Schattenspeicher

Seminar: Implementierungstechniken
für Hauptspeicherdatenbanksysteme

Daniel Kutasi

6. Oktober 2017

1 Motivation

In [2] wird die Motivation für Schattenspeicher erläutert. Datenbankmanagementsysteme (DBMS) wurden anfangs hauptsächlich zur Verarbeitung von Transaktionen genutzt, wie etwa zur Erfassung von Bestellungen oder Banküberweisungen. Dieses Vorgehen wird *Online Transaction Processing* (OLTP) genannt: Transaktionen greifen nur auf einen kleinen Teil aller Daten zu und können deshalb sehr schnell durchgeführt werden.

Mit der Zeit wuchs der Wunsch nach Geschäftsberichten (z.B. Statistiken zu Verkaufszahlen). Dafür muss auf einen Großteil der Daten zugegriffen werden, weshalb diese *Online Analytical Processing* (OLAP) genannten Transaktionen viel langsamer ablaufen als OLTP-Transaktionen. Um bei letzteren keine spürbaren Verzögerungen durch OLAP-Transaktionen zu verursachen, führt man sie klassischerweise auf einem zweckbestimmten OLTP-DBMS durch. Die Änderungen der OLTP-DB werden regelmäßig (z.B. nachts) in ein OLAP-DBMS überführt. Diese Überführung kann nur periodisch stattfinden; also basieren OLAP-Transaktionen stets auf veralteten Daten.

Häufig möchte man Geschäftsberichte jedoch in Echtzeit erstellen können, also basierend auf dem aktuellen Stand der Daten, den auch OLTP-Transaktionen sehen. Dafür müssen OLTP- und OLAP-Transaktionen auf demselben DBMS durchgeführt werden können. Das kann durch *Hybrid Transactional/Analytical Processing* (HTAP) Datenbanken geschehen. Greifen beide Arten von Transaktionen auf den gleichen Speicherbereich der Daten zu, und werden alle Transaktionen in einem einzigen Thread durchgeführt, so müssen kurze OLTP-Transaktionen in der Warteschlange oft auf die Beendigung von langen OLAP-Transaktionen warten.

Ein Ansatz, um die parallele Ausführung von OLTP- und OLAP-Transaktionen auf demselben DBMS möglichst effizient zu gestalten, ist die Erstellung von Schattenspeichern (Snapshots) der Daten. Die Snapshots werden kurz vor Ausführung von OLAP-Transaktionen erstellt und sind logisch gesehen Kopien, die von den ursprünglichen Daten isoliert sind. Es müssen dabei aber nicht notwendigerweise physische Kopien erzeugt werden. Durch Schattenspeicher können OLTP- und OLAP-Transaktionen parallel und konfliktfrei ohne Synchronisationsaufwand durchgeführt werden: OLAP-Transaktionen greifen nur lesend auf den Schattenspeicher zu, während OLTP-Transaktionen auf dem ursprünglichen Speicherbereich schreiben dürfen.

2 Explizit verwalteter Schattenspeicher

Bei dieser Art von Schattenspeichern müssen deren Adressräume und Zustände explizit durch das DBMS verwaltet werden. Im folgenden werden drei Vorgehensweisen für solche Schattenspeicher sowie deren Vor- und Nachteile aufgezeigt.

2.1 Tupel-Schattenspeicher

In [1] wird beschrieben, dass Änderungen an Daten (Updates, Inserts, Deletes) durch OLTP-Transaktionen bei Tupel-Schattenspeichern nicht auf den Tupeln selbst, sondern auf Kopien von ihnen durchgeführt werden. Dieses *Delta* wird regelmäßig in die für Lesezugriffe optimierte Haupt-Datenbank (*main*) eingebracht. Das geschieht durch einen Mischvorgang (*merge*). Ein Tupel in *main* ist somit in der Zeit zwischen seiner Änderung

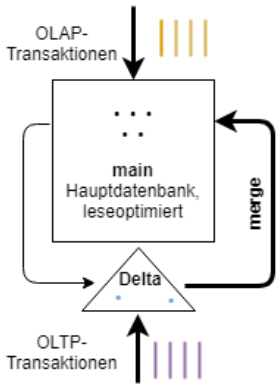
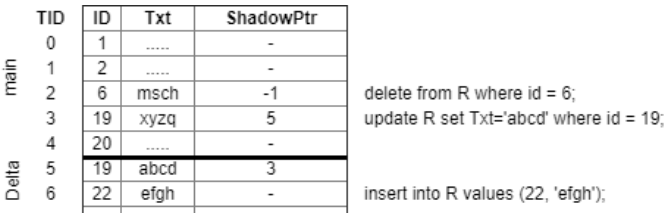


