die momentane Anzahl der Daten in den Speichern $h_1(x)$ und $h_2(x)$ und platzieren x in denjenigen der beiden Speicher, der zurzeit am wenigsten Daten enthält.

Dieser Trick heißt auch *Minimum-Regel*. Sie hat die Eigenschaft, dass für jede Teilmenge $S \subseteq U$ der Größe m die maximale Last höchstens

$$\frac{m}{n} + O(\log \log n)$$

mit hoher Wahrscheinlichkeit ist [14].

Für $m = o(n \log \log n)$ empfiehlt es sich, die Kollisionsmethode von Dietzfelbinger und Meyer auf der Heide zu verwenden [7, 12], die bis auf einen Faktor 3 eine optimale maximale Last für *alle* m liefert:

Wir nutzen wieder zwei zufällige und unabhängig gewählte Hash-Funktionen h_1 und h_2 . Solange noch nicht alle Daten platziert sind, platziere alle noch übriggebliebenen Daten $d \in S$ in den Speichern $h_1(d)$ und $h_2(d)$. Für jeden Speicher, bei dem die dadurch erzeugte Last höchstens $3\lceil m/n \rceil$ ist, belasse alle Daten dort. Die restlichen Daten (d. h. diejenigen Daten, für die sowohl $h_1(d)$ als auch $h_2(d)$ eine Last über $3\lceil m/n \rceil$ haben), nehmen an der nächsten Platzierungsrunde teil.

Wenn die Kollisionsmethode terminiert, hat jeder Speicher garantiert eine Last von höchstens $3\lceil m/n \rceil$. Allerdings ist es nicht klar, ob die Kollisionsmethode tatsächlich terminiert. Es konnte jedoch nachgewiesen werden, dass die Kollisionsmethode mit hoher Wahrscheinlichkeit nach $O(\log \log n)$ Platzierungsrunden terminiert [7, 12].

Falls die Daten auch nach ihrer Platzierung noch bewegt werden dürfen, können wir sogar eine noch bessere Verteilung mithilfe der Strategie in Abb. 1 erreichen. Für jede Teilmenge $S \subseteq U$ der Größe m gilt dabei, dass wenn die Selbstbalancierung genügend

wähle zufällig zwei Speicher v_1 und v_2
falls es ein Datum in v_1 mit alternativem
Ort v_2 gibt, dann
nimm ein beliebiges Datum d mit dieser
Eigenschaft
platziere d im Speicher mit der
geringsten Last (von v_1 und v_2)
falls beide dieselbe Last haben, dann
platziere d in einem zufälligen der
beiden Speicher

Abb. 1 Die Selbstbalancierungsstrategie

lange läuft, dann die maximale Anzahl der Daten in einem Speicher höchstens

$$\lceil m/n \rceil + 1$$

mit hoher Wahrscheinlichkeit ist [6].

Zwei zufällige Hash-Funktionen reichen also, um eine beliebige Teilmenge in U nahezu perfekt auf die Speicher zu verteilen. Wirklich zufällige Hash-Funktionen sind nicht speichereffizient konstruierbar, aber in der Praxis zeigen kryptografische Hash-Funktionen wie z. B. SHA-1 ein Verteilungsverhalten, das denen zufälliger Hash-Funktionen sehr nahekommt. Statisches Hashing (d. h. die Anzahl der Speicher ist fest) arbeitet also sehr effektiv. Aber was können wir tun, wenn sich die Speichermenge über die Zeit verändert? Dazu brauchen wir geeignete dynamische Hashing-Verfahren.

Hashing für dynamische Speichersysteme

In einem verteilten Speichersystem kann sich die Menge und die Kapazität der verfügbaren Speicher stark über die Zeit ändern. Daher benötigen wir Hashing-Strategien, die sich effizient an diese Veränderungen anpassen können. Der naive Ansatz wäre, einfach eine neue Hash-Funktion bei jeder Veränderung zu wählen, aber das würde bedeuten, dass fast alle Daten umverteilt werden müssen, was für große verteilte Systeme nicht verkraftbar ist. Wie wir sehen werden, gibt es viel bessere Strategien, aber zunächst müssen wir uns Gedanken darüber machen, wie wir die Qualität einer dynamischen Hashing-Methode messen wollen.

