# Übungsblatt 3: Optimistische Nebenläufigkeit mit ZFS-Snapshots

**Hinweis**

Der zugehörige Code befindet sich im folgenden Github Repository: https://github.com/kilianbartz/bs_ueb3.

## 1. Aufgabe 1

Zur Bearbeitung dieser Aufgabe habe ich die im Ordner `a1_a3` befindliche Java Bibilothek *Transaction* erstellt. Um eine möglich breite Kompatibilität zu anderen Programmen zu bieten, stellt die Anwendung ein **zustandsloses** *Command Line Interface* bereit. Implementiert sind die folgenden 5 Operationen:

- **transaction** start *name*: Startet eine Transaktion, speichert den Ausgangszustand des *zfs directory*[1] und legt einen ZFS Snapshot an.
- **transaction** Commit *name*: Falls keine Konflikte vorliegen (also seit Start der Transaktion keine Änderungen im *zfs directory* durchgeführt wurden), persistiere die writes innerhalb der Transaktion und lösche anschließend den zugehörigen Snapshot und die Transaktions-Datei. Andernfalls: Führe ein Rollback zum Transaktions-Start aus.
- **write** *name datei inhalt*: Plane *datei* mit *inhalt* als Teil der Transaktion *name* zu schreiben. Diese Änderung wird erst beim Commit persisitiert.
- **read** *name datei*: Lese *datei* als Teil der Transaktion *name*. Hat sich der Zustand des *zfs directory* seit Beginn der Transaktion geändert, führe ein Rollback durch.
- **remove** *name datei*: Plane *datei* als Teil der Transaktion *name* zu löschen. Diese Änderung wird erst beim Commit persisitiert.
- **redo**: Vervollständige den Commit jeder Transaktion, bei der der Commit schon begonnen wurde.

Die ZFS-Befehle werden mittels `ProcessBuilder` ausgeführt. Dies spart einerseits eine weitere Abhängigkeit, liegt aber vor allem daran, dass es keine populäre Java-Bibliothek für ZFS gibt. Zur Konfiguration der Transaktions-Logik steht eine `transactions_config.toml` zur Verfügung, in der das *zfs directory* über die beiden Parameter `zfs_filesystem` und `file_root` angepasst werden kann.

### 1.1. Was ist eine Transaktion und wie wird Atomizität und Isolation gewahrt?

Wird eine Transaktion erstellt (`TransactionNoBuffering.java`), wird in der HashMap `fileTimestamps` der ursprüngliche Zustand des *zfs directory* festgehalten (für jede Datei wird der **Timestamp** der letzten Modifikation gespeichert) und ein ZFS Snapshot erstellt. Wird eine Datei verändert (read/write/remove), wird ihr Name zunächst in der Liste `relevantFiles` gespeichert und ihr Timestamp aktualisiert. Beim Commit wird für alle Dateien, die relevant für diese Transaktion sind, verifiziert, dass es keine Konflikte gibt.

[1] *zfs directory* meint ein Verzeichnis innerhalb eines zfs pools (hier werden später die Operationen innerhalb der Transaktionen persistiert) und ist $\neq$ *working directory* (Hilfsverzeichnis für temporäre Transaktions-Dateien)

- Ist kein Konflikt aufgetreten, können die Änderungen dieser Transaktion erhalten bleiben und sowohl die Transaktions-Datei, als auch der Snapshot können gelöscht werden.
- Gab es einen Konflikt, wird diese Transaktion durch einen ZFS Reroll zurückgesetzt und die Transaktions-Datei gelöscht.

Zudem wird ein Flag `startedCommit` verwaltet, um Transaktionen zu finden, die noch **nicht vollständig commitet** wurden (wenn die Transaktions-Datei nach einem Crash existiert, jedoch dieses Flag gesetzt ist, ist der Crash während des Commit passiert). Diese Transaktionen sind relevant für die Redo-Operation.

