



Datenbanken und Informationssysteme

7. Transaktionsverwaltung

Prof. Dr. Stefan Decker

📍 RWTH Aachen



7. Transaktionsverwaltung

Korrekte Daten trotz Nebenläufigkeit und Fehlern

Von der Transaktion als Arbeitseinheit bis zu Sperrprotokollen und Recovery

🗺 Kapitelfahrplan

- 1 **Transaktionen und Nebenläufigkeit**
- 2 Schedules und Serialisierbarkeit
- 3 Konfliktserialisierbarkeit (CSR)
- 4 Fehlersicherheit und Recoverability

- 5 **Scheduler** und Sperrprotokolle
- 6 **2PL-Korrektheit und Varianten**
- 7 Recovery-Grundlagen

🎯 **Leitidee:** Transaktionsverwaltung sorgt dafür, dass Transaktionen auch bei parallelen Zugriffen und technischen Fehlern geordnet, vollständig und dauerhaft wirken.

1. Transaktionen

Grundlagen und Probleme der Nebenläufigkeit

Leitfrage: Wie macht ein DBMS aus mehreren Lese- und Schreiboperationen eine verlässliche Arbeitseinheit?

Fokus dieses Abschnitts

Wir klären Begriff, Ziel und Ende einer Transaktion; danach wird sichtbar, warum parallele Ausführung ohne Isolation zu Anomalien führt.

Kapitelpfad

1. **Transaktionen und Nebenläufigkeit**
2. Schedules und Serialisierbarkeit
3. Konfliktserialisierbarkeit (CSR)
4. Fehlersicherheit und Recoverability
5. Scheduler und Sperrprotokolle
6. 2PL-Korrektheit und Varianten
7. Recovery-Grundlagen

≡ Fahrplan für den Abschnitt

- ▶ Welche Aufgabe löst Transaktionsmanagement im DBMS?
- ▶ Was ist eine Transaktion? Konzept und Beispiel
- ▶ Das Fundament: die ACID-Prinzipien
- ▶ Commit und Rollback: zwei saubere Enden
- ▶ Read/Write/Commit/Abort-Modell für die Analyse
- ▶ Welche Anomalien entstehen bei Nebenläufigkeit?

Didaktischer Pfad: Wir starten mit Motivation und Begriffen, ordnen die Qualitätsziele als ACID ein und abstrahieren anschließend auf ein formales Modell, das später für Schedules und Konflikte gebraucht wird.

Die Rolle des Transaktionsmanagements

Ein DBMS muss die Korrektheit der Daten auch dann bewahren, wenn viele Transaktionen gleichzeitig laufen oder ein Fehler mitten in der Arbeit passiert.

Synchronisation

Concurrency Control ordnet parallele Transaktionen so, dass sie sich logisch nicht gegenseitig zerstören.

- ▶ Konfliktierende Operationen müssen kontrolliert werden.
- ▶ Nebenläufigkeit soll trotzdem hohe Auslastung ermöglichen.

Fehlertoleranz

Recovery stellt nach Fehlern wieder einen zulässigen Datenbankzustand her.

- ▶ Abstürze dürfen keine halbfertigen Effekte hinterlassen.
- ▶ Bereits bestätigte Ergebnisse müssen erhalten bleiben.

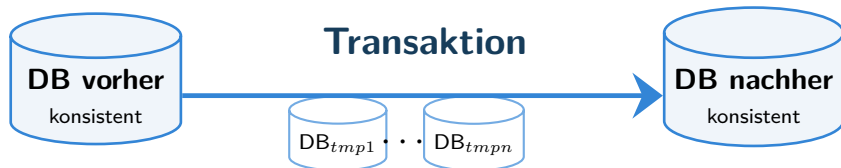
Fokus zunächst: Wir betrachten Synchronisation und die Probleme verzahnter Transaktionen.
Recovery folgt später als eigener Themenblock.

Transaktion = logische Arbeitseinheit

Definition in Worten

Eine **Transaktion** ist eine Folge von Datenbankoperationen, die vom DBMS als eine logische Arbeitseinheit behandelt wird. Bei Erfolg führt sie die Datenbank von einem konsistenten Zustand in den nächsten.

Kernidee: Nach außen zählt nicht der einzelne Zwischenschritt, sondern die ganze *unit of work*: Sie gelingt oder wird verworfen.



Beispiel: Banküberweisung braucht Atomicity

Szenario: 200 EUR werden von Huber (1500 EUR) an Meier (1000 EUR) überwiesen. Die Gesamtsumme muss 2500 EUR bleiben.

Schritt 1

```
UPDATE Konto  
SET Stand = Stand - 200  
WHERE Kunde = 'Huber';
```

Zwischenzustand: Huber = 1300, Meier = 1000,
Summe = 2300 **inkonsistent**.

Schritt 2

```
UPDATE Konto  
SET Stand = Stand + 200  
WHERE Kunde = 'Meier';
```

Endzustand: Huber = 1300, Meier = 1200, Summe
= 2500 **konsistent**.

Vorher

Kunde	Stand
Meier	1000
Huber	1500



Nach Schritt 1

Kunde	Stand
Meier	1000
Huber	1300



Absturz vor Schritt 2

Schlussfolgerung: Die komplette Sequenz muss atomar wirken. Ein Fehler zwischen Schritt 1 und 2 darf die Invariante nicht verletzen.

Das Fundament: ACID-Eigenschaften

A - Atomicity

„**Alles oder nichts**“: Eine Transaktion wird vollständig wirksam oder komplett verworfen.

I - Isolation

Nebenläufige Transaktionen beeinflussen sich logisch nicht. Jede Transaktion erscheint, als lief sie allein im System.

C - Consistency

Eine korrekt programmierte Transaktion führt die Datenbank von einem konsistenten in einen wieder konsistenten Zustand.

D - Durability

Die Effekte einer mit **COMMIT** bestätigten Transaktion bleiben dauerhaft gespeichert und überstehen Systemfehler.

Merke: ACID beschreibt das Sollverhalten einer einzelnen Transaktion. Die eigentliche Herausforderung ist, dieses Versprechen unter Nebenläufigkeit und bei Fehlern technisch einzuhalten.

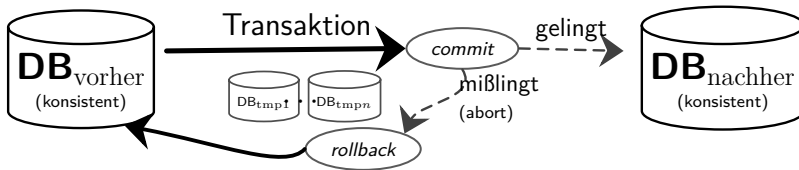
Ende einer Transaktion: Erfolg oder Fehlschlag

✓ COMMIT

- ▶ COMMIT bestätigt eine erfolgreiche Transaktion.
- ▶ Alle Änderungen werden dauerhaft gespeichert.
- ▶ Danach liegt ein neuer konsistenter Zustand vor.
- ▶ Das DBMS akzeptiert COMMIT nur bei erfüllten Bedingungen.

↶ ROLLBACK / ABORT

- ▶ Geplanter oder ungeplanter Abbruch.
- ▶ Seit Beginn erzeugte Effekte werden vollständig rückgängig gemacht.
- ▶ Die Datenbank kehrt in den vorherigen konsistenten Zustand zurück.



Analysemodell: Read/Write/Commit/Abort

Für die formale Analyse von Nebenläufigkeit und Recovery modellieren wir Transaktionen $T_1, T_2, \dots$ als Folgen atomarer Operationen auf Objekten $x, y, z, \dots$.

Bausteine des Modells

Operationen

- ▶ $r_i(x)$: Transaktion T_i liest Objekt x .
- ▶ $w_i(x)$: Transaktion T_i schreibt Objekt x .

Abschlussaktionen

- ▶ c_i : Transaktion T_i bestätigt erfolgreich.
- ▶ a_i : Transaktion T_i bricht ab und rollt zurück.

Beispieltransaktion

Beispiel

$T_1 = r_1(y) \ w_1(y) \ r_1(x) \ w_1(x) \ c_1$

Interne Berechnungen werden ausgeblendet; sichtbar bleiben nur die für Konflikte, Schedules und Recovery relevanten DB-Ereignisse.

Zweck: Wenn Transaktionen so beschrieben sind, können wir ihre Verzahnung später als *Schedule* analysieren.

Zielkonflikt: Nebenläufigkeit vs. Isolation

Isolation

- ▶ Jede Transaktion soll wirken, als wäre sie allein im System.
- ▶ Sie sieht keine halbfertigen Änderungen anderer Transaktionen.

Nebenläufigkeit

- ▶ Das DBMS verzahnt Operationen mehrerer Transaktionen.
- ▶ Dadurch steigen Durchsatz und Ressourcenauslastung.

⚠ Herausforderung: Nicht jede Verzahnung ist korrekt; verletzte Isolation erzeugt Anomalien.

❓ Frage: Wenn nur Korrektheit zählt: Wie führt man mehrere Transaktionen am einfachsten aus? Was kostet das?

Beispieldatenbank für Anomalien

Die nächsten Anomalien verwenden eine kleine Flugdatenbank. Für jeden Flug soll *AnzPass* der Anzahl der zugehörigen Passagierzeilen entsprechen.

Fluginfo	
Flug#	AnzPass
LH745	3
BA932	3

Passagiere		
Flug#	Name	Platz
LH745	Müller	3A
LH745	Meier	6D
LH745	Huber	5C
BA932	Schmidt	9F
BA932	Huber	5C
BA932	Wagner	2B

Invariante: *Fluginfo.AnzPass* und die Anzahl der Passagierzeilen pro Flug müssen zusammenpassen.

Anomalie 1: Lost Update

Szenario: BA932 hat $AnzPass = 3$. T_1 und T_2 buchen je einen zusätzlichen Passagier. Korrekt wäre am Ende also 5.

T_1	T_2
$r_1(AnzPass)$; lese $a = 3$	
	$r_2(AnzPass)$; lese $b = 3$
	$b := b + 1$; also $b = 4$
	$w_2(AnzPass)$; schreibe $b = 4$
$a := a + 1$; also $a = 4$	
$w_1(AnzPass)$; schreibe $a = 4$	

Ergebnis: Obwohl zwei Buchungen stattfinden, steht am Ende nur 4 statt 5 in $AnzPass$. Die Änderung von T_2 geht verloren, weil T_1 denselben alten Ausgangswert später überschreibt.

Anomalie 2: Dirty Read

Szenario: T_1 erhöht *AnzPass*, committet aber noch nicht. T_2 liest diesen unbestätigten Wert und committet. Danach bricht T_1 mit Rollback ab.

