KAPITEL 3
Speichern und Abrufen

Wer Ordnung hält, ist nur zu faul zum Suchen.
– Deutsches Sprichwort

Eine Datenbank muss grundsätzlich zwei Dinge tun: Wenn man ihr Daten übergibt, sollte sie die Daten speichern, und wenn man sie später wieder abfragt, sollte sie die Daten zurückgeben.

In Kapitel 2 haben wir Datenmodelle und Abfragesprachen besprochen – d. h. das Format, in dem Sie (der Anwendungsentwickler) der Datenbank Ihre Daten anbieten, und den Mechanismus, mit dem Sie später danach fragen können. In diesem Kapitel besprechen wir das Gleiche aus dem Blickwinkel der Datenbank: Wie können wir die übergebenen Daten speichern und wie können wir sie wiederfinden, wenn wir danach gefragt werden?

Warum sollten Sie sich als Anwendungsentwickler darum kümmern, wie die Datenbank das Speichern und Abrufen intern realisiert? Wahrscheinlich werden Sie nicht Ihre eigene Storage-Engine von Grund auf neu implementieren, doch von den vielen verfügbaren Storage-Engines müssen Sie eine *auswählen*, die für Ihre Anwendung geeignet ist. Um eine Storage-Engine so zu konfigurieren, dass sie die vorgesehene Arbeitsbelastung gut bewältigt, brauchen Sie eine grobe Vorstellung davon, was die Storage-Engine hinter den Kulissen macht.

Insbesondere besteht ein großer Unterschied zwischen Storage-Engines, die für transaktionale Arbeitslasten optimiert sind, und denjenigen, die für Datenanalyseaufgaben optimiert sind. Auf diesen Unterschied gehen wir später ein im Abschnitt »Transaktionsverarbeitung oder Datenanalyse?« auf Seite 97, und im Abschnitt »Spaltenorientierte Speicherung« auf Seite 103 befassen wir uns mit einer Familie von Storage-Engines, die für Analytik optimiert ist.

Zu Beginn dieses Kapitels sprechen wir aber über Storage-Engines von Datenbanken, mit denen Sie wahrscheinlich vertraut sind. Dabei geht es um herkömmliche relationale Datenbanken und auch einen Großteil der sogenannten NoSQL-Datenbanken. Wir untersuchen zwei Familien von Storage-Engines: *protokollstruk-*

turierte Storage-Engines und *seitenorientierte* Storage-Engines wie zum Beispiel B-Bäume.

Datenstrukturen, auf denen Ihre Datenbank beruht

Sehen Sie sich die einfachste Datenbank der Welt an, die in Form zweier Bash-Funktionen implementiert ist:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Diese beiden Funktionen implementieren einen Schlüssel-Wert-Speicher. Mit dem Aufruf `db_set key value` können Sie den Schlüssel `key` und den Wert `value` in der Datenbank speichern. Schlüssel und Wert können (fast) alles sein, was Sie möchten – zum Beispiel könnte der Wert ein JSON-Dokument sein. Dann können Sie mit dem Aufruf `db_get key` nach dem neuesten Wert suchen, der diesem Schlüssel zugeordnet ist, und ihn zurückgeben.

Und es funktioniert:

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

Das zugrunde liegende Speicherformat ist sehr einfach: eine Textdatei, in der jede Zeile ein Schlüssel-Wert-Paar, getrennt durch ein Komma, enthält (etwa wie eine CSV-Datei, ohne Berücksichtigung von Escape-Zeichenfolgen). Jeder Aufruf von `db_set` fügt die Daten an das Ende der Datei an. Wenn Sie also einen Schlüssel mehrmals aktualisieren, werden die alten Versionen des Werts nicht überschrieben – Sie müssen nach dem letzten Vorkommen eines Schlüssels in einer Datei suchen, um den neuesten Wert zu finden (daher das `tail -n 1` in der Funktion `db_get`):

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

Unsere Funktion `db_set` zeigt eigentlich eine gute Performance für etwas so Einfaches, weil das Anfügen an eine Datei im Allgemeinen sehr effizient vonstatten geht. Ähnlich dem, was `db_set` tut, verwenden viele Datenbanken intern ein *Protokoll*, das als Datei implementiert ist, an die nur am Ende angefügt werden kann. Echte Datenbanken haben mit mehr Problemen zu tun (zum Beispiel Nebenläufigkeit steuern, Festplattenplatz freigeben, damit das Protokoll nicht unendlich wächst, und Fehler sowie unvollständig geschriebene Datensätze behandeln), doch das Grundprinzip ist das gleiche. Protokolle sind unglaublich nützlich, und wir werden sie im Rest dieses Buchs noch mehrmals antreffen.

Unter *Protokoll* (englisch: *log*, im Sinne eines Logbuchs) versteht man oftmals ein Anwendungsprotokoll, in das eine Anwendung Text ausgibt, um Ereignisse zu beschreiben. In diesem Buch verwenden wir *Protokoll* in einem allgemeiner gefassten Sinn: als Folge von Datensätzen, die ausschließlich angefügt werden. Die Daten müssen nicht notwendigerweise im Klartextformat vorliegen, sondern können auch binär codiert und nur zum Lesen durch andere Programme vorgesehen sein.

Andererseits legt unsere Funktion `db_get` eine furchtbare Performance an den Tag, wenn die Datenbank eine große Anzahl von Datensätzen speichert. Bei jeder Suche nach einem Schlüssel muss `db_get` die gesamte Datenbankdatei von Anfang bis Ende durchgehen und nach den Vorkommen des Schlüssels suchen. Im Sprachgebrauch der Algorithmenanalyse sagt man, dass die Kosten einer solchen Suche *O(n)* betragen: Wenn Sie die Anzahl der Datensätze *n* in Ihrer Datenbank verdoppeln, dauert die Suche doppelt so lange. Das ist nicht gut.

Um effizient den Wert für einen bestimmten Schlüssel in der Datenbank zu finden, brauchen wir eine andere Datenstruktur: einen *Index*. In diesem Kapitel sehen wir uns ein paar Indexstrukturen an und vergleichen sie. Prinzipiell liegt ihnen die Idee zugrunde, zusätzliche Metadaten mitzuführen, die als Wegweiser dienen und Ihnen helfen, die gewünschten Daten aufzufinden. Wenn Sie dieselben Daten auf verschiedene Art und Weise suchen wollen, brauchen Sie gegebenenfalls mehrere verschiedene Indizes, die verschiedene Bestandteile der Daten abdecken.

Ein Index ist eine *zusätzliche* Struktur, die aus den primären Daten abgeleitet wird. Viele Datenbanken erlauben es Ihnen, Indizes hinzuzufügen und zu entfernen. Dies wirkt sich nicht auf den Inhalt der Datenbank aus, sondern beeinflusst nur die Performance von Abfragen. Die Verwaltung zusätzlicher Strukturen bringt speziell bei Schreibvorgängen einen Overhead mit sich. Beim Schreiben ist es schwer, die Performance eines einfachen Anfügens an eine Datei zu übertreffen, weil das die einfachste mögliche Schreiboperation ist. Jede Art von Index bremst Schreibvorgänge normalerweise ab, weil der Index bei jedem Schreiben von Daten ebenfalls aktualisiert werden muss.

Dies ist eine wichtige Abwägung in Speichersystemen: Gut gewählte Indizes beschleunigen Leseabfragen, aber jeder Index bremst die Schreibvorgänge. Aus die-

sem Grund indizieren Datenbanken standardmäßig nicht einfach alles, sondern verlangen von Ihnen – dem Anwendungsentwickler oder Datenbankadministrator –, die Indizes manuell auszuwählen, wobei Sie Ihr Wissen über die typischen Abfragemuster der Anwendung einfließen lassen. Dann können Sie die Indizes auswählen, von denen Ihre Anwendung am meisten profitiert, ohne mehr Overhead als notwendig einzubringen.

Hash-Indizes

Beginnen wir mit Indizes für Schlüssel-Wert-Daten. Dies ist nicht die einzige Art von Daten, die man indizieren kann, aber sie kommt häufiger vor, und dieser Index ist ein nützlicher Baustein für komplexere Indizes.

Schlüssel-Wert-Speicher ähneln stark dem *Wörterbuchtyp* (*Dictionary*), den Sie in den meisten Programmiersprachen finden und der normalerweise als Hashtabelle (engl. Hash Map) implementiert ist. Hashtabellen werden in vielen Lehrbüchern für Algorithmen beschrieben [1, 2], sodass wir hier nicht im Detail auf ihre Arbeitsweise eingehen. Da wir bereits über Hashtabellen für unsere speicherinternen Datenstrukturen verfügen, sollten wir dann nicht auch unsere Daten auf der Festplatte damit indizieren?

Nehmen wir an, in unserem Datenspeicher werden die Einträge wie im vorherigen Beispiel immer nur an eine Datei angefügt. Dann sieht die einfachste Indizierungsstrategie so aus: Führen einer speicherinternen Hashtabelle, in der jeder Schlüssel auf einen Byteoffset in der Datendatei abgebildet wird – dem Ort, an dem der Wert gefunden werden kann, wie es Abbildung 3-1 veranschaulicht. Immer wenn Sie ein neues Schlüssel-Wert-Paar an die Datei anfügen, aktualisieren Sie auch die Hashtabelle, sodass sie den Offset der eben geschriebenen Daten widerspiegelt (dies funktioniert sowohl beim Einfügen von neuen Schlüsseln als auch beim Aktualisieren vorhandener Schlüssel). Wenn Sie einen Wert nachschlagen möchten, suchen Sie in der Hashtabelle den Offset auf die Datendatei heraus, suchen diese Position auf und lesen den Wert.