Abbildung 1

und dem nächsten *merge* eine Schattenkopie. Abb. 1 zeigt diese Architektur. Typischerweise werden Tupel-Schattenspeicher in spaltenorientierten *Column Stores* verwendet. Durch den hohen Zeit- und Platzbedarf gilt das *merge* als Flaschenhals. OLTP-Transaktionen, die auf ein Datum in *main* zugreifen wollen, müssen für den jüngsten Zustand bei jedem Zugriff zunächst im Delta nachsehen, ob das Datum geändert wurde, das Tupel in *main* also invalidiert ist. Das führt beim Scan zu Leistungseinbußen und Problemen beim Einfügen in sortierte Datenstrukturen. Deshalb wurde ein effizienter, positionsbasierter Index entwickelt, der *Positional Tree*. Er enthält in sortierter Reihenfolge die Positionen der invalidierten Einträge. Somit kann zwischen zwei nicht invalidierten Tupeln der Scan in voller Zugriffsgeschwindigkeit erfolgen. Nur bei den wenigen, invalidierten Positionen ist der Zugriff aufwendiger.

OLAP-Transaktionen hingegen werden nur auf dem Schattenspeicher *main* ausgeführt. Es wird also kein Zusatzaufwand für das Nachschlagen im *Delta* benötigt. Des Weiteren entfällt eine Synchronisation zwischen OLTP-Transaktionen (die nur auf dem *Delta* schreiben) und OLAP-Transaktionen (die nur von *main* lesen).



	TID	ID	Txt	ShadowPtr
main	0	1	.....	-
	1	2	.....	-
	2	6	msch	-1
	3	19	xyzq	5
	4	20	.....	-
Delta	5	19	abcd	3
	6	22	efgh	-

delete from R where id = 6;
update R set Txt='abcd' where id = 19;
insert into R values (22, 'efgh');

Abbildung 2

In [4] wird ein Implementierungsvorgehen für Tupel-Schattenspeicher aufgezeigt. Das *Delta* wird an den Schluss einer Relation angehängt, was in Abb. 2 dargestellt ist. Die ID eines Tupels im *Delta* bleibt bzgl. seiner Version in *main* unverändert; Tupel werden durch eine TID eindeutig identifiziert. Ob ein Tupel in *main* invalidiert ist, lässt sich am Vorhandensein eines Eintrags im Feld ShadowPtr erkennen.

Bei einem Update wird ein Tupel, z.B. das mit TID 3, zunächst ins *Delta* kopiert und im Feld ShadowPtr die TID 5 dieser Kopie vermerkt. Der ShadowPtr der Kopie zeigt auf das ursprüngliche Datum mit TID 3 in *main*. Das *Delta*-Tupel mit TID 5 erhält nun die neuen Einträge, etwa in der Spalte Txt.

Ein Insert bewirkt das Anhängen des zu erstellenden Tupels im *Delta* mit neuer ID und ShadowPtr gleich null, wie bei TID 6 dargestellt.

Ein Delete wird in *main* dadurch gekennzeichnet, dass der ShadowPtr des Tupels auf -1 gesetzt wird, wie TID 2 zeigt.

2.2 Originäres Schattenspeicher-Konzept

Das ursprüngliche Schattenspeicher-Konzept wurde erstmals in [3] beschrieben und später in [1] für Hauptspeicher-DBMS verwendet. Es arbeitet nicht, wie in Abschnitt 2.1 dargestellt, auf Tupelebene, sondern auf Seitenebene. Bevor eine zuvor noch nicht kopierte Seite geändert wird, ist eine Kopie dieser Seite zu erstellen. Das ist in Abb. 3 gezeigt. $V0$ ist die Seitentabelle, die auf den Schattenspeicher zeigt, deren Inhalt sich während der Bearbeitung nicht ändert. $V1$ übernahm vor der Bearbeitung die Seitentabelle von $V0$. Die beiden grau markierten Seiten zeigen dann auf die Adressen 12 und 10, welche zunächst die kopierten Werte enthalten, die hinter den Adressen 0 und 4 in $V0$ stehen. Diese kopierten Werte wurden sodann verändert.