Sei $V = \{1, \dots, N\}$ die Menge der möglichen Identifikationsnummern (z. B. IP-Adressen) für die

Speicher und $U = \{1, \dots, M\}$ der Adressraum für die Daten. Angenommen, die aktuelle Anzahl der Daten sei $m \leq M$ und die Anzahl der Speicher sei $n \leq N$. Zur Vereinfachung werden wir im Folgenden oft annehmen, dass die Daten und Speicher durchgehend von 1 an durchnummeriert sind, aber im Prinzip funktioniert jede eindeutige Nummerierung mit den unten angegebenen Strategien. Sei $c_i \in [0, 1]$ die aktuelle (relative) *Kapazität* von Speicher i , d. h. c_i gibt den Bruchteil der Kapazität bezogen auf das Gesamtsystem an, das Speicher i zur Verfügung stellt. Dann gilt $\sum_{i=1}^n c_i = 1$, und (c_1, \dots, c_n) nennen wir die aktuelle *Kapazitätsverteilung* des Systems.

Das Speichersystem mag sich nun auf vielfältige Weise ändern. Es können neue Daten hinzukommen und alte gelöscht werden, und die Menge der verfügbaren Speicher oder deren Kapazitäten kann sich verändern. In diesem Fall muss eine geeignete dynamische Hashing-Methode mehrere Kriterien erfüllen:

1. *Fairness*: Ein Speicherschema ist *fair*, falls für jeden Speicher i die (erwartete) Anzahl der Daten in i zwischen $\lfloor (1 - \epsilon)c_i \cdot m \rfloor$ und $\lceil (1 + \epsilon)c_i \cdot m \rceil$ liegt, wobei $\epsilon > 0$ eine beliebig kleine Konstante sein kann.
2. *Redundanz*: Wir nennen ein Schema *r-redundant*, falls es jedem Datum r verschiedene Speicher zuweist, in dem seine Bruchstücke bzw. Kopien abgelegt werden können.
3. *Effizienz*: Ein Speicherschema ist *effizient*, falls die Datenstruktur zur Bestimmung der Position eines Datums nur wenig Platz verbraucht (der von N und m höchstens logarithmisch abhängen sollte) und die Position eines Datum in kurzer Zeit bestimmt werden kann.
4. *Adaptivität*: Wir nennen ein Speicherschema *adaptiv*, falls für den Fall einer Veränderung in der Menge der Speicher oder deren Kapazitäten die Umverteilungen der Daten, die notwendig sind, um die Fairness und Redundanz wieder herzustellen, möglichst gering sind. Dafür verwenden wir die kompetitive Analyse. Angenommen, es gibt eine Kapazitätsveränderung von (c_1, \dots, c_n) nach (c'_1, \dots, c'_n) . In diesem Fall benötigt eine optimale, perfekt faire Strategie (d. h. $\epsilon = 0$)

$$\sum_{i: c_i > c'_i} (c_i - c'_i) \cdot m$$

Umplatzierungen von Daten. Wir nennen eine Speicherstrategie also *c-kompetitiv*, falls sie höchstens c -mal so viele Umplatzierungen für jede solche Kapazitätsveränderung benötigt, um zur Fairness und Redundanz zurückzukehren.

Konkret heißt das für die Adaptivität, dass wenn die Kapazitätsverteilung sich z. B. von $(1/2, 1/2, 0)$ nach $(0, 1/2, 1/2)$ verändert (Speicher 1 verlässt das System und Speicher 3 wird hinzugefügt), dann muss bestenfalls die Hälfte der Daten umplatziert werden, um eine faire Verteilung wieder zu gewährleisten. Das ist dadurch auch machbar, dass alle Daten in Speicher 1 an Speicher 3 übergeben werden.

Dynamisches Hashing für uniforme Speichersysteme

Es gibt bereits eine Reihe von Hashing-Strategien für dynamische uniforme Speichersysteme. Zwei Beispiele dafür sind das konsistente Hashing von Karger et al. [8] und die Cut-and-Paste-Strategie von Brinkmann et al. [3]. Im uniformen Fall betrachten wir lediglich die Situation, dass neue Speicher in das System hineinkommen oder alte Speicher das System verlassen.

