

Verwaltung kleiner Datensätze

Motivation

Amazon Dynamo

Zusammenfassung



- Charakteristika
 - Nutzdaten pro Datensatz oftmals weniger als 1 MB
 - Repräsentation als Schlüssel-Wert-Paar
- Typisches Zugriffsmuster: Atomares Schreiben und Lesen
 - Bei Modifikationen wird immer der komplette Datensatz neu geschrieben
 - Kein teilweises Lesen von Datensätzen
- Beispielanwendungen
 - Sicherung von Nutzereinstellungen
 - Speicherung von Informationen zur Verwaltung von Client-Sitzungen
- Herausforderungen
 - Welche Möglichkeiten eröffnet die geringe Größe der Datensätze?
 - Wie können Konsistenzeigenschaften konfigurierbar gemacht werden?
 - Wie lässt sich ein Datenspeichersystem inkrementell skalierbar gestalten?
 - Wie kann der Heterogenität von Hardware Rechnung getragen werden?



Amazon Dynamo

- Anwendungsbeispiel: Warenkorb
- Anforderungen
 - Inkrementelle Skalierbarkeit des Gesamtsystems
 - Hohe Verfügbarkeit der gespeicherten Daten
 - Leistungsabhängige Lastverteilung im Umfeld heterogener Hardware
- Amazon Dynamo
 - Dezentraler Peer-to-Peer-Ansatz
 - Partitionierung der Daten
 - Fehlertoleranz durch Replikation über mehrere Datenzentren
 - Konsistenzgarantie: *Letztendliche Konsistenz* (*Eventual Consistency*)

Literatur



Giuseppe DeCandia, Deniz Hastorun, Madan Jampani et al.
Dynamo: Amazon's highly available key-value store
Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07),
S. 205–220, 2007.



Anforderungsdetails

- Zu verwaltende Datensätze
 - Binärdaten (*Blobs*)
 - Weniger als 1 MB pro Schlüssel-Wert-Paar
 - Atomares Schreiben und Lesen erforderlich
- Zugriffsmuster: Schreibzugriffe wichtiger als Lesezugriffe
 - Hohe Verfügbarkeit beim Schreiben von Daten („*Always writeable*“)
 - Beispiel: Aktualisierung des Warenkorbs sollte immer möglich sein
- Dienstgüte
 - Maximalwert für die Bearbeitungsdauer von 99.9 % aller Anfragen
 - Durchschnittswert der Antwortzeiten ist zweitrangig
- Weitere Charakteristika
 - Unterschiedliche Dynamo-Instanzen für unterschiedliche Anwendungen
 - Keine Authentifizierung für Clients erforderlich



Schnittstelle

- Datentypen
 - Nutzdaten in Form von Schlüssel-Wert-Paaren aus Keys und Objects
 - Kapselung von systeminternen Metadaten in Context-Objekten
- Schreibzugriff

```
void put(Key key, Context context, Object object)
```

 - Atomares Speichern eines Schlüssel-Wert-Paars
 - Nutz- und Metadaten werden gemeinsam verwaltet
- Lesezugriff

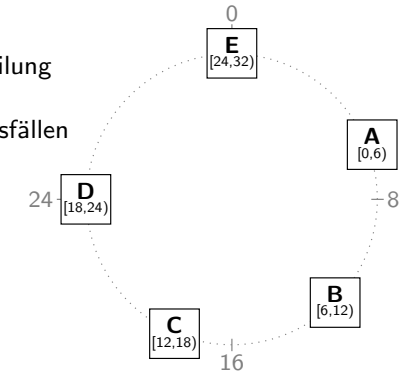
```
{List<Object>, Context} get(Key key)
```

 - Atomarer Zugriff auf Datensätze über Schlüssel
 - Rückgabewert: Tupel aus Ergebnisliste und Context-Objekt
 - Normalfall: Ein Element in Ergebnisliste
 - Ausnahmefall: Ergebnisliste enthält mehrere Elemente
- Meist geht einem Schreib- ein Lesezugriff auf dasselbe Objekt voraus