Soll der Schattenspeicher aktualisiert werden, sind die alten Schattenseiten, die nicht mehr benötigt werden, in den Freiseiten-Listen $M0$ und $M1$ freizugeben. Außerdem muss die Seitentabelle mit den aktuellen Werten, $V1$, nach $V0$ kopiert werden.

Ursprünglich wurde dieses Verfahren für Hintergrundspeicher-DB entwickelt. Durch den Kopiervorgang für zu verändernde Seiten geht jedoch die physische Nachbarschaft zusammengehörender Seiten verloren. In Abb. 3 ist das anhand der veränderten Seiten mit i und j' sowie a' und b gezeigt. Wegen der teuren Suche nach diesen Seiten hat sich das Vorgehen für Hintergrundspeicher-DB nicht durchgesetzt. Für Hauptspeicher-DB macht es aber keinen Unterschied, ob die Seiten benachbart sind oder nicht. Über die Seitentabelle des Schattenspeichers kann man daher einen konsistenten Zustand der DB für die von OLTP-Transaktionen unabhängige Bearbeitung von OLAP-Transaktionen erreichen.

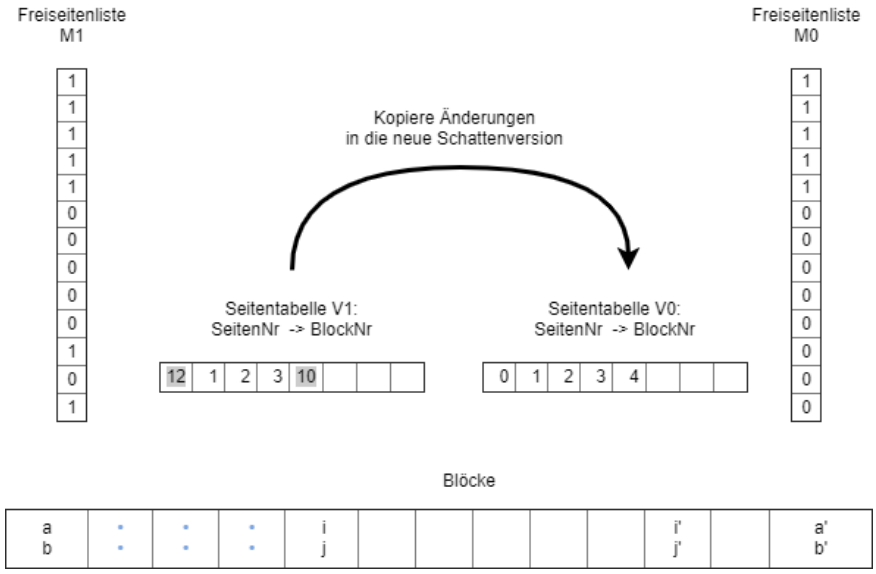


Abbildung 3

Überlasst man dem Betriebssystem die Speicherverwaltung, kann diese hocheffizient durchgeführt werden, wie wir in Abschnitt 3 sehen werden. Dabei entfällt die Verwaltung der Seitentabellen und Freispeicher-Listen, die hier noch gesondert im DBMS implementiert werden müssen.

2.3 Twin-Objekt-Verfahren

Beim Twin-Objekt-Verfahren werden laut [4] für jedes Datenobjekt zwei Kopien vorgehalten. Das ist in Abb. 4 gezeigt. Zwei Bitvektoren geben an, auf welche Kopie OLTP-Transaktionen zugreifen müssen: Lesevorgänge werden auf der durch das Bit R (*read*), Schreibvorgänge auf der durch das Bit W (*write*) angezeigten Kopie ausgeführt. OLAP-Transaktionen können die Schattenkopie lesen, die nicht verändert wurde - sie wird durch die Negation von W, also $\neg W$, angezeigt. Gelöschte Tupel werden durch einen Eintrag im Bitvektor D (*deleted*) angegeben.

