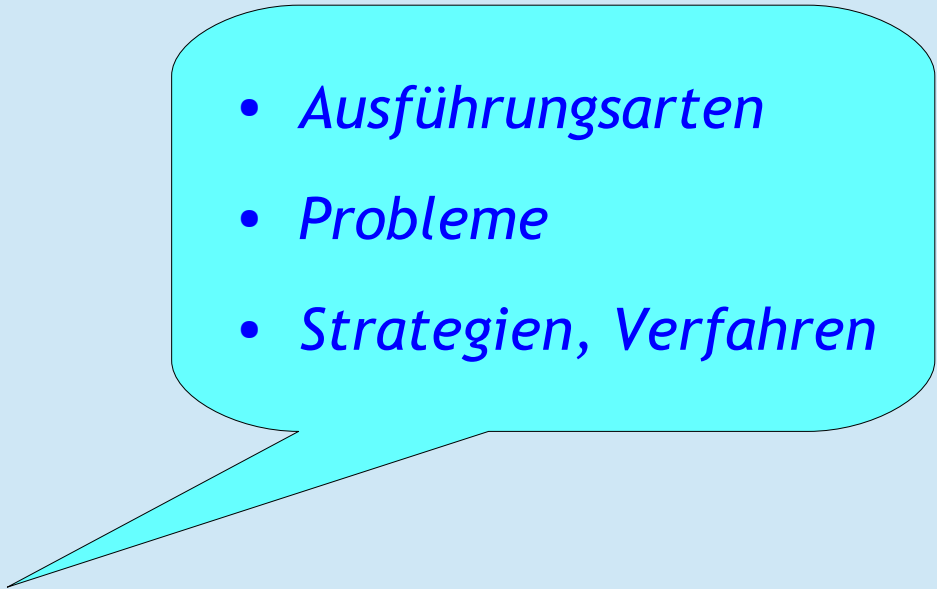




## *Inhalt:*

- *Definition*
- *Beispiel*
- *Anforderungen*
- *Zustände*
- *Synchronisation*

- 
- *Ausführungsarten*
  - *Probleme*
  - *Strategien, Verfahren*



*Integrität (Konsistenz) der Datenbank ist ein Zustand der Daten, in dem sie korrekt, vollständig und widerspruchsfrei sind.*

*Unter einer Transaktion versteht man mehrere Operationen, die:*

- entweder alle erfolgreich durchgeführt werden,*
- oder, falls ein Fehler vorliegt, die schon durchgeführten rückgängig gemacht werden müssen.*

*Transaktionen gewährleisten einen konsistenten Zustand der Datenbank. Somit versetzen die Transaktionen den Datenbestand von einem konsistenten Zustand in einen anderen konsistenten Zustand.*

*Eine Transaktion wird als eine atomare (ununterbrechbare) Operation betrachtet.*

*Die Komponenten einer Transaktion im Falle eines Fehlers werden eigentlich nicht rückgängig gemacht, weil es oft zeitintensiv ist, vielmehr wird der zuvor gespeicherte Zustand der Daten einfach wiederhergestellt.*



*Man betrachte zwei ganze Zahlen A und B (Datenbestand). Man muss in einer Plus-Operation zu allen Komponenten des Datenbestandes (zu A und zu B) die 100 addieren, und in einer Mult-Operation – alle Komponenten mit 2 multiplizieren.*

$T1 = \{ A=A+100; B=B+100; \}$

$T2 = \{ A=A*2; B=B*2; \}$

*Werden diese Transaktionen sequenziell ausgeführt, dann ist die Konsistenz gewährleistet. Werden die Operationen der Transaktionen gemischt, ist dies nicht mehr der Fall.*