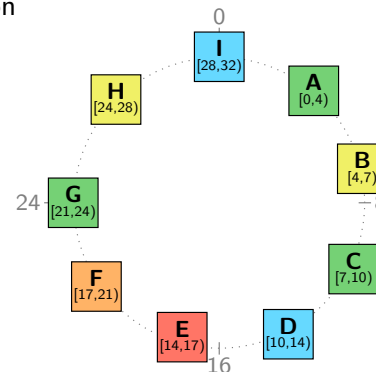
Partitionierung mittels *konsistentem Hashing*

- Zuteilung von Daten zu Rechnern (*Knoten*)
 - Anwendung einer Hash-Funktion auf die Schlüssel von Nutzdaten
 - Abbildung des Wertebereichs der Hash-Funktion auf einen Ring
 - Jeder Knoten ist für einen Teilbereich verantwortlich
 - Vorteile
 - In der initialen Konfiguration: Gleichmäßige Lastverteilung
 - Einfaches Hinzufügen bzw. Entfernen von Knoten
 - Nachteile
 - In der Praxis: Ungleichmäßige Lastverteilung
 - Ungeeignet für heterogene Knoten
 - Asymmetrische Belastung bei Knotenausfällen
- Einfügen eines Schlüssel-Wert-Paars
 1. Berechnung des Schlüssel-Hash-Werts
 2. Senden der Schreibanfrage an den Knoten an der nächsthöheren Position



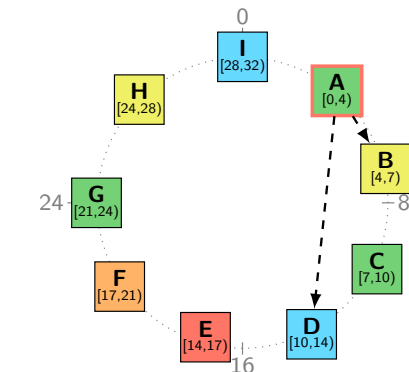
Virtuelle Knoten

- Ziele
 - Lastverteilung abhängig von der Leistungsfähigkeit der Knoten
 - Gleichmäßige Belastung bei der Tolerierung von Knotenausfällen
- Ansatz: Einsatz von *virtuellen Knoten*
 - Mehrere virtuelle Knoten pro physischem Knoten
 - Anzahl der virtuellen Knoten abhängig von der Leistungsfähigkeit eines Rechners
- Konsequenzen
 - Leistungsfähigere Rechner sind für größere Wertebereiche zuständig
 - Beim Hinzufügen und Entfernen von physischen Knoten sind mehrere Rechner beteiligt



Replikation

- Redundante Speicherung auf mehreren Knoten
 - Verwaltung von Replikatinformationen in *Präferenzlisten*
 - Von jedem Knoten im System abhängig vom Datensatzschlüssel berechenbar
 - Referenzen auf N für den Datensatz verantwortliche Replikate
 - Referenzen auf weitere Ersatzreplikate
 - Zusammenstellung der Präferenzliste
 - Für den Teilbereich des Schlüssel-Hash-Werts zuständiger Knoten
 - Nachfolger dieses Knotens im Ring
 - Auslassen von virtuellen Knoten bereits involvierter physischer Rechner
- Bearbeitung von Client-Anfragen
 - Einsatz von *Sloppy-Quoren*
 - Leseanfragen: R Replikate
 - Schreibanfragen: W Replikate
 - Wahl der Parameter
 - In der Praxis meistens $R, W < N$
 - Beispiel: $N = 3, R = 2, W = 2$



Datenzugriffe

- Allgemeines Vorgehen
 1. Auswahl eines *Koordinators* aus den ersten N Knoten der Präferenzliste
 - Per Client-Library
 - Über einen vom System bereitgestellten Load-Balancer
 2. Senden der Anfrage an den Koordinator
 3. Koordinator leitet Anfrage an die $N - 1$ anderen Knoten weiter
 4. Bearbeitung der Anfrage
 5. Koordinator sammelt Ergebnisse
 6. Koordinator antwortet dem Client sobald R bzw. W Resultate verfügbar
 - Letztendliche Konsistenz in Amazon Dynamo
 - $N - W$ Replikate bearbeiten Schreibaufwurf evtl. erst nach der Bestätigung
 - Replikate liefern möglicherweise voneinander abweichende Antworten
 - Kein global eindeutiger Koordinator pro Datensatz
 - Replikate führen Anfragen potentiell in unterschiedlicher Reihenfolge aus
 - Es existieren eventuell mehrere Versionen eines Datensatzes im System
- Strategie zur Auflösung von Konflikten (*Reconciliation*) erforderlich