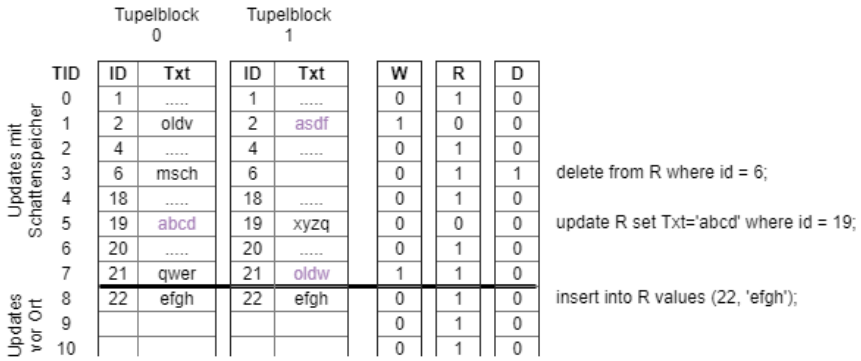


Abbildung 4

Der Vorteil des Twin-Objekt-Verfahrens ist, dass das Umschalten von einer Schattenkopie auf die nächste im Vergleich zu Tupel-Schattenspeichern sehr schnell geht, weil ein aufwendiges *merge* entfällt. Dafür wird allerdings der doppelte Speicherplatz benötigt - auch beim Großteil der Daten, der sich nicht mehr ändern wird.

Abb. 4 zeigt anhand von TID 3, dass Löschungen durchgeführt werden, indem sie als gelöscht markiert werden: der Bitvektor D des Objekts wird auf 1 gesetzt.

Einfügungen wie TID 8 werden an die Speicherblöcke angehängt: Initial ist Block 0 die veränderte Kopie (W-Eintrag). Block 1 ist die ursprüngliche Version und gleichzeitig Schattenkopie (R-Eintrag = $\neg W$-Eintrag).

Bei einer Aktualisierung wird der durch R angegebene Zustand gelesen und der aktualisierte Wert in den durch W angegebenen Block geschrieben. TID 5 veranschaulicht, dass nach der erstmaligen Aktualisierung eines Objekts zusätzlich der R-Eintrag auf den Wert des W-Eintrags gesetzt werden muss, denn nachfolgende OLTP-Transaktionen müssen den neuen Zustand des Objekts lesen.

3 Betriebssystem eigener Schattenspeicher

Bei dieser Art von Schattenspeichern übernimmt das Betriebssystem die effiziente Verwaltung der Adressräume und Zustände, so dass sie nicht explizit durch Implementierungen im DBMS geschehen muss. Zunächst wird das Vorgehen für alleinige OLTP- und OLAP-Threads beschrieben und dieses anschließend auf mehrere Threads erweitert.

3.1 Alleiniger OLTP-Thread mit einziger OLAP-Session

In [2] wird diese einfachste Situation beschrieben. Weil alle Daten im Hauptspeicher verfügbar sind, muss ein OLTP-Thread zu keinem Zeitpunkt angehalten werden, um auf Daten aus dem Hintergrundspeicher zu warten, die noch nicht in den Hauptspeicher oder einen Cache geladen wurden. Daher können alle OLTP-Transaktionen nacheinander in einem einzigen Thread ausgeführt werden. So spart man sich das teure Sperren und Freigeben von Daten, da die jeweils einzige, gerade ausgeführte Transaktion die Kontrolle über die gesamte DB hat. Die OLTP-Transaktionen müssen kurze Antwortzeiten sicherstellen, um lange Wartezeiten für nachfolgende Transaktionen in der Warteschlange zu vermeiden.

Die Dauer einer typischen OLTP-Transaktion bewegt sich, abhängig von DBMS und Komplexität der Transaktion, in den Größenordnungen von 10 bis 100 μs. OLAP-Transaktionen hingegen laufen deutlich länger. Würde man diese also in die selbe Warteschlange einreihen wie die OLTP-Transaktionen, müssten letztere sehr lange warten, bis eine solche OLAP-Transaktion beendet werden würde. Selbst wenn eine OLAP-Transaktion in nur 30 ms durchläuft, ist der Prozess für eine Dauer blockiert, in der möglicherweise tausende von OLTP-Transaktionen hätten ausgeführt werden können.

