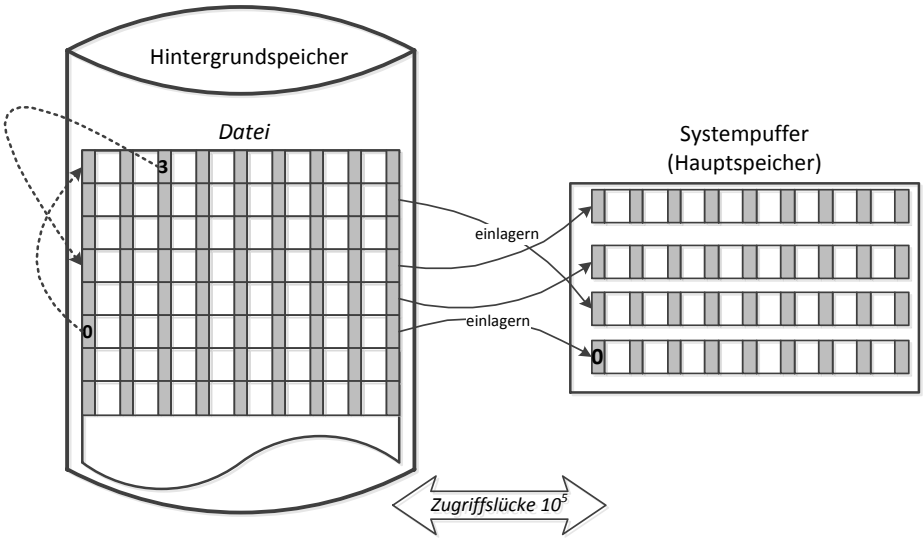

Abb. 7.18: Zusammenspiel des Hintergrundspeichers und des Systempuffers bei B-Bäumen

7.12 Log Structured Merge-Bäume

Ein offenkundiges Problem der B-Bäume besteht in der so genannten *Write Amplification*. Darunter versteht man, dass das Einfügen eines neuen Schlüssels sehr viele Hintergrundspeicherzugriffe verursacht. Bei (realistisch) großen Datenbankanwendungen werden in der Regel nur die inneren Knoten eines B-Baums im Hauptspeicher-Puffer „residieren". Ein neuer Schlüssel wird im Blatt eingefügt, das dazu in den Puffer tranferiert werden muss. Bei realistischer Blattgröße mit $k = 100$ würde also das Einfügen eines neuen Schlüssels den Transfer von mindestens 100 und bis zu 200 schon bestehenden Schlüsseln verursachen. Wir sprechen deshalb in diesem Fall von einer mindestens 100-fachen Schreib-Amplifikation.

Diese Amplifikation wird durch die mehrstufige Anordnung der B^+-Bäume im Log Structured Merge-Baum reduziert. Ein kleinerer B^+-Baum mit einer Kapazität von, sagen wir M, ist Hauptspeicher-resident. Sobald diese Kapazität erschöpft ist, wird der gesamte obere B^+-Baum nach unten transferiert. Der B^+-Baum dieser zweiten Ebene befindet sich schon auf dem Hintergrundspeicher und hat eine maximale Kapazität von $k*M$, wobei k durchaus im Bereich von 10 angesiedelt sein kann. Beim Transfer von oben nach unten (auch *staging* genannt) wird beim Level-LSM-Baum, der in Abbildung 7.19 gezeigt ist, ein Mischvorgang (Engl. *merge*) durchgeführt, welcher in Abbildung 7.20 skizziert ist.

Dieses Prozedere wiederholt sich dann auf allen weiteren Ebenen des LSM-Baums, der auf der dritten Ebene dann schon eine Kapazität von $k * k * M = k^2M$ hat.

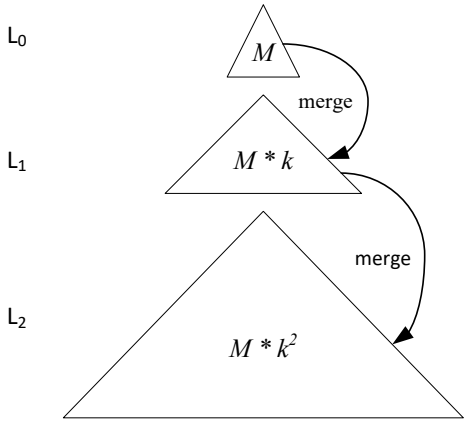

Abb. 7.19: Ein LSM-Baum mit Leveling

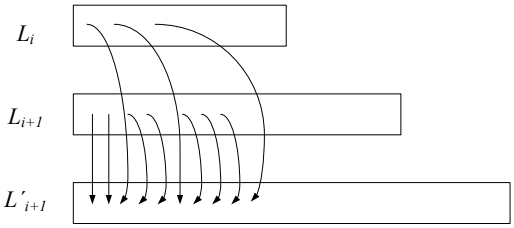

Abb. 7.20: Der Mischvorgang (merge) beim LSM-Baum

Wir wollen jetzt die Schreib-Amplifikation beleuchten. Die Anzahl r der Ebenen des LSM-Baum ist natürlich abhängig von der Datengröße I sowie der Vervielfachung k:

$$r = log_k(I/M)$$

Bei einer Datengröße von 100 * M ergibt sich dann für $k = 5$ folgende Ebenenanzahl: für

$$r = log_5(I/M) = log_5(100) = 3$$

Bei $k = 10$ reduziert sich die Anzahl der Ebenen schon auf $r = log_{10}(100) = 2$.

Jeder Eintrag wird pro Level im Durchschnitt $k/2$ mal von unten gelesen und zurückgeschrieben bzw. gemischt. Warum $k/2$ mal? Intuitiv kann man sich das wie folgt klarmachen: Der allererste Eintrag wandert durch alle Level durch und wird maximal oft gemischt, nämlich k mal pro Level. Der allerletzte Eintrag wird gar nicht nach unten durchgereicht, so dass man im Durchschnitt jeden Eintrag pro Level $k/2$ mal lesen (für das Mischen) und schreiben (für das nachfolgende *staging*) muss. Also hat man folgende

Formel für die Schreib-Amplifikation, also die Anzahl der I/Os bis ein Eintrag letztlich in seiner „Heimat“ ankommt:

$$k/2 * 2 * log_k(I/M)$$

Also ergibt sich bei $k = 5$ und einer Datenbank in der hundert-fachen Hauptspeicher-Puffergröße $I = 100 * M$ und $k = 5$ eine Amplifikation nur für das Schreiben von 7, 5, unter Berücksichtigung auch des Lesens für den *merge*-Vorgang also $w = 15$. Bei realistischerem $k = 10$ ergibt sich sogar ein Amplifikations-Wert von $w = 20 = (k/2) * 2 * r = (10/2) * 2 * 2$.

