

7 Transaktionen

7.1 Das Konzept einer Transaktion

Eine Transaktion ist die abstrakte Sicht des DBMSs über ein Benutzerprogramm: eine Sequenz von Lese- und Schreibbefehle.

- Lesen: Datenbank-Objekte (Lese- und Schreibeinheiten) sind aus dem Speicher geladen, und Werte werden in Programm-Variablen kopiert
- Schreiben: das im Hauptspeicher modifizierte Objekt (d.h. Dateneinheit) wird auf die Disk geschrieben

Gegenüber gleichzeitiger Ausführung und Systemabbrüchen müssen einige Eigenschaften der Transaktionen sichergestellt werden

7.2 ACID Eigenschaften

ACID bedeutet: Atomicity, Consistency, Isolation, Durability, auf Deutsch: Unteilbarkeit, Konsistenz, Isolation, Dauerhaftigkeit

- Unteilbarkeit: Benutzer sollen jede Transaktion als ungeteilt betrachten können: entweder sind alle oder keine Handlungen ausgeführt
- Konsistenz: Jede Transaktion, allein ausgeführt, soll die Konsistenz der Datenbank erhalten (unter Verantwortung des Benutzers)
- Isolation: Benutzer sollen eine Transaktion ohne die Wirkung von gleichzeitiger Behandlung verstehen können
- Dauerhaftigkeit: Auch falls ein Systemabbruch stattfindet, soll eine ausgeführte Transaktion anhalten [persist]

7.3 Transaktionen

Die gleichzeitige Ausführung von Benutzerprogrammen ist für gute DBMS Performanz absolut nötig.

Weil Diskzugriffe *häufig* und relativ *langsam* sind, ist es wichtig, dass die CPU mit mehreren Prozessen beschäftigt wird, d.h. mehrere Benutzer können gleichzeitig bedient werden

Ein Benutzerprogramm kann viele Operationen auf Daten, die aus der Datenbank herausgeholt worden sind, durchführen. Der DBMS aber ist nur daran interessiert, was in der Datenbank geschrieben wird und aus der Datenbank gelesen wird

7.4 Gleichzeitige Ausführung im DBMS

Ein Benutzer oder eine Benutzerin gibt Transaktionen ab; dabei kann er oder sie von jeder Transaktion denken, dass sie allein ausgeführt wird.

Die gleichzeitige Behandlung ist vom DBMS kontrolliert. Das System mischt die Handlungen (d.h. Lese- und Schreibbefehle von DB „Objekte“) von verschiedenen Transaktionen

Jede Transaktion muss die Datenbank in einem konsistenten Zustand lassen, falls sie schon vor der Transaktion in einem konsistenten Zustand war.

Das DBMS stellt einige Integritäts-Einschränkungen [integrity constraints] sicher, in Abhängigkeit von welchen Einschränkungen im **CREATE TABLE** Befehl deklariert worden sind. Das DBMS versteht aber nicht die Semantik der Daten (z.B. es versteht nicht wie der Zins auf einem Bankkonto berechnet wird).

Wegen der gleichzeitiger Ausführung von Transaktionen können zwei Arten von Problemen auftreten:

- Effekte, die von der *Interleaving* [Abwechslung] von Transaktionen verursacht werden;
- Effekte, die bei *crashes* [Abstürze] vorkommen können.

Mit „Problem“ wird hier gemeint, dass es Situationen vorkommen können, wo die ACID – Eigenschaften nicht mehr gewährleistet sind.

7.5 Unteilbarkeit von Transaktionen

Eine Transaktion kann entweder bestätigt werden [*commit*], nachdem alle Handlungen ausgeführt worden sind, oder sie kann abbrechen [*abort*] (oder vom DBMS abgebrochen werden), nachdem einige Aktionen ausgeführt worden sind. halb-durchgeführte Transaktionen sollen – auch nach einem Crash – nicht zugelassen werden.