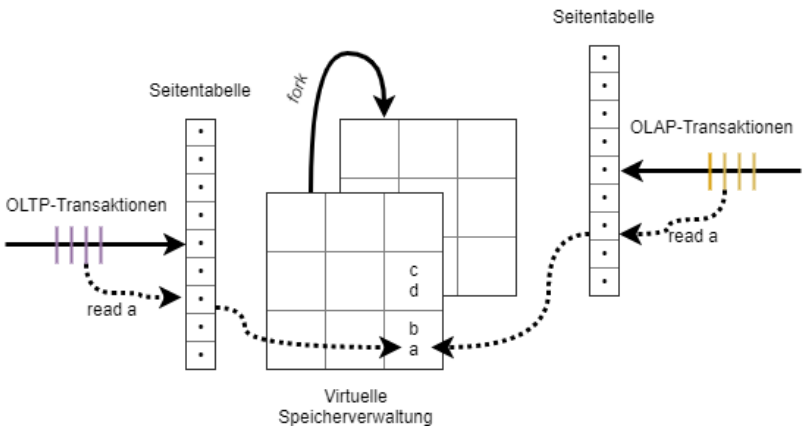


Abbildung 5

Daher kann, wie in [1] beschrieben, mithilfe des Unix-Systembefehls *fork* aus dem OLTP-Prozess ein neuer Kindprozess für die Ausführung der OLAP-Transaktionen erzeugt werden. Dessen virtueller Adressraum entspricht dem gleichen Speicherabbild wie der seines Vaterprozesses, weshalb der Adressraum auf einen Schattenspeicher zeigt. Der *fork*-Befehl kann ohne weitere Hilfsmittel nur zwischen zwei OLTP-Transaktionen ausgeführt werden, also wenn aktuell keine Transaktion aktiv ist. Dann ist der Zustand des Schattenspeichers automatisch transaktionskonsistent. Ein *fork* während aktiver Transaktionen führt nur zu einem aktionskonsistenten Schattenspeicher. In Abschnitt 5 wird ein kurzer Ausblick gegeben, wie sich dieser in einen

transaktionskonsistenten überführen lässt. Abb. 5 zeigt die Warteschlangen der OLTP- und OLAP-Prozesse sowie die beiden zugehörigen, virtuellen Adressräume.

Moderne Betriebssysteme kopieren nicht den Adressraum des Vaterprozesses, wenn *fork* ausgeführt wird - sondern nur die Seitentabelle des virtuellen Speichers. Die Einträge beider virtueller Seitentabellen verweisen auf die gleichen Adressen des physischen Speichers: die eigentlichen Datenseiten werden gemeinsam benutzt. Abb. 5 zeigt dies für das Datum a: die beiden Leseoperationen greifen auf die gleiche, physische Adresse zu.

Erst bei einer Änderung wird die entsprechende Seite physisch kopiert und der Verweis in der Tabelle des OLAP-Prozesses auf die Kopie gesetzt. Dieses Verhalten nennt man *copy on update* oder *copy on write*. Abb. 6 zeigt, dass eine OLTP-Transaktion ein Datum a zu a' geändert hat. Zuvor wurde a kopiert. Die OLAP-Transaktionen arbeiten nun mit der Seite, auf der sich der alte Zustand a befindet, während die OLTP-Transaktionen auf den aktuellen Zustand a' zugreifen.

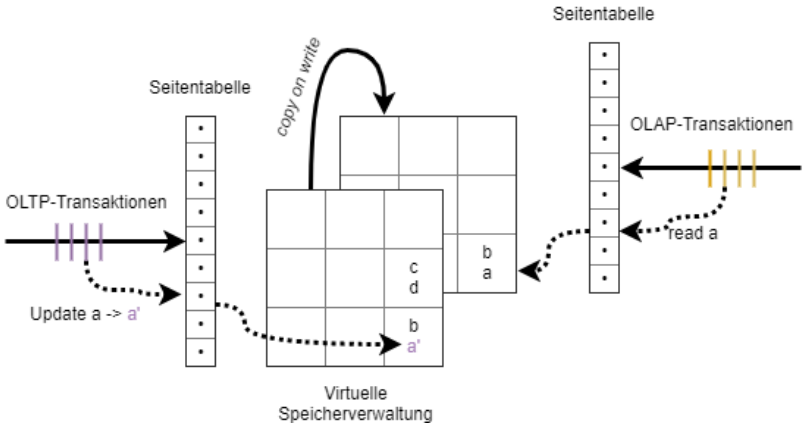


Abbildung 6

Ein solcher Kopiervorgang dauert, abhängig von Hardware und Betriebssystem, nur einige μs. Die hohe Geschwindigkeit resultiert daraus, dass die Erkennung der Notwendigkeit und die Durchführung des Kopiervorgangs durch die *Memory Management Unit* (MMU) hardwareunterstützt erfolgt. Das ist deutlich schneller, als es softwaregesteuert möglich wäre. Zudem nutzt die MMU einen *Translation Lookaside Buffer* (TLB), in dem sich die zuletzt aufgelösten Seitennummern befinden. Das beschleunigt den Vorgang, weil sich viele Zugriffe im DBMS auf dieselbe Seite beziehen, wie etwa bei einem Scan.