Dies mag grob vereinfachend erscheinen, doch der Ansatz ist brauchbar. Tatsächlich ist dies im Wesentlichen das, was Bitcask (die Standard-Storage-Engine in Riak) tut [3]. Bitcask bietet hochperformante Lese- und Schreiboperationen, die der Forderung unterliegen, dass alle Schlüssel in den verfügbaren Arbeitsspeicher (RAM) passen, da die Hashtabelle gänzlich im Arbeitsspeicher gehalten wird. Die Werte dürfen mehr Platz belegen als Arbeitsspeicher vorhanden ist, da sie sich von der Festplatte mit lediglich einem einfachen Suchvorgang laden lassen. Falls sich dieser Teil der Datendatei bereits im Cache des Dateisystems befindet, ist für ein Lesen überhaupt keine Festplatten-E/A erforderlich.

Eine Storage-Engine wie Bitcask ist gut geeignet für Situationen, in denen der Wert für jeden Schlüssel häufig aktualisiert wird. So könnte der Schlüssel die URL eines Katzenvideos sein und der Wert eine Zahl, die angibt, wie oft es wiedergege-

ben wurde (jedes Mal inkrementiert, wenn jemand auf die *Play*-Schaltfläche klickt). Bei einer derartigen Arbeitsbelastung gibt es jede Menge Schreibvorgänge, wobei aber die Anzahl unterschiedlicher Schlüssel überschaubar bleibt – man hat eine große Anzahl von Schreibvorgängen pro Schlüssel, doch es ist praktikabel, alle Schlüssel im Arbeitsspeicher zu halten.

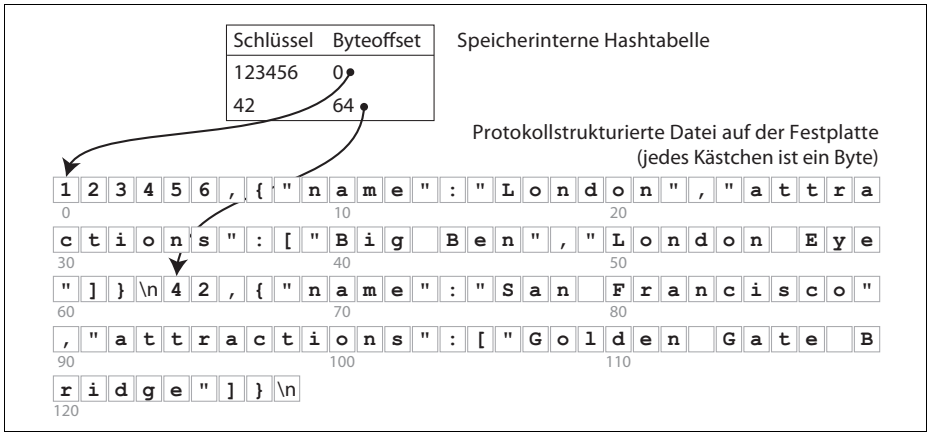

Abbildung 3-1: Ein Protokoll von Schlüssel-Wert-Paaren wird in einem CSV-ähnlichen Format gespeichert, indiziert mit einer speicherinternen Hashtabelle.

Da wir wie bisher beschrieben ausschließlich an eine Datei anfügen, stellt sich die Frage: Wie vermeiden wir, dass uns letztendlich der Festplattenplatz ausgeht? Eine gute Lösung ist es, das Protokoll in Segmente einer bestimmten Größe aufzuteilen, indem man eine Segmentdatei schließt, wenn sie eine bestimmte Größe erreicht hat, und die nächsten Schreibvorgänge in eine neue Segmentdatei ausführt. Diese Segmente können wir dann *komprimieren*, wie Abbildung 3-2 zeigt. Komprimierung bedeutet, doppelte Schlüssel aus dem Protokoll zu entfernen und nur die letzte Aktualisierung für jeden Schlüssel zu behalten.

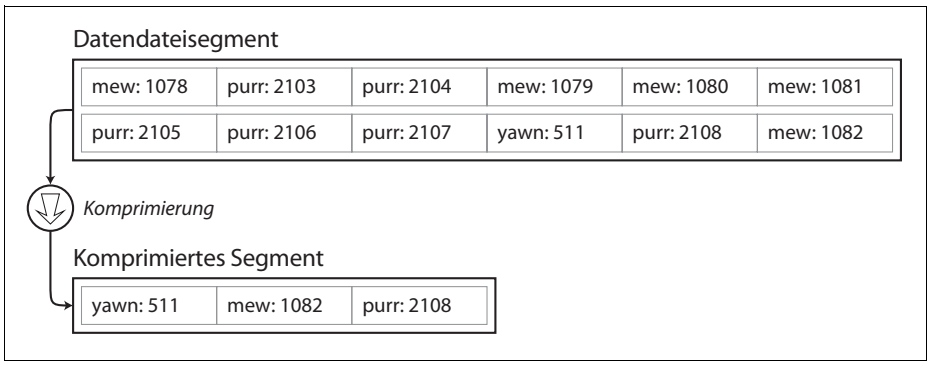

Abbildung 3-2: Komprimierung eines Schlüssel-Wert-Aktualisierungsprotokolls (das zählt, wie oft jedes Katzenvideo wiedergegeben wurde), wobei nur der neueste Wert für jeden Schlüssel beibehalten wird

Da zudem die Segmente beim Komprimieren deutlich kleiner werden (unter der Annahme, dass ein Schlüssel im Durchschnitt mehrmals innerhalb eines Segments überschrieben wird), können wir während der Komprimierung auch mehrere Segmente zusammenführen (siehe Abbildung 3-3). Da Segmente niemals geändert werden, nachdem sie geschrieben wurden, schreiben wir das zusammengeführte Segment in eine neue Datei. Das Zusammenführen und Komprimieren von eingefrorenen Segmenten kann in einem Hintergrund-Thread erfolgen. Und während das geschieht, können wir Lese- und Schreibanfragen wie üblich weiterhin bedienen, indem wir die alten Segmentdateien verwenden. Wenn das Zusammenführen abgeschlossen ist, leiten wir die Leseanfragen auf die neuen, zusammengeführten Segmente anstelle der alten Segmente – und dann können die alten Segmentdateien einfach gelöscht werden.

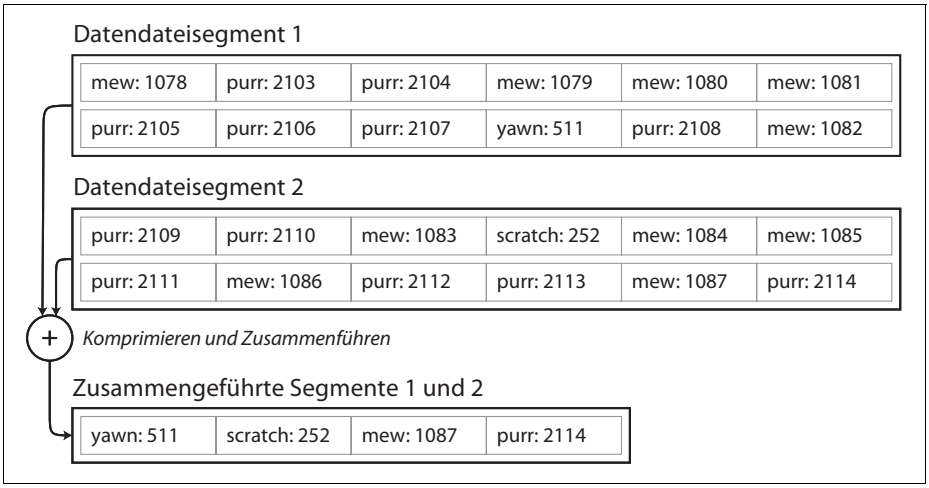

Abbildung 3-3: Komprimieren und Zusammenführen von Segmenten gleichzeitig durchführen

Jedes Segment besitzt jetzt seine eigene speicherinterne Hashtabelle, die Schlüssel auf Dateioffsets abbildet. Um den Wert für einen Schüssel zu finden, überprüfen wir zuerst die Hashtabelle des neuesten Segments. Wenn der Schlüssel nicht vorhanden ist, sehen wir im zweitjüngsten Segment nach usw. Durch das Zusammenführen bleibt die Anzahl der Segmente gering, sodass Nachschlageoperationen nicht sehr viele Hashtabellen überprüfen müssen.

Damit diese einfache Idee in der Praxis auch funktioniert, sind noch jede Menge Details zu berücksichtigen. In einer wirklichen Implementierung sind unter anderem folgende wichtige Punkte zu klären:

Dateiformat
: CSV ist nicht das beste Format für ein Protokoll. Schneller und einfacher zu verwenden ist ein binäres Format, das zuerst die Länge eines Strings in Bytes codiert und daran anschließend den rohen String speichert (ohne dass Escapezeichen notwendig sind).

Datensätze löschen
: Wenn Sie einen Schlüssel und seinen zugeordneten Wert löschen möchten, müssen Sie einen speziellen Löschdatensatz an die Datendatei anfügen (auch als *Tombstone* [Grabstein] bezeichnet). Beim Zusammenführen von Protokollsegmenten erkennt der Zusammenführungsprozess am Löschdatensatz, dass alle vorherigen Werte für den gelöschten Schlüssel zu verwerfen sind.