Die Unteilbarkeit wird vom DBMS für alle Transaktionen garantiert. Das bedeutet, dass ein Benutzer eine Transaktion immer so sehen kann, wie wenn entweder alle Handlungen in einmal ausgeführt wurden, oder keine Handlungen überhaupt ausgeführt wurden.

Das DBMS schreibt alle Handlungen in ein Logbuch, sodass sie rückgängig gemacht werden könnten, wenn eine Transaktion abbrechen muss.

7.5.1 Beispiel

Wir betrachten zwei Transaktionen:

T1: **BEGIN A=A+100, B=B-100 END**

T2: **BEGIN A=1.06*A, B=1.06*B END**

Intuitiv: die erste Transaktion transferiert 100 Franken vom Konto B auf Konto A. Die zweite Transaktion gibt 6% Zins auf beide Konten

Es gibt keine Garantie, dass T1 vor T2, oder umgekehrt, ausgeführt wird, falls beide gleichzeitig gestartet worden sind. Aber die Wirkung auf der Datenbank von der gleichzeitigen Ausführung beider Transaktionen muss gleich sein, wie wenn die zwei Transaktionen nacheinander (Seriell) ausgeführt worden wären, also T1, und dann T2, sowie T2, und dann T1, sind erlaubte Resultate.

Im Beispiel gibt es aber eine Situation, wo Zahlen auskommen können, die keinem „seriellen“ Szenario entspricht. Man betrachtet ein möglicher Zeitplan [schedule] einer abwechselnden Ausführung [interleaving] der Transaktionen:

T1: **BEGIN A=A+100, B=B-100 END**

T2: **BEGIN A=1.06*A, B=1.06*B END**

kann nicht zugelassen werden.

Das DBMS sieht den zweiten Zeitplan wie folgt:

T1: R(A),W(A), R(B),W(B)
 T2: R(A),W(A), R(B),W(B)

7.6 Zeitliche Planung von Transaktionen

–Serieller Zeitplan [serial schedule]: Zeitplan, wo die Handlungen von verschiedenen Transaktionen nicht abwechseln

–Gleichwertige Zeitpläne [equivalent schedules]: Für irgendein Datenbankzustand, die Auswirkung der Ausführung des ersten Zeitplans auf die Menge von Objekten in der Datenbank ist gleich wie die Auswirkung der Ausführung des zweiten Zeitplans.

–Serialisierbarer Zeitplan [serializable schedule]: Ein Zeitplan, der einer seriellen Ausführung von Transaktionen gleichwertig ist. (Bemerkung: falls jede Transaktion die Konsistenz des DBMSs erhält, wird jeder serialisierbare Zeitplan die Konsistenz auch erhalten.)

7.7 Anomalien bei abwechselnder Ausführung

Drei Arten von Anomalien können auftreten. Sie sind alle mit dem *schreiben* von Datensätze verbunden.

Unbestätigte [uncommitted] Daten lesen: (WR Konflikte, “schmutziger Lesebefehl [dirty read]”):

T1: R(A), W(A), R(B), W(B), Abort
 T2: R(A), W(A), Commit

Unwiederholbares Lesen (RW Konflikte):

T1: R(A), R(A), W(A), Commit
 T2: R(A), W(A), Commit

Überschreibung von unbestätigten Daten (WW Konflikte):

T1: W(A), W(B), Commit
 T2: W(A), W(B), Commit

Diese Konflikte müssen verhindert werden. Die Grundidee bei der Verhinderung ist, dass das DBMS nur serialisierbare Zeitpläne zulässt – und die nichtserialisierbare Zeitpläne so umändert, dass sie in serialisierbare geführt werden.

7.8 Schloss-Basierte [lock-based] Steuerung der gleichzeitigen Ausführung

Striktes zwei-phasen Abschlusprotokoll [Strict Two-phase Locking (Strict 2PL) Protocol]:

–Jede Transaktion muss ein geteiltes [shared] Schloss [S lock] auf einen Objekt bekommen, bevor sie es lesen kann

–Jede Transaktion muss ein ungeteiltes [exclusive] Schloss [X lock] auf einen Objekt bekommen, bevor sie es schreiben kann

–Alle Schlösser, die von einer Transaktion gebraucht werden, werden losgelassen [released], wenn die Transaktion am Ende ist.