Updates von Einträgen in niedrigeren Leveln im LSM-Baum werden durch implizite Invalidierung durchgeführt, wie in Abbildung 7.21 skizziert. Um sogenannte „time travel“-Anfragen zu ermöglichen, können die Änderungen eines Datenelements auch noch mit einem Zeitstempel annotiert werden, so dass man den Zustand eines Datenelements zu einem beliebigen zurückliegenden Zeitpunkt ermitteln kann. Die dafür notwendige Suchalgorithmik überlassen wir den Lesern.

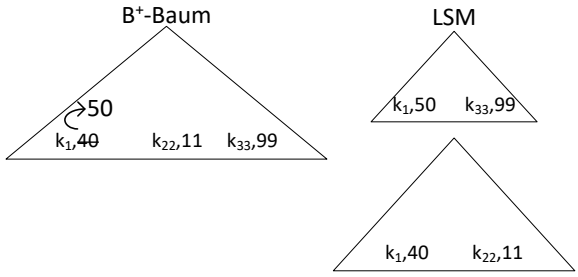

Abb. 7.21: Updates im LSM-Baum im Vergleich zum update-in-place im B⁺-BAum

Das Auffinden eines Eintrags im Level-LSM-Baum erfolgt von oben nach unten. Der zuerst gefundene Eintrag ist der aktuelle, alle möglicherweise weiter unten befindliche Einträge sind veraltet. Die Suche wird durch sogenannte Bloom-Filter unterstützt, die autoritativ das Nichtvorhandensein eines Eintrags in einem B⁺-Baum markieren. Ein Bloom-Filter ist ein Bit-Vektor, dessen Bits durch eine Hashfunktion, die auf dem Suchschlüssel des B⁺-Baum-Eintrags angewendet wird, gesetzt wird. Bei einer Suche nach einem Schlüssel S würde man dieselbe Hashfunktion h anwenden. Wenn an der Position $h(S)$ im Bitvektor eine Null angetroffen wird, ist der Schlüssel mit Sicherheit nicht in dem Baum enthalten. Eine Eins könnte durch eine Hash-Kollision gesetzt worden sein, Wenn also zwei Suchschlüssel denselben Hash-Wert haben, also $h(S_1) = h(S_2)$ gilt. Wenn nur, saagen wir S_1 in dem Baum vorhanden ist, könnte der Bloomfilter die erfolglose Suche nach dem anderen Schlüssel nicht vorzeitig beenden sondern erst nach dem Suchen im Baum.

Schwieriger gestaltet sich eine Bereichssuche, bei der man den Suchbereich auf aallen Ebenen vereinigen muss. Hierbei muss man weiter unten im Suchbereich gefundende (also invalidierte) Einträge mit gleichem Schlüsselwert wie in Bereichen weiter oben aus dem Ergebnis bereinigen.

7.12.1 Tiered (Gestufter) LSM-Baum

Diese Amplifikationswerte sind immer noch recht hoch für schreibintensive Workloads, so dass man den Schreibaufwand gegen einen höhren Suchaufwand im sogenannten Tiered-LSM austarieren kann. Diese LSM-Variante ist in Abbildung 7.22 skizziert. Man verzichtet hierbei auf das sofortige Mischen der B$^+$-Bäume beim „staging“ ab Ebene L_1. Erst wenn eine Ebene L_i vollgelaufen ist, also k B$^+$-Bäume beinhaltet, werden diese k B$^+$-Bäume zusammengefügt (*merged*) und auf die nächstniedrigere Ebene L_{i+1} heruntergereicht. Dadurch reduziert sich die Schreib-Amplifikation, da jeder Eintrag pro Level nur noch einmal geschrieben und (für den abschließenden *merge*) gelesen wird. Also kommt man auf eine Schreib-Amplifikation von $2 * r$. Bei $k = 10$ und $I = 100 * M$ ergibt sich somit ein Wert von $w = 2 * r = 2 * 2 = 4$ gegenüber $w = 20$ beim Level-LSM-Baum. Aber – wie man im Englischen sagt, „there is no free lunch“ – gibt es einen Nachteil beim Lesen. Wir müssen nämlich jetzt auf jeder Ebene, außer der obersten L_0, bis zu k B$^+$-Bäume durchsuchen. Natürlich kommen uns hier jetzt insbesondere die Bloom-Filter zuhilfe um erfolglose Suchen direkt zu erkennen.

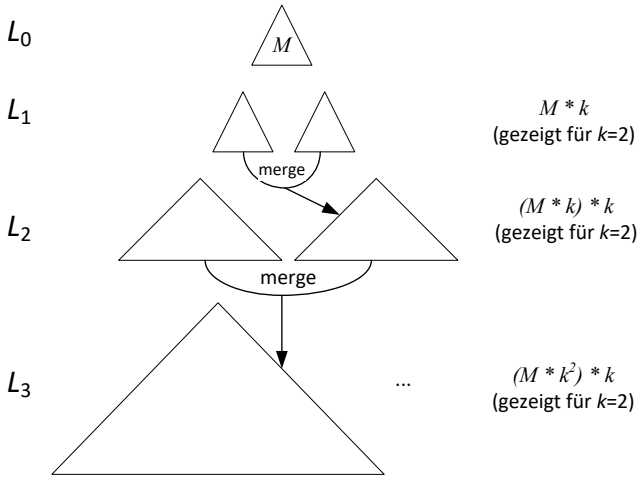

Abb. 7.22: Tiering anstatt sofortiges Mischen

7.12.2 Partitionierter LSM-Baum

Man kann den Aufwand zwischen Lese- und Schreibaufwand auch noch durch Bereichspartinierung austarieren. Diese partitionierten Level-Bäume sind in Abbildung 7.23 skizziert. Hierbei ist für eine Punktanfrage nur ein B$^+$-Baum pro Level „zuständig“. Allerdings sind die Kosten für das „staging“ jetzt höher, da der Transfer von Level L_i zu L_{i+1} immer mit einem Mischen der B$^+$-Bäume verbunden ist, die mit dem Bereich des transferierten B$^+$-Baums aus Level L_i überlappen. Dies ist in der Abbildung angedeutet, wo jeweils zwei B$^+$-Bäume der nächst-niedrigeren Ebene betroffen sind. Wir überlassen es (mal wieder) den Lesern, die hierbei anfallende Schreib-Amplifikation abzuschätzen.

