Inhaltsverzeichnis

Beschleunigung des Datenbanksystems mit SSDs 2
Philipp Thau
1 Einleitung 2
2 SSDs in Datenbanken 3
2.1 Speicher auf gleicher Hierachieebene wie die Festplatte 3
2.2 Write-Back-Cache 4
2.3 SSD Buffer Pool Extension 4
3 SSD-Cache 5
3.1 Systemaufbau 6
3.2 Metadaten 7
3.3 Seitenersetzung 8
3.4 Ergebnisse 9
3.4.1 Seitenersetzungsstrategie 9
3.4.2 Aufbau 10
3.5 Weitere Ansätze und Verbesserungen 11
3.5.1 Eine kalte Seite in konstanter Zeit bestimmen 11
3.5.2 Die Schreibzugriffe auf die Festplatte puffern 11
4 Recovery 12
4.1 Motivation 12
4.2 Lösungsansätze für die Metadaten und Recovery 14
4.2.1 Update-Write-Update 14
4.2.2 Write-Update bei Facebooks Flashcache 18
4.2.3 Lazy-Update Following an Update-Write nach Yang und Yang 20
5 Fazit 23

Beschleunigung des Datenbanksystems mit SSDs

Philipp Thau

Technische Universität Kaiserslautern

Zusammenfassung. In dieser Arbeit wird das Konzept von SSD-Caches in Verbindung mit dem Aspekt des Recoverys vorgestellt. Hierfür wird zunächst ein Ansatz zum SSD-Caching von Canim et al. vorgestellt ([2]), welcher im Anschluss daran um mehrere Ansätze erweitert wird, die den Cache über einen Neustart hinaus und damit für den Recoveryprozess zur Verfügung stellen. Hierfür muss zunächst erklärt werden, was das Problem für den SSD-Cache bei einem Neustart des Systems ist ([3]). Vorgestellte Konzepte für das bereitstellen des Caches sind: Update-Write-Update [1], Write-Update [4][5], Lazy-Update Following an Update-Write [6].

1 Einleitung

Die Verwaltung von Daten in Datenbanken erfordert Datenträger. Dieser Satz mag sich banal anhören, doch haben die Datenträger einer Datenbank einen großen Einfluss darauf, wie Datenbanksysteme funktionieren, mit welcher Geschwindigkeit sie arbeiten, welche monetären Ausgaben die Datenbankhardware erfordert und wie zuverlässig sie arbeitet. Die Datenträger spielen also eine entscheidende Rolle im Aufbau und Design eines Datenbanksystems und müssen deswegen auch die entsprechende, ihnen zustehende, Aufmerksamkeit bekommen.

In dieser Arbeit werden konkret so genannte Solid-State-Drives (SSDs) betrachtet. Neben Festplatten können SSDs als die zentrale Größe im Bereich der Datenträger betrachtet werden: Sie besitzen eine hohe Datendichte, speichern die Daten permanent, sind nach einigen Jahren der Entwicklung inzwischen als robust zu betrachten (auch mechanisch, da sie im Gegensatz zu Festplatten keine beweglichen Bauteile besitzen) und verbrauchen relativ zu Festplatten betrachtet sehr wenig Strom. Außerdem, und dies ist für die vorliegende Arbeit elementar, besitzen sie wesentlich kürzere Zugriffszeiten: Aktuelle Consumer-SSDs erreichen bis zu 100.000 IOPS (Input/Output Operations Per Second). Festplatten hingegen besitzen eine natürliche Grenze bei den Zugriffszeiten: Auf die Daten,

welche sich auf einer rotierenden Scheibe befinden, kann nur zugegriffen werden, wenn diese am Schreib-/Lesekopf vorbeikommen. Da die Datenschreiben in einer Festplatte natürlich nicht unbegrenzt schnell drehen können (in einer aktuellen Desktopfestplatte ca. 7200 Umdrehungen pro Minute), lässt sich die Technik hier nicht weiter ausreizen, weswegen selbst Topmodelle nur ca. 200 IOPS erreichen. Umgerechnet auf die Einheit $\frac{\text{IOPS}}{\text{Euro}}$ leisten SSDs also ca. 1000 $\frac{\text{IOPS}}{\text{Euro}}$, Festplatten hingegen nur ca. 2 $\frac{\text{IOPS}}{\text{Euro}}$.

Aber natürlich besitzen SSDs neben ihren Vorteilen auch einen Nachteil: Sie sind wesentlich teurer wenn man als Vergleichseinheit $\frac{\text{Euro}}{\text{Gigabyte}}$ heranzieht. So kostet eine Festplatte[1] ca. 0,03 $\frac{\text{Euro}}{\text{Gigabyte}}$, eine SSD[2] hingegen mindestens 0,35 $\frac{\text{Euro}}{\text{Gigabyte}}$. Es ist wichtig dies im Auge zu behalten, denn der Gedanke, einfach die gesamte Datenbank auf SSDs zu speichern, zieht sonst schnell ökonomische Probleme nach sich. Es gilt, die optimale Balance aus SSDs und Festplatten zu finden und beide Datenträgertypen so zu verwenden, dass sie sich ergänzen.

Neben dem Kosten- und Leistungsfaktor von SSDs und Festplatten muss jedoch auch die Problematik eines Absturzes bedacht werden: Sind die gespeicherten Daten danach noch verwertbar? Und welche Schritte müssen unternommen werden um die Leistung auch nach dem Absturz im Rahmen des Recoveryprozesses auf einem gleichbleibend hohen Niveau zu halten? Diese Fragen werden in der vorliegenden Arbeit beantwortet.

2 SSDs in Datenbanken

Neben der bereits genannten offensichtlichen Möglichkeit, die Datenbank komplett auf SSDs zu speichern, gibt es noch weitere Möglichkeiten, SSDs in Verbindung mit klassischen Festplatten in Datenbanksystemen einzusetzen. Ziel muss dabei sein, die Überlegenheit von SSDs in Bezug auf IOPS gegenüber Festplatten in Verbindung mit der Eigenschaft der Persistenz optimal auszunutzen.

Dazu gibt es drei Möglichkeiten die im Folgenden beschrieben werden.

2.1 Speicher auf gleicher Hierachieebene wie die Festplatte

Hierbei wird die SSD als weiterer Datenträger eingebunden, welcher auf einer Stufe mit der Festplatte steht ([2]). Auf diesem Datenträger werden dann hochfrequentierte Daten gespeichert, wie zum Beispiel Indexstrukturen oder die Faktentabelle eines Data-Warehouse. Natürlich ist es auch möglich, zur Laufzeit zu

[1] Modell „WD30EZRX“, 3TB, 92,90 Euro
[2] Modell „MX100“. 256GB, 89,90 Euro

messen, welche Tabellen das Datenbanksystem dadurch ausbremsen, dass sie auf der Festplatte abgelegt sind, und diese dann gezielt auf die SSD zu verschieben. Der gemeinsame Nachteil dieser Ansätze ist, dass ein Administrator gezielt in die Datenbank eingreifen muss, um Tabellen zu verschieben. Außerdem müssen die nötigen Aktivitäten wiederholt werden, sobald sich die Anforderungen oder Zugriffsmuster der Datenbank verändern.

Bei diesem Ansatz ist Recovery nach einem Absturz oder Neustart kein Problem, da die SSD als normaler Datenträger auftritt.

2.2 Write-Back-Cache

Eine weitere Möglichkeit, SSDs im Datenbanksystem zu nutzen, besteht darin, Schreibzugriffe auf die Festplatte unter Zuhilfename von SSDs in sequentielle Schreibzugriffe umzuwandeln ([1]), sodass diese von der Festplatte schneller ausgeführt werden können, und so auszunutzen, dass SSDs ihre große Stärke bei Random-IO haben und Festplatten bei sequentiellem IO. Hierfür werden die Schreibzugriffe zunächst an eine SSD weitergereicht, welche eine vorher bestimmte Anzahl von ihnen zwischenspeichert. Dann wird zu einem geeigneten Zeitpunkt, möglicherweise während einer Phase geringer Last und in geeigneter Reihenfolge, die Menge der Speicherzugriffe an die Festplatte weitergegeben.