–Falls eine Transaktion ein X-Lock auf ein Objekt besitzt, kann keine andere Transaktion (weder S noch X) ein X-Lock auf dem gleichen Objekt bekommen.

Strickt 2PL lässt nur serialisierbare Zeitpläne zu

7.9 Transaktionen abbrechen

Falls eine Transaktion T_i abbricht, müssen alle schon ausgeführten Handlungen von T_i rückgängig gemacht werden.

Nicht nur das: Falls eine Transaktion T_j ein Objekt gelesen hat, das von T_i ebenso geschrieben worden ist, muss auch T_j abgebrochen werden!

Die meisten Systeme erlauben sogenannte lawinenartige [cascade] Abbrüche, wobei die Schlösser erst bei Commit-Zeit freigegeben werden. Das heisst also, dass falls T_i ein Objekt schreibt, kann T_j das Objekt erst nach dem Commit von T_i lesen.

Um Handlungen von einer abgebrochenen Transaktion rückgängig zu machen, muss der DBMS ein Logbuch benutzen, wo alle Schreibebefehle aufgeschrieben sind. Dieser Mechanismus wird auch benutzt, um die Effekte von Abstürzen zu beheben [recover]: alle aktiven Transaktionen zur Absturzzeit werden rückgängig gemacht, wenn das System neu gestartet wird.

7.10 Lock Management

–Bedürfnisse für locking und unlocking werden vom Lock Manager verwaltet

–Einträge in der Lock-Tabelle:

- Anzahl Transaktion, die ein Lock haben
- Typ von lock
- Zeiger zur Queue von Lock Requests

–Locking und Unlocking müssen atomare (unteilbare) Operationen sein

–Lock Upgrade: Transaktionen, die ein geteiltes (S) Lock besitzen, können um ein ungeteiltes (X) Lock anfragen

7.11 Deadlocks

Deadlock: Zyklus von Transaktionen, die auf Freigabe von Locks warten, die zu Transaktionen gehören, die auf Freigabe von Locks warten, die zu den ersten Transaktionen gehören.

Zwei Arten, um mit Deadlocks umzugehen:

- Prävention
- Entdecken

7.11.1 Deadlock Prävention

Man setzt Prioritäten durch Zeitstempel [timestamps].

Man betrachte die Situation, wo T_i ein Lock will, das im Moment von T_j benutzt wird. Es gibt zwei Möglichkeiten, mit dem umzugehen:

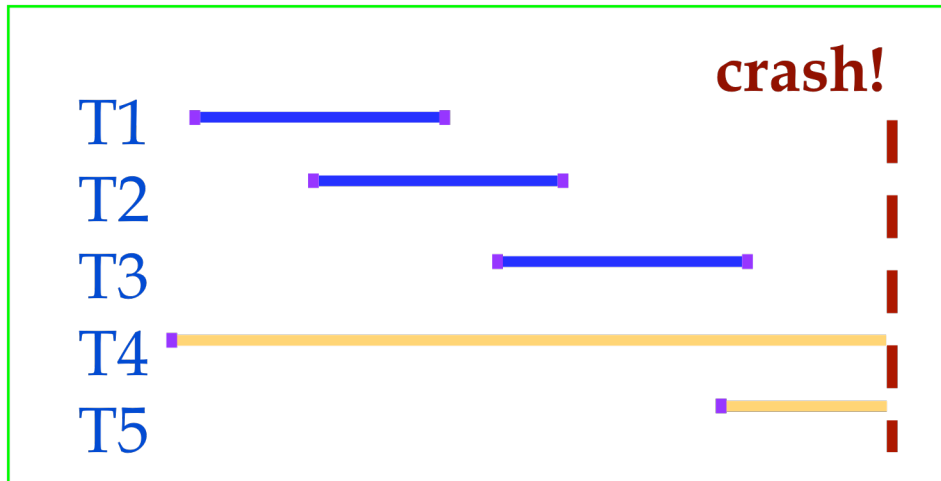
1. Warte-Sterbe [wait-die]: Falls T_i höhere Priorität hat, wartet T_i auf T_j , sonst bricht T_i ab
2. Verletze-Warte [wound-wait]: Falls T_i eine höhere Priorität hat, bricht T_j ab, sonst wartet T_i