Problematische Verzahnung:

T_1	T_2
$r_1(\text{AnzPass})$; lese $a = 3$	
$a := a + 1$; also $a = 4$	
$w_1(\text{AnzPass})$; schreibe $a = 4$	
	$r_2(\text{AnzPass})$; lese $b = 4$ (dirty)
	c_2 ; COMMIT von T_2
a_1 ; ROLLBACK auf 3	

Ergebnis: T_2 hat einen noch nicht bestätigten Wert gelesen. Nach dem Rollback von T_1 basiert der Commit von T_2 auf einer Sicht, die nie dauerhaft gültig war.

Anomalie 3: Non-Repeatable Read

Szenario: T_1 liest die Passagierzahl *AnzPass* für Flug BA932. Danach bucht T_2 einen Platz und committet. Ein zweites Lesen desselben Objekts durch T_1 liefert nun ein anderes Ergebnis.

Problematische Verzahnung:

T_1	T_2
$r_1(\text{AnzPass})$; lese $a = 3$	
	$r_2(\text{AnzPass})$; lese $b = 3$
	$b := b + 1$; $w_2(\text{AnzPass})$; schreibe 4
	c_2 ; COMMIT
$r_1(\text{AnzPass})$; lese jetzt $a = 4$	
c_1 ; COMMIT von T_1	

Ergebnis / Problem: T_1 erhält für dasselbe Objekt *AnzPass* zwei unterschiedliche, jeweils bestätigte Werte. Die Sicht von T_1 bleibt innerhalb der Transaktion nicht stabil.

Anomalie 4: Phantom Read

Szenario: T_1 liest alle Passagiere für LH745 und ermittelt später per COUNT(*) die Anzahl. Dazwischen fügt T_2 einen neuen LH745-Passagier ein und committet.

T_1	T_2
SELECT * FROM Passagiere WHERE Flug# = 'LH745';	
	INSERT INTO Passagiere VALUES ('LH745', 'Phantomas', '7D');
	c2; COMMIT
SELECT COUNT(*) FROM Passagiere WHERE Flug# = 'LH745';	

Ergebnis: Die zweite Anfrage von T_1 berücksichtigt ein neues Tupel, das dasselbe Prädikat Flug# = 'LH745' erfüllt. Diese neue Zeile war bei der ersten Anfrage noch nicht sichtbar: ein Phantom.

Zusammenfassung: Warum Schedules?

☰ Definitionen der vier Anomalien

Lost Update	Zwei Transaktionen lesen x und schreiben es auf Basis des alten Werts zurück; die spätere überschreibt die frühere Änderung ($r_i(x) <_s w_j(x) <_s w_i(x)$).
Dirty Read	Eine Transaktion liest einen noch nicht committeten Wert einer anderen ($w_j(x) <_s r_i(x)$, vor c_j); bricht die Quelle ab, war er nie gültig.
Non-Repeatable Read	Dasselbe x wird zweimal gelesen; dazwischen ändert und committet eine andere Transaktion x ($r_i(x) <_s w_j(x) <_s c_j <_s r_i(x)$).
Phantom Read	Eine wiederholte Anfrage mit Prädikat P liefert eine andere Treffermenge, weil eine andere Transaktion passende Tupel einfügt (oder löscht) und committet.

Kernidee: Jede Anomalie entsteht aus problematischer *Verzahnung*, nicht aus einer einzelnen Transaktion.

Ausblick: Aus Beispielen werden nun formale Folgen von Operationen (*Schedules*); danach Serialisierbarkeit und Konflikte.

2. Schedules & Serialisierbarkeit

Formale Modelle für korrekte verzahnte Ausführungen

Leitfrage: Wann ist eine nebenläufige Ausführung logisch so gut wie eine serielle Abarbeitung der beteiligten Transaktionen?



Fokus dieses Abschnitts

Ein Schedule notiert die Reihenfolge von Read-, Write-, Commit- und Abort-Aktionen. Korrektheit heißt: gleicher Effekt wie bei serieller Ausführung.



Kapitelpfad

1. Transaktionen und Nebenläufigkeit
2. **Schedules und Serialisierbarkeit**
3. Konfliktserialisierbarkeit (CSR)
4. Fehlersicherheit und Recoverability
5. Scheduler und Sperrprotokolle
6. 2PL-Korrektheit und Varianten
7. Recovery-Grundlagen

Fahrplan für den Abschnitt

- ▶ Rückblick: Anomalien entstehen durch Verzahnung
- ▶ Referenzfall: serielle Ausführung
- ▶ Modell: Was ist ein Schedule?
- ▶ Korrektheitsziel: Serialisierbarkeit
- ▶ Gegenbeispiel: Wann Verzahnung scheitert
- ▶ Definitionen für die Konfliktanalyse

Ziel: Am Ende können wir für einen gegebenen Schedule begründet fragen, ob er wie eine serielle Ausführung wirkt.

Motivation: Warum Serialisierbarkeit?

Rückblick: Die Anomalien entstehen nicht durch einzelne Transaktionen, sondern durch ihre unkontrollierte Verzahnung.

✓ Ausgangspunkt

Annahme: Jede Transaktion T_i ist für sich korrekt und erhält die Konsistenz.

Sicherer Referenzfall: Serielle Ausführung erfüllt Isolation trivial: Transaktionen laufen vollständig nacheinander.

⚖ Zielkonflikt

Problem: Reine Serialisierung ist oft ineffizient, insbesondere bei vielen Nutzern oder langen Transaktionen.

Ziel: Verzahnte Ausführungen sollen erlaubt sein, aber logisch wie eine serielle Ausführung wirken.

Gesucht: ein formales Kriterium für verzahnte Ausführungen, die trotz Nebenläufigkeit seriell erklärbar bleiben.

Definition: Schedule

Ein **Schedule** ist die globale Reihenfolge der Operationen $r_i(x)$, $w_i(x)$, c_i und a_i aller beteiligten Transaktionen. **Wichtig:** Innerhalb jeder Transaktion bleibt die ursprüngliche Reihenfolge erhalten.

Beispiel: seriell

T_1	T_2
$r_1(x)$	
$w_1(x)$	
$r_1(y)$	
$w_1(y)$	
c_1	
	$r_2(z)$
	$w_2(x)$
	$w_2(z)$
	c_2

Beispiel: verzahnt

T_1	T_2
$r_1(x)$	
	$r_2(z)$
$w_1(x)$	
	$w_2(x)$
$r_1(y)$	
$w_1(y)$	
c_1	
	$w_2(z)$
	c_2

Merke: Ein Schedule ist zunächst nur eine Abfolge von Operationen. Über Korrektheit sagt er erst einmal nichts aus; das prüfen wir erst mit Serialisierbarkeit.

Serialisierbarkeit: Vergleich mit seriellen Abläufen

Intuition: Eine verzahnte Ausführung ist korrekt, wenn ihr Effekt durch eine serielle Reihenfolge derselben Transaktionen erklärbar ist. Damit ist die Isolationsbedingung erfüllt.

Definition

Ein Schedule s über $T_1, \dots, T_n$ heißt **serialisierbar**, wenn es einen seriellen Schedule s' aus genau diesen Transaktionen gibt und $s \equiv s'$ gilt.

X¹ Formal

s serialisierbar $\iff \exists s' \text{ seriell} : s \equiv s'$

$$s' = T_{p_1}; \dots; T_{p_n}$$

Offene Frage: Was bedeutet $s \equiv s'$ genau? Gleiches Endergebnis wäre schwer direkt zu prüfen; deshalb brauchen wir einen strukturellen Äquivalenzbegriff.

Beispiel: Serialisierbarkeit testen

Gegeben: zwei Transaktionen

$T_1 : r_1(A), A := A + 100, w_1(A), r_1(B), B := B + 100, w_1(B), c_1$

$T_2 : r_2(A), A := A * 2, w_2(A), r_2(B), B := B * 2, w_2(B), c_2$

Ausgangszustand

$A = 100, \quad B = 100$

Analyseauftrag: Zuerst berechnen wir die beiden seriellen Referenzergebnisse $T_1; T_2$ und $T_2; T_1$. Danach vergleichen wir damit einen konkreten verzahnten Schedule s .

Kriterium im Beispiel: Passt das Ergebnis von s zu keinem seriellen Referenzergebnis, dann ist s nicht serialisierbar.

Serielle Referenzen: $T_1; T_2$ und $T_2; T_1$

Schedule $T_1; T_2$

Schr.	T_1	T_2	A	B
1	$r_1(A)$ [100]		100	100
2	$w_1(A)$ [200]		200	100
3	$r_1(B)$ [100]		200	100
4	$w_1(B)$ [200]		200	200
5	c_1		200	200
6		$r_2(A)$ [200]	200	200
7		$w_2(A)$ [400]	400	200
8		$r_2(B)$ [200]	400	200
9		$w_2(B)$ [400]	400	400
10		c_2	400	400

Schedule $T_2; T_1$

Schr.	T_1	T_2	A	B
1		$r_2(A)$ [100]	100	100
2		$w_2(A)$ [200]	200	100
3		$r_2(B)$ [100]	200	100
4		$w_2(B)$ [200]	200	200
5		c_2	200	200
6	$r_1(A)$ [200]		200	200
7	$w_1(A)$ [300]		300	200
8	$r_1(B)$ [200]		300	200
9	$w_1(B)$ [300]		300	300
10	c_1		300	300

Seriell mögliche Endzustände: $(A, B) = (400, 400)$ oder $(A, B) = (300, 300)$.



Frage: Ist das in Ordnung? Warum oder warum nicht?

Verzahrter Schedule s : drittes Ergebnis

Schr.	T_1	T_2	A	B	Kommentar
1	$r_1(A)$ [100]		100	100	
2		$r_2(A)$ [100]	100	100	
3	$w_1(A)$ [200]		200	100	T_1 schreibt A
4		$w_2(A)$ [200]	200	100	T_2 schreibt auf Basis des alten Werts
5	$r_1(B)$ [100]		200	100	
6		$r_2(B)$ [100]	200	100	
7	$w_1(B)$ [200]		200	200	T_1 schreibt B
8		$w_2(B)$ [200]	200	200	T_2 schreibt wieder auf Basis des alten Werts
9	c_1		200	200	
10		c_2	200	200	

Endergebnis von s : $(A, B) = (200, 200)$. Das passt weder zu $(400, 400)$ noch zu $(300, 300)$; s ist in dieser Ergebnisprüfung also nicht serialisierbar.

Vom Beispiel zum Prüfverfahren

Im Beispiel

- ▶ Serielle Referenzen berechnen
- ▶ Verzahntes Ergebnis vergleichen
- ▶ Abweichung sichtbar machen

Allgemein

- ▶ Viele mögliche Verzahnungen
- ▶ Viele Startdaten und Programme
- ▶ Ergebnisvergleich kaum praktikabel

Kernidee: Wir wollen Schedules direkt an ihrer Operationsstruktur prüfen, statt jedes mögliche Endergebnis auszurechnen.

Nächster Schritt: Begriffe präzisieren, dann strukturelle Serialisierbarkeitstests formulieren.