Möchte man diesen Ansatz in einer Datenbank verwenden, ist es wichtig ebenfalls Recovery zu betrachten: Es kann zu der Situation kommen, dass das System abstürzt, während noch nicht alle Daten auf die Festplatte geschrieben wurden. Es ist deshalb erforderlich, den Zustand der Festplatte nicht bedingungslos als konsistent zu betrachten; dies ist nur in Verbindung mit der SSD und den beschreibenden Informationen über den Inhalt der SSD möglich.

2.3 SSD Buffer Pool Extension

Die dritte Möglichkeit ist eine SSD Buffer Pool Extension (kurz: SSD-Cache) ([2,1]): Die Gemeinsamkeit der verschiedenen dazu existierenden Ansätze ist, dass die SSD als weitere Ebene zwischen Arbeitsspeicher und Festplatte eingeführt und dann vollautomatisch mit Daten von der Festplatte gefüllt wird. Das System muss selbstständig sich verändernde Zugriffsmuster erkennen und sich daran anpassen. Ziel ist es, mit einem kleinen SSD-Cache durch dynamische Anpassungen eine möglichst große Wirkung in Bezug auf die Beschleunigung des Systems zu erzielen. Dieser Ansatz wird in dieser Arbeit behandelt.

Der kritische Punkt in Bezug auf Recovery nach einem Absturz oder Neustart sind hier die Metadaten des Caches, also der Inhalt und aktuelle Statistiken zur Nutzung. Diese müssen so gespeichert werden, dass sie nach einem Neustart mit keinem oder wenig Aufwand wiederhergestellt werden können, sodass der Cache direkt „warm“ ist. Bei einer geeigneten Struktur und nach sorgfälliger Abwägung ist es natürlich auch möglich, die Metadaten nicht persistent zu speichern und nach dem Neustart mit einem „kalten“ Cache zu beginnen. Auch dann wird der Cache den Recoveryprozess beschleunigen, da er im Laufe des Recoveryprozesses ebenfalls dynamisch gefüllt und genutzt wird. Dies kann zum Beispiel Sinn machen, wenn man die Notwendigkeit der Unterstützung des Recoveryprozesses als weniger relevant gegenüber der veringerten Komplexität durch nicht-persistente Metadaten erachtet.

3 SSD-Cache

In diesem Kapitel soll das Konzept „SSD-Cache“ näher betrachtet werden. Zunächst wird das grundlegende Konzept anhand eines Beispielansatzes erläutert, um dieses dann später um den Aspekt des Recovery erweitern zu können.

Betrachtet wird das Konzept des SSD-Caches von Canim et al. ([2]). In diesem Aufbau wird die SSD als Lesepuffer zwischen der Festplatte und dem Arbeitsspeicher eingesetzt. Die Festplatte selbst soll nicht ersetzt werden, da sie nach aktuellem Stand noch immer erheblich günstigeren Speicherplatz bereitstellt. Es soll vielmehr der signifikante Nachteil von Festplatten in Bezug auf Random-IO mithilfe der Stärken von SSDs bei Datenzugriffszeiten ausgeglichen werden, um die Gesamtperformance mithilfe der Synergieeffekte erheblich zu steigern. Hierbei geht der Ansatz von Canim et al. davon aus, dass die Performance der SSD bei Datenzugriffen sowohl für Random-IO als auch für sequentielles IO besser ist. Ist diese Voraussetzung nicht erfüllt, so ist der Aufbau suboptimal, da der SSD-Cache auch sequentielle Zugriffe bedient — vorausgesetzt, die zu lesenden Seiten sind in ihm gespeichert. Als Ersetzungsstrategie wird dabei ein temperaturbasierter Algorithmus, welcher zwischen so genannten „warmen“ und „kalten“ Regionen unterscheidet, verwendet. Der Cache ist als Write-Through-Cache konzipiert, was bedeutet, dass Änderungen direkt auf die darunter liegende Ebene — in diesem Fall die Festplatte — geschrieben werden. Sinnvoll ist dieser Cache-Aufbau wenn der Arbeitsspeicher nicht den gesamten Datenbestand fassen kann und oft Daten von der Festplatte benötigt welche nicht im Arbeitsspeicher enthalten sind. Diese werden dann in der Zwischenschicht „SSD-Cache“ abgelegt.

3.1 Systemaufbau

Betrachten wir zunächst den Systemaufbau in Abbildung 1: Die Grafik stellt einen Überblick über das System dar und Pfeile repräsentieren den Datenfluss.

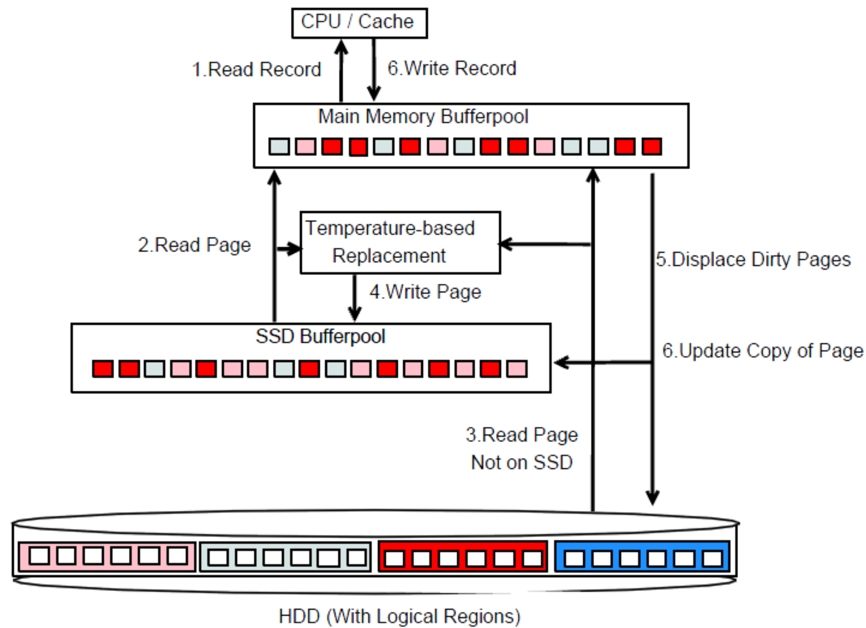


Abb. 1. Systemaufbau des Ansatz von Canim et al. (Quelle: [2])

Eine Leseoperation läuft dabei folgendermaßen ab (die Zahlen verweisen auf die Pfeile der Grafik):

1. Prüfe, ob die zu lesenden Daten im Arbeitsspeicher vorhanden sind. Kann die Leseoperation mithilfe des Arbeitsspeichers bedient werden, muss außerdem nichts mehr getan werden. Kann sie nicht bedient werden, wird als nächstes der SSD-Cache geprüft.
2. Aktualisiere zunächst die Temperatur der angefragten Region und prüfe dann, ob die zu lesenden Daten im SSD-Cache vorhanden sind. Ist dies der Fall, so werden die Daten in den Arbeitsspeicher kopiert und die Operation von dort bedient. Sind die Daten nicht im SSD-Cache, so wird als nächstes die Festplatte geprüft.

3. Alle Daten sind immer auf der Festplatte vorhanden. Deswegen ist eine Leseoperation hier immer erfolgreich und die zu lesenden Daten werden in den Arbeitsspeicher kopiert, um die Leseoperation von dort zu bedienen. Außerdem werden die Daten an die Ersetzungsfunktion des SSD-Cache weitergegeben.
4. Sollte die Seite laut der temperaturbasierten Seitenersetzungsstrategie wärmer sein als die kälteste Seite im SSD-Cache, so verdrängt die gelesene Seite ebendiese kälteste Seite im SSD-Cache.