Wenn eine Transaktion erneut startet, behält sie ihre ursprüngliche Zeitmarke

7.12 Absturzbehebung [crash recovery]: Motivation

Unteilbarkeit: Transaktionen können abbrechen [abort] ("Rollback").

Dauerhaftigkeit: Was passiert, wenn der DBMS aufhört zu laufen?



Erwünscht nach dem Absturz:

- T1, T2 & T3 sollen dauerhaft sein.
- T4 & T5 sollen abgebrochen werden (Auswirkung wird nicht gemerkt).

7.13 Verwaltung von Diskplatz

Die tiefste Schicht des DBMSs Software verwaltet den Diskplatz. Die höheren Schichten benutzen diese Schicht um:

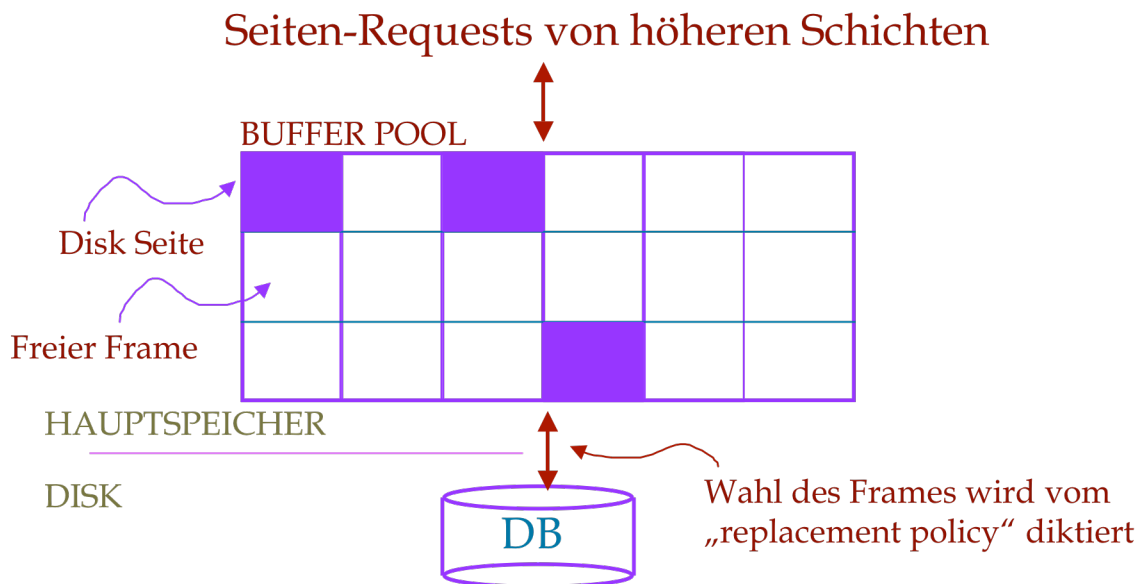
- Seiten zu allozieren oder deallozieren
- Seiten zu lesen und zu schreiben.

Beispiel für die Verwaltung:

Falls eine Sequenz von Seiten gefragt wird, müssen diese Seiten auf die Disk auch sequenziell gespeichert sein. Die höheren Schichten des DBMS sollen sich nicht kümmern, wie dies gemacht wird.

Ein Buffer Manager wird für solche Aufgaben eingesetzt. Das Hauptelement des Buffer Managers ist der Buffer Pool.

7.13.1 Buffer Pool



Hauptkonzepte sind die Disk-Seite und die Frames. Ein Frame ist ein Speicherplatzreservation im Hauptspeicher, die genau eine Disk-Seite enthalten kann.