Eine derartige Implementierung gewährleistet zwar Konsistenz und Atomizität, jedoch **keine Isolation**: Dadurch, dass eine Transaktion auch Effekte anderer unvollständiger Transaktionen sehen kann, werden in der Situation aus Abbildung 1 durch das Zurücksetzen von Transaktion A **auch die Änderungen von Transaktion B** gelöscht, welche gemäß des Durability-Prinzips erhalten bleiben müssten. Wären beide Transaktionen isoliert und sequenziell ausgeführt worden, hätte a.txt am Ende entweder den Inhalt, der in Transaktion A oder B geschrieben wurde; in diesem Fall würde der ZFS Rollback a.txt jedoch komplett löschen. Korrekterweise müsste der Rollback von Transaktion A den Zustand nach dem Abschluss von Transaktion B wiederherstellen, jedoch hat Transaktion A keine Möglichkeit auf diesen zuzugreifen.

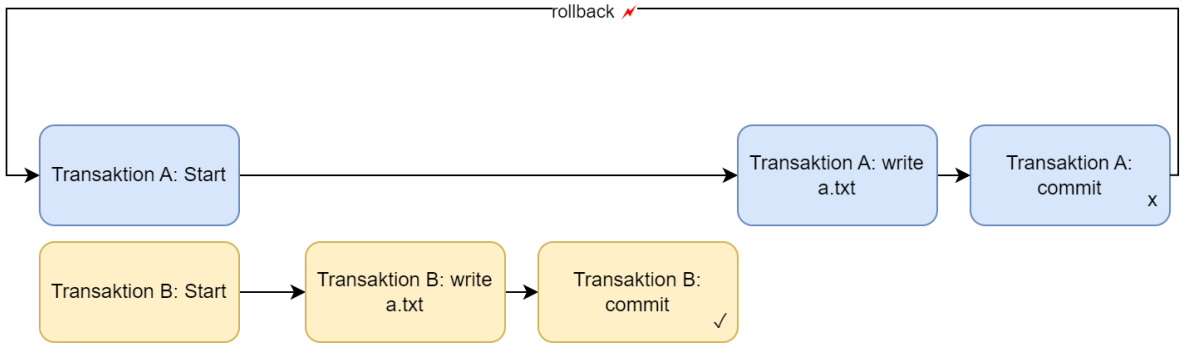



Abbildung 1: Am Anfang existiert a.txt noch nicht. Transaktionen A und B starten gleichzeitig, Transaktion B wird erfolgreich commitet, Transaktion A muss durch den Konflikt zurückgesetzt werden und löscht damit auch die Änderungen von Transaktion B.

Als Alternative, die besser mit den ACID Eigenschaften klar kommt und nicht auf ZFS angewiesen ist, habe ich daher in `Transaction.java` eine Variante der Transaktion implementiert, welche darauf beruht sämtliche (geplanten) Änderungen zunächst in den Datenstrukturen `writes` und `removes` im Sinne des *Write Ahead Logging* zu puffern. Erst wenn zum Commit-Zeitpunkt kein Konflikt vorliegt, werden diese Änderungen tatsächlich im *zfs directory* geschrieben. Kommt es stattdessen zu einem Konflikt, so wird einfach die Transaktions-Datei verworfen und keine Datei im *zfs directory* wurde von der Transaktion modifiziert. Diese zweite Implementierung bietet zudem eine Referenz, mit der die Performance der ZFS Lösung verglichen werden kann.

Diese Implementierung hat neben den Limitierungen von *Write Ahead Logging* (z. B. nur Redo möglich, Änderungen müssen 2x geschrieben werden) zusätzlich den Nachteil, dass die Größe des Logs nicht begrenzt ist und alle geplanten Änderungen aus einer Transaktions-Datei gelesen werden müssen, was vermutlich je nach Situation die Performance deutlich verschlechtern könnte.

## 1.2. Praxisbeispiel zur Veranschaulichung der Funktionsweise beider Transaktions-Implementierungen

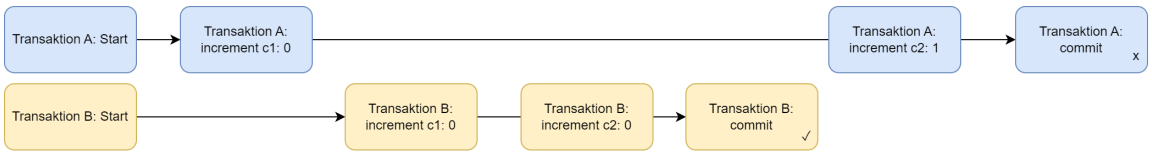
In diesem Praxisbeispiel möchte ich zeigen, dass `TransactionNoBuffering.java` zwar **Konsistenz** einhält, jedoch durch die Seiteneffekte, die bei einem ZFS Rollback auftreten können (i.e., es werden zu viele Dateien, mitunter auch erfolgreiche Transaktionen zurückgesetzt) auch Wiederholungen von verworfenen Transaktionen **keine Isolation** gewährleisten können.