Wird eine Dirty Page (eine Seite mit Änderungen) aus dem Arbeitsspeicher verdrängt, so läuft dies folgendermaßen ab:

5. Verdrängung der Dirty Page aus dem Arbeitsspeicher
6. Aktualisierung der Dirty Page auf der Festplatte und -falls vorhanden- im SSD-Cache. Der Zustand der Seiten im SSD-Cache ist also immer identisch zu dem der Seiten auf der Festplatte.

3.2 Metadaten

Die Metadaten werden bei diesem Ansatz nicht-persistent im Arbeitsspeicher verwaltet. Dies führt bei einem Neustart/Absturz dazu, dass das Datenbanksystem im Laufe des Recoveryprozesses nicht auf einen bereits gefüllten SSD-Cache zurückgreifen kann, sondern dieser erst erneut aufgebaut werden muss. Nichtsdestotrotz wird der Recoveryprozess beschleunigt, da bereits hier der Cache dynamisch gefüllt und genutzt wird. Natürlich wäre es sehr viel nützlicher, bereits auf einen gefüllten Cache von vor dem Neustart/Ausfall zurückgreifen zu können. Dies wird in den Kapiteln 4.1 auf Seite 12 und 4.2 auf Seite 14 betrachtet.

Die Datenverwaltung selbst erfolgt mithilfe einer Hashtabelle, welche zu jeder Position der Seite auf der Festplatte die Position der Seite auf der SSD speichert, sofern dort vorhanden. Um zu prüfen ob eine Seite im SSD-Cache vorhanden ist, muss nur die Hashtabelle geprüft werden.

Die Temperaturstatistiken der Seitenersetzungsstrategie werden ebenfalls in einer Hashtabelle gespeichert, wobei zu jeder Region (eine Region entspricht einer vorher definierten Menge von Seiten, in der Arbeit von Canim et al. 32 Seiten), auf die ein Zugriff erfolgt, ist die Temperatur gespeichert wird. Regionen werden verwendet, da die Erfassung von Statistiken zu jeder Seite zu viel Speicherplatz benötigt und erst nach längerer Zeit aussagekräftig ist, da weniger Daten pro Seite anfallen als pro Region. Außerdem ist die Erfassung von Regionen vorausschauender, da Zugriffe auf Nachbarseiten die Wahrscheinlichkeit erhöhen, dass eine Seite im Cache verbleibt.

Neben der Hashtabelle wird außerdem, um in kurzer Zeit die kälteste Seite zu identifizieren, ein Heap für alle gecachten Seiten geführt. Da das Aktualisieren der Temperaturinformationen in diesem die Laufzeit stark verschlechtern würde, wird der Heap *lazy* aktualisiert: Sobald eine kälteste Seite identifiziert werden muss (also beim Ersetzen einer Seite oder laden einer neuen Seite in den Puffer), wird die Temperaturinformation der Wurzel aktualisiert und diese erneut eingefügt. Dies wird eine vorher bestimmte Anzahl an Ausführungen wiederholt (in der Arbeit von Canim et al. fünf mal), danach wird die Wurzel als kälteste Wurzel ausgewählt. Die Laufzeit ist durch die konstante Anzahl an Einfügeoperationen immernoch logarithmisch ($O(5 * \log(\mathrm{n})) = O(\log(\mathrm{n}))$)

3.3 Seitenersetzung

Für die Seitenersetzung wird ein temperaturbasierter Algorithmus (Temperature-Aware Caching / TAC) verwendet. Dieser arbeitet wie folgt:

- So lange der SSD-Cache noch nicht voll ist, wird jede Seite akzeptiert und abgespeichert.
- Ist der SSD-Cache voll, so wird die Seite akzeptiert und abgespeichert, wenn sie wärmer ist als die kälteste Seite im SSD-Cache. Diese kältere Seite wird dann verdrängt. Das Ablehnen von zu kalten Seiten führt dabei dazu, dass der gesamte Cache besser ausgenutzt werden kann, da bei kalten Seiten das Verhältnis von verwendetem Speicher zu Anzahl der Anfragen ungünstiger ist als bei warmen Seiten. Dieses bessere Ausnutzen des Caches wäre zum Beispiel bei LRU (Least recently used) nicht gegeben, da hier einfach die zuletzt genutzte Seite verdrängt wird, unabhängig davon welche Seite vermutlich früher wieder gelesen wird. Eine Seite, die nur einmal gelesen wurde, verdrängt hier womöglich eine Seite die bald wieder gebraucht werden würde, und diese nur einmal gelesene Seite kann erst wieder aus dem Cache entfernt werden, wenn dieser komplett mit neuen Daten aufgefüllt wurde.

Die Temperaturen werden hierbei online bei jedem Lesen von Daten aus dem SSD-Cache oder der Festplatte in den Arbeitsspeicher aktualisiert, unabhängig davon, ob sie auch in den Cache aufgenommen wurden. Dies führt dazu, dass sich das System selbstständig an sich verändernde Zugriffsmuster anpasst.

Um die Zugriffsmuster genauer definieren zu können und neben den Seiten, auf die zugegriffen wird, auch die Art der Zugriffe (Random-IO oder sequentielles IO) zu erkennen, wird eine so genannte *windowing technique* angewandt. Für eine bestimmte Anzahl an Anfragen wird geprüft, ob sich in dieser Menge von Anfragen

mehr als eine weitere bestimmte Anzahl auf eine einzelne Region beziehen, um zu erkennen ob die Zugriffe sequentiell sind. Canim et al. verwenden hierfür eine Fenstergröße von 20 und eine Zahl von 2 aufeinander folgenden Zugriffen auf eine Region. Wird so eine Region erkannt, auf die tendenziell eher Random-IO auf sich zieht, wird diese vom Seitenersetzungsalgorithmus bevorzugt, da trotz der Prämisse, dass SSDs sowohl für Random-IO als auch für sequentielles IO schneller sind als Festplatten, SSDs auf jeden fall schneller in Bezug auf Random-IO sind. Dies ist ein großer Vorteil von TAC gegenüber LRU, welches alle Zugriffe gleich behandeln würde.

3.4 Ergebnisse

3.4.1 Seitenersetzungsstrategie Canim et al. vergleichen in ihrer Arbeit diesen mit anderen Algorithmen mithilfe eines TPC-H Benchmarks, um die Leistungsfähigkeit zu belegen und die Auswahl zu begründen. Konkret wird der TAC-Algorithmus mit den folgenden verglichen:

- First in first out (FIFO): Die Seite, die zuerst geschrieben wurde, ist die erste Seite welche überschrieben wird.
- Least recently used (LRU): Die Seite, auf welche am längsten nicht mehr zugegriffen wurde, wird als erste überschrieben.
- Clock: Funktioniert ähnlich wie LRU, ist aber leichter zu implementieren: Jede Cache-Seite besitzt ein Flag, welches angibt, ob auf die Seite zugegriffen wurde und ein Zeiger durchläuft alle Cache-Seiten. Besitzt dann die aktuell vom Zeiger referenzierte Seite ein nicht aktiviertes Cache-Flag, so wird diese ersetzt. Andernfalls läuft der Zeiger weiter bis eine solche Seite gefunden wird.
- Adaptive replacement (ARC): Ist eine Erweiterung von LRU: Es gibt vier Listen: Eine für Seiten auf die zuletzt zugegriffen wurde, eine für Seiten auf die oft zugegriffen wird (Seite war in der „zuletzt zugegriffen“-Liste und es wurde erneut auf sie zugegriffen) und zwei Listen, die verwalten, was in den vorherigen Listen einmal vorhanden war, aber nicht mehr vorhanden ist. Mit den letzten zwei Listen passt ein Algorithmus dann selbstständig die Größe der ersten zwei Listen an. Die Seitenersetzung in den ersten zwei Listen funktioniert für gewöhnlich mit LRU.
- Optimal replacement (OPT / MIN): Ein Offline-Algorithmus, welcher mithilfe aller zukünftigen Seitenzugriffe die optimale Seitenersetzungsstrategie berechnet.
- Kein Cachealgorithmus als Referenz-/Kontrollstrategie um eine Untergrenze zu haben

Mithilfe des Vergleichs wurde gezeigt, dass TAC am besten arbeitet (siehe Abbildung 2)