Konsistentes Hashing. Das konsistente Hashing arbeitet wie folgt:

Angenommen, wir haben eine zufällige Hash-Funktion $f : U \rightarrow [0, 1)$ und eine Menge unabhängiger, zufälliger Hash-Funktionen $g_1, \dots, g_k : V \rightarrow [0, 1)$, wobei k ein genügend großer Wert ist. Die Funktion f bildet die Daten auf $[0, 1)$ ab, und die Funktionen g_1, \dots, g_k bilden die Speichermedien auf $[0, 1)$ ab. Datum i wird demjenigen Speicher j zugeordnet, für den es ein ℓ gibt, sodass $g_\ell(j)$ der nächste Nachfolger von $f(i)$ für alle $g_{\ell'}(j')$ ist (wobei $[0, 1)$ als Ring angesehen wird). Siehe auch Abb. 2.

Bei 1-redundanter Speicherung der Daten, d. h. jedes Datum wird nur einem Speicher zugewiesen, ist das konsistente Hashing perfekt fair und 2-kompetitiv und benötigt bei geeigneter Datenstruktur nur erwartet konstant viel Zeit bei $O(n \log^2 N)$ Speicher, um das Speichermedium für ein Datum zu berechnen [8]. Diese Eigenschaften können auch bei r -redundanter Speicherung mit $r > 1$ beibehalten werden, falls jedes Datum i den r Speichern mit den nächsten Nachfolgern $g_\ell(j)$ zu $f(i)$ zugeordnet wird.

Man mag denken, dass sich das konsistente Hashing einfach auf den nichtuniformen Fall erweitern

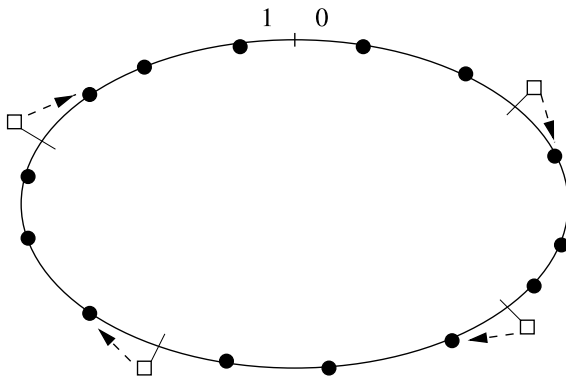


Abb. 2 Das konsistente Hashing

lässt, indem man Speichern mit höherer Kapazität mehr Punkte in $[0, 1)$ erlaubt. Allerdings würde das $\Omega(\min[c_{\max}/c_{\min}, m])$ Punkte benötigen, um fair zu sein, wobei c_{\max} die maximale Kapazität und c_{\min} die minimale Kapazität eines Speichers ist. Im schlimmsten Fall kann das bedeuten, dass ein einzelner Speicher bis zu $\Theta(m)$ Punkte haben kann, was unsere Anforderungen an die Speichereffizienz deutlich verletzen würde.

Dynamisches Hashing für nichtuniforme Speichersysteme

Es gibt bereits verschiedene Hashing-Verfahren, die auch für nichtuniforme dynamische Speichersysteme effizient anwendbar sind. Dazu gehören SHARE, SIEVE [4], Trichter-Strategien [15], Redundantes SHARE [2] und SPREAD [11]. Während SHARE, SIEVE und die Trichter-Strategien nur die Fairness bei beliebigen Kapazitätsverteilungen garantieren können, können Redundantes SHARE und SPREAD sowohl die Fairness als auch eine redundante Speicherung der Daten garantieren, solange das prinzipiell möglich ist. Wir werden exemplarisch für diese Strategien die SHARE-Strategie vorstellen.

Die SHARE-Strategie. Die SHARE-Strategie baut auf konsistentem Hashing auf. Sie besitzt (neben den Hash-Funktionen für das konsistente Hashing) zwei Hash-Funktionen: eine Hash-Funktion $h: U \rightarrow [0, 1)$, um jedem Datum einen Punkt in $[0, 1)$ zuzuweisen und eine Hash-Funktion $g: V \rightarrow [0, 1)$, um jedem Speicher ein Intervall in $[0, 1)$ zuzuweisen. SHARE arbeitet wie folgt:

Angenommen, die momentane Kapazitätsverteilung sei (c_1, \dots, c_n) . Jedem Speicher i wird ein Intervall I_i der Länge $s \cdot c_i$ zugewiesen, wobei s ein fes-

ter *Dehnungsfaktor* ist. I_i startet im Punkt $g(i)$ und endet in $(g(i) + s \cdot c_i) \bmod 1$, wobei $[0, 1)$ als Ring angesehen wird. Ist $s \cdot c_i > 1$, dann bedeutet das, dass das Intervall $\lceil s \cdot c_i \rceil$ -mal um $[0, 1)$ herumgewickelt ist. Zur Vereinfachung der Präsentation werden wir für $\lceil s \cdot c_i \rceil > 1$ annehmen, dass I_i aus $\lceil s \cdot c_i \rceil$ Intervallen $I_{i,j}$ besteht, wobei $\lceil s \cdot c_i \rceil$ von diesen die Länge 1 haben und das letzte Intervall die Restlänge übernimmt.