Man stelle sich einen einfachen Counter vor. Er verfügt über zwei Zählerstände $c_1$ und $c_2$. In einem Durchlauf führt er zunächst `update(c1)` und anschließend `update(c2)` gemäß dem untenstehenden Pseudocode aus. Ein Zustand ist dann konsistent, wenn beide Zählerstände gleich (oder beide nicht gesetzt) sind. Initial existieren beide Dateien nicht.

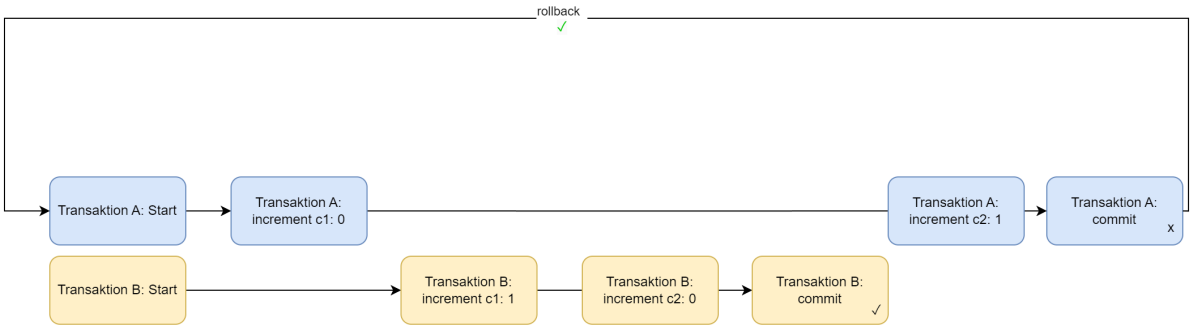
```
update(ci){
  if (read ci.txt results in file_not_found)
    write ci.txt 0
  else
    write ci.txt ((read ci.txt) + 1)
}
```

Der Counter werde nun in 2 Threads mit 2 Transaktionen parallel ausgeführt.

- Sequenziell: Die erste Transaktion würde beide Zähler mit 0 initialisieren, die zweite beide Zähler auf 1 inkrementieren.
- `Transaction.java`: Dadurch, dass die Änderung von c1 in Transaktion A nur geplant und noch nicht persistiert ist, sieht Transaktion B diese Änderung nicht und legt ebenfalls c1 neu mit 0 an. Nach dem Commit von Transaktion B existiert für Transaktion A jedoch c2.txt und wird somit auf 1 erhöht. Beim Commit von Transaktion A wird der Konflikt festgestellt und nur Transaktion A wird verworfen. Am Ende beträgt also der Wert beider Zähler 0. Lediglich die verworfene Transaktion muss wiederholt werden, um Isolation zu erreichen, sodass beide Zähler auf 1 stehen.




- `TransactionNoBuffering.java`: Beide Transaktionen sind nicht atomar, da beide Änderungen der jeweils anderen Transaktion referenzieren. Wird nun das Commit von Transaktion A ausgeführt, werden korrekterweise beide nicht-atomaren Transaktionen zurückgesetzt. Weder c1 noch c2 existieren am Ende also müssten zur Erreichung beide Transaktionen wiederholt werden. Da die erfolgreich ausgeführte Transaktion B jedoch keine Chance hat zu erfahren, dass sie als Seiteneffekt zurückgesetzt wurde, würde nur Transaktion A wiederholt werden und der Durchlauf mit den Zählerständen 0, 0 enden, was nicht der sequenziellen Lösung entspricht.




# 2. Aufgabe 2

Um zu demonstrieren, wie simpel man mit einer generischen Programmiersprache auf die Transaktions-Logik der Java-Applikation aus Aufgabe 1 zugreifen kann, habe ich mich dazu entschieden, das Brainstorming-Tool in Python zu schreiben. Es handelt sich um ein textbasiertes Programm, das zu jeder Idee einen *Namen*, eine *Beschreibung* und *Notizen* erfragt, welche simpel in der Command Line eingegeben werden. Die Idee wird als JSON-Datei gespeichert. Existiert bereits eine Idee mit demselben Namen, werden die alte Beschreibung und Notizen als Hinweis angezeigt.