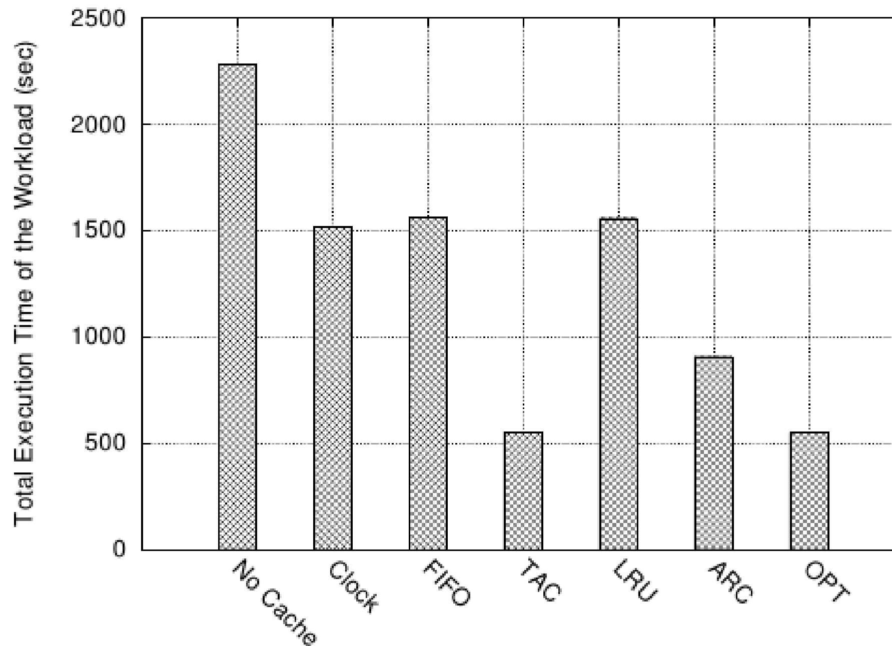


Abb. 2. Vergleich verschiedener Ersetzungsalgorithmen (Quelle: [2])

3.4.2 Aufbau Um den Aufbau zu prüfen, verwenden Canim et al. sowohl den TPC-C (simuliert Handelstransaktionen) als auch den TPC-H (simuliert hochkomplexe Anfragen mit langer Ausführungszeit) Benchmark, wobei der Ablauf der Benchmarks aufgezeichnet und dann mit und ohne Cache wiederholt abgespielt wird. Der Cache selbst wurde dabei direkt in DB2 LUW implementiert.

Folgende Ergebnisse wurden gemessen ([2]):

- Die Ausführung der Versuchstransaktionen benötigt bis zu 12x weniger Zeit.
- Verglichen mit anderen Seitenersetzungsalgorithmen benötigt die Ausführung mit dem temperaturbasierte Algorithmus bis zu 3x weniger Zeit.
- Durch den temperaturbasierten Seitenersetzungsalgorithmus ist die SSD in der Lage bis zu 83% der potentiellen Lesezugriffe auf die Festplatte (also Lesezugriffe, bei denen die Daten nicht im Arbeitsspeicher waren) selbst zu bedienen, wobei die Datenbank für diese Ergebnisse 5 GB, der Arbeitsspeicher 200 MB und der Cache 1,2 GB misst.

3.5 Weitere Ansätze und Verbesserungen

Neben dem hier dargestellten Aufbau schlagen Canim et al. einige weitere Ansatzmöglichkeiten vor, die die Leistung weiter verbessern können:

3.5.1 Eine kalte Seite in konstanter Zeit bestimmen

Wenn es ausreicht, eine sehr kalte Seite im Cache (und nicht die kälteste Seite) zu finden, so ist dies in konstanter Zeit möglich (zur Erinnerung: die kälteste Seite finden, benötigt logarithmische Zeit). Hierfür wird die Temperaturbandbreite in eine konstante Anzahl sogenannter „Bänder" aufgeteilt, welche technisch als verkettete Listen realisiert werden. Sobald die Temperaturinformationen einer Seite aktualisiert werden müssen, wird gleichzeitig entschieden, ob die Seite in der gleichen Liste verbleibt oder ob sie verschoben werden muss. Um jetzt eine sehr kalte Seite zu finden, müssen die Listen von kalt nach warm durchlaufen werden bis eine nichtleere Liste gefunden wird (konstante Laufzeit), dann wird aus dieser Liste das erste Element entfernt (mit zusätzlicher Variable in konstanter Laufzeit), es wird also LRU für die Listen verwendet. Canim et al. schlagen 50-100 Listen vor, da weniger Listen das Spektrum zu stark egalisieren und mehr die Performance zu sehr verschlechtern. Dieser Ansatz hat sich jedoch trotz der theoretisch besseren Laufzeit in der Praxis als nicht wesentlich performanter herausgestellt, weswegen Canim et al. den einfacher zu realisierenden Ansatz der Heaps verwenden, bei dem eine Seite aktzeptiert wird, wenn sie mindestens 1% wärmer ist als die kälteste Seite innerhalb des Heaps.

3.5.2 Die Schreibzugriffe auf die Festplatte puffern

Wie bereits in Kapitel 2 auf Seite 3 angedacht, besteht die Möglichkeit, mit einer SSD Schreibzugriffe auf die Festplatte in sequentielle Schreibzugriffe umzuwandeln und nicht auf das Schreiben auf die Festplatte warten zu müssen. Canim et al. nennen hier die folgenden konkreten Möglichkeiten:

1. Einen Teil der SSD hierfür reservieren und diesen, sobald komplett gefüllt, auf die Festplatte schreiben.
2. Die Seiten direkt im SSD-Cache als *dirty* markieren und dann, sobald ein bestimmter Prozentsatz dirty ist, diese Seiten auf die Festplatten schreiben und die Seiten im Cache als *clean* markieren. Der Vorteil dieses Ansatzes ist, dass mehr SSD-Speicherplatz für den tatsächlichen SSD-Cache bereitsteht. Die Nachteile sind mehr Lesezugriffe auf die SSD und ein höherer Verwaltungsaufwand.

In jedem Fall werden die gepufferten Seiten vor dem Schreiben entsprechend der Zieladresse auf der Festplatte sequentiell vorsortiert. Es konnte durch Versuche

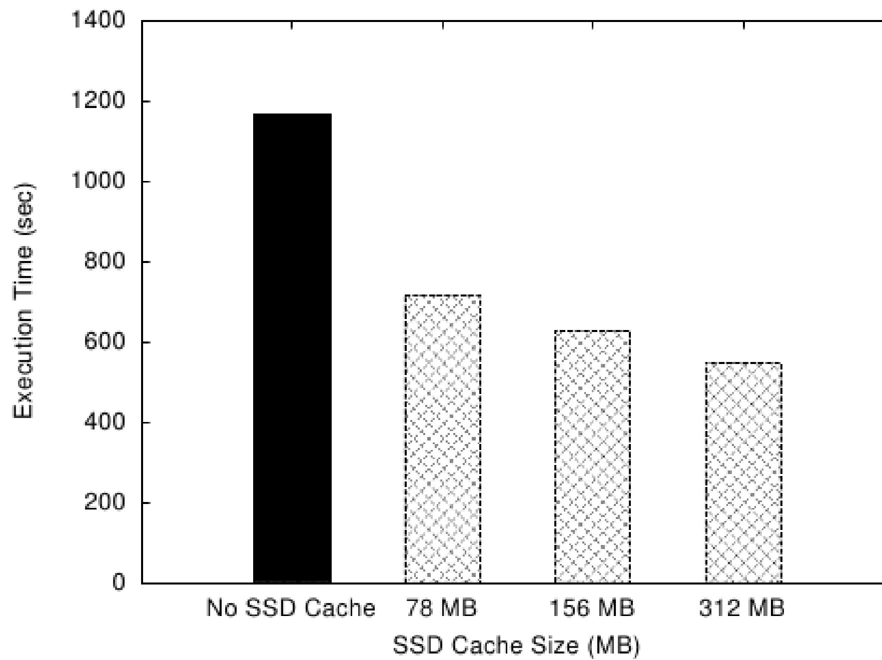


Abb. 3. Ausführungszeit mit verschiedenen Größen des SSD-Schreibzugriff-Puffers (Quelle: [2])

bei denen nur Schreibzugriffe verwendet wurden gezeigt werden, dass dieser Puffer eine erhebliche Auswirkung auf die Ausführungszeit hat (siehe Abbildung 3). Canin et al. verwendeten diesen Ansatz trotzdem nicht, da dadurch, dass die Daten von SSD-Cache und Festplatte nicht mehr immer konsistent zueinander sind eben auch der Aufwand in Bezug auf Recovery stark ansteigt.