Anforderung an Serialisierbarkeitstests

Erhalten bleiben muss

- ▶ Reihenfolge innerhalb jeder Transaktion
- ▶ Wirkung der Lese- und Schreiboperationen
- ▶ korrekter Abschluss durch Commit oder Abort

Geprüft werden soll

- ▶ Konflikte auf gleichen Datenobjekten
- ▶ daraus erzwungene Reihenfolgen
- ▶ Äquivalenz zu einem seriellen Ablauf

Leitgedanke: Eine Verzahnung ist nicht automatisch falsch; entscheidend ist, ob sie sich wie ein serieller Ablauf erklären lässt. Dafür brauchen wir präzise Begriffe.

Grundbegriffe: Transaktion, Schedule, seriell

Transaktion T_i

Geordnete Folge von Operationen einer logischen Arbeitseinheit.

$$T_1 = r_1(A) \ w_1(A) \ r_1(B) \ w_1(B)$$

Schedule s

Globale Operationsfolge mehrerer Transaktionen; die lokale Reihenfolge bleibt erhalten.

Serieller Schedule

Keine Verzahnung: Transaktionen laufen vollständig nacheinander, z. B. $T_1; T_2$.

Serialisierbarkeit fragt, ob ein möglicherweise verzahnter Schedule dieselbe Wirkung hat wie einer der seriellen Referenzabläufe.

Formale Definitionen: Shuffle und Abschluss

$shuffle(T)$

Alle Mischungen der r - und w -Operationen aus $T = \{T_1, \dots, T_n\}$.

- ▶ lokale Reihenfolge bleibt erhalten
- ▶ keine Commit- oder Abort-Operationen
- ▶ Element: s_{basic}

$shuffle_{ac}(T)$

Vollständige Schedules: zu jeder Transaktion wird genau ein Abschluss ergänzt.

- ▶ Commit c_i oder Abort a_i
- ▶ nach der letzten r/w -Operation
- ▶ Element: s_{comp}

Merke: Shuffle beschreibt zulässige Reihenfolgen, aber noch keine Korrektheit.

Definition: Schedule, seriell, serialisierbar

Schedule s

Ein Schedule ist ein beliebiges Präfix eines vollständigen Schedules

$$s_{comp} \in shuffle_{ac}(T).$$

Serieller Schedule

Ein vollständiger Schedule ist seriell, wenn eine Permutation p existiert mit

$$s_{comp} = T'_{p_1}; T'_{p_2}; \dots; T'_{p_n}.$$

Transaktionsblock T'_k

T'_k besteht aus den r/w -Operationen von T_k , gefolgt von genau einem Abschluss: c_k oder a_k .

Serialisierbar: Ein Schedule s ist serialisierbar, wenn er zu mindestens einem seriellen Schedule s' äquivalent ist. Der Äquivalenzbegriff wird als Nächstes präzisiert.

Mengen zu einem Schedule s

$op(s)$	alle Aktionen in s
$trans(s)$	beteiligte Transaktionen
$commit(s)$	$\{T_i \mid c_i \in s\}$
$abort(s)$	$\{T_i \mid a_i \in s\}$
$active(s)$	beteiligte, aber noch nicht abgeschlossene Transaktionen

Beispiel

$T = \{T_1, T_2, T_3\}$
 $T_1 = r_1(x) w_1(x) r_1(y) w_1(y)$
 $T_2 = r_2(z) w_2(x) w_2(z)$
 $T_3 = r_3(x) r_3(y) w_3(z)$
 $s_1 = r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y)$
 $r_1(y) w_1(y) w_2(z) w_3(z) \in shuffle(T)$
 $s_2 = s_1 c_1$ ist ein Schedule
 $s_3 = T_1 c_1 T_3 a_3 T_2 c_2$ ist ein serieller Schedule
 $trans(s_2) = \{T_1, T_2, T_3\}$
 $commit(s_3) = \{T_1, T_2\}$
 $abort(s_3) = \{T_3\}$
 $active(s_2) = \{T_2, T_3\}$

Intuition: $active(s) = trans(s) \setminus (commit(s) \cup abort(s))$: beteiligt, aber noch nicht abgeschlossen.

Zwischenstand: Von Serialisierbarkeit zu CSR

Was feststeht

- ▶ Serielle Ausführung erfüllt Isolation.
- ▶ Verzahnung ermöglicht Nebenläufigkeit.
- ▶ Serialisierbarkeit vergleicht mit seriellen Referenzen.

Was noch fehlt

- ▶ Ergebnisvergleich ist praktisch unhandlich.
- ▶ Wir brauchen einen strukturellen Test.
- ▶ Relevant sind nur Operationen, die sich beeinflussen können.

Nächster Abschnitt: Konflikte zwischen Operationen, daraus Konfliktserialisierbarkeit (CSR) und der Test über den Konfliktgraphen.

Transaktionsverwaltung

1. Transaktionen und Nebenläufigkeit
2. Schedules und Serialisierbarkeit
3. **Konfliktserialisierbarkeit (CSR)**
4. Fehlersicherheit und Recoverability
5. Scheduler und Sperrprotokolle
6. 2PL-Korrektheit und Varianten
7. Recovery-Grundlagen

Aktueller Fokus

Wie erkennt man effizient, ob ein Schedule sich wie ein serieller Ablauf verhält?

Werkzeug: Konflikte zwischen Operationen und der Konfliktgraph.

Agenda: Konfliktserialisierbarkeit (CSR)

1. Konflikte

- ▶ kritische Operationen
- ▶ RW, WR, WW
- ▶ Reihenfolge zählt

2. Äquivalenz

- ▶ gleiche Konfliktordnung
- ▶ CSR-Definition
- ▶ Vergleich mit seriell

3. Test

- ▶ Konfliktgraph
- ▶ Azyklizität
- ▶ Aussage und Grenzen

Leitfrage: Wann erzwingt die Reihenfolge der Konflikte eine serielle Reihenfolge der Transaktionen?

Serialisierbarkeit als Ziel

- ▶ korrekt, wenn äquivalent zu einem seriellen Ablauf
- ▶ Endergebnisse zu vergleichen ist anschaulich
- ▶ praktisch aber kaum testbar

Strukturelle Prüfung

- ▶ Welche Operationen beeinflussen sich?
- ▶ Welche Reihenfolgen müssen erhalten bleiben?
- ▶ Reicht das für einen effizienten Test?

Kritische Operationen heißen: Konflikte.

Konflikte zwischen Operationen

Definition

Operationen $p_i(x)$ aus T_i und $q_j(y)$ aus T_j stehen in Konflikt, wenn:

1. $i \neq j$
2. $x = y$
3. mindestens eine Operation schreibt

Geordnete Konflikttypen

Für ein geordnetes Paar im Schedule gilt jeweils $i \neq j$ und $p <_s q$:

RW $r_i(x) <_s w_j(x)$ Read vor Write

WR $w_i(x) <_s r_j(x)$ Write vor Read

WW $w_i(x) <_s w_j(x)$ Write vor Write

Kein Konflikt: Read/Read-Paare oder Operationen auf verschiedenen Objekten.

Intuition: Konflikte markieren genau die Paare, bei denen die Reihenfolge gelesene Werte oder den Endzustand beeinflussen kann.

Konfliktmenge eines Schedules

$C(s)$: Konfliktmenge

$C(s)$ enthält alle geordneten Paare (p, q) von Operationen in s , sodass:

- ▶ $p \in T_i, q \in T_j$ und $i \neq j$
- ▶ p und q stehen in Konflikt
- ▶ p steht vor q im Schedule: $p <_s q$

$conf(s)$: bereinigte Konfliktmenge

Aus $C(s)$ werden Paare entfernt, sobald eine beteiligte Transaktion in s abgebrochen wird.

$$conf(s) = \{(p, q) \in C(s) \mid T_i, T_j \notin abort(s)\}$$

Bereinigung: $conf(s)$ enthält nur Konfliktpaare, deren Transaktionen in s nicht abgebrochen wurden.

Beispiel: Konfliktmenge bereinigen

Schedule

$$s = w_1(x) \ r_2(x) \ w_2(y) \ r_1(y) \ w_1(y) \ w_3(x) \ w_3(y) \ c_1 \ a_2 \ c_3$$

$C(s)$ nach Datenobjekt

$$\begin{aligned} x : & (w_1(x), r_2(x)), (w_1(x), w_3(x)), (r_2(x), w_3(x)) \\ y : & (w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_2(y), w_3(y)), \\ & (r_1(y), w_3(y)), (w_1(y), w_3(y)) \end{aligned}$$

Bereinigung

$$\begin{aligned} abort(s) &= \{T_2\} \\ conf(s) &= \{(w_1(x), w_3(x)), \\ & \quad (r_1(y), w_3(y)), \\ & \quad (w_1(y), w_3(y))\} \end{aligned}$$

Merke: Alle Konfliktpaare mit der abgebrochenen Transaktion T_2 fehlen in $conf(s)$.

Konfliktäquivalenz von Schedules

Intuition

Zwei Schedules sind strukturell gleichwertig, wenn sie dieselben Operationen enthalten und dieselben relevanten Konfliktpaare erzeugen.

Nichtkonfligierende Operationen dürfen also anders angeordnet sein.

Definition

Schedules s und s' heißen **konfliktäquivalent**, geschrieben $s \approx_c s'$, falls:

$$s \approx_c s' \iff \begin{aligned} op(s) &= op(s') \\ \wedge \quad conf(s) &= conf(s') \end{aligned}$$

Merke: Konfliktäquivalenz hält nur die Reihenfolge konfliktträchtiger Operationen fest. Unabhängige Operationen bleiben frei beweglich.

Beispiel: Konfliktäquivalenz

$T_1 = r_1(x) \ w_1(x) \ r_1(y) \ w_1(y), \quad T_2 = r_2(z) \ w_2(x) \ w_2(y), \quad \text{beide committen}$

Schedule s

$$s = r_1(x) \ w_1(x) \ r_2(z) \ r_1(y) \\ w_2(x) \ w_2(y) \ w_1(y)$$

Schedule s'

$$s' = r_1(x) \ r_2(z) \ w_1(x) \ r_1(y) \\ w_2(x) \ w_2(y) \ w_1(y)$$

Gemeinsame Konfliktmenge

$$\text{conf}(s) = \text{conf}(s') = \{(r_1(x), w_2(x)), (w_1(x), w_2(x)), \\ (r_1(y), w_2(y)), (w_2(y), w_1(y))\}$$

Vertauscht wurden nur $w_1(x)$ und $r_2(z)$. Sie greifen auf verschiedene Objekte zu; daher bleibt $s \approx_c s'$.

Definition: Konfliktserialisierbarkeit (CSR)

Definition

Ein vollständiger Schedule s heißt **konfliktserialisierbar**, wenn er konfliktäquivalent zu einem seriellen Schedule s' ist.

$$s \in CSR \iff \exists s' \text{ seriell} : s \approx_c s'$$

Einordnung

- ▶ CSR ist ein strukturelles Kriterium.
- ▶ Hinreichend für Serialisierbarkeit: $CSR \subseteq SR$.
- ▶ Nicht notwendig: Manche serialisierbaren Schedules liegen außerhalb von CSR .