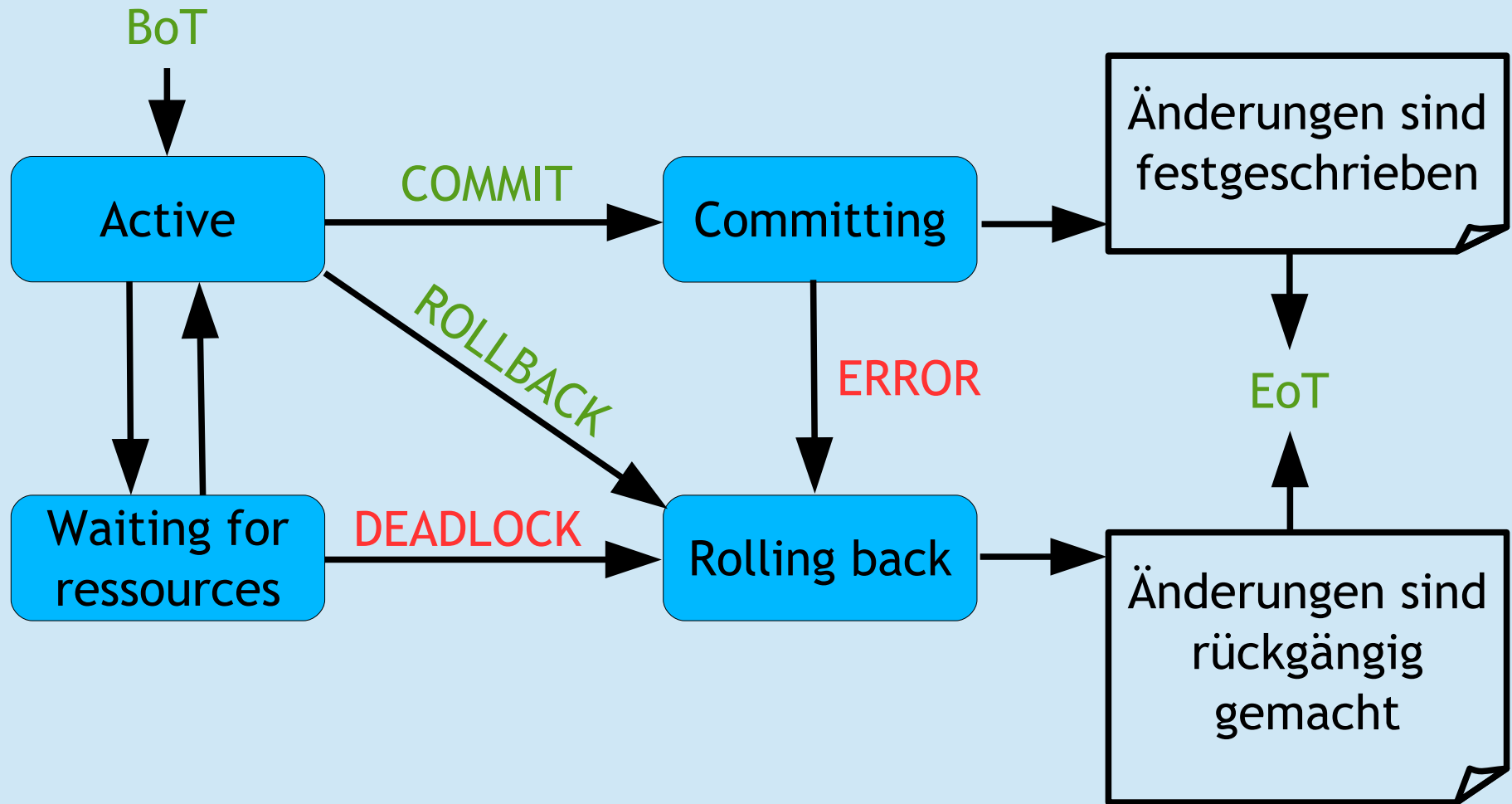
## Anforderungen an die Verwaltung der Transaktionen:

- *Mehrere Transaktionen müssen gleichzeitig (quasi-parallel) verarbeitet werden.*
- *Nach den Fehlern im System müssen die abgeschlossenen Transaktionen bestehen bleiben, die offenen Transaktionen müssen rückgängig gemacht werden.*
- *Transaktionen können manuell (im Source-Code) oder automatisch organisiert werden.*
- *Transaktionen können Zwischenpunkte haben, an denen die Daten der erfolgreich durchgeführten Operation gespeichert werden.*
- *Transaktionen haben nur zwei Arten des Abschlusses: erfolgreich (COMMIT) und erfolglos (ROLLBACK). Erfolgreicher Abschluss passiert aufgrund von Systemfehlern, Integritätsverletzungen, Deadlock, oder durch Anweisung des Benutzers.*



## Anforderungen (Eigenschaften) der Transaktionen ACID:

- *Atomicity (Atomarität) – Entweder alle Befehle oder gar keine werden ausgeführt.*
- *Consistency (Konsistenz) – Eine Transaktion hinterlässt nach ihrer Beendigung einen konsistenten Datenzustand, ansonsten wird sie komplett zurückgesetzt.*
- *Isolation – Nebenläufige (quasi-parallele) Transaktionen beeinflussen sich nicht gegenseitig. Jede Transaktion wird logisch so ausgeführt, als wäre sie die einzig aktive Transaktion.*
- *Durability (Dauerhaftigkeit) – Die Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in dem Datenbestand erhalten. Dies ist auch nach einem Systemfehler gewährleistet.*