4 Recovery

4.1 Motivation

Nachdem nun der innere Aufbau von SSD-Caches beleuchtet wurde, werde ich als nächstes auf den Recoveryprozess nach einem Neustart oder Systemcrash eingehen.

Das grundsätzliche Problem bei einem Neustart/Crash ist, dass der vorher gut gefüllte Cache entsprechend einem Aufbau aus Kapitel 3 auf Seite 5 unbrauchbar wird, obwohl die Daten noch immer vorhanden sind, da keine Informationen über die darin enthalteten Daten (Metadaten) persistent gespeichert wurden; alle Verwaltungsinformationen befanden sich nur nicht-persistent im Arbeitsspeicher. Zwar wird der SSD-Cache den Recoveryprozess nichtsdestotrotz unterstützen, denn auch bei diesem werden möglicherweise mehr Daten gelesen, als in den zur Verfügung stehenden Arbeitsspeicher passen und es handelt sich um sehr viel Random-IO. Diese Art der Unterstützung ist jedoch nicht optimal, denn die Daten, welche während des Recoveryprozess benötigt werden, müssen ohne einen gefüllten SSD-Cache zunächst von der sehr viel langsameren Festplatte gelesen werden. Würde der SSD-Cache hingegen noch so, wie er vor dem Crash gefüllt war, zur Verfügung stehen, so würde er genau die Daten enthalten welche im Recoveryprozess benötigt werden — die Daten welche kurz vor dem Crash verarbeitet wurden.

Ein gefüllter SSD-Cache würde den Recoveryprozess also beschleunigen — doch warum ist dies notwendig? Ist ein Crash so häufig, dass die Vorteile des beschleunigten Recoveryprozesses den Nachteil der erhöhten Komplexität des Gesamtsystems überwiegen? Diese Frage ist mit Ja zu beantworten: Laut [3] sollte man damit rechnen, dass ein Systemcrash mehrmals pro Woche auftritt. Jeder dieser Crashs sorgt dafür, dass das gesamte System nicht verfügbar ist. Es kommt also zu Produktivitätsverlusten, da nicht gearbeitet werden kann und möglicherweise sogar zu Vertragsstrafen, wenn ein bestimmter, vorher vertraglich garantierter Prozentsatz bei der Verfügbarkeit unterschritten wird.

Nehmen wir einmal an, das System durchläuft drei Mal pro Woche einen Crash inklusive Recoveryprozess, welcher etwa 6 Minuten benötigt (dies ist die Dauer, welcher der Recoveryprozess im Versuchsaufbau in [2] ohne Cache benötigt). Dann ist das System nur dadurch bereits 936 Minuten bzw ca. 16 Stunden pro Jahr offline, erreicht also nur noch maximal eine Verfügbarkeit von 99,8% — oder anders ausgedrückt: An zwei Arbeitstagen pro Jahr wird das Personal bezahlt ohne tatsächlich zu arbeiten, nur weil die Datenbank offline ist. Hierbei wurde die Zeit die benötigt wird, bis das System einen performancemäßig stabilen Zustand erreicht hat, noch nicht mit eingerechnet.

Es ist also von großer Bedeutung, den Recoveryprozess zu beschleunigen. Trägt ein persistenter Cache zu dieser Beschleunigung bei, so ist die erhöhte Komplexität gerechtfertigt.

4.2 Lösungsansätze für die Metadaten und Recovery

In diesem Abschnitt soll das bereits erläuterte Konzept des SSD-Caching um den Aspekt des Recoveryprozesses erweitert werden. Zentraler Punkt ist dabei das Speichern der Metadaten über den Neustart hinaus. Hierfür werden drei Verfahren vorgestellt und näher beschrieben: „Update-Write-Update“, „Write-Update“ und „Lazy-Update Following an Update-Write“ (LUFUW). Ersteres ist ein Konzept, welches sich stark am Transaktionskonzept ausrichtet und den gesamten Cache nach dem Neustart zur Verfügung stellen möchte, dadurch aber weniger Performance leisten kann. Write-Update versucht, einen der drei Schreibzugriffe pro Datenschreibzugriff zu vermeiden und geht dabei auch das Risiko ein, dass große Teile des Caches nach dem Crash unbrauchbar werden. LUFUW versucht hingegen möglichst viel, aber nicht unbedingt den gesamten Inhalt des Caches „warm“ zu halten, um so eine Steigerung der Performance zu erreichen.

4.2.1 Update-Write-Update Zunächst wird der Ansatz von Bhattacharjee, Canim et al. ([1]) betrachtet. Dieser Ansatz stellt eine Erweiterung von [2] dar, zu welchem in Kapitel 3 auf Seite 5 ein Überblick gegeben wurde. Diese ursprüngliche Funktion als SSD-Cache mit nicht-persistenten Metadaten beschleunigt zwar das System im laufenden Betrieb, der Cache kann aber nach einem Neustart, welcher möglicherweise durch einen Crash bedingt wurde, nicht einfach weitergenutzt werden, sondern muss neu gefüllt werden. Die Erweiterung ([1]) beschäftigt sich deswegen hauptsächlich damit, die Metadaten persistent zu speichern und die Daten konsistent zur Festplatte zu halten, sodass der SSD-Cache nach einem Neustart noch „warm“ ist und sowohl den Recoveryprozess beschleunigt, als auch nach diesem weiter zur Verfügung steht, sodass das System insgesamt schneller wieder den Zustand von vor dem Neustart erreicht. Es handelt sich um ein Update-Write-Update Konzept, da vor und nach jedem Daten-Write jeweils eine zusätzliche Write-Operation benötigt wird. Der Write wird also erst angekündigt und dann danach bestätigt.

Systemaufbau

Gegenüber dem ursprünglichen Ansatz verändert sich das System in Bezug auf das Schreiben von Seiten in den SSD-Cache, genauer die Verwaltung von Metadaten (vergleiche Abbildung 4 auf der nächsten Seite). Dies ist notwendig, damit der SSD-Cache immer, auch nach einem Neustart, zuverlässig verwendet werden kann und der exakte Inhalt, sowie dessen Zustand in Bezug auf seine Konsistenz zur Festplatte bekannt ist. Konkret wird vor dem Schreiben einer neuen Seite in den SSD-Cache die zu beschreibende Seite in den Metadaten als nicht valide und erst nach dem erfolgreichen Schreiben wieder als valide markiert. Dies

Abb. 4. Systemaufbau des Ansatz von Bhattacharjee, Canim et al. (Quelle: [1])

bringt allerdings auch den Nachteil mit sich, dass Seiten im Cache, welche valide sind, aber noch nicht als solche markiert wurden, nach dem Crash womöglich nicht verwendet werden. Bhattacharjee et al. sahen dies als vernachlässigbar gegenüber der andererseits immer bestehenden Konsistenz an.