Die Daten müssen im Hauptspeicher transferiert werden, bevor die höhere Schichten des DBMS sie benutzen können.

Dafür wird eine Tabelle mit Datensätze <frame#, pageid> angewendet.

Ein Requestor von einer Seite muss sie nach Benutzung unmarkieren [unpin], und sagen, ob sie modifiziert worden ist oder nicht. Dafür sind **dirty bits** benutzt

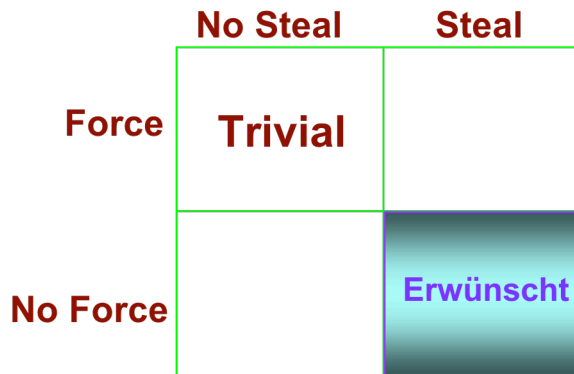
Eine Seite im Pool kann mehrmals abgefragt werden. Ein **pin count** wird benutzt. Eine Seite ist ein Kandidat für Ersatz falls pin count = 0.

CC und recovery können weitere I/Os benötigen (z. B. Write Ahead Logging, siehe nachher).

7.13.2 Verwaltung vom Buffer Pool

Werden jede Schreibeoperation W auf die Disk **gezwungen**, ist die Dauerhaftigkeit garantiert, aber es entsteht schlechte Antwortzeiten. Mit zwingen wird gemeint, dass jede Write-operation sofort die Daten auf die Disk geschrieben werden.

–Gegenüber dem Konzept von zwingen ist das Konzept von **stehlen [steal]**. Mit stehlen wird gemeint, dass Seiten können aus dem Buffer Pool auf die Disk geschrieben, **auch wenn die Transaktion (noch) unbestätigt ist**. Das kann nötig sein, um Platz im Buffer Pool zu schaffen. Falls stehlen nicht erlaubt wird, wird das System uneffizient (throughput). Falls stehlen erlaubt wird, wie kann die Unteilbarkeit garantiert werden?



Figur 8: Steal und No-Force

STEAL (oder: wieso die Unteilbarkeit schwierig zu gewährleisten ist):

Um ein Frame F zu stehlen: Die laufende Seite in F (sagen wir S) ist auf die Disk geschrieben; irgendeine Transaktion hält ein Lock auf S.

- Was passiert, falls die Transaktion mit dem Lock auf S abbricht?
- **UNDO**: Man muss den alten Wert von S zur Zeit wo der Frame gestohlen worden ist, irgendwo behalten, um den Write rückgängig machen zu können.

NO FORCE (oder: wieso die Dauerhaftigkeit schwierig zu gewährleisten ist) :

- Was passiert, falls das System abstürzt, bevor eine Seite auf die Disk geschrieben ist?
- **REDO**: Man schreibt die Information (so wenig wie möglich), auf einen sicheren Platz, bei Bestätigungszeit, um die Modifikationen wieder ausführen zu können.

7.14 ARIES Algorithmus



Es gibt drei Phasen im ARIES Absturzbehebungs-Algorithmus:

Analysis: Man geht durch das Logbuch nach vorne, und notiert

- welche Transaktionen aktiv waren,

- welche Seiten im Buffer Pool dirty waren, als das System abgestürzt ist.

Die Analyse startet an einem Punkt, der Checkpoint genannt ist. Ein checkpoint ist ein „snapshot“ des Datenbank-Zustandes zu einem bestimmten Zeitpunkt. Checkpointing wird periodisch durchgeführt.

Redo: Man macht alle Änderungen auf dirty Seiten im Buffer Pool erneut, um sicher zu sein, dass alle aufgeschriebenen Änderungen tatsächlich umgesetzt worden sind und auf die Disk geschrieben worden sind.