Für jedes $x \in [0, 1)$ sei $C_x = \{i \mid x \in I_i\}$ und $c_x = |C_x|$. Da die Gesamtanzahl der Endpunkte der Intervalle höchstens $2(n + s)$ ist, muss das $[0, 1)$ -Intervall in höchstens $2(n + s)$ Rahmen $F_j \subseteq [0, 1)$ aufgeteilt werden, sodass für jeden Rahmen F_j gilt, dass C_x dieselbe Menge für jedes $x \in F_j$ ist. Das ist wichtig, um sicherzustellen, dass die Datenstruktur für SHARE eine geringe Speicherkomplexität hat. Die Berechnung der Position eines Datums d erfolgt einfach dadurch, dass zunächst die Speicher Menge $C_{h(d)}$ bestimmt wird und dann auf $C_{h(d)}$ das konsistente Hashing angewendet wird, um zu bestimmen, welcher der Speicher in $C_{h(d)}$ für d zuständig ist (siehe auch Abb. 3).

Unter der Annahme, dass $s = \Theta(\log N)$ genügend groß ist (damit jeder Punkt in $[0, 1)$ durch ungefähr gleich viele Intervalle überdeckt ist), ist SHARE fair und 2-kompetitiv und benötigt bei geeigneter Datenstruktur nur erwartet konstant viel Zeit bei $\Theta(n \log^2 N)$ Speicher, um das Speichermedium für ein Datum zu berechnen [4].

Eine wichtige Eigenschaft der SHARE-Strategie ist, dass sie *vergesslich* ist, d. h. die Datenverteilung hängt nur von der aktuellen Knotenmenge ab und *nicht* davon, wie diese Knotenmenge zustande gekommen ist. Das vereinfacht die Verwaltung der Daten. Einige andere Strategien wie die Cut-and-Paste-Strategie [3] und die SIEVE-Strategie [4]

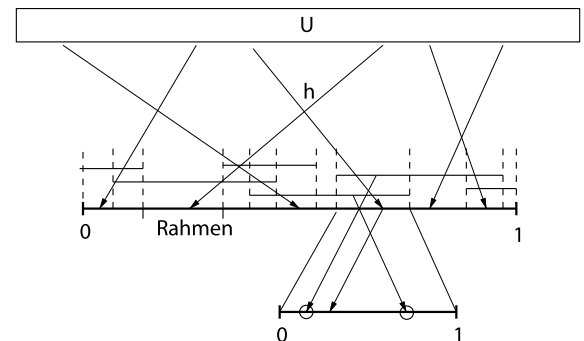


Abb. 3 Illustration des SHARE-Algorithmus

von Brinkmann et al. erfüllen diese Eigenschaft nicht. Leider kann SHARE nicht ohne weiteres auf eine redundante Datenspeicherung erweitert werden. Betrachten wir z. B. die Kapazitätsverteilung $(1/2, 1/4, 1/4)$ mit Redundanz $r = 2$. In diesem Fall sollte für alle Daten gelten, dass die beiden Kopien entweder in Speicher 1 und 2 oder in Speicher 1 und 3 abgelegt werden, um eine faire Speicherauslastung sicherzustellen. Das kann die SHARE-Strategie nicht leisten, da sie auch Kopien auf Speicher 2 und 3 für einige Daten ablegt.

Vernetzung dynamischer Speichersysteme

Bis jetzt haben wir uns noch nicht damit beschäftigt, wie wir die Speichersysteme vernetzen können, um schnelle Zugriffe darauf zu erlauben und sich schnell an eine Veränderung in der Speichermenge anpassen zu können. Für uniforme Speichersysteme hat sich der kontinuierlich-diskrete Ansatz als sehr hilfreich erwiesen.