Detektion von Konflikten

- Versionierung mittels *Vektoruhren* [Weiterführende Informationen in *Verteilte Systeme*.]
 - Tupel $t = \{k, z\}$: Kombination aus Knoten-ID k und Zähler z
 - Vektoruhr $V = \{t_1, \dots, t_n\}$: Vektor aus Tupeln (initial sind alle Zähler 0)
 - Definition von Relationen
 - $<$ für Tupel: $t_x < t_y \Leftrightarrow t_x.k = t_y.k \wedge t_x.z < t_y.z$
 - \leq für Tupel: $t_x \leq t_y \Leftrightarrow t_x.k = t_y.k \wedge t_x.z \leq t_y.z$
 - \prec für Vektoruhren: $V_A \prec V_B \Leftrightarrow (\forall i : t_{A,i} \leq t_{B,i}) \wedge (\exists i : t_{A,i} < t_{B,i})$
 - Einsatz von Vektoruhren als Zeitstempel für Modifikationen
- Anwendung von Vektoruhren bei Schreibanfragen
 - Koordinator
 - Empfang einer Schreibanfrage mit dem Zeitstempel (\rightarrow Context-Parameter der `put`-Methode) der letzten dem Client bekannten Version
 - Inkrementieren des Zählers im Tupel des Koordinators
 - Verteilung des aktualisierten Zeitstempels zusammen mit der Anfrage
 - Replikate
 - Falls $V_{Datensatz} \prec V_{Anfrage}$: Überschreiben der alten Version des Datensatzes
 - Sonst: Paralleles Speichern beider Versionen des Datensatzes



Detektion von Konflikten

Beispiel

- Anlegen eines Datensatzes mit dem Wert w_1
 - Koordinator: Knoten A
 - Vektoruhr der Anfrage: $\{\{A, 1\}\}$
 - Replikate: Speicherung von $(\{\{A, 1\}\}, w_1)$
- Zuweisung eines neuen Werts w_2 nach dem Lesen von w_1
 - Koordinator: Knoten A
 - Referenzzeitstempel: $\{\{A, 1\}\} \rightarrow$ Vektoruhr der Anfrage: $\{\{A, 2\}\}$
 - Replikate
 - Vektoruhrenvergleich zeigt Abhängigkeit \rightarrow Überschreiben des alten Werts
 - Speicherung von $(\{\{A, 2\}\}, w_2)$
- Zuweisung eines neuen Werts w_3 nach dem Lesen von w_1
 - Koordinator: Knoten B
 - Referenzzeitstempel: $\{\{A, 1\}\} \rightarrow$ Vektoruhr der Anfrage: $\{\{A, 1\}, \{B, 1\}\}$
 - Replikate
 - Vektoruhrenvergleich zeigt keine Abhängigkeit \rightarrow Aufheben des alten Werts
 - Speicherung von $(\{\{A, 2\}\}, w_2)$ und $(\{\{A, 1\}, \{B, 1\}\}, w_3)$



Auflösung von Konflikten

- Bearbeitung von Leseanfragen durch den Koordinator
 - Sammlung aller Datensatzversionen von R Replikaten
 - Aussortieren veralteter/abhängiger Versionen (*Syntactic Reconciliation*)
 - Inhalt der Antwortnachricht an den Client
 - Alle voneinander unabhängigen Versionen des Datensatzes
 - Kombinierte Vektoruhr als Teil des Context-Objekts
 - Auflösung des Konflikts liegt in der Verantwortung des Client
 - Auswahl der zukünftig verwendeten Version (*Semantic Reconciliation*)
 - Kombinierte Vektoruhr als Referenz für nächste Schreibanfrage
- Replikate verwerfen die alten, unabhängigen Versionen
- Anwendungsbeispiel: Warenkorb
 - Standardoperationen: Hinzufügen bzw. Herausnehmen einer Ware
 - Möglicher Konflikt: Eine Version des Warenkorbs enthält eine bereits gelöschte Ware noch, die andere nicht mehr
 - Angewandte Strategie: Wahl des Warenkorbs, der die Ware enthält