Metadaten

Bei diesem Ansatz werden die Metadaten, genauer gesagt die Zuordnung von Seiten auf der Festplatte zu Seiten auf der SSD und die Temperaturaturinformationen des temperaturbasierten Seitenersetzungsalgorithmus, persistent auf der SSD gespeichert (im nicht erweiterten Ansatz werden sie nur im Arbeitsspeicher gehalten). Die Seitenzuordnung wird technisch betrachtet als Datei gespeichert, welche eine vorher festgelegt Größe hat. Diese Datei wird dann, wie bereits in Kapitel 4.2.1 auf der vorherigen Seite dargestellt, angepasst, sobald der Inhalt des SSD-Caches sich ändert, sei dies durch einen Austausch einer gecachten Seite oder durch das hinzukommen von neuen Seiten in einen noch nicht komplett gefüllten Cache. Da die Datei relativ klein ist (konstant 23MB bei einem SSD-Cache mit 3,6GB potentiellem Speicher im Versuchsaufbau von Bhattacharjee et al, also $< 1\%$) kann sie auch auf einem schnelleren Gerät als einer SSD gespeichert werden, z.B. auf PCM-Speicher.

Ergebnisse

Um den Systemaufbau zu testen, vergleichen Bhattacharjee et al. ihn mit dem Ursprünglichen, ohne Erweiterung nach einem Neustart ([2]):

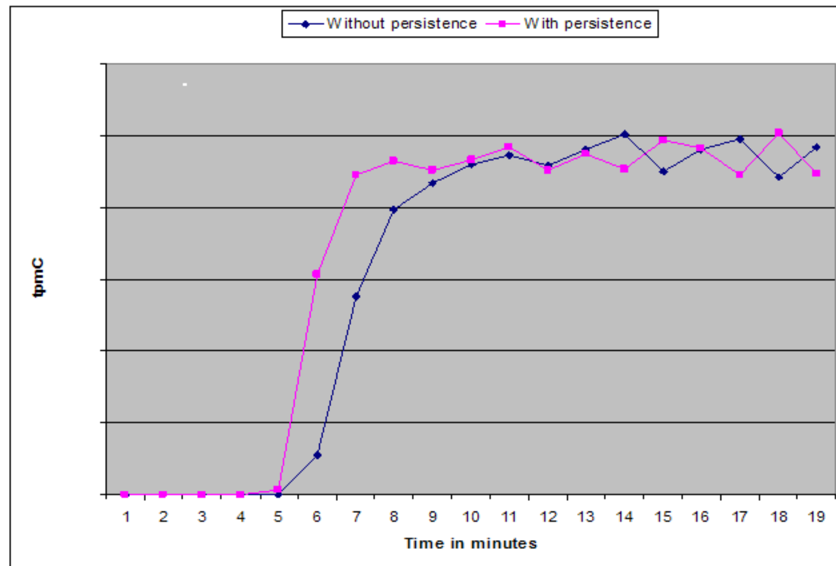


Abb. 5. Durchsatz im Zeitablauf (Quelle: [1])

In Abbildung 5 kann man erkennen, dass mit den Anpassungen der Recoveryprozess nach 4 Minuten abgeschlossen wird und nach 8 Minuten die Performance ein stabiles Level erreicht. Ohne die Anpassungen hingegen endet der Recoveryprozess nach 5 Minuten und die Performance stabilisiert sich nach 10 Minuten. Insgesamt wird also 20% weniger Zeit für den Recoveryprozess benötigt.

In Abbildung 6 auf der nächsten Seite ist zu erkennen, dass ohne die Erweiterung, selbst nach dem Recoveryprozess, welcher etwa bis zum x-Wert 50 (= 250 Sekunden) dauert, der Cache noch nicht genutzt wird, und er mit der Recoveryerweiterung auch schon während des eigentlichen Recoveryprozesses genutzt wird. Außerdem stabilisiert sich die Nutzung mit der Erweiterung wesentlich früher als ohne die Erweiterung.

In Abbildung 7 auf der nächsten Seite kann man den Nachteil der Erweiterung erkennen: Die CPU-Last ist sowohl während der Recoveryphase, als auch darüber hinaus wesentlich höher.

Außerdem wurde wärend des Testens erkannt, dass die IO-Last durch das Schreiben der Metadaten auf die SSD gegenüber dem Schreiben der Metadaten in den

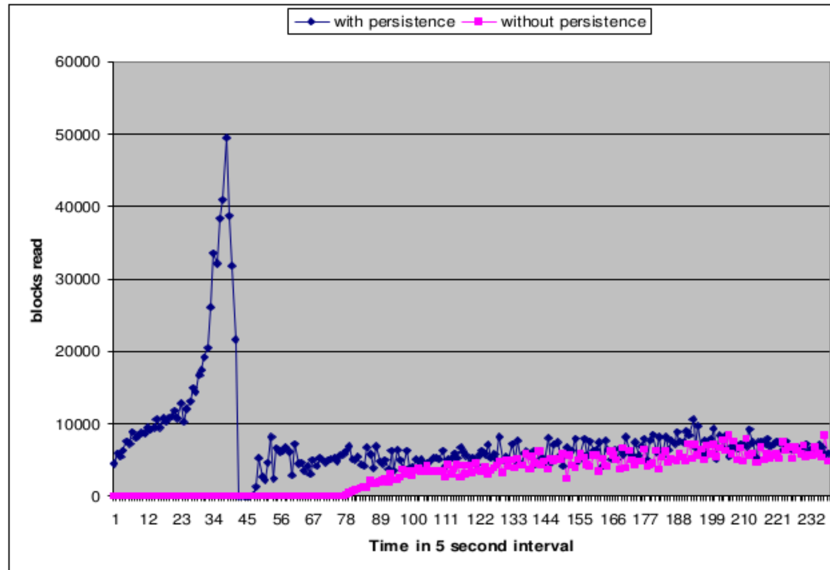


Abb. 6. SSD-Leseoperationen im Zeitablauf (Quelle: [1])

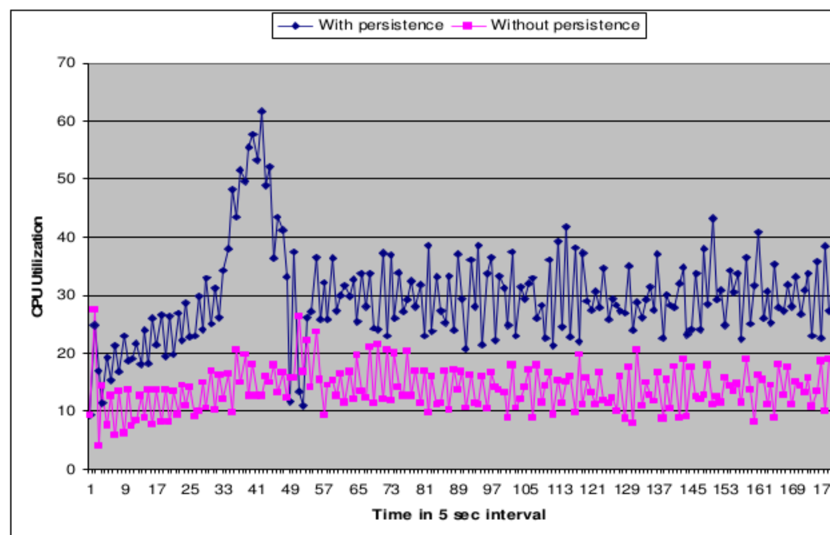


Abb. 7. CPU-Last im Zeitablauf (Quelle: [1])

Arbeitsspeicher vernachlässigbar ist und das System nicht wesentlich langsamer macht.

Mögliche Verbesserung

Zum hier vorgestellten Aufbau sehen Bhattacharjee et al. noch eine weitere mögliche Anpassungen, welche vorgenommen werden könnte: Durch das Entfernen der Zuordnungsdatei (Festplattenseiten zu SSD-Seiten) von der SSD, wodurch die Verwaltung ausschließlich nicht-persistent im Arbeitsspeicher geschieht, ist es möglich weiter die Performance zu steigern. Bei einem Neustart muss dann vor dem Recoveryprozess der SSD-Cache gescannt werden und die Daten entsprechend ihrer auf der SSD gespeicherten Seiten-ID mit den Daten von der Festplatte abgeglichen werden. Der Zeitbedarf hierfür wird mit wenigen Minuten beziffert. Der unterschied zum ursprünglichen Ansatz mit einem nicht persistenten Cache besteht hierbei in dem Versuch den Cache neu zu indexieren.