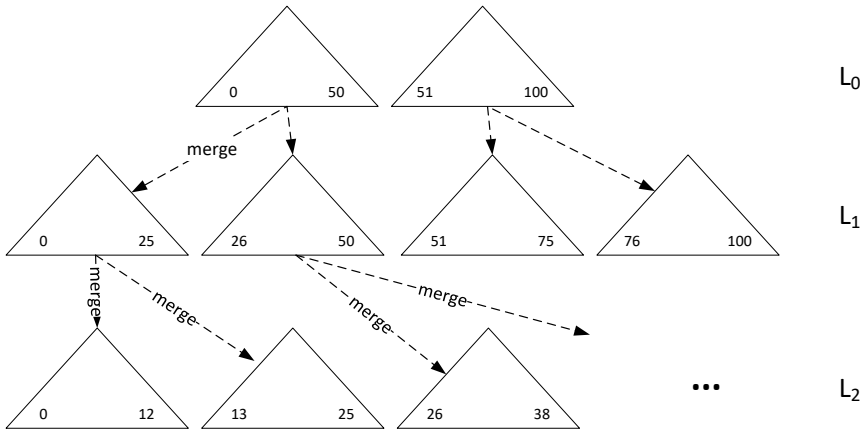

Abb. 7.23: Leveling in Kombination mit Bereichs-Partitioniering

7.13 Hashing

Das ultimative Ziel aller Bemühungen um ein gutes physisches Design ist es, wirklich nur diejenigen Seiten vom Hintergrundspeicher zu lesen, die absolut benötigt werden. Hash-Verfahren ermöglichen es, ein bestimmtes Datum im Durchschnitt mit einem bis zwei Seitenzugriffen zu finden. Bäume benötigen Seitenzugriffe in der Ordnung von $\log_k(n)$, wobei k der durchschnittliche Verzweigungsgrad und n die Anzahl der eingetragenen Datensätze ist.[4]

4 Diese Zahlen werden meist durch Pufferungseffekte relativiert.

8 Neue Inhalte

8.1 Raft: Konsensus-Protokoll

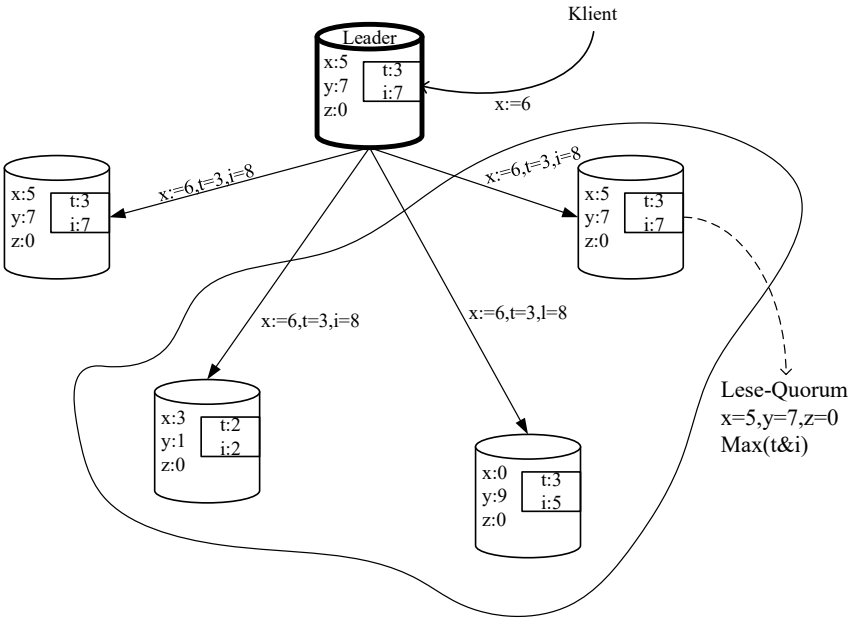

Abb. 8.1: Visualisierung eines Quorums in Raft mit fünf verteilten Stationen

In hoch-verfügbaren Informationssystemen ist es oft nötig, dass wichtige Informationen über viele (geographisch) verteilte Stationen verfügbar gemacht werden. Dabei ist natürlich unabdingbar, dass ein Leser dieser Information einen konsistenten Zustand zu sehen bekommt. Deshalb ist hierfür ein Quorum-Modell nötig, in dem die Mehrheit der Stationen sich in einem konsistenten Zustand befindet. Man nennt eine solche Absicherung dementsprechend auch Konsensus-Prokoll. Wir wollen hier nur ein solches Protokoll, das sich in der Praxis wegen seiner Einfachheit durchgesetzt hat, beschreiben: das Raft-System. In Abbildung 8.1 ist ein Raft-System mit fünf Stationen beschrieben. In Raft ist eine Station designiert als *Leader*, der alle Updates von Klienten entgegen nimmt. Der Leader darf ein Update nur dann *committ*'en, wenn er eine Mehrheit der Stationen dafür gewinnt, also in unserem Beispiel-Szenarion neben seiner eigenen Station noch

https://doi.org/10.1515/978311ooooooo-002

mindestens zwei weitere. Leser müssen ebenfalls drei Stationen kontaktieren, um garantiert den neuesten konsistenten Zustand bei mindestens einer dieser drei Stationen sehen zu können. Der neueste konsistente Zustand zeichnet sich dadurch aus, dass die betreffende Station den höchsten Wert für die Kombination der term- und index-Werte hat. Also muß sie für den Term t sowie nachfolgend für den Index i den höchsten Wert vorweisen, also $t\&i$ muss maximal sein. In unserem Beispiel wäre also die Station am rechten Rand mit $t : 3, i : 7$ maßgeblich für die konsistenten $x = 5, y = 7, z = \ldots$-Werte. Das nächste Update für $x := 6$ mit dem Termwert $t = 3$ und dem Indexwert $i = 8$ wurde zwar vom Leader an die Stationen geschickt, hate aber bislang noch keine Mehrheit gefunden. Deshalb gilt dieses Update noch nicht als committ'et.

Das Raft-Protokoll muss für den fehlertolernten Ablauf also mehrere Garantien sicherstellen:

1. Es gibt nur einen Leader und wenn ein Leader ausfällt müssen sich alle auf einen neuen Leader verständigen.
2. Alle Follower werden – über kurz oder lang – denselben Zustand der Datenbasis annehmen, der sich aus den committ'eten Updates ergibt.

Bei 5 Stationen könnte Raft den Ausfall von bis zu 2 Stationen verkraften, weil dann immer noch ein Quorum von 3 Stationen zusammenkommen kann. Was passiert aber beim Ausfall des Leaders. In diesem Fall müssen die verbleibenden Stationen sich auf einen neuen Leader einigen, das heißt einen neuen Leader wählen. Die verschiedenen Phasen des Raft-Betriebs sind in Abbildung 8.2 gezeigt. Der normale Betrieb wechselt also in eine Leader-Wahl sobald mindestens eine Station glaubt, der Leader sei abgestürzt oder unerreichbar. Diese Station würde dann selbst versuchen neuer Leader zu werden. Dazu benötigt sie aber die Zustimmung von mindestens zwei weiteren Stationen. Gelingt ihr das nicht, ist die Leader-Wahl gescheitert.