Undo: Man macht alle Schreibebefehle für alle aktive Transaktionen rückgängig (wobei die ursprüngliche Werte aus dem Logbuch geholt werden). Undo fängt am Ende des Logbuches an.

7.14.1 Logbuch führen

Man schreibt in dem Logbuch die Information über Modifikationen, die REDO und UNDO Operationen (beim Restart) erfordern.

Folgende Einträge sind im Logbuch vorhanden:

-Falls Ti ein Objekt schreibt : der alte und der neue Wert. Der Eintrag muss auf die Disk geschrieben werden, bevor die Seite geändert wird!

-Falls Ti bestätigt [commit] oder abgebrochen wird: Ein Eintrag im Logbuch erwähnt diese Handlung.

Die Einträge im Logbuch sind mit der Transaktions-ID assoziiert, sodass es einfach ist, eine bestimmte Transaktion rückgängig zu machen. (auch z.B. um einen deadlock zu beheben).

Der Log ist oft dupliziert und auf einem „stabilen“ Medium gespeichert. Das Logbuch sollte nicht auf der gleichen Disk wie die Daten sein. Das Schreiben ins Logbuch erfolgt sequentiell.

Alle Log-verbundenen Handlungen (und eigentlich alle CC-bezogene Handlungen wie Schloss geben, Schloss befreien, Leerläufe beheben usw.) sind vom DBMS transparent behandelt.

Minimale Information muss geschrieben werden, sodass mehrere Modifikationen auf einer einzigen Log-Seite Platz haben

Das Logbuch kann auch als eine geordnete Liste von REDO und UNDO Aktionen gesehen werden.

Jeder Log Eintrag hat ein eindeutiges Log Sequence Number (LSN). Der Log Sequence Number ist eine Zahl die stetig wächst für jeden Log-Eintrag. Es erlaubt, den Einträge im Logbuch eindeutig zu identifizieren. Das heißt, jeder Log Eintrag hat ein eindeutiges Log Sequence Number (LSN).

Zusätzlich zur Log erhält Jede Datenseite einen pageLSN. Das ist der LSN vom letzten log Eintrag für eine Modifikation auf dieser Seite

Ein Update Logbuch-Eintrag enthält, unter anderen, folgende Attribute:

<LSN, prevLSN, XID, pageID, offset, length, old data, new data>

wobei XID = Transaktions-ID (eine Zahl, die die Transaktion eindeutig identifiziert), pageID = die ID der Seite (auch pro Seite eindeutig), offset = Anzahl Bytes von Anfang der Seite, length = Anzahl Bytes, die auf der Seite zu modifizieren sind, old data = der Wert nach der Modifikation, auch before-image genannt, und new data = der neue Wert, auch after-image genannt.

Mögliche Typen von log Einträge:

- Update
- Commit
- Abort
- End (bedeutet Ende von Bestätigung oder Abbruch)
- Compensation Log Records (CLRs) für UNDO

Andere Log-bezogene Elemente

Transaction Table: Eine Tabelle, die es erlauben, die aktiven Transaktionen zu nachfolgen. Die Transaction Table enthält einen Eintrag für jede aktive Transaktion. Ein Eintrag enthält:

XID, Status (running/committed/aborted), und lastLSN.

Der LastLSN zeigt auf dem letzten Log-Eintrag, der von dem entsprechenden Transaction assoziiert ist.

Dirty Page Table:

Es gibt einen Eintrag für jede Dirty Page im Buffer Pool. Jeden Eintrag enthält, neben der pageID, einen recLSN -- der LSN vom Log-Eintrag, der zuerst die Seite dirty gemacht hat.

[Beispiel]

7.14.2 Write-Ahead Logging (WAL) [Vorwärtsschreiben]

Der Write-Ahead Logging Protokoll:

1. Der Log-Eintrag für eine Modifikation wird gezwungen geschrieben, bevor die entsprechende Datenseite auf der Disk geschrieben wird
2. Alle Log-Einträge müssen im Logbuch geschrieben werden, bevor eine Transaktion bestätigt ist.