Warum nützlich? Konfliktserialisierbarkeit ist konservativ, aber effizient prüfbar. Genau dafür dient im nächsten Schritt der Konfliktgraph.

Effizienter Test: Der Konfliktgraph $G(s)$

Idee

Konflikte erzwingen Reihenfolgen zwischen Transaktionen.

Der Konfliktgraph sammelt diese Zwänge als gerichtete Kanten. Ein Zyklus verhindert eine konsistente serielle Reihenfolge.

Definition: $G(s) = (V, E)$

$$V = \text{commit}(s)$$

$$T_i \rightarrow T_j \in E \iff \exists (p, q) \in \text{conf}(s) : \\ p \in T_i, q \in T_j, i \neq j$$

Kantenbedeutung: $T_i \rightarrow T_j$ heißt: T_i stand in einem Konflikt vor T_j . Diese Reihenfolge muss in jedem konfliktäquivalenten seriellen Schedule erhalten bleiben.

Beispiel: Konfliktgraph konstruieren

Schedule

$$s = r_1(x) \ r_2(x) \ w_1(x) \ r_3(x) \ w_3(x) \ w_2(y) \ c_3 \ c_2 \ w_1(y) \ c_1$$

1. Knoten

$$\text{commit}(s) = \{T_1, T_2, T_3\}$$

$$\text{abort}(s) = \emptyset$$

$$V = \{T_1, T_2, T_3\}$$

2. Konflikte sammeln

$$\begin{aligned} x : & (r_2(x), w_1(x)), (r_1(x), w_3(x)), (r_2(x), w_3(x)), \\ & (w_1(x), r_3(x)), (w_1(x), w_3(x)) \\ y : & (w_2(y), w_1(y)) \end{aligned}$$

Keine Transaktion bricht ab: $\text{conf}(s) = C(s)$. Als Nächstes wird jedes Konfliktpaar in eine gerichtete Kante übersetzt.

Beispiel: Konfliktgraph konstruieren

Konflikte in Kanten übersetzen

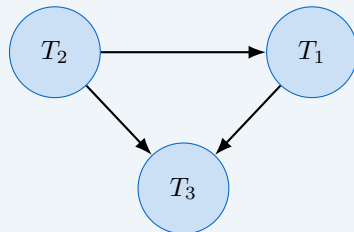
$$V = \{T_1, T_2, T_3\}$$

$$T_2 \rightarrow T_1 : (r_2(x), w_1(x)), (w_2(y), w_1(y))$$

$$T_1 \rightarrow T_3 : (r_1(x), w_3(x)), (w_1(x), r_3(x)), \\ (w_1(x), w_3(x))$$

$$T_2 \rightarrow T_3 : (r_2(x), w_3(x))$$

Resultierender Graph



Mehrere Konfliktpaare können dieselbe Kante erzeugen. Für den Zyklentest zählt die Kante im Graphen trotzdem nur einmal.

Der Serialisierbarkeitssatz

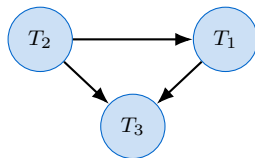
Satz: Ein vollständiger Schedule s ist konfliktserialisierbar genau dann, wenn sein Konfliktgraph $G(s)$ azyklisch ist.

$$s \in CSR \iff G(s) \text{ ist azyklisch}$$

Algorithmischer Test

1. $G(s)$ konstruieren.
2. Auf Zyklen prüfen.
3. Topologische Sortierung als serielle Reihenfolge lesen.

Beispiel



Azyklisch, also $s \in CSR$. Eine mögliche serielle Reihenfolge ist $T_2; T_1; T_3$.

Beweis: $CSR \Rightarrow G(s)$ azyklisch

Kontraposition: Wenn $G(s)$ einen Zyklus enthält, dann gilt $s \notin CSR$.

Annahme

Der Konfliktgraph enthält einen Zyklus:

$$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$$

Jede Kante steht für mindestens einen relevanten Konflikt, dessen Reihenfolge erhalten bleiben muss.

Widerspruch

Der Zyklus erzwingt gleichzeitig:

$$T_1 < T_2 < \dots < T_n < T_1$$

Ein serieller Schedule ordnet Transaktionen aber linear. Eine lineare Ordnung kann diese zyklische Forderung nicht erfüllen.

Also existiert kein konfliktäquivalenter serieller Schedule. Damit folgt aus $s \in CSR$, dass $G(s)$ azyklisch sein muss.

Beweis: $G(s)$ azyklisch $\Rightarrow CSR$

Zu zeigen: Ist $G(s)$ azyklisch, dann existiert ein serieller Schedule, der zu s konfliktäquivalent ist.

1. Topologisch sortieren

Da $G(s)$ azyklisch ist, existiert eine topologische Sortierung:

$$T_{i_1}, T_{i_2}, \dots, T_{i_n}$$

Alle Kanten zeigen darin von links nach rechts.

2. Seriellen Schedule bauen

Setze die Transaktionen in dieser Reihenfolge seriell:

$$s_{ser} = T'_{i_1} T'_{i_2} \dots T'_{i_n}$$

Jeder relevante Konflikt behält seine Richtung; nur unabhängige Operationen werden umsortiert.

Damit gilt $s \approx_c s_{ser}$. Also ist s konfliktserialisierbar: $s \in CSR$.

Was verhindert CSR (und was nicht?)

CSR erkennt Konfliktzyklen, aber nicht alle Abbruchprobleme.

Wird erkannt

Lost Update

$$r_1(x) \ r_2(x) \ w_1(x) \ w_2(x) \ c_1 \ c_2$$

$$G(s) : T_1 \leftrightarrow T_2 \Rightarrow s \notin CSR$$

Non-Repeatable Read durch Write

$$r_1(x) \ w_2(x) \ c_2 \ r_1(x) \ c_1$$

$$G(s) : T_1 \leftrightarrow T_2 \Rightarrow s \notin CSR$$

Grenze: Dirty Read

$$s = r_1(x) \ w_1(x) \ r_2(x) \ a_1 \ w_2(x) \ c_2$$

$$\text{conf}(s) = \emptyset$$

$G(s)$ hat keine Kanten und ist azyklisch.
Trotzdem hat T_2 einen Wert gelesen, den T_1 später zurückrollt.

Fazit: CSR ist ein Isolationstest über Konflikte, aber noch kein Recovery-Kriterium. Dafür folgen Recoverability, ACA und Striktheit.

Mitnehmen: CSR

- ▶ Konflikte: r/w , w/r , w/w auf demselben Objekt.
- ▶ Konfliktäquivalenz vergleicht die Reihenfolge relevanter Konflikte.
- ▶ *CSR* heißt: konfliktäquivalent zu einem seriellen Schedule.
- ▶ Test: $s \in CSR \iff G(s)$ azyklisch.

Offene Lücke

- ▶ Abbrüche werden nur begrenzt berücksichtigt.
- ▶ $conf(s)$ blendet abgebrochene Transaktionen aus.
- ▶ Dirty Reads können trotz *CSR* auftreten.

Nächster Schritt: Fehlersicherheit bei Abbrüchen durch Recoverability (RC), ACA und Striktheit (ST).

Übergang: Bisher ging es um korrekte Reihenfolgen bei Nebenläufigkeit. Jetzt geht es zusätzlich um korrekte Reaktionen auf Abbrüche.

Bisher

1. Transaktionen und Nebenläufigkeit
2. Schedules und Serialisierbarkeit
3. Konfliktserialisierbarkeit (CSR)

Jetzt

Fehlersicherheit und Recoverability

Welche Commit- und Abort-Reihenfolgen sind zulässig, wenn Transaktionen Werte anderer Transaktionen gelesen haben?

Danach

Scheduler und Sperrprotokolle, 2PL-Korrektheit und Varianten, Recovery-Grundlagen.

Agenda: Fehlersicherheit und Recoverability

Leitfrage

Wann bleibt ein Schedule korrekt, wenn Transaktionen abbrechen und andere Transaktionen bereits deren Werte gelesen haben?

Fahrplan

1. Rückblick: Grenzen von CSR
2. Problem: Transaktionsabbrüche (a_i)
3. Grundlage: die „Liest-von“-Beziehung
4. Kriterien: Rücksetzbarkeit (RC), ACA, Striktheit (ST)
5. Hierarchie und korrekte Schedules

Motivation: Warum CSR nicht ausreicht

Dirty Read trotz CSR

$$s = w_1(x) \ r_2(x) \ a_1 \ c_2$$

CSR-Sicht

- ▶ $\text{conf}(s) = \emptyset$
- ▶ $G(s)$ ist azyklisch
- ▶ also $s \in \text{CSR}$

Fehlersicht

- ▶ T_2 liest den unbestätigten Wert von T_1
- ▶ T_1 bricht ab
- ▶ T_2 hat den gelesenen Wert bereits committet

Lektion

- ▶ *CSR* sichert Serialisierbarkeit, aber keine Fehlersicherheit.
- ▶ Bei Abbrüchen zählt, wer welchen noch nicht gesicherten Wert gelesen hat.
- ▶ Dafür brauchen wir Recoverability-Kriterien.

Basis für Fehlersicherheit: „Liest-von“

Definition

Für $T_i, T_j \in \text{trans}(s)$ mit $i \neq j$ liest T_i Objekt x von T_j in s , wenn:

1. $w_j(x) <_s r_i(x)$
2. $a_j \not<_s r_i(x)$
3. für alle T_k mit $k \neq j$ gilt:

$$w_j(x) <_s w_k(x) <_s r_i(x) \Rightarrow a_k <_s r_i(x)$$

T_i liest von T_j , falls dies für irgendein Objekt gilt.

Intuition

Die Quelle eines Reads ist der letzte Schreibzugriff vor dem Lesen, dessen Transaktion bis dahin nicht abgebrochen wurde.

#	T_1	T_2	T_3
1	$w_1(x)$		
2	$w_1(y)$		
3		$r_2(u)$	
4		$w_2(x)$	
5		$r_2(y)$	
6			$r_3(x)$
7		$w_2(z)$	
8		a_2	
9	$r_1(z)$		
10	c_1		
11			c_3

AbleSEN

- ▶ T_3 liest x von T_2 (dirty read), nicht von T_1 .
- ▶ T_2 liest y von T_1 .
- ▶ T_1 liest z nicht von T_2 , da $a_2 <_s r_1(z)$.

Fehlersicherheit (1): Rücksetzbarkeit (RC)

Definition: *RC*

Eine Transaktion soll nicht committen, solange sie von noch nicht bestätigten Daten abhängt.