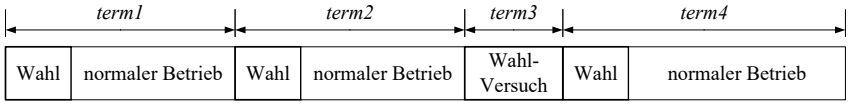

Abb. 8.2: Raft-Betrieb in verschiedenen Phasen (Terms)

Um eine Leaderwahl fehlertolerant realisieren zu können, wird ein *Term*-Wert mit jedem Wahl-Antrag mitgeschickt. Dieser Wert wird immer um eins erhöht im Vergleich zu dem Wert, den eine Station zuletzt wahrgenommen hat. Der Wahlvorgang ist in Abbildung 8.3 als Zustandsübergngs-Diagramm skizziert. Sobald eine Raft-Station keine Verbindung mehr zum designierten Leader aufbauen kann, wird sie (sich selber) in den Zustand Kandidat transferieren und versuchen, selbst Leader zu werden. Dazu wird

der (ihr bisher bekannter) Term-Wert t inkrementiert zu t + +. Sodann kontaktiert sie alle anderen erreichbaren Stationen um Zustimmung zu erhalten. Natürlich darf ein angefragter Follower nur zustimmen, wenn er nicht schon mit gleichem oder höherem Term-Wert einem anderen Kandidaten zugestimmt hat. Diese Stationen würden aber auch nur dann zustimmen, wenn die vom Kandidaten übermittelten Protokolleinträge, bestehend aus dem bisherigen Term-Wert und dem Index i mindestens so aktuell sind wie ihre eigenen. Sonst hätte der Kandidat veraltete Information und könnte nicht selbst Leader werden. In diesem Fall wäre die Leaderwahl gescheitert, da ja mindestens eine Station im benötigten Quorum den aktuellen Zustand besitzen muss. Dann würde nach zufällig gewählter Wartezeit eine andere Station zum Kandidat für die Leaderwahl. Wenn diese Wahl gelingt, würde ein ehemaliger Leader nach Wiederanlauf anhand des jetzt höheren Term-Werts herausfinden, dass es einen neuen Leader gibt und sich selbst zum Follower degradieren. Stationen würden nach einem Absturz oder einer Netzwerkpartitionierung beim aktuellen Leader den aktuellen committ'eten Zustand „erfragen".

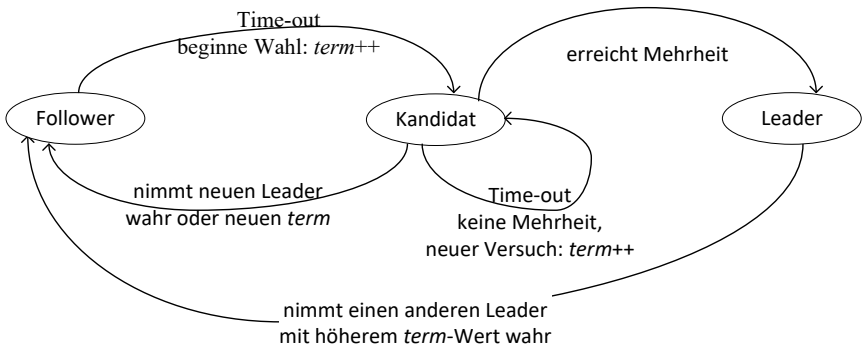

Abb. 8.3: Der Wahlvorgang in Raft

In Abbildung 8.4 sind die Protokoll-Daten gezeigt die für die Fehlertoleranz sowohl vom Leaader als auch den zustimmenden Followern persistent geschrieben werden müssen. Hier ist das aus der Recovery bekannte WAL (write ahead log)-Prinzip einzuhalten: Bevor ein Follower zustimmen kann, muss er diese beabsichtigte Zustimmung in einem dauerhaften Log geschrieben haben. Der Leader darf ein update erst committ'en, sobald er die Zustimmung des Quorums, in diesem Fall also zwei weiterer Stationen eingesammelt hat. Deshalb ist der letzte gezeigte update von $x = 6$ noch nicht committ'et. Wohingegen das update von $x = 5$ mit $t : 3, i : 7$ sich schon jenseits der commit-Marke befindet, da das Quorum von 3 Stimmen schon erreicht wurde, wie man an den Log-Einträgen von zwei der vier Follower erkennen kann. Diese Commit-Marke wird

den Followern vom Leader mitgeteilt – also ähnlich dem 2-Phasen-Commit-Protokoll. Man beachte, dass alle früheren Updates auch schon committ'et sind, da dieser Vorgang streng sequentiell abläuft. Nachdem ein Update committ'et ist, kann dies auch auf die eigentliche Datenbank fortgeschrieben werden, wie in Abbildung 8.1 illustriert.

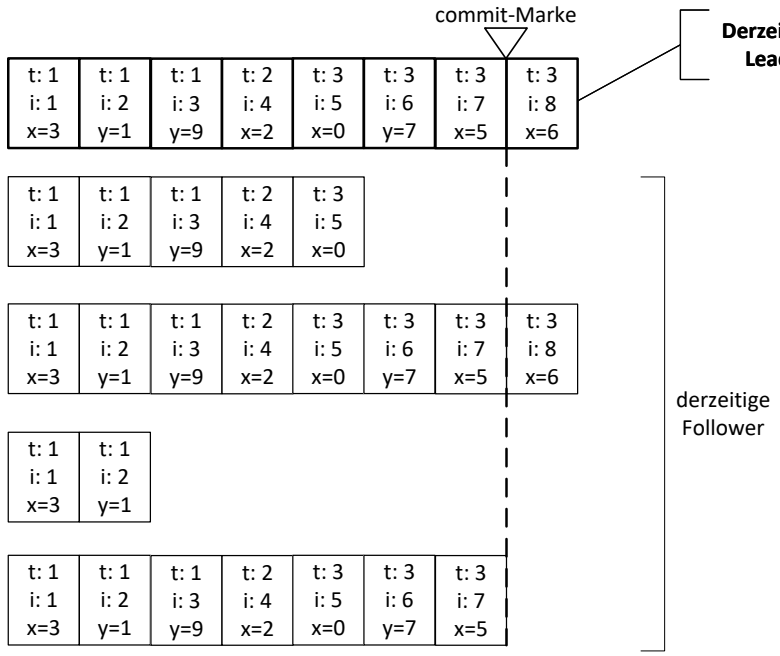

Abb. 8.4: Die verteilten Log-Dateien in Raft

8.2 JSON

XML gilt als sehr „verbos", so dass die eigentlichen Inhalte sich in relativ langen Dokumenten befinden. Deshalb hat sich in der letzten Zeit ein einfacheres Format im Datenbankbereich durchgesetzt: JSON. Das Akronym steht für JavaScript Object Notation, obwohl es sich um ein Programmiersprachen-unabhängiges Textformat handelt. Genau wie XML ist JSON ein hierarchisches Modell mit beliebig tief geschachtelten Unterobjekten. Aber anders als XML hat JSON nur noch Name/Wert- bzw. Key/Value-Paare als Informationseinheiten; die Unterscheidung in Attribute und Unterelemente entfällt. Als Strukturelemente gibt es in JSON also die *Objekte*, die mit geschweiften Klammern {…}