-#1 stellt Unteilbarkeit sicher.

-#2 stellt Dauerhaftigkeit sicher.

7.14.3 Beispiel

Zeichnen Sie das Logbuch mit den folgenden Elemente:

Transaktion T1000 ändert Bytes 21 - 23 auf Seite P500 von „ABC“ auf „DEF“

Transaktion T2000 ändert Bytes 41 - 43 auf Seite P600 von „HIJ“ auf „KLM“

Transaktion T2000 ändert Bytes 20 - 22 auf Seite P500 von „GDE“ auf „QRS“

Transaktion T1000 ändert Bytes 21 - 23 auf Seite P505 von „TUV“ auf „XYZ“

7.14.4 Checkpointing

Periodisch generiert der DBMS einen Checkpoint, um die Zeit, die für die Erhebung nach einem Absturz nötig ist, zu minimieren.

Geschrieben im Logbuch werden folgende Einträge

- begin_checkpoint: Wann das Checkpoint begonnen hat.

- end_checkpoint: Die laufende Transaktionstabelle und die Dirty Page Table. Das ist ein so genannten „fuzzy checkpoint“:

Andere Transaktionen laufen weiter, d.h. diese Tabellen sind nur bei der Zeit vom begin_checkpoint aktuell.

Es gibt keinen Versuch, schmutzige Seiten auf Disk zu zwingen; deshalb ist die Effektivität des Checkpoints mit dem ältesten ungezwungene Modifikation in einer schmutzigen Seite begrenzt. Deshalb ist es gut, periodisch schmutzige Seiten zu forcieren.

Man speichert das LSN vom Checkpoint an einem sicheren Platz (*master record*).

7.14.5 Transaktion Bestätigen

- Das Bestätigungs-Log-Eintrag wird im Logbuch geschrieben
- Alle Log-Einträge bis die Transaktions-lastLSN werden auf Disk geschrieben
- Es gibt mehrere Log-Einträge per Log-Seite.
- Bestätigung() endet
- Man schreibt einen „End“ Log-Eintrag

7.14.6 The Big Picture: Was ist wo gespeichert

ARIES: Analysis Phase

Siehe Beispiel von vorher, **plus im Log Tail:**

- Transaktion T2000 bestätigt
- Transaktion T1000 ändert Bytes 21 - 23 auf Seite P700 von „ABC“ auf „DEF“
- Das System stürzt ab, d.h. DPT und TT sind verloren
- Früheste Checkpoint: am Anfang

Aries: REDO Phase

- Man betrachte weiter unseren Beispiel von vorher

Aries: UNDO Phase

- Gleich wie für einen expliziten Abbruch der Transaktion
- UNDO: Wir wollen das Logbuch vom Ende aus lesen, d.h. wir starten mit dem letzten Eintrag, und machen alle Modifikationen rückgängig:
- Man nimmt der lastLSN von einer Transaktion in der Transaktionstabelle
- Man folgt die Kette von Log-Einträge rückwärts mit dem prevLSN Feld
- Bevor wir mit dem UNDO starten, schreiben wir ein Abort Log-Eintrag, um uns von einem Abbruch während der UNDO zu retten
- Um ein UNDO zu ausführen, müssen wir ein lock auf die Daten haben (Kein Problem!)
- Bevor wir die alte Werte auf einer Seite wieder schreiben können, schreiben wir ein Compensation Log Record CLR
- Wir loggen während dem UNDO weiter
- CLR hat ein extra Feld: undonextLSN, zeigt auf dem nächsten LSN zu UNDOen (d.h. zu prevLSN vom Rekord, dass wir ebenso UNDOen). CLR

brauchen selbst nie rückgängig gemacht zu werden, aber sie können erneut gemacht werden (REDO), mit repeating history: Unteilbarkeit ist so gewährleistet

- Nach dem UNDO, man schreibt ein „End“ Log-Eintrag