Ein Schedule s ist *rücksetzbar* (recoverable), wenn für alle T_i, T_j mit $i \neq j$ gilt:

$$T_i \text{ liest von } T_j \text{ in } s \wedge c_i \in s \\ \Rightarrow c_j <_s c_i$$

Wenn T_i von T_j liest und T_i committet, muss T_j vorher committen.

Bedeutung

- ▶ Verhindert Commits auf Basis später zurückgerollter Werte.
- ▶ Nach Recovery bleiben keine committeten Werte erhalten, die auf zurückgerollten Werten beruhen.
- ▶ Ist die Basiseigenschaft; ACA und ST werden stärker sein.

Beispiel: Recoverable (RC) vs. Nicht-RC

Schedule s_I (nicht RC)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7		c_2
8	$w_1(z)$	
9	c_1	

Analyse

T_2 liest y von T_1 , $c_2 <_{s_I} c_1$

T_2 committet vor der Quelle T_1 :

$$s_I \notin RC$$

Schedule s_{II} (RC)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

Analyse

T_2 liest y von T_1 , $c_1 <_{s_{II}} c_2$

T_2 wartet bis zum Commit der Quelle T_1 :

$$s_{II} \in RC$$

Problem mit RC: Kaskadierende Abbrüche

Ausgangspunkt: ein recoverable Schedule

$$s_{II} = w_1(x) w_1(y) r_2(u) w_2(x) \textcolor{red}{r_2(y)} w_2(y) w_1(z) c_1 c_2$$

RC ist erfüllt, weil T_2 erst nach c_1 committet.

Was trotzdem passiert

- ▶ T_2 liest y von T_1 : $r_2(y)$.
- ▶ Zu diesem Zeitpunkt ist T_1 noch nicht committet.
- ▶ Der gelesene Wert ist also schmutzig.

Abort-Szenario

Bricht T_1 nach $r_2(y)$, aber vor c_1 ab, wird der von T_2 gelesene Wert ungültig.

Damit müsste auch T_2 zurückgesetzt werden.

RC verhindert falsche Commit-Reihenfolgen, aber nicht das Lesen schmutziger Daten. Ein Abort kann deshalb weitere Aborts auslösen (kaskadierende Aborts).

Fehlersicherheit (2): Avoiding Cascading Aborts (ACA)

Definition: ACA

Kaskadierende Abbrüche werden verhindert, indem das Lesen schmutziger Daten von vornherein verboten wird.

Ein Schedule s ist *ACA*, wenn für alle T_i, T_j mit $i \neq j$ gilt:

$$T_i \text{ liest } x \text{ von } T_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

Also: Die Quelle eines gelesenen Werts muss vor dem Lesen bereits committet haben.

Wirkung

- ▶ Keine Dirty Reads
- ▶ Keine kaskadierenden Abbrüche
- ▶ Spätere Aborts gefährden gelesene Werte nicht

Einordnung

$$ACA \subset RC$$

ACA prüft früher als RC: schon beim Lesen, nicht erst beim Commit.

Beispiel: ACA vs. nur RC

Schedule s_{II} (RC, nicht ACA)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

Analyse

$$r_2(y) <_{s_{II}} c_1$$

T_2 liest y vor dem Commit der Quelle T_1 :

$$s_{II} \notin ACA$$

Trotzdem gilt $s_{II} \in RC$, weil $c_1 <_{s_{II}} c_2$.

Schedule s_{III} (ACA)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

Analyse

$$c_1 <_{s_{III}} r_2(y)$$

T_2 liest y erst nach dem Commit der Quelle T_1 :

$$s_{III} \in ACA$$

Damit ist s_{III} auch recoverable.

Problem mit ACA: Überschreiben uncommitteter Daten

Ausgangspunkt: $s_{III} \in ACA$

$s_{III} = w_1(x) w_1(y) r_2(u) w_2(x) w_1(z) c_1 r_2(y) w_2(y) c_2$

T_2 liest y erst nach c_1 ; ACA ist also erfüllt.

Neues Problem

- ▶ $w_2(x)$ überschreibt $w_1(x)$, bevor T_1 beendet ist.
- ▶ Bricht T_1 danach ab, muss $w_1(x)$ per UNDO entfernt werden.
- ▶ Der aktuelle Wert von x stammt aber bereits von T_2 .

Striktheit soll auch solche *Dirty Writes* vermeiden.

Kritischer Präfix

#	T_1	T_2
1	$w_1(x) = v$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x) = w$
5	$a_1?$	

Bei a_1 müsste Recovery den Effekt von $w_1(x)$ entfernen, ohne den späteren Schreibzugriff $w_2(x)$ zu verlieren.

Fehlersicherheit (3): Striktheit (ST)

Definition: *ST*

Striktheit vermeidet jeden Zugriff auf Daten, die von noch aktiven Transaktionen geschrieben wurden.

Ein Schedule s ist *strikt*, wenn für alle T_i, T_j mit $i \neq j$ und alle $p_i(x) \in \{r_i(x), w_i(x)\}$ gilt:

$$w_j(x) <_s p_i(x) \Rightarrow (c_j <_s p_i(x) \vee a_j <_s p_i(x))$$

Nach einem Schreibzugriff von T_j darf T_i erst lesen oder schreiben, wenn T_j beendet ist.

Wirkung

- ▶ Keine Dirty Reads
- ▶ Keine Dirty Writes
- ▶ Einfacheres UNDO bei Abbrüchen

Einordnung

$$ST \subset ACA \subset RC$$

ST prüft sowohl Lese- als auch Schreibkonflikte mit aktiven Transaktionen.

Beispiel: Strikt (ST) vs. Nicht-ST

Schedule s_{III} (ACA, nicht ST)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

Analyse

$$w_1(x) <_{s_{III}} w_2(x) <_{s_{III}} c_1$$

T_2 überschreibt x , bevor T_1 beendet ist:

$$s_{III} \notin ST$$

Schedule s_{IV} (ST)

#	T_1	T_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4	$w_1(z)$	
5	c_1	
6		$w_2(x)$
7		$r_2(y)$
8		$w_2(y)$
9		c_2

Analyse

$$c_1 <_{s_{IV}} w_2(x), \quad c_1 <_{s_{IV}} r_2(y)$$

Alle Konflikte mit T_1 erfolgen erst nach dessen Commit:

$$s_{IV} \in ST$$

Zusammenfassung: Fehlersicherheit & Korrektheit

Fehlersicherheitsklassen

- ▶ *RC*: Commits warten auf Commits gelesener Quellen.
- ▶ *ACA*: Reads erfolgen nur aus committeten Quellen.
- ▶ *ST*: Reads und Writes warten auf das Ende der Quelle.

$$ST \subset ACA \subset RC$$

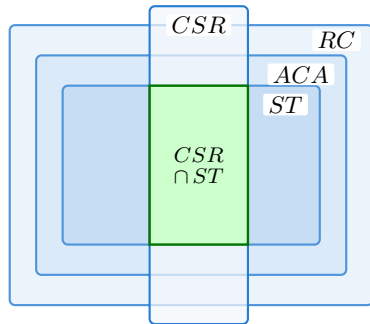
Korrektheit

CSR und Fehlersicherheit beantworten unterschiedliche Fragen:

$$s \text{ korrekt} \implies s \in CSR \cap F$$

mit $F \in \{RC, ACA, ST\}$.

In der Praxis ist häufig $CSR \cap ST$ das robuste Ziel.



Lesart: Serialisierbarkeit und Fehlersicherheit werden getrennt geprüft; brauchbare Schedules liegen in der Schnittmenge.

Mitnehmen

- ▶ *CSR* allein reicht bei Transaktionsabbrüchen nicht aus.
- ▶ Fehlersicherheit ordnet Reads, Writes, Commits und Aborts.
- ▶ Die Klassen werden schrittweise stärker:

$$ST \subset ACA \subset RC$$

- ▶ Korrekte Schedules brauchen Serialisierbarkeit *und* Fehlersicherheit.

Nächste Frage

Wie kann ein Scheduler solche Kriterien in der Praxis garantieren?

Nächstes Thema: Sperrprotokolle, insbesondere das 2-Phasen-Sperrprotokoll (2PL).

Kapitelüberblick

Bisher

1. Transaktionen und Nebenläufigkeit
2. Schedules und Serialisierbarkeit
3. Konfliktserialisierbarkeit (CSR)
4. Fehlersicherheit und Recoverability

Jetzt

5. **Scheduler und Sperrprotokolle**

Danach

6. 2PL-Korrektheit und Varianten
7. Recovery-Grundlagen

Wir wechseln von Kriterien für gute Schedules zu Mechanismen, die solche Schedules erzeugen.

Agenda: Scheduler & Sperrprotokolle

Leitfrage

Wie erzeugt ein DBMS aus konkurrierenden Transaktionsoperationen automatisch korrekte Schedules?

Fahrplan

1. Ziel: *CSR* und Fehlersicherheit
2. Rolle und Eingriffsmöglichkeiten des Schedulers
3. Grundstrategien: pessimistisch vs. optimistisch
4. Sperrkonzepte: Lese-/Schreibsperrn und Kompatibilität
5. Sperrregeln: L1 bis L4
6. Warum 2-Phasen-Sperren (2PL) gebraucht werden

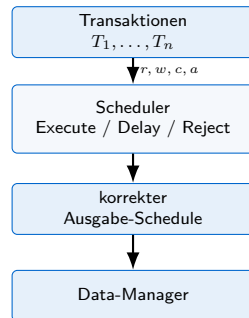
Rückblick & Aufgabe des Schedulers

Vom Kriterium zum Mechanismus

- ▶ Ziel: Ausgabe-Schedules mit *CSR* und Fehlersicherheit.
- ▶ Bisher: gegebene Schedules analysieren.
- ▶ Jetzt: korrekte Schedules aktiv erzeugen.
- ▶ Eingaben: $r_i(x)$, $w_i(x)$, c_i , a_i paralleler Transaktionen.

Aktionen pro Operation

- ▶ **Execute:** Operation ausführen.
- ▶ **Delay:** Operation vorläufig warten lassen.
- ▶ **Reject:** Transaktion abbrechen.



Sicherheit & Ausdruckstärke von Scheduling

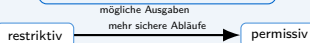
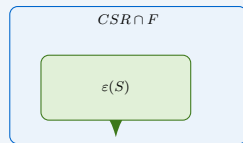
Erzeugnis: $\varepsilon(S) = \{ s \mid s \text{ kann von Scheduler } S \text{ ausgegeben werden} \}$

Sicherheit: nichts Falsches zulassen

- ▶ **s-sicher:** $\varepsilon(S) \subseteq CSR$
- ▶ **f-sicher:** $\varepsilon(S) \subseteq F$ mit $F \in \{RC, ACA, ST\}$
- ▶ **gesamt sicher:** $\varepsilon(S) \subseteq CSR \cap F$

Merksatz: Alle möglichen Ausgaben müssen innerhalb der geforderten Korrektheitsklasse liegen.