notiert werden und Arrays, die mit eckigen Klammern […] gekennzeichnet sind. Arrays selbst können auch wieder Objekte enthalten. Anders als in XML werden Arrays ab 0 indexiert. Ein Objekt ist, anders als ein Array eine ungeordnete Liste von Key/Value-Paaren, wobei der Value durch einen Doppelpunkt vom Key abgetrennt ist. Als Key kommen nur Strings in Hochkommata infrage, wohingegen der Value-Teil „alles mögliche“ sein kann, also eine Zahl, ein String, ein Boole'scher Wert, ein Objekt, oder ein Array.

Wir wollen die JSON-Datenmodellierung an einem Ausschnitt unseres vereinfachten Universitätsbeispiels der “Grösten Denker“ demonstrieren:

```
{
    "Universität": {
      "Name": "Virtuelle Universität der Grössten Denker",
      "UniLeitung": {
        "Rektor": "Sokrates",
        "Kanzler": "Erhard" },
      "Fakultäten": [
         {"FakName": "Theologie",
          "Professoren": [ {
            "Name": "Augustinus",
            "Rang": "C3",
            "Raum": 309,
            "Vorlesungen": [
               {"VorlNr": "V5022",
                "Titel": "Glaube und Wissen",
                "SWS": 2 } ] } ] } ,
         {"FakName": "Physik",
          "Professoren": [
             {"Name": "Curie",
              "Rang": "C4",
              "Raum": 36 } ,
             {"Name": "Kopernikus",
              "Rang": "C3",
              "Raum": 310,
              "Vorlesungen": [
                 {"VorlNr": "V5023",
                  "Titel": "Alfonsinische Tafeln",
                  "SWS": 2 } ,
                 {"VorlNr": "V5024",
                  "Titel": "Astronomie",
                  "SWS": 2 } ] } ] } ] }
}
```

8.2.1 JSONpath

Mit JSONpath scheint sich eine standardisierte Anfragesprache für JSON-Dokumente im Datenbankbereich durchzusetzen. Leider weicht die Syntax doch deutlich von XPATH Pfadausdrücken ab; auch wenn es semantisch sehr ähnlich ist. So verwendet man in Pfadausdrücken in JSONPath für die Adressierung eines Kind-Objekts den Punkt statt eines Backslash. Beispielsweise wird der Rektor der Uni durch diesen JSONPath-Ausdruck ermittelt:

```
$.Universität.UniLeitung.Rektor
```

Hierbei steht das Dollarzeichen $ für das Wurzelobjekt. Alternativ zur Punkt/Dot-Notation kann man auch eckige Klammern verwerden, dann muss der Key aber in Hochkommata gesetzt werden, wie hier:

```
$['Universität']['UniLeitung']['Rektor']
```

Der rekursive Abstieg, also die *descendant-or-self*-Achse wurde in XPATH abgekürzt durch „//“. Aber in JSONpath verwendet man zwei Punkte, wie in diesem Programmbeispiel:

```
$..Professoren..Vorlesungen..Titel
```

Hier werden die Titel aller von Professoren gehaltenen Vorlesungen ausgegeben. Man beachte, dass dieser rekursive Abstieg also auch durch die zwischengelagerten Arrays iteriert – die Vorlesungen sind in unserem Beispieldokument den Professoren in einem (geschachtelten) Array zugeordnet ebenso, wie die Fakultäten in einem Array der Universität zugeordnet sind.

Will man nur die Vorlesungen von Kopernikus ausgeben, so benötigt man ein Filterprädikat, das mit einem Fragezeichen eingeleitet wird. Das Prädikat steht dann in (…)-Klammern. In der Regel wird das Prädikat auf der Basis des schon erreichten Objekts formuliert, was mit dem „Klammeraffen“ notiert wird. Es entspricht in XPATH also dem **self** oder, abgekürzt, dem Punkt.

```
$..Professoren[?(@.Name=='Kopernikus')]..Vorlesungen..Titel
```

Das Ergebnis für unsere Beispiel-Datenbank ist dann das Array mit den beiden Vorlesungstiteln.

```
[
  "Alfonsinische Tafeln",
  "Astronomie"
]
```

Will man nur die wichtigste, also die erste Vorlesung von Kopernikus ausgeben, muss man die Indexposition 0 abfragen

```
$..Professoren[?(@.Name=='Kopernikus')]..Vorlesungen[0].Titel
```

und erhält ein Array mit nur einem Eintrag:

```
[
  "Alfonsinische Tafeln"
]
```

Man kann auch Array-Bereiche angeben. Wenn man beispielsweise die ersten beiden Vorlesungen von Kopernikus ausgeben will, geht das wie folgt:

```
$..Professoren[?(@.Name=='Kopernikus')]..Vorlesungen[0:2].Titel
```

Das vorhesehbare Ergebnis ist:

```
[
  "Alfonsinische Tafeln",
  "Astronomie"
]
```

Wenn man explizit durch alle Elemente eines Arrays iterieren will, kann man das mit [*] ausdrücken. Bei den Bereichsangaben der Array-Indizes kann man auch die untere bzw. auch die obere Schranke weglassen, wie beispielsweise [:1] wenn man nur das erste Element haben möchte:

```
$.Universität.Fakultäten[*].Professoren[?(@.Name=='Kopernikus')]..
                                                    Vorlesungen[:1].Titel
```

Wiederum haben wir ein vorhersehbares Ergebnis als Ausgabe:

```
[
  "Alfonsinische Tafeln"
]
```

Wie bereits erwähnt kann man statt der Dot-Notation auch die eckigen Klammern für die Traversierung zum Kind-Objekt verwenden, muss dann aber den Key in Hochkomma angeben:

```
$['Universität']['Fakultäten'][*]['Professoren']
```

```
[?(@['Name']=='Kopernikus')]..['Vorlesungen'][:1]['Titel']
```

Das Ergebnis ist natürlich wieder wie zuvor:

```
[
  "Alfonsinische Tafeln"
]
```

Wenn man die wichtigste (also jeweils erste) Vorlesung aller Professoren ermitteln möchte, geht das wie folgt:

```
$['Universität']['Fakultäten'][*]['Professoren']..
                                  ['Vorlesungen'][0]['Titel']
```

oder ohne rekursiven Abstieg könnte man es so formulieren:

```
$['Universität']['Fakultäten'][*]['Professoren'][*]
                                ['Vorlesungen'][0]['Titel']
```

