Logging und Recovery

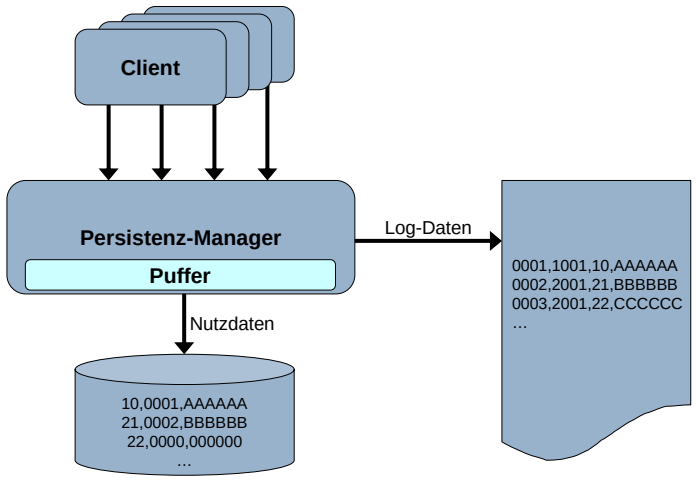
Übungen zu DIS, Sommersemester 2009
Marc Holze

Aufgabenstellung

- Realisierung eines vereinfachten „Datenbanksystems“
- Mehrere Clients greifen auf einen Persistenz-Manager zu

- Logging im Normalbetrieb
  - Non-atomic: direktes Überschreiben der Nutzdaten
  - No-steal: keine Sicherung der Rücksetzbarkeit notwendig
  - No-force: verzögertes Ausschreiben der Nutzdaten
- Crash-Recovery
  - Nur Redo-Recovery
  - Wiederholung verloren gegangener Änderungen

Architektur



Persistenz-Manager

- Persistierung von Seiten
- Struktur der Seiten: [PageID, LSN, Data]
  - PageID: Kennung der Seite
  - LSN: Log Sequence Number
  - Data: Nutzdaten
- Puffer: verzögertes Ausschreiben
  - Ausschreiben der Daten **beendeter** Transaktionen, wenn mehr als fünf Seiten im Puffer sind
  - Prüfung auf „vollen“ Puffer nach Schreibzugriffen (unabhängig von Transaktionsgrenzen)
  - Aktualisierung von Seiten im Puffer möglich!
  - Kein Ausschreiben schmutziger Seiten (No-Steal)!
- Sofortiges Logging von Änderungsoperationen (Commit-Regel!)
- Anforderung: Unterstützung von Crash-Recovery

Persistenz-Manager: Logging

- Physisches Zustands-Logging
- Log-Granulat: Seite
- neue Zustände (After-Images) geänderter Objekte werden in die Log-Datei geschrieben
- Satzarten
  - BOT- und Commit-Satz
  - Änderungssatz (After-Images)
- Struktur der Log-Einträge: [LSN, TAID, PageID, Redo]
  - LSN: Log Sequence Number (monoton aufsteigend)
  - TAID: Transaktionskennung
  - PageID: Kennung der betroffenen Seite
  - Redo: gibt an, wie die Änderung nachvollzogen werden kann

Clients

- 5 gleichartige Clients
  - Parallel laufende Threads mit ClientIDs 1..5
- Zugriff auf Persistenz-Manager
  - Persistenz-Manager als Singleton: eine Instanz
  - Jeder Client greift schreibend auf Seiten zu
    `beginTransaction()… write()… write()… … commit()`
  - Keine konkurrierenden Zugriffe → keine Sperren notwendig (Client 1: Seiten 10..19, Client 2: Seiten 20..29 etc.)
  - TAID wird vom Persistenz-Manager vergeben

Crash-Recovery

- Ein Block der permanenten DB ist entweder
  - aktuell
  - oder veraltet (noforce) → Redo
- Analyse-Phase
  - Vorwärtslesen des Log
  - Bestimmung von Gewinner-TA
- Redo-Phase
  - Vorwärtslesen des Log
  - Selektives Redo (redo winners)
- Protokollierung des Recovery-Fortschritts durch Eintragen der LSNs durchgeführter Redo-Schritte in die jeweiligen Seiten

Crash-Recovery

- Redo ist nur erforderlich, wenn
  Seiten-LSN < LSN des Redo-Log-Satzes
- Seiten-LSN wird bei Redo aktualisiert (wächst monoton)

```
if LSN(LS)  >  LSN(Page) then
  redo (Änderung aus LS);
  LSN(Page) := LSN(LS);
end;
```

Thread-Safe Singleton

```
class Singleton {

    static final private Singleton singleton;
    static {
        try {
            singleton = new Singleton();
        }
        catch (Throwable e) {
            throw new RuntimeException(e.getMessage());
        }
    }

    private Singleton() {}

    static public Singleton getInstance() {
        return singleton;
    }
}
```


Erzeugung von Threads (1/2)

```
public class HelloThread extends Thread {
    String name;

    public HelloThread(String name){
        this.name = name;
    }

    public void run() {
        while(true){
            System.out.println("Hello from " + name);
            try{
                Thread.sleep(2000);
            }catch(InterruptedException e){
                return;
            }
        }
    }
}
```

Erzeugung von Threads (2/2)

```
public class ThreadCreator {

	public static void main(String[] args) {
		Thread t1 = new HelloThread("Thread1");
		Thread t2 = new HelloThread("Thread2");

		t1.start();
		t2.start();

		try {
			Thread.sleep(10000);
		} catch (InterruptedException e) {
		}

		t1.interrupt();
		t2.interrupt();
	}
}
```


Gegenseitiger Ausschluss

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```