Wiederherstellung nach Absturz
: Wenn die Datenbank neu gestartet wird, gehen die speicherinternen Hashtabellen verloren. Im Prinzip können Sie die Hashtabelle für jedes Segment wiederherstellen, indem Sie die gesamte Segmentdatei von Anfang bis Ende lesen und sich dabei den Offset des neuesten Werts für jeden Schlüssel notieren. Dies kann jedoch recht lange dauern, wenn die Segmentdateien groß sind, was Serverneustarts besonders schmerzhaft macht. Bitcask beschleunigt die Wiederherstellung, indem ein Snapshot der Hashtabelle für jedes Segment auf der Festplatte abgelegt wird. Diese lässt sich dann schneller in den Arbeitsspeicher laden.

Teilweise geschriebene Datensätze
: Die Datenbank kann jederzeit abstürzen, auch mitten im Anfügen eines Datensatzes an das Protokoll. Bitcask-Dateien enthalten Prüfsummen, sodass sich beschädigte Teile des Protokolls erkennen und ignorieren lassen.

Steuerung der Nebenläufigkeit
: Da die Schreibvorgänge in streng sequenzieller Reihenfolge an das Protokoll angefügt werden, wird dies üblicherweise mit nur einem Schreib-Thread implementiert. An Datendateisegmente kann ausschließlich am Ende angefügt werden, und sie sind anderweitig unveränderlich, sodass sie von mehreren Threads parallel gelesen werden können.

Ein Protokoll im Anfügemodus scheint auf den ersten Blick verschwenderisch zu sein: Warum aktualisiert man die Datei nicht an Ort und Stelle, indem man den alten Wert mit dem neuen Wert überschreibt? Doch ein Konzept, das nur Anfügen am Ende erlaubt, erweist sich aus mehreren Gründen als vorteilhaft:

- Das Anfügen und Zusammenführen von Segmenten sind sequenzielle Schreiboperationen, die im Allgemeinen wesentlich schneller ablaufen als Schreiboperationen mit wahlfreiem Zugriff. Das trifft insbesondere auf Festplattenlaufwerke mit rotierenden Magnetscheiben zu. Zum Teil sind sequenzielle Schreiboperationen auch bei Flash-basierten SSD-Laufwerken (Solid State Drives) zu bevorzugen [4]. Im Abschnitt »B-Bäume und LSM-Bäume im Vergleich« auf Seite 89 gehen wir näher auf dieses Thema ein.
- Nebenläufigkeit und Wiederherstellung bei Abstürzen lassen sich wesentlich einfacher realisieren, wenn die Segmentdateien nur Anfügen erlauben oder unveränderlich sind. Zum Beispiel brauchen Sie sich nicht um den Fall zu kümmern, wenn ein Absturz passiert ist, während ein Wert überschrieben

wird, wobei eine Datei zurückbleibt, die einen Teil des alten und einen Teil des neuen Werts enthält, die miteinander verschweißt sind.

- Das Zusammenführen alter Segmente vermeidet das Problem, dass Datendateien mit der Zeit fragmentiert werden.

Allerdings sind beim Hashtabellenindex ebenfalls Einschränkungen zu beachten:

- Die Hashtabelle muss in den Arbeitsspeicher passen. Wenn Sie also eine sehr große Anzahl von Schlüsseln verwalten müssen, haben Sie Pech. Prinzipiell ließe sich eine Hashtabelle auf der Festplatte verwalten, doch leider ist es schwierig, mit einer festplattengestützten Hashtabelle eine brauchbare Performance zu erzielen. Es sind jede Menge E/A-Operationen mit wahlfreiem Zugriff erforderlich; es ist aufwendig, die Tabelle zu erweitern, wenn sie voll ist; und Hashkollisionen verlangen nach einer ausgefeilten Logik [5].
- Bereichsabfragen sind nicht effizient. Zum Beispiel können Sie nicht einfach nach allen Schlüsseln zwischen `kitty00000` und `kitty99999` suchen – Sie müssten jeden Schlüssel einzeln in den Hashtabellen nachschlagen.

Im nächsten Abschnitt sehen wir uns eine Indexstruktur an, die diese Beschränkungen nicht aufweist.

SSTables und LSM-Bäume

In Abbildung 3-3 ist jedes protokollstrukturierte Speichersegment eine Sequenz von Schlüssel-Wert-Paaren. Diese Paare erscheinen in der Reihenfolge, in der sie geschrieben wurden, und Werte, die weiter hinten in der Protokolldatei stehen, haben Vorrang vor den Werten für denselben Schlüssel weiter vorn im Protokoll. Abgesehen davon spielt die Reihenfolge der Schlüssel-Wert-Paare in der Datei keine Rolle.

Jetzt können wir eine einfache Änderung am Format unserer Segmentdateien vornehmen: Wir fordern, dass die Sequenz der Schlüssel-Wert-Paare nach dem Schlüssel sortiert wird. Auf den ersten Blick sieht es so aus, als ob diese Forderung unsere Fähigkeit unterbindet, sequenzielle Schreibvorgänge zu verwenden, doch dazu kommen wir gleich. Wir nennen dieses Format *Sorted String Table* (kurz *SSTable*). Des Weiteren fordern wir, dass jeder Schlüssel nur einmal innerhalb jeder zusammengeführten Segmentdatei erscheinen darf (das sichert bereits die Komprimierung ab). SSTables bieten mehrere entscheidende Vorteile gegenüber Protokollsegmenten mit Hashindizes:

1. Das Zusammenführen von Segmenten ist einfach und effizient, selbst wenn die Dateien größer als der verfügbare Arbeitsspeicher sind. Das Konzept entspricht dem beim *Mergesort*-Algorithmus verwendeten Vorgang und ist in Abbildung 3-4 zu sehen: Man liest zunächst die Eingabedateien nebeneinander ein, sucht nach dem ersten Schlüssel in jeder Datei, kopiert den kleinsten

Schlüssel (entsprechend der Sortierreihenfolge) in die Ausgabedatei und wiederholt das Ganze. Dabei entsteht eine neue zusammengeführte Segmentdatei, die ebenfalls nach dem Schlüssel sortiert ist.

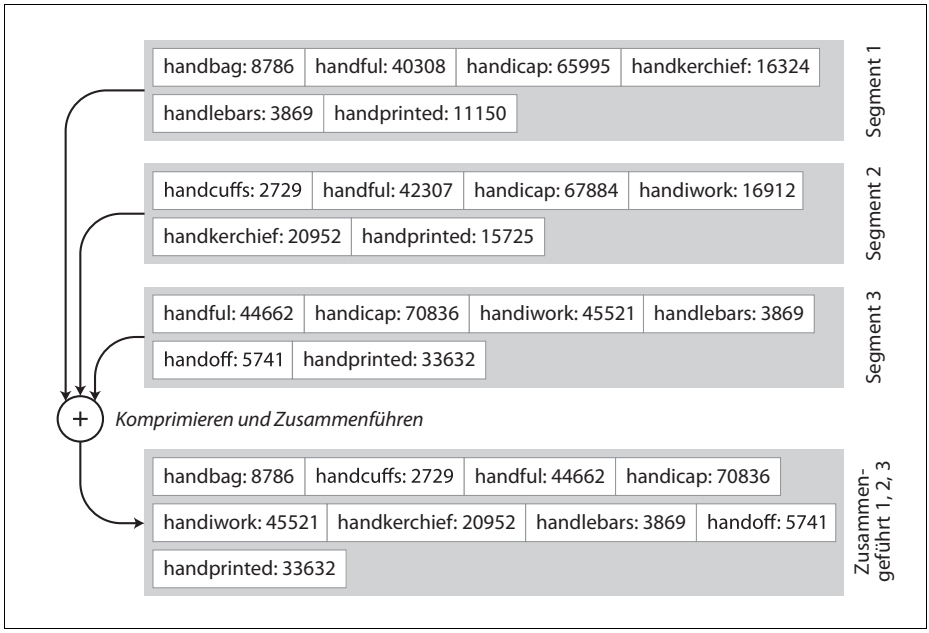

Abbildung 3-4: Zusammenführen mehrerer SSTable-Segmente, wobei nur der neueste Wert für jeden Schlüssel beibehalten wird

Wie sieht es aus, wenn der gleiche Schlüssel in mehreren Eingabesegmenten vorkommt? Denken Sie daran, dass jedes Segment alle Werte enthält, die in einer bestimmten Zeitspanne in die Datenbank geschrieben wurden. Das heißt, dass alle Werte in dem einen Eingabesegment neuer sein müssen als alle Werte im anderen Segment (unter der Annahme, dass wir immer aufeinanderfolgende Segmente zusammenführen). Wenn mehrere Segmente den gleichen Schlüssel enthalten, können wir den Wert aus dem neuesten Segment behalten und die Werte in älteren Segmenten verwerfen.

2. Um einen bestimmten Schlüssel in der Datei zu finden, brauchen Sie nicht mehr den kompletten Index aller Schlüssel im Speicher zu halten. Sehen Sie sich das Beispiel in Abbildung 3-5 an: Angenommen, Sie suchen nach dem Schlüssel `handiwork`, kennen aber nicht den genauen Offset dieses Schlüssels in der Segmentdatei. Allerdings kennen Sie die Offsets für die Schlüssel `handbag` und `handsome`, und aufgrund der Sortierung ist bekannt, dass `handiwork` zwischen diesen beiden liegen muss. Das heißt, Sie können zum Offset für `handbag` springen und von dort aus suchen, bis Sie `handiwork` finden (oder eben nicht, wenn der Schlüssel nicht in der Datei vorhanden ist).