Ausdruckstärke: viel Richtiges zulassen



Gute Scheduler sind sicher und lassen trotzdem viele konfliktfreie Nebenläufigkeiten zu.

Grundlegende Synchronisationsstrategien

Sperrende Verfahren

Pessimistischer Ansatz

- ▶ Konflikte werden im Voraus vermieden.
- ▶ Vor dem Zugriff fordert eine Transaktion Sperren an.
- ▶ Inkompatible Sperre vorhanden? Dann wartet die Transaktion.
- ▶ Vorteil: weniger Abbrüche wegen Synchronisationskonflikten.
- ▶ Kosten: Wartezeiten und mögliche Deadlocks.

Nicht-sperrende Verfahren

Optimistischer Ansatz

- ▶ Konflikte werden erst nachträglich erkannt.
- ▶ Transaktionen laufen zunächst ohne Blockieren.
- ▶ Beim Commit wird validiert, ob Konflikte entstanden sind.
- ▶ Konflikt gefunden? Dann Rollback und Wiederholung.
- ▶ Kosten: teuer bei häufigen Konflikten.

Fokus jetzt: **Locking** als klassische Grundlage für Sperrprotokolle und Zwei-Phasen-Sperren.

Das Sperrkonzept: Grundlagen

Idee: Transaktionen reservieren Datenobjekte durch Sperren, bevor sie darauf lesen oder schreiben.

Sperroperationen

$rl_i(x)$ Lesesperre für x anfordern

$wl_i(x)$ Schreibsperre für x anfordern

$ru_i(x)$ Lesesperre für x freigeben

$wu_i(x)$ Schreibsperre für x freigeben

Der Scheduler fügt diese Operationen in den Schedule ein.

Zugriffsmuster

Lesen $rl_i(x) \rightarrow r_i(x) \rightarrow ru_i(x)$

Schreiben $wl_i(x) \rightarrow w_i(x) \rightarrow wu_i(x)$

Vor jedem Datenzugriff steht eine passende Sperre; danach wird sie wieder freigegeben.

Sperroperationen kommen nicht aus der Anwendung, sondern aus der Steuerlogik des Schedulers.

Korrekte Sperrverwendung (Basisregeln L1 bis L3)

Jede Transaktion T_i muss ihre eigenen Sperren wohldefiniert verwenden.

(L1) Korrekte Paarung und Reihenfolge

Jeder Datenzugriff liegt zwischen passender Sperre und passender Freigabe:

$$\begin{aligned}rl_i(x) <_s r_i(x) <_s ru_i(x) \\wl_i(x) <_s w_i(x) <_s wu_i(x)\end{aligned}$$

(L2) Vollständiges Sperren

Jeder Zugriff auf x hat genau eine passende vorangehende Sperre.

(L3) Keine überflüssigen Freigaben

Keine Freigabe $ru_i(x)$ oder $wu_i(x)$ ohne zuvor erworbene Sperre.

Notation:

$DT(s)$ Projektion von s auf Daten- und Abschlussoperationen, also ohne Sperroperationen.

$CP(s)$ *Committed Projection*: nur Operationen von Transaktionen, die in s committen.

Sperrkompatibilität (Regel L4) & Sperrprotokoll

Leitfrage: Wann dürfen zwei verschiedene Transaktionen Sperren auf dasselbe Objekt x halten?

(L4) Kompatibilitätsregel

Eine Sperranforderung von T_j wird nur gewährt, wenn sie mit allen bereits von anderen Transaktionen gehaltenen Sperren auf x kompatibel ist.

Sperrprotokoll: Ein Scheduler arbeitet nach einem Sperrprotokoll, wenn er die Regeln L1–L4 befolgt.

Kompatibilitätsmatrix für x

	$rl_j(x)$	$wl_j(x)$
T_i hält $rl_i(x)$	Ja	Nein
T_i hält $wl_i(x)$	Nein	Nein

- ▶ Read/Read ist kompatibel.
- ▶ Eine Schreibsperre ist exklusiv.

L1–L3 regeln die eigene Sperrverwendung; L4 regelt die Konkurrenz zwischen Transaktionen.

Problem: Ad-hoc-Sperren reichen nicht!

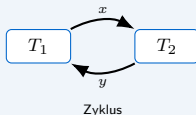
L1–L4 allein garantieren noch nicht *CSR*.

Datenprojektion

$$DT(s) = r_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1$$

Konfliktgraph

- ▶ $T_1 \rightarrow T_2$ wegen $r_1(x) <_s w_2(x)$.
- ▶ $T_2 \rightarrow T_1$ wegen $w_2(y) <_s w_1(y)$.



Zyklus: $DT(s)$ ist nicht konfliktserialisierbar.

Folgerung: Wir brauchen ein strengeres Protokoll, das die Phasen des Sperrens und Freigebens regelt:

2-Phasen-Sperrprotokoll (2PL).

Beispiel-Schedule s

#	T_1	T_2
1	$rl_1(x)$	
2	$r_1(x)$	
3	$ru_1(x)$	
4		$wl_2(x)$
5		$w_2(x)$
6		$wl_2(y)$
7		$w_2(y)$
8		$wu_2(x)$
9		$wu_2(y)$
10		c_2
11	$wl_1(y)$	
12	$w_1(y)$	
13	$wu_1(y)$	
14	c_1	

Das 2-Phasen-Sperrprotokoll (2PL): Definition

2PL-Regel: Nach der ersten Freigabe $ru_i(x)$ oder $wu_i(x)$ darf T_i keine neue Sperre mehr anfordern.

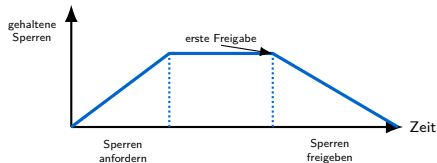
1. Wachstumsphase (Growing Phase)

- ▶ Sperren dürfen angefordert werden.
- ▶ Datenoperationen dürfen ausgeführt werden.
- ▶ Sperren werden noch nicht freigegeben.

2. Schrumpfphase (Shrinking Phase)

- ▶ Sperren dürfen freigegeben werden.
- ▶ Datenoperationen dürfen weiter ausgeführt werden.
- ▶ Neue Sperranforderungen sind verboten.

Phasenwechsel



Kipppunkt: Mit der ersten Freigabe endet die Wachstumsphase.

Regeln für einen 2PL-Scheduler

2PL-Scheduler = korrekte Sperrverwendung (L1–L4) + Zweiphasigkeit.

L1–L3: Sperren korrekt benutzen

Schreibkonvention: Wir kürzen mit $p \in \{r, w\}$ ab: $p_i(x)$ steht für $r_i(x)$ oder $w_i(x)$; $pl_i(x)$ und $pu_i(x)$ für passende Sperre und Freigabe.

- ▶ **L1: Korrekte Paarung:**
 $pl_i(x) <_s p_i(x) <_s pu_i(x)$.
- ▶ **L2:** Jedes genutzte Objekt wird gesperrt.
- ▶ **L3:** Keine unnötigen Freigaben.

L4 + 2PL: Nebenläufigkeit steuern

- ▶ **L4:** Gleichzeitige Sperren verschiedener Transaktionen auf demselben Objekt müssen kompatibel sein.
- ▶ **2PL:** Nach der ersten Sperrfreigabe darf keine neue Sperre mehr angefordert werden.

Ziel: Jeder erzeugte Schedule soll eine konfliktserialisierbare Datenprojektion haben.

Zusammenfassung

- ▶ Scheduler steuern die nebenläufige Ausführung.
- ▶ Locking ist ein pessimistischer Ansatz zur Konfliktvermeidung.
- ▶ L1–L4 definieren korrekte Sperrverwendung und Kompatibilität.
- ▶ Ad-hoc-Sperren reichen nicht aus, um *CSR* sicherzustellen.
- ▶ 2PL erzwingt Wachstums- und Schrumpfphase.

Ausblick

- ▶ Warum stellt 2PL tatsächlich *CSR* sicher?
- ▶ Welche Varianten gibt es, etwa C2PL oder S2PL?
- ▶ Welche zusätzlichen Garantien bieten sie, etwa Striktheit oder Deadlock-Freiheit?

Nächster Schritt: 2PL nicht nur definieren, sondern seine Korrektheit und Varianten verstehen.

Kapitelüberblick

Bisher

1. Transaktionen und Nebenläufigkeit
2. Schedules und Serialisierbarkeit
3. Konfliktserialisierbarkeit (CSR)
4. Fehlersicherheit und Recoverability
5. Scheduler und Sperrprotokolle

Jetzt

6. **2PL-Korrektheit und Varianten**

Danach

7. Recovery-Grundlagen

Wir wechseln von der Definition des 2PL-Protokolls zum Beweis seiner Korrektheit und zu wichtigen Varianten.

Agenda: 2PL-Korrektheit und Varianten

Korrektheit von 2PL

1. Rückblick: 2PL-Regeln und Beweisidee
2. Beweisziel: $\varepsilon(2PL) \subseteq CSR$
3. Lemmas: Konflikte erzwingen eine Ordnung
4. Theorem: der Konfliktgraph ist azyklisch

Varianten und Praxis

1. Deadlocks als zentrales Laufzeitproblem
2. Konservatives 2PL (C2PL)
3. Strenges 2PL (S2PL) und sehr strenges 2PL (SS2PL)
4. Bezug zu SQL-Isolationsebenen

Leitfrage: Welche zusätzliche Struktur macht 2PL korrekt, und welche Varianten verbessern Laufzeit- oder Recovery-Eigenschaften?

Was 2PL vorgibt

- ▶ L1–L4: korrekte Sperrverwendung und Kompatibilität
- ▶ Jede Transaktion hat Wachstums- und Schrumpfphase.
- ▶ Der *Lock Point* ist das Ende der Wachstumsphase.

Zu zeigen

Für jeden 2PL-Schedule gilt:

$$CP(DT(s)) \in CSR$$

also insgesamt $\varepsilon(2PL) \subseteq CSR$.

Beweisfahrplan

1. Basiseigenschaften **E1–E3** aus L1, L4 und der Zweiphasigkeit sammeln; Beweisobjekt ist $CP(DT(s))$, nicht der rohe Sperr-Schedule.
2. **Lemma 1:** Jede Konfliktkante erzwingt *Unlock vor Lock* zwischen den beteiligten Transaktionen.
3. **Lemma 2:** Diese Reihenfolge überträgt sich per Induktion (mit E3) entlang ganzer Pfade.
4. **Lemma 3:** Ein Zyklus erzwingt *Unlock vor Lock* innerhalb *einer* Transaktion – Widerspruch zu E3; also ist $G(CP(DT(s)))$ azyklisch.
5. **Theorem:** Mit dem Serialisierbarkeitssatz folgt $CP(DT(s)) \in CSR$.