Der Vorteil dieses Ansatzes ist, dass zur Laufzeit weniger Performance durch die zusätzliche IO-Last der Zuordnungsdatei verloren geht, was jedoch bedingt durch die vergleichsweise geringe IO-Last nur einen geringe Performancesteigerung ermöglicht. Es kann so jedoch andererseits nicht sichergestellt werden, dass der Inhalt der SSD immer konsistent zum Inhalt der Festplatte ist, da es beispielsweise nach dem Schreiben auf die SSD aber vor dem Schreiben auf die Festplatte zum Absturz kam. Dies wird zum Beispiel dann zum Problem, wenn während des Recoveryprozess eine Seite von der SSD gelesen wird, welche auf der Festplatte veraltet ist. Die LSN der Seite verhindert dann Operationen an der Seite, da diese ja aktuell ist. Im Anschluss daran wird die Seite dann aber möglicherweise noch während dem Recoveryprozesses aus dem SSD-Cache verdrängt wodurch, die aktuelle Version verloren geht und nur die veraltete auf der Festplatte zurückbleibt — die Transaktion, die zu Änderungen führte, war also nicht dauerhaft.

4.2.2 Write-Update bei Facebooks Flashcache

Nachdem nun mit Update-Write-Update der grundlegendste Ansatz beleuchtet wurde, soll nun ein weiterer vorgestellt werden, welcher von Facebook im Rahmen der Entwicklung an dem von ihnen genutzten Flashcache erarbeitet wurde ([4,5]). Dieser Ansatz ist eine Optimierung von Update-Write-Update mit dem Ziel, die IO-Last pro Datenschreiboperation von drei auf zwei zu verringern. Hierfür wird als Nachteil in Kauf genommen, dass beim Recoveryprozess nicht mehr der gesamte Datenbestand wiederhergestellt werden kann.

Ablauf

Der Aufbau des Caches selbst ist folgender: Die gesamte zur Verfügung stehende Speichermenge wird in Sets („Buckets“) mit einer vorher festgelegten Größe aufgeteilt. Innerhalb eines solchen Sets wird ein Block via Hashing mit Linear Probing gefunden.

Es sei zu beachten: Es handelt sich bei Flashcache um einen Write-Back Cache. Das bedeutet, dass Datenschreibeoperationen zunächst nur auf der SSD stattfinden. Außerdem hat jeder Block einen Zustand: *dirty*, valide oder invalide. Die Zustandsmetadaten werden dabei auf der SSD nur bei Schreiboperationen („dirty“) oder beim Schreiben auf die Festplatte („valid“) markiert. Metadatenupdates werden dabei — wenn möglich — zusammengefasst, um unnötige mehrfache Schreiboperationen in den gleichen Block zu vermeiden.

Eine Schreiboperation läuft folgendermaßen ab:

1. Schreibe die Daten auf die SSD und markiere gleichzeitig den geschriebenen Block persistent als dirty, falls nicht bereits geschehen.
2. Die als dirty markierten Daten werden von Background-Threads auf die Festplatte geschrieben. Findet sich kein geeigneter Zeitpunkt hierfür, aber wurde eine vorher bestimmte Anzahl an dirty Daten überschritten oder wurden die Daten zu lange gepuffert und nicht geschrieben (Standardwert: 15 Minuten), so wird das Schreiben auf die Festplatte erzwungen. Die Blöcke, bzw das Set, welches geschrieben werden soll, wird basierend auf FIFO oder LRU (Standardeinstellung: FIFO) ausgewählt. Das ausgewählte Set wird dann sortiert und weitere zusammenhängende Blöcke werden ebenfalls geschrieben (eine große IO-Last, statt vieler kleiner). Im Anschluss an das Schreiben werden die Daten in den Metadaten persistent als Valid markiert.

Bei einem geplanten, vom Administrator durchgeführten Neustart, werden die gesamten Metadaten aus dem Arbeitsspeicher und ein Flag, welches den sauberen Neustart markiert, persistent auf der SSD gespeichert, um dann nach dem Neustart sowohl die validen Daten als auch die dirty Daten vom Cache wiederzuverwenden. Bei einem Crash, also einem unsauberen Neustart ohne gesetztes Flag, werden nur die als dirty markierten Daten weiter verwendet, da die „validen“ Daten möglicherweise gerade überschrieben wurden, jedoch noch nicht als dirty markiert wurden. Dies ist der Nachteil gegenüber Update-Write-Update, da das erste Update bei Write-Update fehlt.

Außerdem gibt es zwei unterschiedliche Möglichkeiten des Herunterfahrens: Bei der schnellen Methode werden nur die Metadaten geschrieben und nach dem Neustart werden sowohl dirty als auch auf die Festplatte geschriebene Blöcke vorgefunden. Bei der langsamen Methode werden alle als dirty markierten Blöcke

auf die Festplatte geschrieben, danach werden die Metadaten geschrieben, und nach einem Neustart werden nur geschriebene Blöcke vorgefunden.

Leseanfragen zu Daten gehen immer zuerst an den Cache. Dieser wird dann bei einem Cache Miss, basierend auf der vorher festgelegten Ersetzungsstrategie, gefüllt.

Probleme bei partiellem Schreiben

Es besteht das Problem, dass bei einem Crash oder einem Stromausfall partiell geschriebene Daten bereits synchron als dirty markiert wurden. Um diesem Problem zu begegnen, ist es möglich, bei Flashcache Prüfsummen zu verwenden, welche jedoch in den Standardeinstellungen deaktiviert sind, da die Metadaten dadurch um 33% größer sind (32 Byte statt 24 Byte))

4.2.3 Lazy-Update Following an Update-Write nach Yang und Yang

In diesem Abschnitt soll eine weitere Lösung für das persistente Speichern der Metadaten dargestellt werden. Diese Lösung geht zurück auf [6]. Yang und Yang präsentieren in dieser Arbeit die von ihnen LUFUW (Lazy-Update Following an Update-Write) genannte Methode, welche sie als weitere Methode neben Update-Write-Update (Kapitel 4.2.1 auf Seite 14) und Write-Update (Kapitel 4.2.2 auf Seite 18) sehen.

Yang und Yang leiten damit ein, dass die bisher bekannten und bereits genannten Methoden nicht optimal sind: Update-Write-Update benötigt für eine Schreibeoperation für Daten insgesamt drei Schreibeoperationen und Write-Update ist nicht in der Lage, ausreichend große Mengen der Daten (laut Yang und Yang nur insgesamt 14%) auch nach dem Neustart weiterzuverwenden, da nur als dirty markierte Daten genutzt werden.

Ablauf

Eine Schreiboperation für Daten (siehe Abbildung 8 auf der nächsten Seite) läuft bei der LUFUW-Methode folgendermaßen ab:

1. Schreibe in die Metadaten (sowohl im Arbeitsspeicher als auch auf der SSD) eine Flag, dass die betreffenden Daten gerade geschrieben werden
2. Nach erfolgreichem Schreiben wird dieses Flag in der Metadatenkopie im Arbeitsspeicher wieder zurückgesetzt und erst dann wird dem System der Abschluss des Schreibvorgangs gemeldet
3. Wenn das nächste mal Schritt 1 ausgeführt oder anderweitig in die Metadaten auf der SSD im selben Block geschrieben wird, werden ebenfalls die

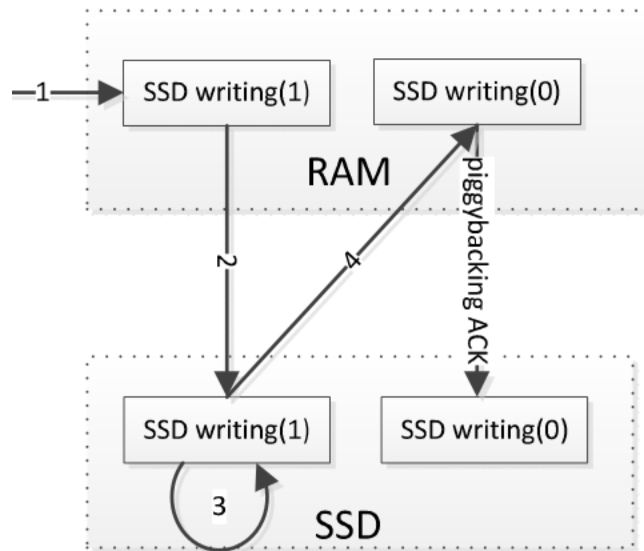


Abb. 8. Ablauf einer Schreiboperation bei LUFUW (Quelle: [6])

zurückgesetzten Flags von anderen Schreiboperationen auf die SSD geschrieben (Lazy Update)