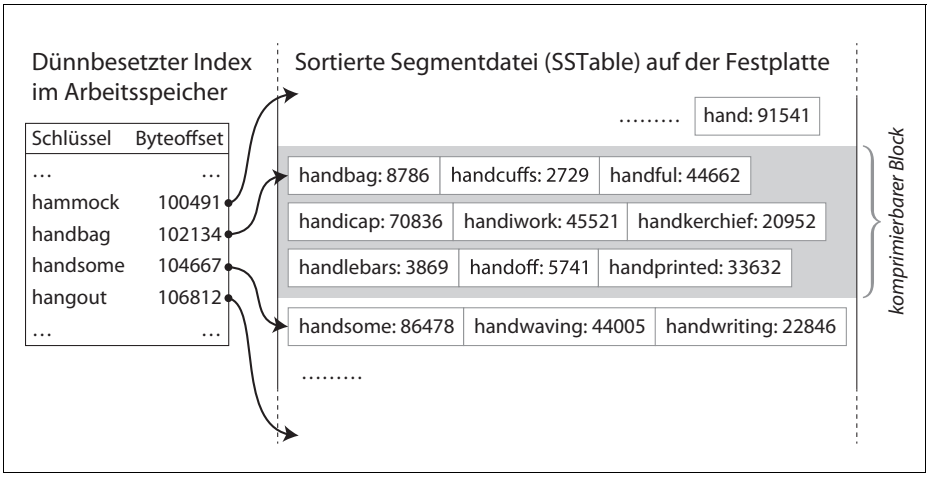

Abbildung 3-5: Eine SSTable mit einem speicherinternen Index

Trotzdem brauchen Sie noch einen speicherinternen Index, der Ihnen die Offsets für einige der Schlüssel angibt, aber er kann dünnbesetzt ausfallen: ein Schlüssel für alle paar Kilobyte der Segmentdatei genügt, weil sich einige Kilobyte sehr schnell durchsuchen lassen.[1]

3. Da Leseanfragen ohnehin über mehrere Schlüssel-Wert-Paare im angefragten Bereich scannen müssen, ist es möglich, diese Datensätze in einem Block zu gruppieren und zu komprimieren, bevor die Daten auf die Festplatte geschrieben werden (durch den schattierten Bereich in Abbildung 3-5 angezeigt). Jeder Eintrag des reduzierten speicherinternen Index zeigt dann auf den Beginn eines komprimierten Blocks. Abgesehen von einer Einsparung an Speicherplatz verringert die Komprimierung auch den E/A-Bandbreitenbedarf.

SSTables konstruieren und verwalten

So weit, so gut – doch wie bekommen Sie Ihre Daten überhaupt erst einmal nach Schlüsseln sortiert? Unsere eingehenden Schreibvorgänge können in beliebiger Reihenfolge auftreten.

Es ist durchaus möglich, eine sortierte Struktur auf der Festplatte zu verwalten (siehe Abschnitt »B-Bäume« auf Seite 85), doch im Arbeitsspeicher lässt sie sich einfacher verwalten. Es gibt viele etablierte Baumdatenstrukturen, die Sie verwenden können, beispielsweise Rot-Schwarz-Bäume oder AVL-Bäume [2]. Mit diesen Datenstrukturen können Sie Schlüssel in jeder Reihenfolge einfügen und sie in sortierter Reihenfolge wieder auslesen.

1 Hätten alle Schlüssel eine feste Größe, könnte man eine binäre Suche auf der Segmentdatei ausführen und auf den speicherinternen Index gänzlich verzichten. Allerdings sind die Schlüssel in der Praxis normalerweise unterschiedlich lang, sodass sich schwer sagen lässt, wo der eine Datensatz endet und der nächste beginnt, wenn man keinen Index hat.

Unsere Storage-Engine können wir nun wie folgt zum Laufen bringen:

- Wenn Schreibdaten eintreffen, werden diese in eine speicherinterne balancierte Baumdatenstruktur eingefügt (zum Beispiel in einen Rot-Schwarz-Baum). Dieser speicherinterne Baum wird auch als *MemTable* bezeichnet.
- Wenn die MemTable größer als ein bestimmter Schwellenwert wird – typischerweise einige Megabyte –, wird sie als SSTable-Datei auf die Festplatte geschrieben. Das lässt sich effizient bewerkstelligen, weil der Baum bereits die Schlüssel-Wert-Paare nach Schlüsseln sortiert verwaltet. Die neue SSTable-Datei wird zum neuesten Segment der Datenbank. Während die SSTable auf Festplatte geschrieben wird, können weitere Schreibvorgänge in eine neue MemTable-Instanz erfolgen.
- Um eine Leseanforderung zu bedienen, wird zuerst versucht, den Schlüssel in der MemTable zu finden, dann im neuesten Segment auf der Festplatte, dann im nächst älteren Segment usw.
- Gelegentlich werden im Hintergrund die Segmentdateien zusammengeführt und komprimiert, um überschriebene oder gelöschte Werte zu verwerfen.

Dieser Ansatz funktioniert gut. Es leidet nur an einem Problem: Wenn die Datenbank abstürzt, gehen die letzten Schreibvorgänge (die zwar in der MemTable stehen, aber noch nicht auf Festplatte geschrieben wurden) verloren. Um dieses Problem zu vermeiden, können wir ein separates Protokoll auf der Festplatte führen, in das jeder Schreibvorgang sofort angefügt wird, genau wie im vorherigen Abschnitt. Dieses Protokoll ist nicht sortiert, doch das spielt keine Rolle, weil sein einziger Zweck darin besteht, die MemTable nach einem Absturz wiederherzustellen. Jedes Mal, wenn die MemTable in eine SSTable übernommen wird, kann das entsprechende Protokoll verworfen werden.

Einen LSM-Baum aus SSTables erstellen

Der hier beschriebene Algorithmus ist praktisch das, was in LevelDB [6] und RocksDB [7] verwendet wird. Diese Storage-Engine-Bibliotheken für Schlüssel-Wert-Paare sind dafür konzipiert, in andere Anwendungen eingebettet zu werden. Unter anderem lässt sich LevelDB in Riak als Alternative zu Bitcask einsetzen. Ähnliche Storage-Engines werden in Cassandra und HBase verwendet [8], die beide durch den *BigTable*-Artikel von Google [9] inspiriert wurden (der die Begriffe *SSTable* und *MemTable* eingeführt hat).

Ursprünglich wurde diese Indexstruktur von Patrick O'Neil et al. unter dem Namen *Log-Structured Merge-Tree* (oder LSM-Tree) [10] beschrieben, aufbauend auf einer früheren Arbeit an protokollstrukturierten Dateisystemen [11]. Storage-Engines, die auf diesem Prinzip von Zusammenführen und Komprimieren sortierter Dateien aufbauen, werden oftmals LSM-Storage-Engines genannt.

Lucene, eine von Elasticsearch und Solr eingesetzte Index-Engine für Volltextsuchen arbeitet nach einer ähnlichen Methode, um ihr *Begriffswörterbuch* zu spei-

chern [12, 13]. Ein Volltextindex ist wesentlich komplexer als ein Schlüssel-Wert-Index, beruht aber auf einer ähnlichen Idee: Finde für ein gegebenes Wort in einer Suchabfrage alle Dokumente (Webseiten, Produktbeschreibungen usw.), in denen das Wort vorkommt. Dies wird mit einer Schlüssel-Wert-Struktur implementiert, bei der der Schlüssel ein Wort (ein *Begriff*) ist und der Wert die Liste der IDs aller Dokumente, die das Wort enthalten (die *Postingsliste*). In Lucene wird diese Abbildung von der Begriffs- auf die Postingsliste in SSTable-ähnlichen sortierten Dateien gehalten, die bei Bedarf im Hintergrund zusammengeführt werden [14].

Performanceoptimierungen

Wie immer ist viel Detailarbeit erforderlich, um in der Praxis eine gute Leistung der Storage-Engine zu erreichen. Zum Beispiel kann der LSM-Baum-Algorithmus beim Nachschlagen von Schlüsseln, die in der Datenbank nicht existieren, langsam sein: Sie müssen die MemTable überprüfen, dann die Segmente bis zurück zum ältesten (und möglicherweise jedes einzelne von der Festplatte lesen), bevor Sie sicher sein können, dass der Schlüssel nicht existiert. Um derartige Zugriffe zu optimieren, verwenden Storage-Engines oftmals zusätzliche *Bloom-Filter* [15]. (Ein Bloom-Filter ist eine speichereffiziente Datenstruktur, um den Inhalt einer Menge anzunähern. Er kann sagen, ob ein Schlüssel in der Datenbank fehlt, und spart somit viele unnötige Festplattenleseoperationen für nicht existierende Schlüssel.)

Es gibt auch andere Strategien, um Reihenfolge und Timing zu bestimmen, wie SSTables komprimiert und zusammengeführt werden. Die gängigsten Optionen sind eine sogenannte *size-tiered* (nach Dateigröße gruppierte) oder eine *leveled* (in Ränge gruppierte) Komprimierung. Die Bibliotheken LevelDB und RocksDB verwenden ranggruppierte Komprimierung (der Name von LevelDB ist vom »leveled« Ansatz hergeleitet), HBase verwendet Dateigrößengruppierung, und Cassandra unterstützt beide [16]. Bei der nach Dateigröße gruppierten Komprimierung werden neuere und kleinere SSTables stufenweise zu älteren und größeren SSTables zusammengeführt. Bei der ranggruppierten Komprimierung wird hingegen der Schlüsselbereich in kleinere SSTables aufgeteilt, und ältere Daten werden in separate »Ränge« verschoben, wodurch die Komprimierung inkrementell ablaufen kann und deshalb weniger Festplattenplatz benötigt.