Basiseigenschaften unter 2PL

Ausgangslage

Sei $s \in \varepsilon(2PL)$ und $s_c = CP(DT(s))$. Für $p \in \{r, w\}$ bezeichnen $pl_i(x)$ und $pu_i(x)$ den passenden Lock- bzw. Unlock-Aufruf zu $p_i(x)$.

Drei Fakten für den Beweis

E1: L1	Jede Datenoperation liegt zwischen passendem Lock und Unlock.	$pl_i(x) <_s p_i(x) <_s pu_i(x)$
E2: L4	Konfliktierende Operationen brauchen inkompatible Sperren; ihre Intervalle überlappen nicht.	$pu_i(x) <_s ql_j(x)$ oder $qu_j(x) <_s pl_i(x)$
E3: 2PL	Alle Lock-Aufrufe einer Transaktion stehen links von allen Unlock-Aufrufen.	$l_i(*) <_s u_i(*)$

Lemma 1: Konflikt erzwingt klare Reihenfolge

Aussage

Sei $s_c = CP(DT(s))$. Für jede Kante $T_i \rightarrow T_j$ in $G(s_c)$ gibt es ein Objekt x und konfliktverursachende Operationen $p_i(x) <_{s_c} q_j(x)$ mit

$$pu_i(x) <_s ql_j(x).$$

Aus E1

Die Operationen liegen in ihren Sperrintervallen:

$$pl_i(x) <_s p_i(x) <_s pu_i(x),$$

$$ql_j(x) <_s q_j(x) <_s qu_j(x).$$

Aus E1, E2 und Reihenfolge

Wegen L4 dürfen sich die Sperrintervalle inkompatibler Sperren nicht überlappen. Die Alternative $qu_j(x) <_s pl_i(x)$ würde aber

$$q_j(x) <_s qu_j(x) <_s pl_i(x) <_s p_i(x)$$

erzwingen, wobei die beiden äußeren Schritte aus E1 kommen. Das widerspricht $p_i(x) <_{s_c} q_j(x)$. Also bleibt nur $pu_i(x) <_s ql_j(x)$.

Lemma 2: Pfad erhält Sortierung

Aussage

Sei $s_c = CP(DT(s))$. Für jeden Pfad $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k$ in $G(s_c)$ gilt: Ein passendes Unlock von T_1 liegt vor einem passenden Lock von T_k .

Induktion über die Pfadlänge

Anfang	Für $k = 2$ direkt mit Lemma 1: $pu_1(x) <_s ql_2(y)$.
Schritt	Die Induktionsvoraussetzung bringt ein Unlock von T_1 vor einen Lock von T_{k-1} . Lemma 1 für $T_{k-1} \rightarrow T_k$ bringt ein Unlock von T_{k-1} vor einen Lock von T_k .
E3-Klebestelle	In T_{k-1} liegen alle Lock-Aufrufe vor allen Unlock-Aufrufen.
Kette	$pu_1(x) <_s l_{k-1}(z) <_s u_{k-1}(z) <_s ql_k(y)$
Folge	Also liegt ein Unlock von T_1 vor einem Lock von T_k .

Lemma 3: Konfliktgraph ist azyklisch

Aussage

Für jeden 2PL-Schedule s mit $s_c = CP(DT(s))$ ist der Konfliktgraph $G(s_c)$ azyklisch.

Widerspruchsbeweis

Annahme $G(s_c)$ enthalte einen Zyklus $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k \rightarrow T_1$.

Lemma 2 Der Zyklus ist ein Pfad von T_1 nach T_1 . Also gibt es passende Aktionen in T_1 mit $pu_1(x) <_s ql_1(y)$.

Widerspruch In derselben Transaktion läge ein Unlock vor einem späteren Lock. Das verletzt E3: Unter 2PL stehen alle Locks vor allen Unlocks.

Folge Der angenommene Zyklus kann nicht existieren: $G(s_c)$ ist azyklisch.

Theorem: 2PL $\Rightarrow$ CSR

Theorem

Für jeden Schedule s , den ein 2PL-Scheduler erzeugt, ist die committete Datenoperationsprojektion konfliktserialisierbar:

$$\varepsilon(2PL) \subseteq CSR.$$

Beweis

1. Betrachte $s_c = CP(DT(s))$.
2. Nach Lemma 3 ist $G(s_c)$ azyklisch.
3. Nach dem Serialisierbarkeitssatz folgt $s_c \in CSR$.
4. Also erzeugt 2PL auf den committeten Datenoperationen nur konfliktserialisierbare Schedules.

Echte Teilmenge

2PL ist konservativer als reine Konfliktserialisierbarkeit:

$$\varepsilon(2PL) \subsetneq CSR.$$

$$w_1(x) \ r_2(x) \ c_2 \ r_3(y) \ c_3 \ w_1(y) \ c_1$$

Der Konfliktgraph ist azyklisch: $T_3; T_1; T_2$. Dennoch verletzt der Schedule 2PL: T_1 müsste x vor $r_2(x)$ freigeben und später für $w_1(y)$ wieder eine neue Sperre anfordern.

Das Deadlock-Problem bei Sperrverfahren

2PL garantiert Konfliktserialisierbarkeit, verhindert aber keine Deadlocks: Transaktionen können sich gegenseitig blockieren.

Deadlock und Umgang

Deadlock

- ▶ Zwei oder mehr Transaktionen warten dauerhaft aufeinander.
- ▶ Jede hält Sperren, die eine andere im Zyklus benötigt.
- ▶ Im Wartegraphen entsteht ein Zyklus.

Strategien

- ▶ **Prävention/Vermeidung:** Regeln so wählen, dass kein Wartezyklus entstehen kann.
- ▶ **Erkennung und Auflösung:** Deadlocks zulassen, periodisch erkennen und eine Transaktion abbrechen.

Klassisches Muster

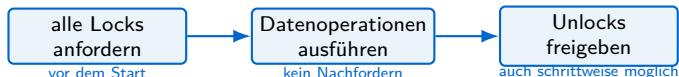
#	T_1	T_2
1	$wl_1(x)$ erfolgreich	
2		$wl_2(y)$ erfolgreich
3	$wl_1(y)$ blockiert: T_2 hält y	
4		$wl_2(x)$ blockiert: T_1 hält x

Wartezyklus: $T_1 \xrightarrow{y} T_2 \xrightarrow{x} T_1$

2PL-Variante: Konservatives 2PL (C2PL)

Regel

Conservative / Static 2PL: Alle benötigten Sperren einer Transaktion müssen **vor der ersten Datenoperation** vollständig angefordert und gewährt werden.



Vorteil

C2PL ist deadlock-frei: Eine Transaktion startet erst, wenn sie alle benötigten Sperren besitzt. Dadurch kann sie nicht mitten in der Ausführung in einen Wartezyklus geraten.

Kosten

- ▶ Alle benötigten Objekte müssen im Voraus bekannt sein.
- ▶ Sperren werden oft früh und unnötig lange gehalten.
- ▶ Starvation ist möglich, wenn nie alle Sperren gleichzeitig verfügbar sind.

2PL-Variante: Strenges 2PL (S2PL / SS2PL)

Freigaberegeln

Abkürzungen: S2PL = Strict 2PL; SS2PL = Strong Strict 2PL (Rigorous 2PL).

2PL	Nach dem ersten Unlock keine neuen Locks.	Garantiert <i>CSR</i> .
S2PL	Alle Schreibsperr en werden erst nach Commit c_i oder Abort a_i freigegeben.	Schreibresultate aktiver Transaktionen bleiben geschützt.
SS2PL	Alle Sperren werden erst nach c_i oder a_i freigegeben.	Einfachere Implementierung; auch Rigorous 2PL.

Wirkung

- ▶ S2PL erzeugt strikte Schedules: $\varepsilon(S2PL) \subseteq CSR \cap ST$.
- ▶ Keine kaskadierenden Aborts durch vorzeitig gelesene oder überschriebene Schreibresultate aktiver Transaktionen.
- ▶ S2PL und SS2PL sind **nicht deadlock-frei**; sie brauchen weiterhin Deadlock-Management oder konservative Sperranforderung.

Einschub: SQL-Isolationsebenen – Motivation

Vom Sperrprotokoll zur Anwendungsschnittstelle

Theorie

CSR und *ST* beschreiben starke formale Garantien, z. B. durch S2PL.

Niedriger: mehr Nebenläufigkeit, mehr erlaubte Anomalien.

SQL-Praxis

Die Anwendung wählt eine **Isolationsebene** als standardisierten Kompromiss.

Höher: stärkere Konsistenz, mehr Synchronisationsaufwand.

Vier Standard-Ebenen: von niedrig bis hoch

READ UNCOMMITTED

kaum Schutz vor Lese-anomalien

READ COMMITTED

nur committete Daten lesen

REPEATABLE READ

gelesene Zeilen bleiben stabil

SERIALIZABLE

serialisierbare Wirkung

Einschub: SQL-Isolationsebenen (1/2) – Niedrige Stufen

READ UNCOMMITTED

Idee	Lesen ohne Rücksicht auf noch aktive Schreiber.
Verhindert	keine der klassischen Leseanomalien zuverlässig.
Möglich	Dirty Read, Non-Repeatable Read, Phantom Read.
Praxis	selten; höchstens für grobe, unkritische Auswertungen.

READ COMMITTED

Idee	Leseoperationen sehen nur bereits committete Versionen.
Verhindert	Dirty Reads.
Möglich	Non-Repeatable Read und Phantom Read.
Praxis	häufiges Default-Level; oft per kurzer Lesesperre oder MVCC.

Einordnung

READ COMMITTED ist der praktische Einstiegspunkt: Es verhindert das Lesen unbestätigter Schreibresultate, garantiert aber noch keine stabile Sicht innerhalb einer ganzen Transaktion.

Einschub: SQL-Isolationsebenen (2/2) – Höhere Stufen

REPEATABLE READ

Idee	Bereits gelesene Zeilen bleiben innerhalb der Transaktion stabil.
Verhindert	Dirty Read und Non-Repeatable Read.
Möglich	Phantom Read, wenn keine Bereichs- oder Prädikatskontrolle greift.
Praxis	länger gehaltene Lesesperren oder Snapshot-Varianten per MVCC.

SERIALIZABLE

Idee	Nebenläufige Ausführung wirkt wie eine serielle Reihenfolge.
Verhindert	Dirty Read, Non-Repeatable Read und Phantom Read.
Umsetzung	striktes 2PL mit Bereichssperren oder serialisierbare MVCC-Verfahren.
Preis	stärkste Konsistenz, aber weniger Nebenläufigkeit.

Merksatz: REPEATABLE READ stabilisiert Tupel; SERIALIZABLE stabilisiert den Schedule.

Einschub: SQL-Isolationsebenen – Zusammenfassung

Welche Anomalien sind noch erlaubt?