Damit die Änderungen atomar und konfliktfrei erfolgen, greift das Python-Skript mittels `subprocess` auf die Befehle `transaction start`, `write` und `transaction commit` meiner Applikation aus Aufgabe 1 zu. Die Transaktions-Logik ist dazu in der Python-Klasse `Transaction` gekapselt, sodass sich der Code für das eigentliche Brainstorming-Tool auf nur 40 Lines of Code beläuft. Startet man das Python-Skript, wird eine Transaktion mit einer *UUID* als Namen gestartet. Hat der Nutzer sämtliche Details zur Idee angegeben, wird der entsprechende JSON-String erstellt und in der Datei `zfs_filesystem/file_root/uname` gespeichert (`uname` entspricht dabei dem *url-kodierten* Namen der Idee, um mit Leer- und Sonderzeichen umzugehen). Anschließend wird die Transaktion commitet. Die Transaktions-Logik aus Aufgabe 1 gewährleistet (zumindest in der Implementierung `Transaction.java`) die geforderte **Konfliktfreiheit** und **Atomizität**: Eine parallele Erfassung unterschiedlicher Ideen ist möglich (diese werden in unterschiedliche Dateien geschrieben). Kommt es jedoch zu einem Konflikt, so wird die Transaktion zurückgesetzt (siehe ).

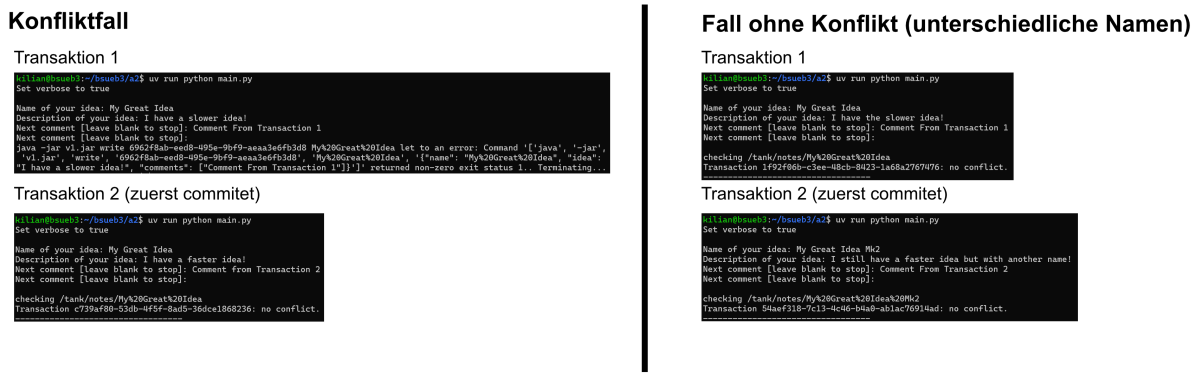



Abbildung 2: Tritt im Brainstorming-Tool der gleiche Fall auf, wie in Figure Abbildung 1 (Transaktion 1 und 2 zeitgleich gestartet, jedoch Transaktion 2 zuerst commitet), wird durch Verwendung der Transaktions-Logik aus Aufgabe 1 Konfliktfreiheit und Atomizität gewährleistet.

Auch für dieses Skript gibt es eine Konfigurations-Datei (`brainstorm_config.toml`), in der die entsprechende Executable für die Transaktions-Verwaltung angegeben werden kann.

# 3. Aufgabe 3

Da die Kommunikation über die Command Line etwa 0.5s an Overhead mit sich bringt, habe ich die Validierung für maximale Performance in Java (`Validate.java`) implementiert, sodass die Transaktions-Logik direkt zugreifbar ist. Beide Implementierungen wurden in je 2 Szenarios getestet:

1. *Only Writes*: Jeder Thread startet eine Transaktion, wählt eine zufällige Datei, beschreibt diese mit einem zufälligen Inhalt und commitet die Transaktion.
2. *Read + Write*: Jeder Thread startet eine Transaktion, wählt eine zufällige Datei und beschreibt diese zu 50% mit einem zufälligen Inhalt (in den restlichen 50% wird die Datei gelesen). Anschließend wird die Transaktion commitet.