Ziel dieses Lazy-Updates ist es, die Gesamtanzahl der Schreiboperationen zu reduzieren. Ausgenutzt wird dabei, dass eine SSD in 4KB Blöcken angesprochen wird, und ein solcher Block nie nur die Metadaten zu einer einzelnen Datenseite umfasst, sondern im Aufbau von Yang z.B. 240 Einträge, weswegen eine Schreiboperation auch gleichzeitig die anderen Flags schreiben kann, welche ebenfalls in dem entsprechenden Block liegen. Die Warscheinlichkeit, die anderen Flags ebenfalls schreiben zu können ist, also sehr hoch, und steigt bedingt dadurch, dass in der Regel Daten, welche in der Nähe der bereits verwendeten Daten liegen, im weiteren Ablauf benötigt werden.

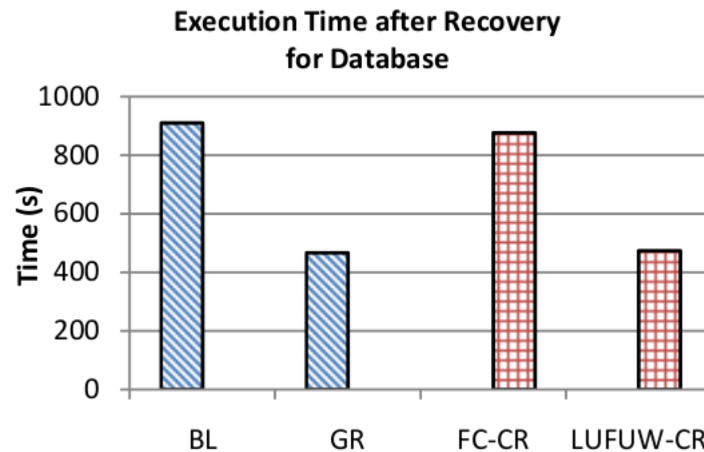
Es wird dabei zugunsten der geringeren IO-Last in Kauf genommen, dass auch konsistente Daten möglicherweise dieses Flag, welches eine mögliche Inkonsistenz symbolisiert, besitzen. Diese nicht geschriebenen Flags werden mithilfe eines sich selbst verschiebenden Fensters, das die Indexnummern aller nicht geschriebener Metadaten auf der SSD enthält und mit einem Timestamp verwaltet, welche mit jedem Metadatenupdate persistent auf die SSD geschrieben werden (es existiert also mehr als einmal physisch auf der SSD). Das Fenster verschiebt sich, sobald das älteste, nicht bestätigte, Flag geschrieben wird. Um die Anzahl der als inkonsistent markierten konsistenten Daten zu reduzieren, wird nach einer

bestimmten Anzahl von nicht persistent geschriebenen Flags (bei Yang 63) das Schreiben dieser erzwungen. Bei einem ungeplanten Neustart muss dann mithilfe des Timestamps das jüngste Fenster auf der SSD gefunden werden, um dann mit diesem als weitere Sicherheitsmaßnahme sicherstellen zu können, dass die enthaltenen Datenblöcke nicht als konsistent angenommen werden.

Kommt es zu einem geplanten Neustart (d.h. vom Administrator ausgelöst), so werden alle Metadaten vorher persistent gesichert.

Ergebnisse

Um den Aufbau zu testen und Ergebnisse präsentieren zu können, wurde ein Prototyp basierend auf Flashcache 1.0.121 erstellt. Dieser wurde mit verschiedenen anderen Recoveryansätzen verglichen (siehe Abbildung 9)



- BL („Baseline“): Neustart mit leerem Cache
- GR („Graceful reboot“): Metadaten wurden vor geplantem Neustart persistent gesichert
- FC-CR: Flashcaches crash recovery
- LUFUW-CR: LUFUWs crash recovery

Abb. 9. Zeitbedarf verschiedener Recoveryansätze (Quelle: [6])

Folgendes wurde dabei festgestellt:

- 99% der gecachten Daten sind nach dem Neustart noch nutzbar
- Der Recoveryprozess ist bis zu zweimal schneller abgeschlossen als bei Write-Update mit Flashcache

– Im laufenden Betrieb sind Flashcache und LUFUW sich im Hinblick auf die Performance sehr ähnlich

5 Fazit

Es wurde in dieser Arbeit sowohl das Konzept des SSD-Caches, als auch das Problem bei einem Crash und damit einhergehende Lösungen beleuchtet. Zusammenfassend lässt sich sagen, dass bei der Verwendung der verschiedenen Methoden zum persistenten Schreiben der Metadaten immer unterschiedliche Nachteile in Kauf genommen werden müssen. Update-Write-Update ermöglicht es, denn gesamten Cache nach dem Neustart wiederzuverwenden, dies wird jedoch mit Performanceeinbußen im laufenden Betrieb bezahlt, da zu jeder einzelnen Schreiboperation sich zwei weitere für die Metadaten gesellen. Write-Update versucht dies zu umgehen, indem das erste Update ausgelassen wird, was jedoch bei einem Crash dazu führt, dass relativ wenige Daten weiter verwendet werden können (14%, siehe Kapitel 4.2.3 auf Seite 20). Lazy-Update Following an Update-Write (LUFUW) versucht, beide Ansätze zu vereinen: Hohe Performance im laufenden Betrieb wie bei Flashcache verbunden mit der Möglichkeit, große (wenn auch nicht alle) Daten nach einem Crash wieder zu verwenden. Dieser Ansatz kann als erfolgreich bezeichnet werden, wie in den Ergebnissen in Kapitel 4.2.3 auf Seite 20 zu erkennen ist.

Diese Arbeit beschäftigt sich dabei jedoch nur mit dem Aspekt, Leseoperationen zu beschleunigen. Der SSD-Cache von Canim et al. dient nicht dazu, Schreiboperationen schneller auszuführen, auch wenn Möglichkeiten hierfür in den weiteren Ansätzen genannt werden, welche auch im zugehörigen Paper erwähnt werden; Flashcache bedient sich ebenfalls dieser Möglichkeit. Als weitere Option, um die Performance des Gesamtsystems zu steigern, sei deswegen vorzuschlagen, ebenfalls die Schreiboperationen zu optimieren; möglicherweise auch durch einen verbundenen, adaptiven Puffer, welcher bei einer großen Anzahl an Schreibzugriffen mehr Speicherplatz zur Verfügung stellt, um selbige zu puffern und dann im Hintergrund auf die Festplatte zu schreiben.

Literatur

1. BHATTACHARJEE, Bishwaranjan, et al. Enhancing recovery using an SSD buffer pool extension. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware. ACM, 2011. S. 10-16.
2. CANIM, Mustafa, et al. SSD bufferpool extensions for database systems. Proceedings of the VLDB Endowment, 2010, 3. Jg., Nr. 1-2, S. 1435-1446.

3. HAERDER, Theo; REUTER, Andreas. Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR), 1983, 15. Jg., Nr. 4, S. 287-317.
4. SRINIVASAN, Mohan; CALLAGHAN, Mark. FlashCache. 2010; http://github.com/facebook/flashcache
5. SRINIVASAN, M.; SAAB, P. Flashcache: a general purpose writeback block cache for linux, 2011.
6. YANG, Jing; YANG, Qing. A New Metadata Update Method for Fast Recovery of SSD Cache. In: Networking, Architecture and Storage (NAS), 2013 IEEE Eighth International Conference on. IEEE, 2013. S. 60-67.