Das Ergebnis für unsere Beispiel-Datenbank sieht dann so aus:

```
[
  "Glaube und Wissen",
  "Alfonsinische Tafeln"
]
```

8.3 Worst Case Optimal Joins

Insbesondere bei Anfragen auf Graph-Daten werden sehr viele sogenannte Self-Joins, also mehrfaches Vorkommen derselben Kanten-Relation, für die Traversierung benötigt. Zur Illustration gehen wir aber zunächst davon aus, dass die Kanten eines ungerichteten GRaphen auf drei Relationen aufgeteilt sind:

$$R : \{[A, B]\} \quad S : \{[B, C]\} \quad T : \{[A, C]\}$$

Als Ausprägung dieser drei Relationen gehen wir von der in Abbildung 8.5 gezeigten Konstellation aus. Diese Ausprägung enthält eine „dramatische“ Schieflage, da a_0, b_0 und c_0 jeweils mit allen anderen Knoten des Graphen verbunden sind – außer mit den Knoten aus der eigenen Knotenmenge. Die Relationen sehen dann wie folgt aus:

R	
A	B
a_0	b_0
a_0	b_1
a_0	b_2
a_0	b_3
a_1	b_0
a_2	b_0
a_3	b_0

S	
B	C
b_0	c_0
b_0	c_1
b_0	c_2
b_0	c_3
b_1	c_0
b_2	c_0
b_3	c_0

T	
A	C
a_0	c_0
a_0	c_1
a_0	c_2
a_0	c_3
a_1	c_0
a_2	c_0
a_3	c_0

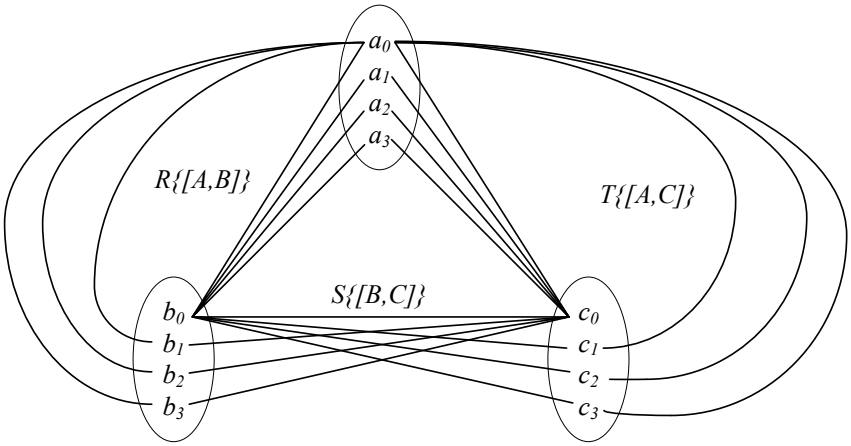

Abb. 8.5: Die Datenbank für eine Dreiecks-Anfrage

Diese Datenbank soll demonstrieren, dass die Zwischenergebnisse bei der („herkömmlichen“) Anfragebearbeitung viel größer sein können als das Endergebnis. Dazu betrachten wir den Dreiwege-Join

$$R \bowtie S \bowtie T$$

oder in SQL-Notation

select *
from R **join** S **on** $R.B = S.B$ **join** T **on** $S.C = T.C$ **and** $T.A = R.A$;

Das Ergebnis dieser Anfrage hat $3m + 1$ Tupel, wenn man jeweils m Knoten pro Knotenmenge hätte – anstelle der nur vier Knoten in unserer Illustration. Für unsere Ausprägung hätten wir also 13 Ergebnistupel – es sei den Lesern überlassen dies nach-

zuprüfen. Jeder binäre Join von jeweils zwei der Relationen hätte aber $(m+1)^2 + m$ Zwischenergebnisse, in unserem Fall also 29. Herkömmliche Anfragesysteme (sogenannte „query engines“) sind auf binäre Joins beschränkt, so dass sie einen der folgenden Pläne auswerten müssten:

$$(R \bowtie S) \bowtie T \quad \textbf{oder} \quad R \bowtie (S \bowtie T) \quad \textbf{oder} \quad (R \bowtie T) \bowtie S)$$

Der binäre Join hätte aber jeweils quadratische Komplexität – egal welche Join-Algorithmik man anwendet, da das (Zwischen-) Ergebnis quadratische Größe hat, nämlich $(m+1)^2+m$. Hier setzen die in jüngster Zeit populär gewordenen Worst-Case Optimal Joins an, indem sie nicht mehr binäre Operatoren verwenden sonderen mehr-stellige Operationen, die quais synchron durch die zugrundeliegenden Datenmengen iterieren.

Das grundlegende Problem liegt darin, dass die binären Joins das Zwischenergebnis „aufblähen“ und erst zum Schluss eine starke Filterung erzielt wird. Dies ist schematisch in Abbildung 8.6 illustriert, in der die Größe des jeweiligen (Zwischen-)Ergebnis in der horizontalen Ausdehnung gezeigt wird. Die Rauten/Diamant-Form stellt das Problem der übergroßen ZWisschenergebnisse dar.

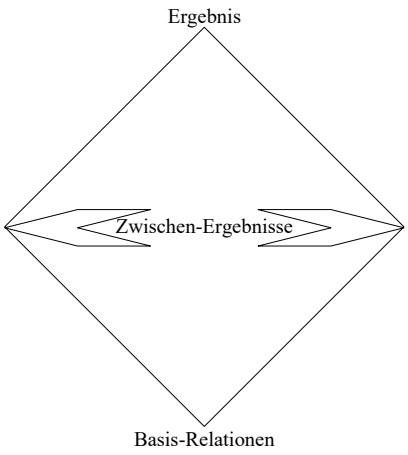

Abb. 8.6: Der Diamond-Join zur Vermeidung der aufgeblähten Zwischenergebnisse

Dies kommt insbesondere bei zyklischen Join-Graphen zum Tragen, wenn also wie in unserem Beispiel R ein Join-Prädikat mit S hat, S mit T und dann zyklisch T zusätzlich ein Joinprädikat mit R besitzt.