Der kontinuierlich-diskrete Ansatz

Die grundlegende Idee hinter dem kontinuierlich-diskreten Ansatz [13] ist es, eine klassische Familie von Graphen in ein kontinuierliches Modell umzuformen und dieses dann zur Vernetzung einer dynamischen Menge von Speichereinheiten anzuwenden. Ein wohlbekanntes Peer-to-Peer-System, das auf ähnlichen Ideen aufbaut, ist z. B. Chord [16].

Betrachte einen beliebigen Raum U . Angenommen, wir haben eine (potenziell unendliche) Menge F an Funktionen $f_i: U \rightarrow U$. Sei

$$E_F = \{(x, y) \in U^2 \mid x = y \text{ oder es gibt ein } i: y = f_i(x)\}.$$

Dann kann (U, E_F) als ungerichteter Graph auf einer unendlichen Menge an Knoten, also als kontinuierliches Graphenmodell, angesehen werden. Für die Menge $S \subseteq U$ sei $\Gamma(S) = \{y \in U \mid \exists x \in S: (x, y) \in E_F\}$ die Nachbarschaftsmenge von S . Falls $\Gamma(S) \neq \emptyset$ für jedes $S \subset U$ ist, dann sagen wir, dass F U mischt. Falls F U nicht mischt, dann gibt es nichtzusammenhängende Bereiche in U .

Betrachte nun eine Menge von Speichern V und sei $R(v)$ der Unterraum von U , der Speicher v zugeordnet wird. Wir verwenden dann die folgende *kontinuierlich-diskrete Verbindungsregel*, um die Speicher miteinander zu vernetzen:

Für jedes Speicherpaar v und w ist v verbunden mit w genau dann, wenn es Punkte $x, y \in U$ gibt mit

$x \in R(v)$, $y \in R(w)$ und entweder $(x, y) \in E_F$ oder $(y, x) \in E_F$.

Sei $G_F(V)$ der aus dieser Bedingung hervorgehende Graph. Dann ist leicht zu zeigen, dass falls F U mischt und $\cup_v R(v) = U$, dann ist $G_F(V)$ stark zusammenhängend. Daher ist es wichtig sicherzustellen, dass F U mischt und dass $\cup_v R(v) = U$. Wir müssen uns also Gedanken um eine geeignete Auswahl von (U, F) und einer geeigneten Partitionierung von U machen. Falls $U = [0, 1)$ ist, dann können wir für die Partitionierung das konsistente Hashing anwenden, das ja jeder Speichereinheit Intervalle in U zur Speicherung von Daten so zuordnet, dass das gesamte U abgedeckt ist. Das hat den folgenden Vorteil:

Wenn ein neuer Speicher i in das System integriert werden muss, dann müssen maximal k alte Speicher (bei k Hash-Funktionen g_1, \dots, g_k) kontaktiert werden (nämlich die Speicher, von denen er Intervallstücke übernehmen wird), um eine Aufteilung von $[0, 1)$ gemäß des konsistenten Hashings zu bewahren und die entsprechenden Daten zu bekommen. Weiterhin kennen diese alten Speicher aufgrund der kontinuierlich-diskreten Verbindungsregel alle Verbindungen, die Speicher i in $G_F(V)$ eingehen muss und können ihn entsprechend informieren.

Also nicht nur speichertechnisch, sondern auch verbindungstechnisch ist es sehr einfach, mithilfe der kontinuierlich-diskreten Methode und konsistentem Hashing einen neuen Speicher in das System zu integrieren.

Auch der Fall, dass ein Speicher das System verlässt, ist sehr einfach zu handhaben. Dieser Speicher muss lediglich seine Daten und Verbindungen an maximal k andere Speicher weitergeben, nämlich diejenigen Speicher, die nach dem konsistenten Hashing die Intervalle des rausgehenden Speichers beerben. Zum Abschluss geben wir ein konkretes Beispiel für F .

Der dynamische de Bruijn-Graph

Der d -dimensionale de Bruijn-Graph $DB(d)$ ist ein ungerichteter Graph $G = (V, E)$ mit Knotenmenge $V = \{v \in \{0, 1\}^d\}$ und Kantenmenge E , die alle Kanten $\{v, w\}$ enthält, mit der Eigenschaft, dass $v = (v_1, \dots, v_d)$ und $w = (x, v_1, \dots, v_{d-1})$ für ein $x \in \{0, 1\}$. Wenn wir einen Knotennamen (x_1, \dots, x_d) als die Zahl $y = \sum_{i \geq 1} x_i / 2^i \in [0, 1)$ interpretieren, dann gilt für $d \rightarrow \infty$, dass

- $V = [0, 1)$ und
- $F = \{f_0, f_1\}$ mit $f_0(x) = x/2$ und $f_1(x) = (1 + x)/2$ die de Bruijn-Verbindungen in V wiedergibt.