Obwohl viele Feinheiten zu beachten sind, ist die Grundidee von LSM-Bäumen – eine Kaskade von SSTables zu verwalten, die im Hintergrund zusammengeführt werden – einfach und effektiv. Selbst wenn die Datenmenge wesentlich größer als der verfügbare Arbeitsspeicher ist, funktioniert das Verfahren weiterhin gut. Da die Daten in sortierter Reihenfolge gespeichert werden, können Sie Bereichsabfragen effizient durchführen (alle Schlüssel oberhalb eines Minimums bis zu einem bestimmten Maximum scannen), und weil die Festplattenschreibvorgänge sequenziell erfolgen, kann der LSM-Baum einen bemerkenswert hohen Schreibdurchsatz unterstützen.

B-Bäume

Die protokollstrukturierten Indizes, die wir bisher besprochen haben, gewinnen zwar an Akzeptanz, sind aber nicht die gebräuchlichste Indexart. Am weitesten verbreitet ist eine ganz andere Indexstruktur: der *B-Baum*.

Die 1970 eingeführten [17] und kaum 10 Jahre später als »allgegenwärtig« [18] titulierten B-Bäume haben sich im Laufe der Zeit gut bewährt. In nahezu allen relationalen Datenbanken bleiben sie die Standardindeximplementierung, und viele nichtrelationale Datenbanken nutzen sie ebenfalls.

Wie SSTables halten B-Bäume Schlüssel-Wert-Paare nach dem Schlüssel sortiert, was effiziente Schlüssel-Wert-Nachschlageoperationen und Bereichsabfragen ermöglicht. Doch an dieser Stelle hört die Ähnlichkeit auf: B-Bäume haben eine gänzlich andere Entwurfsphilosophie.

Die protokollstrukturierten Indizes, wie sie weiter vorn zu sehen waren, gliedern die Datenbank in *Segmente* variabler Größe von typischerweise mehreren Megabyte und schreiben ein Segment immer sequenziell. Im Unterschied dazu gliedern B-Bäume die Datenbank in *Blöcke* oder *Seiten* fester Größe von traditionell 4 KB (manchmal auch mehr) und lesen oder schreiben jeweils eine Seite auf einmal. Dieses Design kommt der zugrunde liegenden Hardware näher, da Festplatten ebenfalls in Blöcke fester Größe unterteilt sind.

Jede Seite kann mit einer Adresse oder einer Position identifiziert werden, wodurch eine Seite auf eine andere verweisen kann – ähnlich einem Zeiger, aber auf Festplatte statt im Arbeitsspeicher. Diese Seitenreferenzen können wir verwenden, um einen Baum von Seiten zu konstruieren, wie Abbildung 3-6 veranschaulicht.

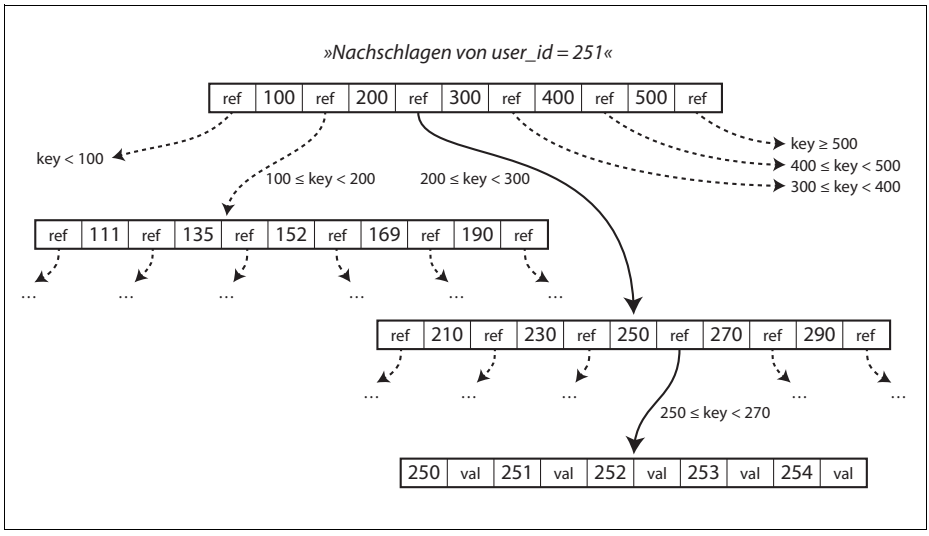

Abbildung 3-6: Nachschlagen eines Schlüssels mit einem B-Baum-Index

Eine bestimmte Seite ist als Wurzel des B-Baums festgelegt. Hier beginnen Sie immer, wenn Sie nach einem Schlüssel im Index suchen. Die Seite enthält mehrere Schlüssel und Referenzen auf untergeordnete Seiten. Jede untergeordnete Seite ist für einen zusammenhängenden Schlüsselbereich zuständig, und die Schlüssel zwischen den Referenzen zeigen an, wo die Grenzen zwischen diesen Bereichen liegen.

Da wir im Beispiel, das Abbildung 3-6 zeigt, nach dem Schlüssel 251 suchen, wissen wir, dass wir der Seitenreferenz zwischen den Grenzen 200 und 300 folgen müssen. Dies bringt uns zu einer ähnlich aussehenden Seite, die den Bereich von 200 bis 300 weiter in Teilbereiche unterteilt.

Schließlich gelangen wir zu einer Seite, die einzelne Schlüssel enthält (eine *Blattseite*), die entweder den Wert für jeden Schlüssel inline speichert oder die Seiten referenziert, wo die eigentlichen Werte gefunden werden können.

Die Anzahl der Referenzen auf untergeordnete Seiten in einer Seite des B-Baums ist der *Verzweigungsgrad*. So ist im Beispiel von Abbildung 3-6 der Verzweigungsgrad gleich 6. In der Praxis hängt der Verzweigungsgrad davon ab, wie viele Bytes benötigt werden, um die Seitenreferenzen und die Bereichsgrenzen zu speichern; typischerweise beträgt der Verzweigungsgrad mehrere Hundert.

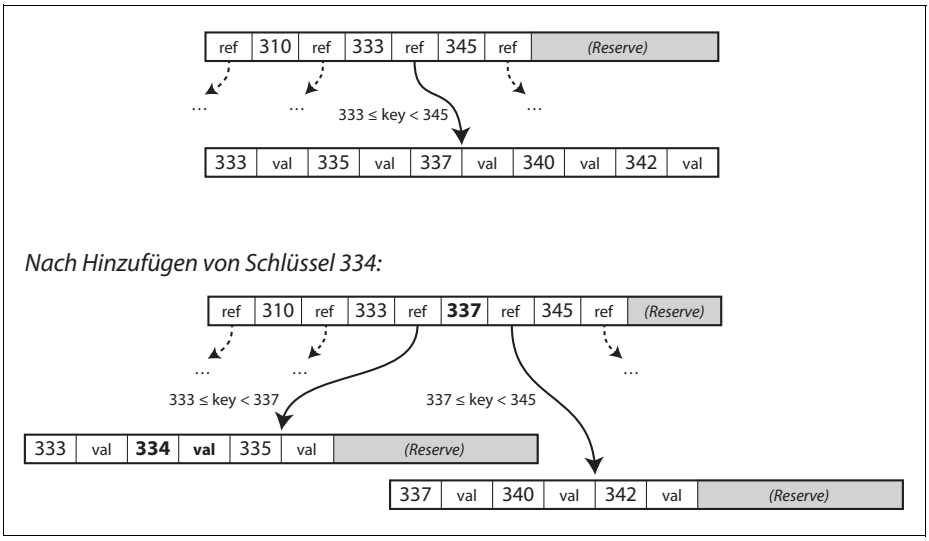

Abbildung 3-7: Wachsen eines B-Baums durch Teilen einer Seite

Wenn Sie den Wert für einen vorhandenen Schlüssel in einem B-Baum aktualisieren wollen, suchen Sie nach der Blattseite, die diesen Schlüssel enthält, ändern den Wert in dieser Seite und schreiben die Seite zurück auf die Festplatte (Verweise von anderen Seiten auf diese Seite bleiben dabei gültig). Möchten Sie einen neuen Schlüssel hinzufügen, müssen Sie die Seite suchen, deren Bereich den neuen Schlüssel umschließt, und ihn zu dieser Seite hinzufügen. Wenn in der Seite nicht genügend Platz für den neuen Schlüssel ist, wird sie in zwei halbvolle Seiten geteilt

und die übergeordnete Seite wird aktualisiert, um die neue Unterteilung der Schlüsselbereiche zu erfassen – siehe Abbildung 3-7.[2]

Dieser Algorithmus stellt sicher, dass der Baum *balanciert* bleibt: Ein B-Baum mit n Schlüsseln hat immer eine Tiefe von $O(\log n)$. Die meisten Datenbanken passen in einen B-Baum, der drei oder vier Ebenen tief ist. Somit brauchen Sie nicht vielen Seitenverweisen zu folgen, um die gesuchte Seite zu finden. (Ein Baum mit vier Ebenen, 4 KB großen Seiten und mit einem Verzweigungsfaktor von 500 kann bis zu 256 TB speichern.)

B-Bäume zuverlässig machen

Beim Schreiben in einem B-Baum wird prinzipiell eine Seite auf der Festplatte mit den neuen Daten überschrieben. Es wird davon ausgegangen, dass sich beim Überschreiben die Position der Seite nicht ändert; d.h., alle Referenzen auf diese Seite bleiben intakt, wenn die Seite überschrieben wird. Dieses Vorgehen steht in krassem Kontrast zu protokollstrukturierten Indizes wie zum Beispiel LSM-Bäumen, die ausschließlich an Dateien anfügen (und letztlich veraltete Dateien löschen), aber niemals Dateien direkt modifizieren.