Es gibt durchaus praktische Anwendungsbeispiele, gerade auf Graph-Daten, wo dies vorkommt. Betrachten wir mal folgende Situation, die möglicherweise unerlaubte/unethische Bewertungsabsprachen bei einem e-Commerce-Unternehmen analysiert. A be-

wertet ein Produkt von B positiv, B bewertet ein Produkt von C positiv und C wiederum bewertet nach Absprache ein Produkt von A positiv. Solche Dreieckskonstellationen kann man in der Datenbank wie folgt aufdecken:

Bewertungen	
Bewerter	Bewerteter
A	B
B	C
C	A
…	…

```
select *
from Bewertungen b1 join Bewertungen b2 on b1.Bewerter=b2.Bewerteter
                    join Bewertungen b3 on b2.Bewerter=b3.Bewerteter and
                                           b3.Bewerter=b1.Bewerteter ;
```

Um die Auswertungskomplexität einer Anfrage durch die Größe des Endergebnis' zu begrenzen wurden mehrsstellige so-genannte Worst Case Joinoperatoren entwickelt. Das Grundprinzip dieser mehr-stelligen Operatoren wollen wir am Beispiel des Leapfrog-Joins bei der Durchschnittsberechnung dreier Zahlenmengen illustrieren: $R \cap S \cap T$. Man beachte, dass die Mengendifferenz ja nichts anderes ist als der natürliche Join auf einstelligen Relationen.

Der Leap-Frog-Join würde diese Mengen zunächst sortieren und dann synchron durch diese Sequenzen iterieren, wie in Abbildung 8.7 skizziert.

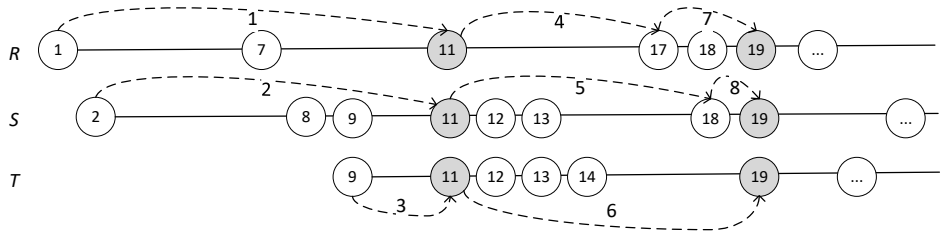

Abb. 8.7: Der Leapfrog-Join angewendet auf drei Sequenzen

Nach dem Sortieren wird für jede Sequenz ein Iterator angelegt. Diese Iteratoren werden einmal gemäß des ersten Elements in der Sequenz sortiert, so dass in unserem Fall die Reihenfolge R, S, T ermittelt wird. Danach springen die Iteratoren abwechselnd in der „wrap around" Reihenfolge $R, S, T, R, S, T, R, \ldots$ durch die Sequenz, und zwar bis

zu der Position die durch das größte Iterator-Element vorgegeben ist. Also „marschiert“ der jeweils aktive iterator bis zu einem Element in der eigenen Sequenz, das größer oder gleich dem größten anderen Element an den derzeitigen Positionen der anderen Iteratoren vorgegeben wird. In unserem Fall springt bzw. „marschiert“ der Iterator R bis zur 11, da dieser Wert (knapp) größer als die 9 des Iterators T ist. Als nächstes ist S dran gemäß der initialen Sortierung und geht ebenfalls bis zur 11. Gleichfalls rückt der Iterator T bis zur 11 vor. Jetzt stehen alle Iteratoren auf der 11, so dass diese Zahl im Ergebnis des Menegendurchschnitts liegt. Die Iteratoren müssen jetzt nicht mehr neu sortiert werden – stehen ja eh alle auf dem selben Wert. Es kommt aber auch vor, wie in unserem Beispiel illustriert, dass ein Iterator mehrmals vorrücken muss. R springt nämlich initial nur bis zur 17, wonach S zur 18 vorrückt und T zur 19. R kommt jetzt gemäß der „wrap around“ Reihenfolge wieder dran und rückt bis zur 19 vor, genau wie S im nächsten Schritt. Es ist unschwer zu erkennen, dass diese Vorgehensweise mit linearer Komplexität in der Länge der Sequenz vonstatten geht, so dass der teuerste Schritt das Sortieren der Sequenzen ist.

Beim allgemeinen Join reicht eine einfache Sortierung nicht mehr aus, so dass der Leapfron-Join die Daten der beteiligten Relationen in einem Trie bzw. Radix-Baum anordnet. Für die ternäre Relation R

R		
A	B	C
1	3	1
1	3	2
1	5	4
3	5	1
3	5	3

sähe der Radixbaum dann für die Sortierreihenfolge A, B, C wie in Abbildung 8.8 gezeigt aus.

Für den allgemeinen Join dreier Relationen würde man dann gemäß der global festgelegten Reihenfolge der Join-Attribute die Tries aller drei Relationen aufbauen. Wir betrachten die folgende Relationenausprägungen:

R	
A	B
7	3
7	**5**
9	…
…	…

S	
B	C
1	8
1	9
5	**1**
5	3
…	…

T	
A	C
6	0
6	9
7	**1**
7	2
7	3
…	…

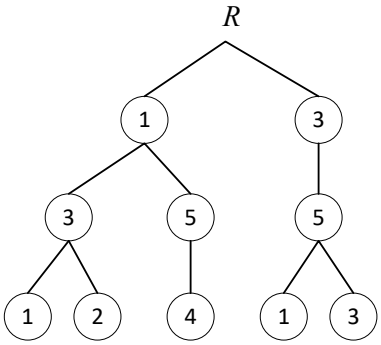

Abb. 8.8: Der Trie/Radixbaum der Relation $R : \{[A, B, C]\}$

Hier ist das erste Ergebnistupel des Dreiwege-Joins durch Fettdruck markiert. In Abbildung 8.9 ist skizziert, wie dieses Ergebnis zustandekommt: In den Tries R und T wird synchron zu dem ersten gemeinsam vorkommenden A-Wert iteriert. Dann wird in

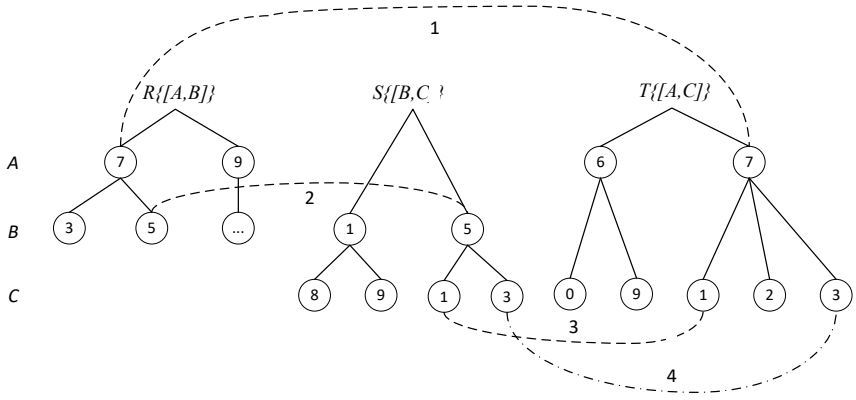

Abb. 8.9: Der Leapfrog/Trie-Join in Aktion

Schritt (2) über die Kindknoten von 7 und die B Werte von S iteriert bis ein gemeinsamer Wert gefunden wird, hier die 5. Schlussendlich wird in Schritt (3) durch Iteration über die Kindknoten von der 5 in S und die Kindknoten der vorher schon „festgezurrten“ 7 im Trie T iteriert und der gemeinsam vorkommende Wert 1 gefunden, um das Ergebnistupel [7,5,1] zu ermitteln. Gleichfalls würde im nächten Schritt (4) durch Back-

tracking auch das Ergebnistupel [7,5,3] ermittelt. Der Trie-Join überspringt also bei der Ermittlung der Ergebnistupel unter Umständen große Teilbäume des Tries, so dass die quadratische Komplexität der binären Joins bei der Dreiecksanfrage vermieden wird.