Es kann dann für den dynamischen Fall gezeigt werden, dass das konsistente Hashing zusammen mit der kontinuierlich-diskreten Methode zu einem Speichernetzwerk mit maximalem Grad $O(\log n)$ und Durchmesser $O(\log n)$ mit hoher Wahrscheinlichkeit führt [13]. Weiterhin können effiziente Routingverfahren verwendet werden, die Botschaften von jedem Punkt zu jedem anderen Punkt des Netzwerks in $O(\log n)$ Kommunikationsschritten versenden können. Diese können dazu verwendet werden, dass beliebige Datenanfragen im Speichernetzwerk in $O(\log n)$ Kommunikationsschritten und damit effizient bedient werden können.

Zusammenfassung

Wie wir gesehen haben, können Hashing-Methoden angewendet werden, um skalierbare dynamische Speichersysteme zu realisieren. Wir haben hier nur einen winzigen Ausschnitt der aktuellen Forschung wiedergegeben. Für weitergehende Informationen stehen wir gerne zur Verfügung.

Literatur

1. Azar Y, Broder A, Karlin A, Upfal E (1994) Balanced allocation. In: Proc of the 26th ACM Symp on Theory of Computing (STOC), pp 593–602
2. Brinkmann A, Effert S, Meyer auf der Heide F, Scheideler C (2007) Dynamic and redundant data placement. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp 29–38
3. Brinkmann A, Salzwedel K, Scheideler C (2000) Efficient, distributed data placement strategies for storage area networks. In: Proc of the 12th ACM Symp on Parallel Algorithms and Architectures (SPAA), pp 119–128
4. Brinkmann A, Salzwedel K, Scheideler C (2002) Compact, adaptive placement schemes for non-uniform capacities. In: Proc of the 14th ACM Symp on Parallel Algorithms and Architectures (SPAA), pp 53–62
5. Carter J, Wegman M (1979) Universal classes of hash functions. J Comput Sys Sci 18(2):143–154
6. Czumaj A, Riley C, Scheideler C (2004) Perfectly balanced allocation. In: 7th Int Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM), pp 240–251
7. Dietzfelbinger M, Meyer auf der Heide F (1993) Simple, efficient shared memory simulations. In: Proc of the 5th ACM Symp on Parallel Algorithms and Architectures (SPAA), pp 110–119
8. Karger D, Lehman E, Leighton T, Levine M, Lewin D, Panigrahy R (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc of the 29th ACM Symp on Theory of Computing (STOC), pp 654–663
9. Karp R, Luby M, Meyer auf der Heide F (1992) Efficient PRAM simulation on a distributed memory machine. In: Proc of the 24th ACM Symp on Theory of Computing (STOC), pp 318–326
10. McDiarmid C (1998) Concentration. In: Habib M, McDiarmid C, Ramirez-Alfonsin J, Reed B (eds) Probabilistic Methods for Algorithmic Discrete Mathematics. Springer, Berlin, pp 195–247
11. Mense M, Scheideler C (2008) SPREAD: an adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In: Proc of the 19th ACM/SIAM Symp on Discrete Algorithms (SODA), pp 1135–1144
12. Meyer auf der Heide F, Scheideler C, Stemmann V (1995) Exploiting storage redundancy to speed up randomized shared memory simulations. In: Proc of the 12th Symp on Theoretical Aspects of Computer Science (STACS), pp 267–278
13. Naor M, Wieder U (2003) Novel architectures for P2P applications: the continuous-discrete approach. In: Proc of the 15th ACM Symp on Parallel Algorithms and Architectures (SPAA), pp 50–59
14. Steger A, Berenbrink P, Czumaj A, Vöcking B (2000) Balanced allocations: The heavily loaded case. In: Proc of the 32nd ACM Symp on Theory of Computing (STOC), pp 745–754
15. Schindelhauer C, Schomaker G (2005) Weighted distributed hash tables. In: Proc of the 17th ACM Symp on Parallel Algorithms and Architectures (SPAA), pp 218–227
16. Stoica I, Morris R, Karger D, Kaashoek F, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for Internet applications. In: Proc of the SIGCOMM '01, pp 149–160