- Ausgangssituation (siehe vorheriges Beispiel)
 - Existenz zweier unabhängiger Versionen desselben Datensatzes
 - Separate Speicherung: $(\{\{A, 2\}\}, w_2)$ und $(\{\{A, 1\}, \{B, 1\}\}, w_3)$
- Lesezugriff auf den Datensatz
 - Koordinator sendet w_2 und w_3 an den Client
 - Kombinierte Vektoruhr [Berechnung: tupelweise Maximumbildung]: $\{\{A, 2\}, \{B, 1\}\}$
- Auflösung des Konflikts und anschließender Schreibzugriff
 - Client
 - Festlegung auf w_2 als gültigen Wert
 - Berechnung eines neuen Werts w_4 auf Basis von w_2
 - Absetzen einer Schreibenanfrage für w_4 mit kombinierter Vektoruhr
 - Koordinator: Knoten $B \rightarrow$ Vektoruhr der Anfrage: $V_{w_4} = \{\{A, 2\}, \{B, 2\}\}$
 - Replikate
 - $\{\{A, 2\}\} \prec V_{w_4}$ und $\{\{A, 1\}, \{B, 1\}\} \prec V_{w_4} \rightarrow$ Verwerfen der alten Versionen
 - Speicherung von $(\{\{A, 2\}, \{B, 2\}\}, w_4)$



- Nichttrivial aus Anwendungs(programmierer)sicht
 - Keine obere Schranke für das Erreichen eines konsistenten Zustands
 - (Parallele) Leseanfragen sehen eventuell unterschiedliche Teilzustände
 - Lesen veralteter Daten auch nach bestätigten Schreiboperationen möglich
 - Neben Fehler- auch Normalfall (\rightarrow konkurrierende Zugriffe) problematisch
- Voraussetzungen für Praktikabilität
 - Vermeidung konkurrierender Schreibzugriffe auf denselben Datensatz
 - Seltener Wechsel des Koordinators

\rightarrow Vermeidung der Entstehung verschiedener Versionen im Nichtfehlerfall
- Mögliche zusätzliche Maßnahmen
 - Automatisierte Auflösung von Konflikten (z. B. *Last Write Wins*)
 - Koordinator einer Leseanfrage aktualisiert Datensätze auf Replikaten, die veraltete Versionen geliefert haben (*Read Repair*)
- Warenkorb: 99,94% der Anfragen sehen nur eine Version [DeCandia et al.]



Weitere Aspekte

- Behandlung temporärer Replikatausfälle: *Hinted Handoff*
 - Ausfall eines Replikats wird vom System erkannt
 - Einbindung eines Ersatzreplikats aus der Präferenzliste
 - Anfragen enthalten Hinweis auf ausgefallenes Replikat
 - Ersatzreplikat bearbeitet betroffene Anfragen in separater Datenbank
 - Zustandstransfer sobald ausgefallenes Replikat wiederhergestellt ist
 - Weiterführende (manuelle) Maßnahmen, falls Replikatausfall dauerhaft
- Optimierte Koordinierung von Leseanfragen (Alternativen)
 - Jeder Knoten kann als Koordinator fungieren \rightarrow Lastbalancierung
 - Client übernimmt selbst die Koordinierung der Anfrage
 - Vorteil: Einsparung einer Indirektionsstufe
 - Nachteil: Client muss aktuelle Sicht auf Systemzusammensetzung haben
- Anwendungsspezifische Wahl der Quoren-Größen
 - Kleine Werte für R und $W \rightarrow$ Höhere Effizienz, schwächere Konsistenz
 - $R = 1, W = N$ für Anwendungen mit hohem Anteil an Leseanfragen



Zusammenfassung

- Verwaltung kleiner Datensätze
 - Nutzdaten pro Datensatz weniger als 1 MB
 - Repräsentation als Schlüssel-Wert-Paare
 - Atomares Lesen und Schreiben von Datensätzen praktikabel
- Amazon Dynamo
 - Dezentraler Peer-to-Peer-Ansatz
 - Hohe Verfügbarkeit für Schreiboperationen
 - Partitionierung mittels konsistentem Hashing
 - Platzierung des Datensatzes (indirekt) von seinem Schlüssel abhängig
 - Virtuelle Knoten zur Einbindung unterschiedlich leistungsfähiger Rechner
 - Bereitstellung von letztendlicher Konsistenz
 - Detektion von Konflikten mittels Vektoruhren
 - Auflösung von Konflikten durch die Anwendung