Das Überschreiben einer Seite auf Festplatte können Sie sich als echte Hardwareoperation vorstellen. Auf einer magnetischen Festplatte verschiebt die Steuerung den Schreib-Lese-Kopf zur richtigen Spur, wartet auf die richtige Position der sich drehenden Magnetplatten und überschreibt dann den jeweiligen Sektor mit neuen Daten. Auf SSDs sind die Vorgänge etwas komplizierter, was mit dem Umstand zusammenhängt, dass eine SSD ziemlich große Blöcke auf dem Speicherchip auf einmal löschen und neu beschreiben muss [19].

Darüber hinaus ist es bei manchen Vorgängen erforderlich, mehrere verschiedene Seiten zu überschreiben. Wenn Sie zum Beispiel eine Seite teilen, weil sie beim Einfügen zu voll geworden wäre, müssen Sie die beiden geteilten Seiten schreiben und außerdem deren übergeordnete Seite überschreiben, um die Referenzen auf die beiden untergeordneten Seiten zu aktualisieren. Diese Operation ist gefährlich, denn wenn die Datenbank abstürzt, nachdem nur einige der Seiten geschrieben worden sind, ist das Ergebnis ein beschädigter Index (zum Beispiel kann eine *verwaiste* Seite entstehen, die zu keiner übergeordneten Seite mehr gehört).

Um die Datenbank gegenüber Abstürzen robust zu machen, verwalten Implementierungen von B-Bäumen üblicherweise eine zusätzliche Datenstruktur auf der Festplatte: ein *Write-Ahead-Log* (WAL) – auch als *Wiederholungsprotokoll* (Redo-Log) bekannt. Dies ist eine Datei, an die jede B-Baum-Modifikation angefügt werden muss, bevor sie auf die Seiten des Baums selbst angewendet werden darf. Wenn die Datenbank nach einem Absturz wieder gestartet wird, lässt sich anhand dieses Protokolls der B-Baum in einen konsistenten Zustand wiederherstellen [5, 20].

2 Einen neuen Schlüssel kann man relativ einfach in einen B-Baum einfügen; einen Schlüssel zu löschen (und dabei den Baum balanciert zu halten), ist dagegen etwas komplizierter [2].

Beim direkten Aktualisieren von Seiten ist zusätzlich zu berücksichtigen, dass eine sorgfältige Nebenläufigkeitssteuerung erforderlich ist, wenn mehrere Threads gleichzeitig auf den B-Baum zugreifen – andernfalls kann ein Thread den Baum in einem inkonsistenten Zustand vorfinden. Dies geschieht in der Regel dadurch, dass die Datenstrukturen des Baums mit *Indexsperren* (engl. *Latches*; leichtgewichtige Sperren) geschützt werden. Protokollstrukturierte Ansätze sind in dieser Hinsicht einfacher, weil sie das gesamte Zusammenführen im Hintergrund abwickeln, ohne mit eingehenden Abfragen zu kollidieren, und alte Segmente von Zeit zu Zeit atomar gegen neue Segmente austauschen.

Optimierungen von B-Bäumen

Da B-Bäume schon so lange existieren, überrascht es nicht, dass im Lauf der Jahre viele Optimierungen entwickelt worden sind. Um nur ein paar zu nennen:

- Anstatt Seiten zu überschreiben und ein WAL für die Wiederherstellung bei Abstürzen zu verwalten, verwenden manche Datenbanken (wie LMDB) ein Kopieren-beim-Schreiben-Prinzip [21]. Eine modifizierte Seite wird an einen anderen Ort geschrieben, und im Baum wird eine neue Version der übergeordneten Seiten angelegt, die auf die neue Position verweisen. Dieses Konzept ist auch nützlich für die Kontrolle der Nebenläufigkeit, wie der Abschnitt »Snapshot-Isolation und Repeatable Read« auf Seite 252 erläutert.
- Auf den Seiten lässt sich Platz sparen, wenn man nicht den gesamten Schlüssel speichert, sondern ihn abkürzt. Speziell auf Seiten im Inneren des Baums müssen Schlüssel nur so viele Informationen liefern, dass sie als Grenzen zwischen Schlüsselbereichen fungieren können. Werden mehr Schlüssel auf eine Seite gepackt, kann der Baum einen höheren Verzweigungsgrad und folglich weniger Ebenen haben.[3]
- Im Allgemeinen können Seiten an beliebigen Stellen auf der Festplatte untergebracht werden; es gibt keinerlei Forderungen, dass Seiten mit benachbarten Schlüsselbereichen auch auf der Festplatte nahe beieinanderliegen müssten. Wenn eine Abfrage über einen großen Teil des Schlüsselbereichs in sortierter Reihenfolge scannen muss, ist dieses seitenorientierte Layout eventuell ineffizient, weil eine Festplattenpositionierung für jede gelesene Seite erforderlich sein kann. Viele B-Baum-Implementierungen versuchen deshalb, den Baum so anzuordnen, dass die Blattseiten auf der Festplatte in sequenzieller Reihenfolge liegen. Allerdings ist es schwierig, diese Reihenfolge aufrechtzuerhalten, wenn der Baum wächst. Dagegen schreiben LSM-Bäume beim Zusammenführen große Segmente des Speicherinhalts in einem Zug, sodass es für sie einfacher ist, aufeinanderfolgende Schlüssel nahe beieinander auf der Festplatte abzulegen.

3 Diese Variante wird auch als B^+-Baum bezeichnet, obwohl die Optimierung so häufig vorkommt, dass man diese Variante oftmals nicht von anderen B-Baum-Varianten unterscheidet.

- Zusätzliche Zeiger können in den Baum eingefügt werden. Zum Beispiel kann jede Blattseite Verweise auf ihre gleichgeordneten Seiten nach links und rechts haben, sodass sich Schlüssel der Reihe nach suchen lassen, ohne zu übergeordneten Seiten zurückspringen zu müssen.
- B-Baum-Varianten wie zum Beispiel *fraktale Bäume* [22] orientieren sich zum Teil an protokollstrukturierten Konzepten, um die Anzahl der Festplattenpositionierungen zu verringern (sie haben allerdings nichts mit Fraktalen zu tun).

B-Bäume und LSM-Bäume im Vergleich

Obwohl B-Baum-Implementierungen im Allgemeinen ausgereifter sind als LSM-Baum-Implementierungen, sind LSM-Bäume auch wegen ihrer Performanceeigenschaften interessant. Als Faustregel gilt, dass Schreibvorgänge bei LSM-Bäumen und Lesevorgänge bei B-Bäumen typischerweise schneller sind [23]. Lesevorgänge sind bei LSM-Bäumen in der Regel langsamer, weil sie mehrere verschiedene Datenstrukturen und SSTables auf verschiedenen Komprimierungsrängen prüfen müssen.

Allerdings sind Ergebnisse von Benchmarks oftmals nicht schlüssig und hängen stark von Details der Arbeitsbelastung ab. Um brauchbare Vergleiche durchführen zu können, müssen Sie die Systeme mit Ihrer konkreten Arbeitsbelastung testen. In diesem Abschnitt gehen wir kurz auf einige Dinge ein, die Sie bei Leistungsmessungen einer Storage-Engine beachten sollten.

Vorteile von LSM-Bäumen

Ein B-Baum-Index muss jedes Datenelement mindestens zweimal schreiben: einmal in das Write-Ahead-Log und einmal auf die Baumseite selbst (und vielleicht erneut, wenn Seiten geteilt werden). Außerdem entsteht ein Overhead, weil eine ganze Seite geschrieben werden muss, selbst wenn sich auf dieser Seite nur wenige Bytes geändert haben. Manche Storage-Engines überschreiben dieselbe Seite sogar zweimal, damit zum Beispiel bei einem Stromausfall keine nur teilweise aktualisierte Seite zurückbleibt [24, 25].

Protokollstrukturierte Indizes schreiben die Daten ebenfalls mehrmals neu, was mit wiederholtem Komprimieren und Zusammenführen von SSTables zusammenhängt. Dieser Effekt – ein Schreibvorgang in der Datenbank führt zu mehreren Schreibvorgängen auf der Festplatte während der Lebenszeit der Datenbank – wird als *Write Amplification* (Schreibverstärkung) bezeichnet. Dies ist besonders kritisch bei SSD-Laufwerken, bei denen die Anzahl der Schreibzyklen technologisch begrenzt ist, Blöcke also nicht beliebig oft überschrieben werden können, bevor die Speicherzellen verschlissen sind.

Bei schreibintensiven Anwendungen kann der Engpass die Rate sein, mit der die Datenbank auf Festplatte schreiben kann. In diesem Fall schlägt sich die Write Amplification direkt in Performancekosten nieder: Je mehr eine Storage-Engine

auf Festplatte schreibt, desto weniger Schreiboperationen pro Sekunde kann sie innerhalb der verfügbaren Festplattenbandbreite verarbeiten.

Darüber hinaus sind LSM-Bäume typischerweise in der Lage, einen höheren Schreibdurchsatz als B-Bäume aufrechtzuerhalten, einerseits, weil sie manchmal eine geringere Write Amplification haben (obwohl diese von der Konfiguration der Storage-Engine und der Arbeitslast abhängt), und andererseits, weil sie sequenziell kompakte SSTable-Dateien schreiben, anstatt mehrere Seiten im Baum überschreiben zu müssen [26]. Dieser Unterschied ist besonders wichtig bei magnetischen Festplatten, wo sequenzielle Schreiboperationen wesentlich schneller als wahlfreie Schreiboperationen sind.