Das *copy on update* Verfahren kommt dem betrieblichen Einsatz von DBMS zugute, da der Großteil der Daten ohnehin nicht mehr geändert wird: sie stellen in der Vergangenheit durchgeführte Transaktionen dar. Ein Schattenspeicher existiert, bis die OLAP-Session beendet wird - üblicherweise nach einigen Sekunden oder Minuten. Danach wird er gelöscht, indem der abgespaltene Kindprozess beendet wird.

3.2 Mehrere OLAP-Sessions

In [2] wird die Erweiterung des Konzepts auf mehrere OLAP-Sessions aufgezeigt. Da OLAP-Transaktionen auf die Daten nur lesend zugreifen, können sie parallel in mehreren Threads, also auch auf mehreren Rechnerkernen ausgeführt werden, die auf den gleichen Adressraum zugreifen. Ohne Schreibzugriffe entstehen keine Konflikte, die durch Sperren vermieden werden müssten.

Das beschriebene Vorgehen der Nutzung mehrerer Threads lässt sich erweitern, indem für neue OLAP-Transaktionen regelmäßig oder auf Abruf mittels *fork* weitere Prozesse mit Schattenspeichern erzeugt werden. Das hat den Vorteil, dass jüngere OLAP-Transaktionen auf aktuellere Snapshots zugreifen können als bereits laufende, ältere. Theoretisch lassen sich so beliebig viele Schattenspeicher generieren. In Abb. 7 ist im vorderen Rechteck der aktuelle Datenbankzustand des alleinigen OLTP-Prozesses gezeigt, der eine Seite mit Datum a''' enthält. Im Hintergrund sieht man die erzeugten Schattenspeicher der OLAP-Sessions mit den Daten a'', a' und a, wobei die älteren Zustände weiter hinten sind.

Wie auch bei alleinigen OLAP-Sessions wird ein OLAP-Prozess beendet und damit sein Snapshot gelöscht, sobald seine letzte OLAP-Transaktion beendet wurde. Es ist nicht notwendig, Schattenspeicher in der gleichen

Reihenfolge zu löschen, wie sie erstellt wurden: Der mittlere Snapshot enthält das Datum c, auf das auch der älteste Snapshot zugreifen kann. Wird der mittlere Snapshot vor dem ältesten gelöscht, kann der älteste immer noch auf c zugreifen, weil das Betriebssystem durch einen Referenzzähler automatisch erkennt, dass die Seite mit Datum c auch mit dem ältesten Schattenspeicher geteilt wurde. Die Seite mit dem Datum a' hingegen "überlebt" das Beenden des mittleren Schattenspeichers nicht, da sie kein anderer Prozess mehr benötigt: Der älteste Snapshot hat selbst eine Kopie mit einem älteren Zustand a.