8.4 Der Diamant-gehärtete Join

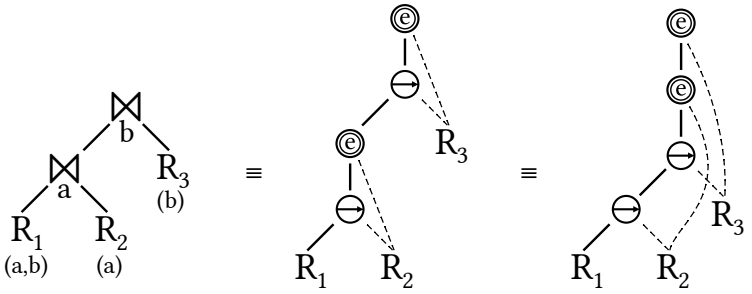

Abb. 8.10: Drei-Wege-Join mit Lookup-Expand

Das Kenrproblem der binären Joins waren die aufgebläten Zwischenergebnisse, die dann gegen Ende der Anfragebearbeitung doch wieder eliminiert wurden. Dies war in Abbildung 8.6 schematisch durch die Diamant/Rauten-Form dargestellt worden. Hier setzt der Diamant-gehärtete Join an, der diese übergroßen Zwischenergebnisse des binären Joins dadurch vermeidet, dass er erst mal nur ein Test-Tupel beim Hash-Join in der Lookup-Phase weiterreicht und schaut ob dieses die weitere Join-Bearbeitung überlebt. Alle weiteren Duplikate werden erst im Erfolgsfall in der Expandier-Phase nachgeholt. Wir gehen von dem Join $R_1 \bowtie_{R_1.a=R_2.a} R_2$ aus. R_2 sei der Build-Input des Hash-Joins, so dass eine Hashtabelle für den Schlüssel $R_2.a$ aufgebaut ist. Weiterhin gehen wir davon aus, dass in R_2 Duplikate für die a-Werte vorkommen, so dass die Hashtabelle (lange) Überlaufketten enthält. Für diesen Zwei-Wege-Join der zwei Relationen ist der Pseudo-Code wie folgt:

```
1 def lookup_consume (produce , tuple , hashTable):
2    # Suche das erste Vorkommen von tuple.a in hashTable
3    iterator = hashTable.find(tuple.key)
4    if iterator.done():
5        return
6 # Geb das gefundene Join-Tuple und einen Iterator für weitere Treffer zurück
7    produce (tuple,iterator)
8
9 def expand_consume(produce , tuple):
```

```
10   # Extrahiere den Iterator vom Tuple
11   currentMatch = tuple.iterator
12   do : # Iteriere über alle Treffer
13      produce(tuple, *currentMatch)
14      currentMatch.step()
15   while not currentMatch.done()
```

Man kann diese Vorgehensweise des Lookup&Expandiere auch über mehrere Joins hinweg anwenden, wie in Abbildung 8.10 gezeigt. Die gestrichelten Linien der beiden rechten Auswertungsbäume symbolisieren das Expandieren. Das Expndieren des unteren Joins kann man nach oben verschieben, so dass man dies nur für solche a-Werte durchführen wird, die jenseits des Joins mit R_3 überleben. Dadurch wird das mögliche Aufblähen des Zwischenergebnis des binären Joins $R_1 \bowtie_{R_1.a=R_2.a} R_2$ vermieden.

8.5 Property Graph Modell

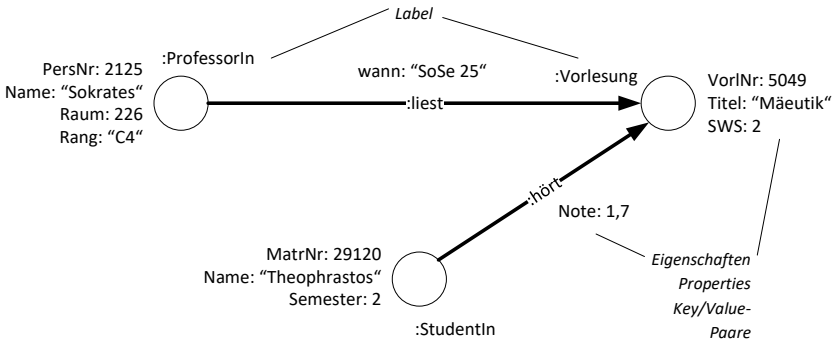

Abb. 8.11: Beispiel eines Property-Graphen

Im Graph-Datenbankbereich scheinen sich das Property-Graph-Modell und die Sprache *Cypher* durchzusetzen. Wir werden hier einige Beispiele in Cypher programmieren, ohne allerdings den Sprachumfang vollständig erläutern zu können. Zunächst wollen wir eine Graph-Datenbank aufbauen:

```
insert( (:ProfessorIn {PersNr:2125, Name:"Sokrates", Raum:226, Rang:"C4"})
              -[:liest {wann: "SoSe25"}]->
                    (:Vorlesung {VorlNr:5049, Titel:"Mäeutik", SWS:2}) )
```

Hiermit wird der obere Teil des Property Graphen aus Abbilfung 8.11 schon mal eingefügt. Also haben sowohl Kanten als auch Knoten Eigenschaften (Properties), die als Key/Value-Paare annotiert werden.

Natürlich gibt es auch Abfragemöglichkeiten in Cypher. Uns interessiert: Wer hört Mäeutik?

```
Match (s:StudentIn) -[:hört]->(v:Vorlesung {Name:"Mäeutik"})
   return s.Name
```

Weiterhin wollen wir herausfinden: Welche Studierende sind entweder als Hörer oder als Tutoren an Mäeutik beteiligt:

```
Match (s:StudentIn) -[beteiligt]->(v:Vorlesung {Name:"Mäeutik"})
   return type(beteiligt) as Rolle, s.Name
```

Hierbei steht *beteilgt* für alle möglichen Beziehungen zwischen *StudentIn* und *Vorlesung*, also auch noch andere denkbar jenseits von *hören* und *tutor*.