Kommt es zu einem Konflikt, wird die Transaktion unabhängig von der Implementierung zurückgesetzt und diese iteration wiederholt, bis kein Konflikt mehr auftritt.

Die wichtigste Metrik für die Performance der beiden Implementierungen ist die durchschnittliche Extrazeit avgExtraTime. Sie berechnet sich wie folgt:

- avgExtraTime $= \frac{\text{total persist time}}{\#\text{ iterations}}$ für `Transaction.java`. total persist time misst dabei die Summe der Zeit, die `Transaction.java` extra für die Persistierung der geplanten Writes benötigt.
- avgExtraTime $= \frac{\text{total reset time}}{\#\text{ iterations}}$ für `TransactionNoBuffering.java`

**Hinweis**

Bewusst ist avgExtraTime für beide Implementierungen über die Anzahl der Iterationen, nicht etwa pro Reset gemittelt. Dies bietet einen faireren Vergleich und lässt abschätzen, wie viel Zeit die Transaktions-Logik in jeder Implementierung für iterations Iterationen benötigt, auch wenn extra Zeit tatsächlich in `Transactions.java` bei jeder Transaktion anfällt und bei `TransactionNoBuffering.java` nur bei Konflikten.

Beide Szenarios wurden mit 1-4 Threads in den untenstehenden 5 Konfigurationen getestet.

| #Verwendete Dateien | #Iterationen |
|---|---|
| 1 | 1000 |
| 1 | 500 |
| 1 | 3000 |
| 5 | 3000 |
| 10 | 3000 |

Die Experimente wurden auf einer Proxmox VM mit der folgenden Spezifikation ausgeführt:

| | |
|---|---|
| CPU | Intel i5 8400T |
| #CPU-Kerne | 4 |
| Hauptspeicher | 12GB DDR4 |
| OS | Ubuntu 24.04.1 LTS |
| zpool Konfiguration | raidz1: 3x8GB |

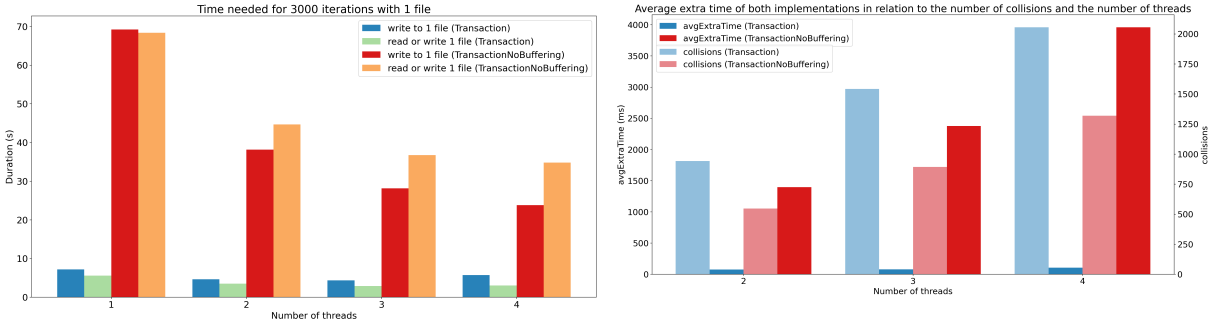



Abbildung 3: Links: Vergleich der Dauer der zwei Szenarios (only write, read/write) mit beiden Implementierungen. Rechts: Anzahl der Kollisionen und durchschnittliche Extrazeit bei 2-4 Threads.

Wie in Abbildung 3 zu sehen, verhalten sich beide Szenarios im Referenzfall mit 1 Thread sehr ähnlich. Für `Transaction.java` ist wie zu erwarten der read/write Workload etwas leichter als der write Workload, da das Lesen selbst keine Extrazeit für die Persistierung verursacht. Dass dies keine sehr große Rolle spielt (read/write ist nicht 50% schneller als write), liegt daran, dass Kollisionen und die damit verbundenen Retries deutlich teurer sind. Außerdem sieht man, dass die parallelen Operationen (zumindest mit bis zu 3 Threads) durchaus einen Performance-Vorteil bringen. Mit 4 Threads scheint der Overhead durch die gestiegenen Kollisionen jedoch den Vorteil der Parallelisierung zu überwiegen.