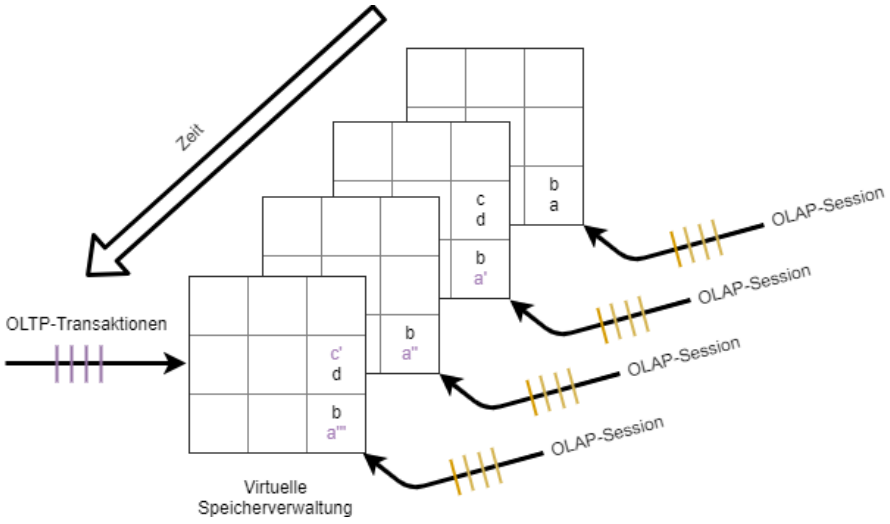


Abbildung 7

3.3 Mehrere OLTP-Threads

In [2] wird auch die Erweiterung auf mehrere OLTP-Threads beschrieben. OLTP-Transaktionen, die nur lesend auf Daten zugreifen, können ohne Mehraufwand auf mehrere Threads verteilt und parallel ausgeführt werden. Sobald eine Transaktion an der Reihe ist, die auch Daten schreibt, werden Konflikte mit ihr vermieden, indem vor ihrer Ausführung wieder alle Transaktionen zur sequentiellen Ausführung in einem einzigen Thread untergebracht werden - solange, bis keine schreibende Transaktion mehr am Anfang der Warteschlange steht. Da in der Praxis viel mehr Transaktionen zu erwarten sind, die nur lesen, als solche, die auch schreiben, kann so ein bestimmtes Maß an Parallelität erreicht werden. Dieses kann sogar erhöht werden, indem die Warteschlange umsortiert wird.

In Anwendungen, die mandantenfähig sind (*Multitenancy-Systeme*), können sogar schreibende Transaktionen in parallelen Threads ausgeführt werden - nämlich dann, wenn diese Transaktionen Daten nur im eigenen Speicherbereich (*Partition*) ihres Mandanten schreiben, und auf mit anderen Mandanten geteilte Daten nur lesend zugreifen. In diesem Fall kann eine Schreibtransaktion pro Partition parallel ausgeführt werden.

OLAP-Schattenspeicher können wie bisher beschrieben erzeugt werden. Es müssen jedoch alle Threads angehalten werden und keine Transaktion darf aktiv sein, bevor das in einer transaktionskonsistenten Art getan werden kann. Wie zuvor sei auf Abschnitt 5 für einen Ausblick verwiesen, wie auch während laufender Transaktionen Snapshots transaktionskonsistent generiert werden können.

3.4 Implementierung

Die beispielhafte Implementierung realisiert in der Programmiersprache C++ die im Abschnitt 3.2 beschriebene Situation mit einem OLTP-Thread und mehreren OLAP-Threads. Die einzige Relation der Hauptspeicher-Datenbank simuliert eine Menge von Bestellungen. Eine Bestellung hat eine ID, die ID des Kunden sowie einen Gesamtbetrag. Die Relation heißt *orders* und ist als *Row Store*, nämlich als *vector* von *order* umgesetzt. *order* ist ein *struct* mit drei Feldern: *orderId* und *customerId* als vorzeichenlose Integer, sowie *totalAmount* als Double.

Abb. 8 zeigt schematisch den Fluss der Anwendung. Zwei Millionen mal wird im Vaterprozess eine OLTP-Transaktion ausgeführt. Innerhalb dieser Schleife wird alle zehntausend mal vom Vaterprozess aus mittels

fork ein neuer Kindprozess erzeugt, in dem eine OLAP-Transaktion ausgeführt wird. Nach deren Ausführung wird der Kindprozess beendet und der Schattenspeicher damit verworfen.

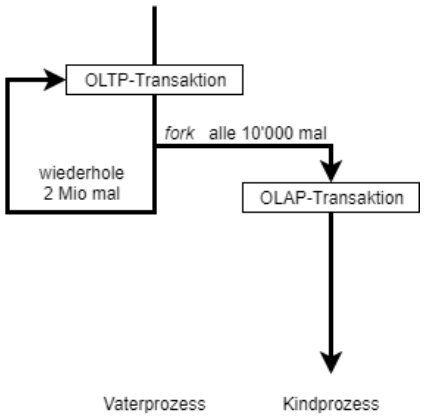


Abbildung 8

Eine OLTP-Transaktion fügt der Relation *orders* eine Bestellung hinzu, die ein *struct* mit einer fortlaufenden *orderId*, einer fortlaufenden *customerId* sowie einem zufälligen *totalAmount* zwischen EUR 1,00 und EUR 999,99 ist.

Eine OLAP-Transaktion berechnet auf dem erzeugten Schattenspeicher den durchschnittlichen Gesamtbetrag sowie die Anzahl aller bisherigen Bestellungen.