Isolationsebene	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED	möglich	möglich	möglich
READ COMMITTED	–	möglich	möglich
REPEATABLE READ	–	–	möglich
SERIALIZABLE	–	–	–

SQL-Syntax

```
SET TRANSACTION ISOLATION LEVEL  
REPEATABLE READ;
```

Trade-off: Striktere Isolation reduziert Anomalien, kann aber Nebenläufigkeit und Durchsatz verschlechtern. Details bleiben DBMS-spezifisch.

Was dieser Block liefert

2PL	Zwei Phasen reichen für Konfliktserialisierbarkeit: $\varepsilon(2PL) \subseteq CSR$.
C2PL	fordert alle Sperren vorab an: deadlock-frei, aber oft unpraktisch.
S2PL / SS2PL	hält Schreibsperren bzw. alle Sperren bis zum Ende: strikte Schedules und einfacheres Recovery, aber weiterhin mögliche Deadlocks.
SQL-Isolation	macht den Konsistenz-/Nebenläufigkeits-Kompromiss für Anwendungen explizit wählbar.

Ausblick: Recovery-Grundlagen

- ▶ Wie funktionieren UNDO und REDO im Detail?
- ▶ Was garantiert Write-Ahead Logging (WAL)?
- ▶ Wie bleiben Atomarität und Dauerhaftigkeit auch nach Systemfehlern erhalten?

Kapitelüberblick

1. Transaktionen und Nebenläufigkeit
2. Schedules und Serialisierbarkeit
3. Konfliktserialisierbarkeit (CSR)
4. Fehlersicherheit und Recoverability
5. Scheduler und Sperrprotokolle
6. 2PL-Korrektheit und Varianten
7. **Recovery-Grundlagen**

Jetzt im Fokus

- ▶ Fehler überstehen
- ▶ Log als Gedächtnis
- ▶ UNDO, REDO und WAL
- ▶ Neustart nach Crash

Agenda: Recovery-Grundlagen

1. Problemraum

- ▶ Warum Recovery? Atomarität und Dauerhaftigkeit.
- ▶ Welche Speicher sind flüchtig, welche persistent?
- ▶ Was kann schiefgehen? Fehlerklassen und Crash-Szenarien.

2. Mechanik

- ▶ Logfile als Gedächtnis der Datenbank.
- ▶ UNDO und REDO nach einem Fehler.
- ▶ Write-Ahead Logging (WAL) als Grundregel.

3. Recovery-Prozess

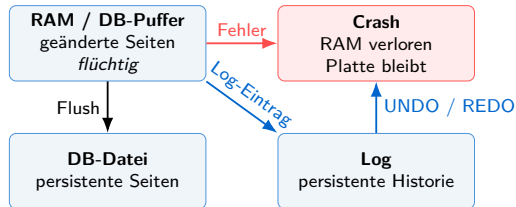
- ▶ Checkpoints begrenzen die Arbeit beim Neustart.
- ▶ Der Recovery-Lauf rekonstruiert einen konsistenten Zustand, z. B. im Stil von ARIES.

Leitfrage: Wie stellt ein DBMS sicher, dass nach einem Crash weder zu viel noch zu wenig in der Datenbank steht?

Recovery: Ziele und technischer Kontext

Ziele und Speicher

A	Transaktionen wirken ganz oder gar nicht.
D	Commit-Effekte bleiben nach einem Crash erhalten.
RAM	schnell, aber flüchtig; hier arbeitet der DB-Puffer.
Platte/Log	persistent; hier liegen Datenbankdatei und Log-Historie.



Problem: Pufferänderungen und Logeinträge erreichen die Platte zu unterschiedlichen Zeitpunkten; Recovery rekonstruiert daraus einen konsistenten Zustand.

Was kann schiefgehen? Fehlerarten

Transaktionsfehler

Ursache

Anwendungsfehler,
Constraint-Verletzung,
Deadlock-Opfer, explizites
ROLLBACK.

Reichweite

Nur die betroffene
Transaktion.

Recovery

UNDO dieser Transaktion.

Systemfehler

Ursache

Stromausfall,
Betriebssystemabsturz, CPU-
oder RAM-Fehler.

Reichweite

RAM geht verloren; Platte
bleibt meist intakt, DB-Datei
ggf. inkonsistent.

Recovery

Log-basiert mit **UNDO** und
REDO.

Medienfehler

Ursache

Zerstörung des
Speichermediums, z. B.
Plattencrash oder Feuer.

Reichweite

Permanente Datenbank ist
teilweise oder ganz verloren.

Recovery

Backup plus Logfiles
nachziehen (*Roll-Forward
Recovery*).

Merksatz: Je dauerhafter der betroffene Speicher, desto schwerer die Wiederherstellung.

Das „Gedächtnis“ der DB: Das Logfile

Grundidee

Recovery-relevante Aktionen werden **sequentiell** und **append-only** auf stabilem Speicher protokolliert.

Typischer Log-Eintrag

Transaktion	ID, z. B. T_i
Typ	START, UPDATE, COMMIT, ABORT, CHECKPOINT
Objekt	Seite, Tupel oder Attribut
Before / After	alter Wert für UNDO, neuer Wert für REDO
Verkettung	optionaler Prev-Zeiger

Warum reicht das?

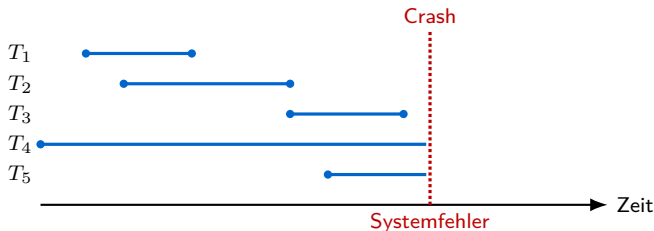
- ▶ Before Images $\Rightarrow$ UNDO.
- ▶ After Images $\Rightarrow$ REDO.
- ▶ Reihenfolge $\Rightarrow$ rekonstruierbare Historie.

Beispiel

```
(T1, START)
(T1, x, v_alt=10, v_neu=20)
(T2, START)
(T2, y, v_alt=50, v_neu=60)
(T1, COMMIT)
...
```

Nach dem Crash: UNDO- und REDO-Notwendigkeit

Szenario: RAM ist verloren, Datenbankdatei und Log liegen auf Platte; die DB-Datei kann trotzdem einen Zwischenzustand enthalten.



Committed vor dem Crash: REDO

- ▶ Beispiele: T_1, T_2, T_3 .
- ▶ Ihre Effekte müssen dauerhaft sein.
- ▶ Fehlende Platteneffekte: **wiederholen**.

Noch aktiv beim Crash: UNDO

- ▶ Beispiele: T_4, T_5 .
- ▶ Ihre Effekte dürfen nicht dauerhaft bleiben.
- ▶ Geschriebene Platteneffekte: **zurücknehmen**.

Die goldenen Regeln: Write-Ahead Logging (WAL)

WAL-Prinzip

Das Log muss die entscheidenden Informationen **früher** dauerhaft speichern als die Datenbankseite oder die Commit-Bestätigung sichtbar wird.

Regel 1: UNDO-Sicherheit

Vor einem Datenflush:

- ▶ Log-Eintrag der Änderung ist stabil.
- ▶ Insbesondere: alter Wert (*Before Image*).
- ▶ Danach kann ein uncommitteter Platteneffekt per **UNDO** entfernt werden.

Regel 2: REDO-Sicherheit

Vor Commit-Erfolg:

- ▶ Alle Log-Einträge der Transaktion sind stabil.
- ▶ Insbesondere: neue Werte (*After Images*).
- ▶ Danach kann ein fehlender Platteneffekt per **REDO** nachgezogen werden.

Kurzform: Erst Log, dann Datenbankseite; erst Log, dann Commit-Erfolg.

Effizienz: Lange Recovery-Zeiten & Checkpoints

Problem ohne Checkpoints

- ▶ Recovery müsste potenziell das gesamte Log durchsuchen.
- ▶ REDO- und UNDO-Kandidaten liegen irgendwo in einer langen Historie.
- ▶ Bei großen Logs wird der Neustart langsam.

Checkpoint-Idee

- ▶ Checkpoint notiert einen Recovery-Startpunkt im Log.
- ▶ Log-Eintrag enthält aktive Transaktionen und relevante Seiten.
- ▶ *Master Record* merkt die Adresse des letzten Checkpoints.
- ▶ Praxis: *Fuzzy Checkpoints* sind kein FORCE; Dirty Pages dürfen offen bleiben.

Effekt für Recovery

- ▶ Log-Scan startet beim Checkpoint bzw. bei der ältesten damals aktiven Transaktion.
- ▶ Nur bei *Sharp/FORCE*-Checkpoints sind früher committete Änderungen sicher auf Platte.
- ▶ Checkpoints verkürzen den Scan; das Log bleibt Quelle für UNDO/REDO.

Recovery-Prozess nach Systemneustart: ARIES

ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) ist das klassische Muster für robustes Crash-Recovery: **Analyse** → **REDO** (*repeat history*) → **UNDO** (*rollback losers*).

1. Analyse

Scan: vorwärts ab Checkpoint.

- ▶ finde aktive Transaktionen: *Loser*
- ▶ rekonstruiere Dirty-Page-Informationen
- ▶ bestimme Startpunkte für REDO

2. REDO

Idee: *repeat history*.

- ▶ scanne das Log wieder vorwärts
- ▶ wende After Images erneut an
- ▶ auch Effekte von Losern werden zunächst wiederholt

3. UNDO

Idee: *rollback losers*.

- ▶ scanne loserbezogen rückwärts
- ▶ mache deren Updates per Before Images rückgängig
- ▶ protokolliere UNDO-Schritte

Merksatz: REDO rekonstruiert nur einen Zwischenstand: die Historie bis zum Crash, auch mit Loser-Updates. Erst UNDO entfernt danach die Effekte der *Loser Transactions*.

Zusammenfassung & Ausblick

Transaktionsmanagement verbindet zwei Schutzlinien: *Isolation* gegen Nebenläufigkeitsfehler und *Recovery* gegen Fehler, Abstürze und verlorene Pufferinhalte.

Recovery-Werkzeugkasten

Baustein	Merksatz
Logfile	protokolliert Updates mit Before und After Images.
WAL	Log-Einträge liegen stabil, bevor DB-Seiten geschrieben werden.
UNDO	entfernt Effekte nicht committeter Transaktionen.
REDO	stellt fehlende Effekte committeter Transaktionen wieder her.
Checkpoint	begrenzt den Log-Bereich, den Recovery durchsuchen muss.
ARIES	Analyse → REDO → UNDO.

ACID-Blick

- ▶ **Atomicity:** UNDO macht halbe Transaktionen unsichtbar.
- ▶ **Durability:** WAL und REDO retten committete Effekte.
- ▶ **Isolation:** Schedules, Serialisierbarkeit und Sperren begrenzen Nebenläufigkeit.

Ausblick: Verteilte DBs: CAP und Replikation; neue Modelle: NoSQL, Graph und In-Memory.