LSM-Bäume können besser komprimiert werden und ergeben folglich oftmals kleinere Dateien auf der Festplatte als B-Bäume. B-Baum-Storage-Engines nutzen aufgrund der Fragmentierung nicht den gesamten Festplattenplatz aus: Wenn eine Seite geteilt wird oder wenn eine Zeile nicht auf eine vorhandene Seite passt, bleibt Platz in einer Seite ungenutzt. Da LSM-Bäume nicht seitenorientiert sind und regelmäßig SSTables neu schreiben, um die Fragmentierung zu beseitigen, haben sie einen geringeren Speicheroverhead, und zwar insbesondere, wenn sie ranggruppierte Komprimierung verwenden [27].

Auf vielen SSDs verwendet die Firmware intern einen protokollstrukturierten Algorithmus, um wahlfreie Schreibvorgänge in sequenzielle Schreibvorgänge auf den zugrunde liegenden Speicherchips umzuwandeln, was dazu führt, dass sich das Schreibmuster der Storage-Engine weniger stark bemerkbar macht [19]. Allerdings sind eine geringere Write Amplification und verringerte Fragmentierung bei SSDs trotzdem von Vorteil: Die kompaktere Darstellung der Daten erlaubt mehr Lese- und Schreibzugriffe innerhalb der verfügbaren E/A-Bandbreite.

Nachteile von LSM-Bäumen

Nachteilig bei der protokollstrukturierten Speicherung ist, dass der Komprimierungs- und Zusammenführungsvorgang manchmal die Performance von laufenden Lese- und Schreiboperationen beeinträchtigen kann. Selbst wenn Storage-Engines versuchen, die Komprimierung inkrementell auszuführen und ohne gleichzeitige Zugriffe zu beeinflussen, kann es durch die begrenzten Ressourcen von Festplatten leicht passieren, dass eine Anforderung warten muss, während die Festplatte einen umfangreichen Komprimierungsvorgang abschließt. Der Einfluss auf den Durchsatz und die mittlere Reaktionszeit ist normalerweise gering, doch bei höheren Perzentilen (siehe Abschnitt »Performance beschreiben« auf Seite 14) kann die Reaktionszeit von Anfragen an protokollstrukturierte Storage-Engines manchmal ziemlich lang sein, und B-Bäume können einheitlichere Performance zeigen [28].

Ein anderes Problem bei der Komprimierung entsteht bei einem hohen Schreibdurchsatz: Die endliche Schreibbandbreite der Festplatte müssen sich der anfäng-

liche Schreibvorgang (Protokollieren und Schreiben einer MemTable auf die Festplatte) und die im Hintergrund laufenden Komprimierungsthreads teilen. Beim Schreiben in eine leere Datenbank steht die volle Bandbreite für den anfänglichen Schreibvorgang zur Verfügung, doch je größer die Datenbank wird, desto mehr Bandbreite ist für die Komprimierung erforderlich.

Wenn bei einem hohen Schreibdurchsatz die Komprimierung nicht sorgfältig konfiguriert ist, kann die Komprimierung gegebenenfalls nicht mehr mit der Rate der eingehenden Schreibanforderungen Schritt halten. In diesem Fall wächst die Anzahl der nicht zusammengeführten Segmente auf der Festplatte, bis kein Festplattenplatz mehr übrig ist. Auch die Leseoperationen laufen langsamer ab, weil sie mehr Segmentdateien prüfen müssen. Typischerweise drosseln SSTable-basierte Storage-Engines die eingehenden Schreibanforderungen nicht, selbst wenn die Komprimierung nicht Schritt halten kann. Diese Situation müssen Sie deshalb explizit überwachen und erkennen [29, 30].

B-Bäume haben unter anderem den Vorteil, dass jeder Schlüssel an genau einer Stelle im Index vorhanden ist, während eine protokollstrukturierte Storage-Engine mehrere Kopien desselben Schlüssels in verschiedenen Segmenten haben kann. Dieser Aspekt macht B-Bäume attraktiv für Datenbanken, die strenge transaktionale Semantik bieten wollen: In vielen relationalen Datenbanken wird die Transaktionsisolation durch Sperren auf Schlüsselbereichen implementiert, und in einem B-Baum-Index können diese Sperren unmittelbar dem Baum zugewiesen werden [5]. In Kapitel 7 kommen wir auf diesen Punkt ausführlich zu sprechen.

B-Bäume sind in der Architektur von Datenbanken tief verwurzelt und bieten eine durchgängig gute Performance für viele Arbeitslasten. Es ist also unwahrscheinlich, dass sie in absehbarer Zeit von der Bildfläche verschwinden werden. In neueren Datenspeichern werden protokollstrukturierte Indizes zunehmend beliebter. Da es keine Regel gibt, nach der Sie schnell und einfach bestimmen könnten, welcher Typ von Storage-Engine für Ihren Anwendungsfall besser geeignet ist, lohnt es sich, empirisch zu testen.

Andere Indizierungsstrukturen

Bislang haben wir uns nur mit Schlüssel-Wert-Indizes beschäftigt, die etwa einem *Primärschlüsselindex* im rationalen Modell entsprechen. Ein Primärschlüssel identifiziert eindeutig eine Zeile in einer relationalen Tabelle, ein Dokument in einer Dokumentendatenbank oder einen Knoten in einer Graphdatenbank. Andere Datensätze in der Datenbank können auf diese Zeile, das Dokument oder den Knoten mit dem entsprechenden Primärschlüssel (oder der ID) verweisen, und der Index dient dazu, solche Verweise aufzulösen.

Es ist auch üblich, *sekundäre Indizes* anzulegen. In relationalen Datenbanken können Sie mehrere sekundäre Indizes auf derselben Tabelle mit der Anweisung `CREATE INDEX` anlegen. Oftmals sind solche Indizes entscheidend, um Joins effizient

ausführen zu können. Zum Beispiel würden Sie in Abbildung 2-1 von Kapitel 2 höchstwahrscheinlich einen sekundären Index auf den `user_id`-Spalten einrichten, um effizient alle Zeilen zu finden, die in jeder der Tabellen zum selben Benutzer gehören.

Ein sekundärer Index lässt sich leicht aus einem Schlüssel-Wert-Index konstruieren. Der Unterschied zu einem Primärindex besteht vor allem darin, dass Schlüssel nicht eindeutig sind; d.h., es kann viele Zeilen (bzw. Dokumente oder Knoten) mit dem gleichen Schlüssel geben. Dies lässt sich nach zwei Methoden auflösen: Entweder macht man jeden Wert im Index zu einer Liste von übereinstimmenden Zeilenbezeichnern (wie zum Beispiel eine Postings-Liste in einem Volltextindex), oder man macht jeden Schlüssel eindeutig, indem man ihm einen Zeilenbezeichner anfügt. In jedem Fall können sowohl B-Bäume als auch protokollstrukturierte Indizes als sekundäre Indizes verwendet werden.

Werte im Index speichern

In einem Index ist es der Schlüssel, wonach Abfragen suchen, doch der Wert kann zweierlei sein: Er könnte die tatsächlich gesuchte Zeile (bzw. Dokument oder Knoten) sein oder ein Verweis auf die Zeile, die an anderer Stelle gespeichert ist. Im zweiten Fall ist der Ort, an dem die Zeilen gespeichert werden, die sogenannte *Heap-Datei*. Sie speichert die Daten in einer unbestimmten Reihenfolge. (Die Heap-Datei verwaltet typischerweise den Speicherplatz in einer Weise, die erlaubt, gelöschte Zeilen später mit neuen Daten zu überschreiben.) Der Ansatz mit Heap-Datei ist gebräuchlich, weil er doppelte Daten vermeidet, wenn mehrere Sekundärindizes vorhanden sind: Jeder Index verweist einfach auf eine Stelle in der Heap-Datei, und die eigentlichen Daten werden an einem Ort für sich gespeichert.

Wird ein Wert aktualisiert, ohne den Schlüssel zu ändern, kann der Ansatz mit Heap-Datei recht effizient sein: Der Datensatz kann an Ort und Stelle überschrieben werden, sofern der neue Wert nicht mehr Platz benötigt als der alte Wert. Die Situation ist komplizierter, wenn der neue Wert größer ist, denn er muss möglicherweise an eine neue Position im Heap verschoben werden, wo genügend Platz zur Verfügung steht. In diesem Fall müssen entweder alle Indizes aktualisiert werden, um auf die neue Heap-Position des Datensatzes zu verweisen, oder es wird an der alten Heap-Position ein *Weiterleitungszeiger* hinterlassen [5].

In manchen Situationen bedeutet der zusätzliche Sprung vom Index zur Heap-Datei zu viel Einbuße an Performance bei Leseoperationen, sodass es wünschenswert sein kann, die indizierte Zeile direkt innerhalb eines Index zu speichern. Dies wird als *gruppierter Index* (clustered index) bezeichnet. Zum Beispiel ist in der Storage-Engine InnoDB von MySQL der Primärschlüssel einer Tabelle immer ein gruppierter Index, und Sekundärindizes verweisen auf den Primärschlüssel (statt auf eine Position in der Heap-Datei) [31]. In SQL Server können Sie pro Tabelle einen gruppierten Index vorgeben [32].

Ein Kompromiss zwischen einem gruppierten Index (der alle Zeilendaten innerhalb des Index speichert) und eines nicht gruppierten Index (der innerhalb des Index nur Verweise auf die Daten speichert) wird als *abdeckender Index* (covering index) oder *Index mit eingeschlossenen Spalten* bezeichnet. Er speichert *ausgewählte* Spalten einer Tabelle innerhalb des Index [33]. Auf diese Weise lassen sich manche Abfragen allein durch Verwendung des Index beantworten (in diesem Fall sagt man, dass der Index die Abfrage *abdeckt*) [32].