4 Zusammenfassung

In dieser Arbeit wurden zwei verschiedene Arten von Schattenspeichern vorgestellt, mit denen sich OLTP- und OLAP-Transaktionen parallel und effizient auf derselben DB ausführen lassen.

Es wurden drei Verfahren für durch DBMS verwaltete Schattenspeicher vorgestellt. Tupel-Schattenspeicher, bei denen nur zu ändernde Daten kopiert werden, kommen auch wegen des aufwendigen *merge*-Vorgangs am Besten in leseoptimierten DBMS zum Einsatz, wenn also einmal erstellte Daten selten geändert werden und Lesevorgänge häufig sind - oft etwa in *Column Stores*. Das ursprüngliche Konzept für Schattenspeicher ähnelt Tupel-Snapshots, sieht aber Kopien auf Seitenebene vor. Für änderungsintensive Daten eignet sich das Twin-Objekt-Verfahren gut, da zwar alle Daten doppelt gehalten werden, das Umschalten auf eine neue Schattenkopie aber sehr effizient ist.

Als Erweiterung des ursprünglichen Schattenspeicher-Konzepts wurden betriebssystem-verwaltete Snapshots vorgestellt: Durch den *fork*-Befehl übernimmt das Betriebssystem die effiziente Erstellung und Adressverwaltung der Schattenspeicher, die physische Kopien nur für geänderte Daten vorhalten. Im Gegensatz zum ursprünglichen Konzept ermöglicht das eine effiziente, parallele Ausführung von OLAP- und OLTP-Transaktionen. Da sich theoretisch beliebig viele solcher Schattenspeicher generieren lassen und OLAP-Transaktionen nur lesen, ist eine Erweiterung auf mehrere, parallele OLAP-Threads ohne Mehraufwand möglich. Innerhalb derselben Datenmenge dürfen dagegen ohne weitere Synchronisation nur lesende OLTP-Transaktionen auch in parallelen Threads ablaufen. In mandantenfähigen Systemen darf sogar pro Mandant ein schreibender OLTP-Thread parallel laufen, wenn dieser auf einen Speicherbereich zugreift, der exklusiv einem Mandanten zugewiesen wurde.

Für die Beispielimplementierung wurde die Situation mit einem OLTP-Thread und mehreren OLAP-Threads gewählt. Die Implementierung wurde dieser Arbeit beigefügt und enthält neben dem Quellcode im Ordner *src* eine in Terminals von UNIX-Betriebssystemen ausführbare Datei im Ordner *Release*.

5 Ausblick

Um Schattenspeicher während aktiver OLTP-Transaktionen zu erzeugen, müssen deren bisherige Aktionen rückgängig gemacht werden, damit die OLAP-Transaktionen einen transaktionskonsistenten Zustand lesen. Das kann z.B. mithilfe von *Undo Buffern* geschehen, die in ein *Multi-Version Concurrency Control* eingebettet sind [5].

Des Weiteren findet man in der Literatur auch andere Verfahren von explizit verwalteten Snapshots [4].

Bemerkungen

Ist eine Quelle aus der Literatur in einem Abschnitt genannt, so ist damit ein indirektes Zitat gekennzeichnet, das sich bis zur Nennung der nächsten Quelle im selben Abschnitt erstreckt - oder bis zum Ende des Abschnitts, wenn in diesem nur eine Quelle genannt ist.

Alle in dieser Arbeit verwendeten Grafiken wurden vom Autor selbst erstellt. Das geschah, mit Ausnahme von Abb. 8, auf Grundlage von ähnlichen Grafiken, die in der im jeweiligen Abschnitt zitierten Arbeit vorhanden sind.

Literatur

[1] Funke F, Kemper A, Mühe H, Neumann T (2011) HyPer: Die effiziente Reinkarnation des Schattenspeichers in einem Hauptspeicher-DBMS. In: Datenbank-Spektrum, 11, S 111-122

[2] Kemper A, Neumann T (2011) HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In: IEEE 27th International Conference on Data Engineering, S 195-206

[3] Lorie R (1977) Physical Integrity in a Large Segmented Database. In: ACM Trans. Database Syst., 2, S 91-104

[4] Mühe H, Kemper A, Neumann T (2011) How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled? In: Proceedings of the Seventh International Workshop on Data Management on New Hardware, 17-26

[5] Neumann T, Mühlbauer T, Kemper A (2015) Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In: ACM SIGMOD International Conference on Management of Data