`TransactionNoBuffering.java` benötigt in jedem Fall deutlich länger, was der Latenz der ZFS Befehle geschuldet ist. Obwohl es durch die langsamere Ausführung zu weniger Kollisionen kommt, ist die extra Zeit durch den ZFS Overhead deutlich länger. Zwei interessante Auffälligkeiten:

1. read/write Workloads mit `TransactionNoBuffering.java` sind überraschenderweise etwas langsamer als write Workloads.
2. Die benötigte Extrazeit von `TransactionNoBuffering.java` steigt scheinbar überproportional zur Anzahl der Threads (damit Anzahl der Konflikte).

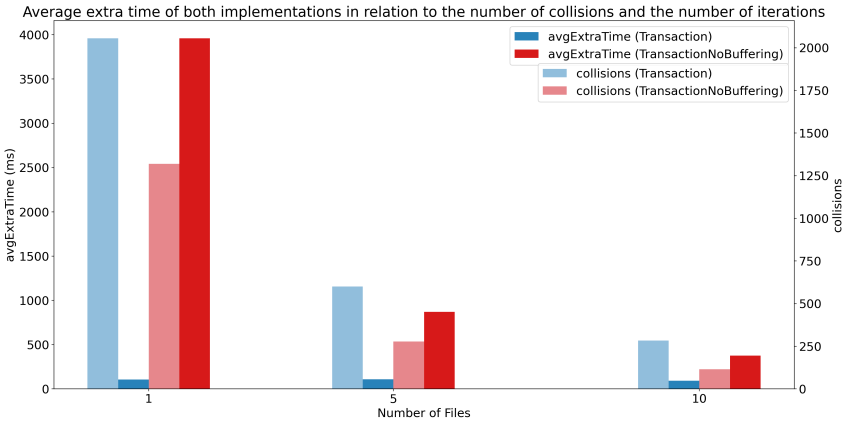



Abbildung 4: Zahl der Kollisionen (hell) und durchschnittliche Extrazeit (dunkel) in Abhängigkeit der Anzahl der verwendeten Dateien, welche 4 Threads lesend oder schreibend zugreifen.

Bezüglich der Anzahl der Dateien habe ich erwartet, dass die durchschnittliche Rücksetzzeit für `TransactionNoBuffering.java` mit der Anzahl der Dateien steigt, da ZFS so im Durchschnitt mehr Blöcke zurücksetzen muss. Wie jedoch in Abbildung 4 zu sehen, scheint dieser Aufwand vernachlässigbar und eine größere Anzahl an verwendeten Dateien verringert

das Risiko einer Kollision, womit proportional die benötigte Extrazeit beider Implementierungen abnimmt.

Insgesamt scheint `Transaction.java` die bessere Wahl zu sein, da es in allen Szenarien deutlich schneller ist als `TransactionNoBuffering.java` und auch besser die ACID-Prinzipien einhält (siehe Abschnitt 1.2). Trotzdem ist die Verwendung von ZFS in `TransactionNoBuffering.java` sehr spannend und ZFS selbst in meinen Tests angenehm schnell, obwohl sich dies durch den Overhead durch die CLI-Aufrufe der ZFS Befehle in meiner Implementierung nicht widerspiegelt. In speziellen Fällen könnte diese Variante trotzdem besser sein, wenn es sehr viele writes gibt (dadurch steigt Extrazeit für Transactions) und sehr wenige Konflikte (dadurch sinkt Extrazeit für TransactionNoBuffering). Praxistauglich sind beide Implementierungen jedoch nicht,

Praxistauglich sind beide Implementierungen in diesem Zustand realistisch gesehen nicht, da nur text-basiertes Interface zur Verfügung steht und selbst die schnellere `Transaction.java` Implementierung mit etwa 8s für 3000 sequenzielle writes einen zu hohen Overhead hat. Trotzdem sieht man, dass optimistische Transaktionen durchaus einen Performance-Vorteil gegenüber Locks, welche die Writes auf eine Datei sequenzialisieren würden, bieten können (siehe write to 1 file, Abbildung 3). Gelingt es z. B. mittels rust, schlankeren Datenstrukturen und asynchronous writes den Overhead zu reduzieren, könnte man durchaus über eine solche *Concurrency Control* nachdenken.