Wie bei jeder Art von Datenduplizierung können gruppierte und abdeckende Indizes Leseoperationen beschleunigen, doch sie erfordern zusätzlichen Speicher und können Overhead bei Schreibvorgängen verursachen. Datenbanken müssen zudem zusätzlichen Aufwand betreiben, um transaktionale Garantien durchzusetzen, weil Anwendungen keine Inkonsistenzen aufgrund der Duplizierung sehen sollten.

Mehrspaltige Indizes

Die bisher vorgestellten Indizes bilden nur einen einzelnen Schlüssel auf einen Wert ab. Das genügt nicht, wenn wir mehrere Spalten einer Tabelle (oder mehrere Felder in einem Dokument) gleichzeitig abfragen müssen.

Die gebräuchlichste Art eines mehrspaltigen Index ist ein sogenannter *zusammengesetzter Index* (concatenated index), der einfach mehrere Felder zu einem Schlüssel zusammenfasst, indem eine Spalte an eine andere angefügt wird (wobei die Indexdefinition festlegt, in welcher Reihenfolge die Felder verkettet werden). Dies ist wie bei einem althergebrachten gedruckten Telefonbuch, das einen Index von (*Nachname*, *Vorname*) zur Telefonnummer bietet. Aufgrund der Sortierreihenfolge eignet sich der Index, um alle Personen mit einem bestimmten Nachnamen zu finden oder alle Leute mit einer bestimmten *Nachname*-*Vorname*-Kombination. Der Index ist jedoch nutzlos, wenn Sie alle Personen mit einem bestimmten Vornamen ermitteln wollen.

Mehrdimensionale Indizes sind eine allgemeinere Methode, mehrere Spalten auf einmal abzufragen, was vor allem für Geodaten/raumbezogene Daten wichtig ist. Zum Beispiel könnte eine Website für Restaurantsuchen eine Datenbank mit den Breiten- und Längengradangaben jedes Restaurants führen. Wenn sich ein Benutzer die Restaurants auf einer Karte ansieht, muss die Website nach allen Restaurants innerhalb des rechteckigen Kartenausschnitts suchen, den der Benutzer gerade betrachtet. Dazu ist eine Abfrage für einen zweidimensionalen Bereich wie im folgenden Beispiel erforderlich:

```
SELECT * FROM restaurants WHERE latitude > 51.4946 AND latitude < 51.5079
                           AND longitude > -0.1162 AND longitude < -0.1004;
```

Ein standardmäßiger B-Baum- oder LSM-Baum-Index ist nicht in der Lage, derartige Abfragen effizient zu beantworten: Er kann Ihnen entweder alle Restaurants in einem Bereich von Breitengraden (aber mit beliebigen Längengraden) oder alle

Restaurants in einem Bereich von Längengraden (aber irgendwo zwischen Nord- und Südpol) liefern, aber nicht beide gleichzeitig.

Eine Möglichkeit besteht darin, einen zweidimensionalen Ort mithilfe einer raumfüllenden Kurve in eine einzelne Zahl zu übersetzen und dann einen normalen B-Baum-Index zu verwenden [34]. Gebräuchlicher sind spezialisierte räumliche Indizes wie zum Beispiel R-Bäume. So implementiert PostGIS Geodatenindizes als R-Baum mit der *Generalized Search Tree*-(GiST-)Schnittstelle von PostgreSQL [35]. Aus Platzgründen können wir nicht im Detail auf R-Bäume eingehen, doch gibt es hierzu jede Menge Literatur.

Interessant sind auch andere Einsatzarten von mehrdimensionalen Indizes, nicht nur für geografische Orte. Zum Beispiel könnten Sie auf einer E-Commerce-Website einen dreidimensionalen Index auf den Dimensionen (*Rot*, *Grün*, *Blau*) verwenden, um nach Produkten in einem bestimmten Farbenbereich zu suchen, oder in einer Datenbank mit Wetterbeobachtungen einen zweidimensionalen Index auf (*Datum*, *Temperatur*) einrichten, um nach allen Beobachtungen im Jahr 2013 zu suchen, bei denen die Temperatur zwischen 25 und 30 °C lag. Mit einem eindimensionalen Index müssten Sie entweder sämtliche Datensätze von 2013 (ohne Beachtung der Temperatur) durchsuchen und sie dann nach Temperatur filtern oder umgekehrt. Mit einem zweidimensionalen Index ließen sich die Daten gleichzeitig nach Zeitstempel und Temperatur einengen. HyperDex verwendet diese Technik [36].

Volltextsuche und Fuzzy-Indizes

Alle bisher behandelten Indizes gehen davon aus, dass Sie über genaue Daten verfügen und nach genauen Werten eines Schlüssels oder einem Bereich von Schlüsselwerten mit einer Sortierreihenfolge abfragen können. Dagegen erlauben sie Ihnen nicht, nach *ähnlichen* Schlüsseln zu suchen, beispielsweise nach falsch geschriebenen Wörtern. Eine derartige *unscharfe* (engl. fuzzy) Abfrage verlangt andere Techniken.

Beispielsweise erlauben es Volltextsuchmaschinen üblicherweise, die Suche nach einem Wort auf Synonyme des Worts auszudehnen, grammatikalische Variationen des Worts zu ignorieren und nach nahe beieinanderliegenden Vorkommen des Worts im selben Dokument zu suchen. Zudem unterstützen sie verschiedene andere Features, die von der linguistischen Analyse des Texts abhängen. Um mit Tippfehlern in Dokumenten oder Abfragen zurechtzukommen, ist Lucene in der Lage, Text nach Wörtern innerhalb einer bestimmten Editierdistanz (auch Levenshtein-Distanz) zu suchen (wobei eine Editierdistanz von 1 bedeutet, dass genau ein Buchstabe hinzugefügt, entfernt oder ersetzt wurde) [37].

Wie in Abschnitt »Einen LSM-Baum aus SSTables erstellen« auf Seite 83 erwähnt, verwendet Lucene eine SSTable-ähnliche Struktur für sein Begriffswörterbuch.

Diese Struktur benötigt einen kleinen speicherinternen Index, dem Abfragen entnehmen können, bei welchem Offset in der sortierten Datei sie nach einem Schlüssel suchen müssen. In LevelDB ist dieser speicherinterne Index eine schwach besetzte Auflistung einiger Schlüssel, während in Lucene der speicherinterne Index durch einen endlichen Automaten über die Zeichen der Schlüssel – ähnlich einem *Trie* (Präfixbaum) – realisiert wird [38]. Dieser Automat lässt sich in einen *Levenshtein*-Automaten transformieren, der eine effiziente Suche nach Wörtern innerhalb einer vorgegebenen Editierdistanz unterstützt [39].

Andere unscharfe Suchtechniken gehen in Richtung Dokumentklassifizierung und maschinelles Lernen. Weitere Details hierzu finden Sie in Lehrbüchern zur Informationsgewinnung [zum Beispiel 40].

Alles im Arbeitsspeicher halten

Die bisher in diesem Kapitel besprochenen Datenstrukturen sind alles Antworten auf Beschränkungen von Festplatten gewesen. Verglichen mit dem Hauptspeicher ist der Umgang mit Festplatten umständlich. Sowohl bei magnetischen Festplatten als auch bei SSDs müssen Sie die Daten sorgfältig auf dem Datenträger anordnen, wenn Sie gute Performance bei Lese- und Schreibvorgängen erzielen wollen. Allerdings tolerieren wir diese Unbequemlichkeit, weil Festplatten zwei erhebliche Vorteile bieten: Sie sind dauerhaft (ihr Inhalt geht nicht verloren, wenn der Strom abgeschaltet wird), und die Kosten pro Gigabyte sind geringer als bei RAM.

Mit billiger werdendem RAM wird das Argument Kosten pro Gigabyte ausgehöhlt. Viele Datensätze sind einfach nicht sonderlich groß, sodass es durchaus machbar ist, sie vollständig im Arbeitsspeicher zu halten, eventuell über mehrere Computer hinweg verteilt. Dies hat zur Entwicklung von speicherinternen Datenbanken geführt.

Einige speicherinterne Schlüssel-Wert-Speicher wie zum Beispiel Memcached sind allein für Caching-Zwecke vorgesehen, wo ein Datenverlust durch einen Neustart des Computers akzeptierbar ist. Andere speicherinterne Datenbanken sind dagegen auf Dauerhaftigkeit ausgelegt, die sich mit spezieller Hardware (wie zum Beispiel batteriegestütztem RAM), durch Schreiben eines Änderungsprotokolls auf Festplatte, durch periodisches Schreiben von Snapshots auf Festplatte oder durch Replizieren des speicherinternen Zustands auf andere Computer erreichen lässt. Wird eine speicherinterne Datenbank neu gestartet, muss sie ihren Zustand erneut laden, und zwar entweder von Festplatte oder über das Netzwerk von einem Replikat (sofern keine spezielle Hardware verwendet wird). Trotz der Schreibvorgänge auf Festplatte handelt es sich immer noch um eine speicherinterne Datenbank, weil sie die Festplatte lediglich als Protokoll für Dauerhaftigkeit nutzt und Lesevorgänge komplett aus dem Arbeitsspeicher bedient. Das Schreiben auf Festplatte hat auch Vorteile für den Betrieb: Dateien auf Festplatte lassen sich leicht von externen Dienstprogrammen sichern, inspizieren und analysieren.