*Unter Synchronisation (Mehrbenutzersynchronisation) versteht man die Koordinierung von mehreren Benutzerprozessen. Das ist eng mit der Ausführung der Transaktionen verbunden.*

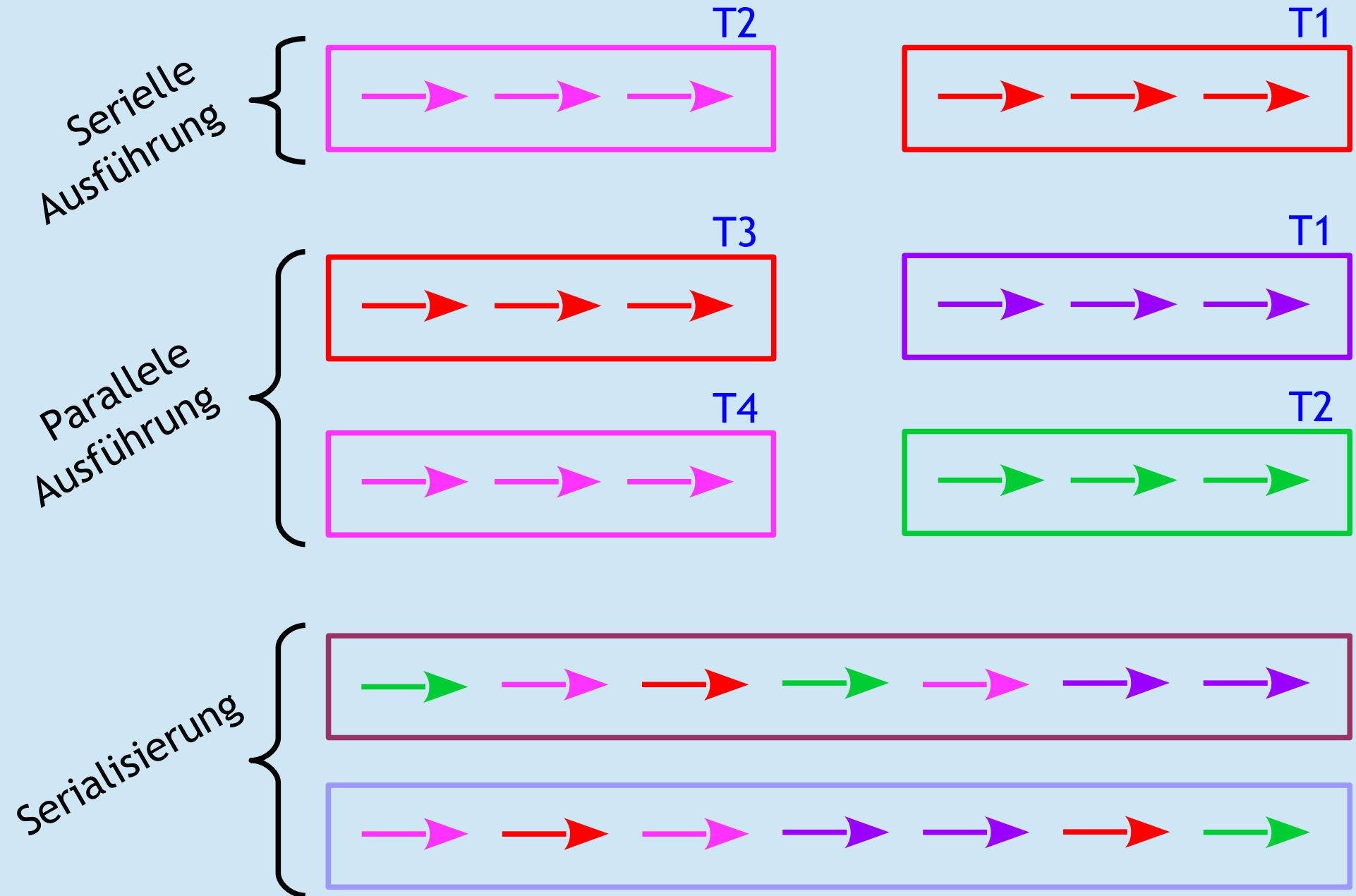
*Ausführung der Transaktionen: seriell und parallel.*

*Serielle Ausführung aller Transaktionen wäre absolut sicher, aber recht langsam. Wartet eine Transaktion auf irgendwelche Ressourcen oder Benutzereingabe, so blockiert sie die nachfolgenden Transaktionen.*

*Parallele Ausführung nutzt die Ressourcen besser aus.*

*Serialisierung ist eine Anordnung der einzelnen Operationen von mehreren parallelen Transaktionen so, dass die Konsistenz der Datenbank gewährleistet wird.*

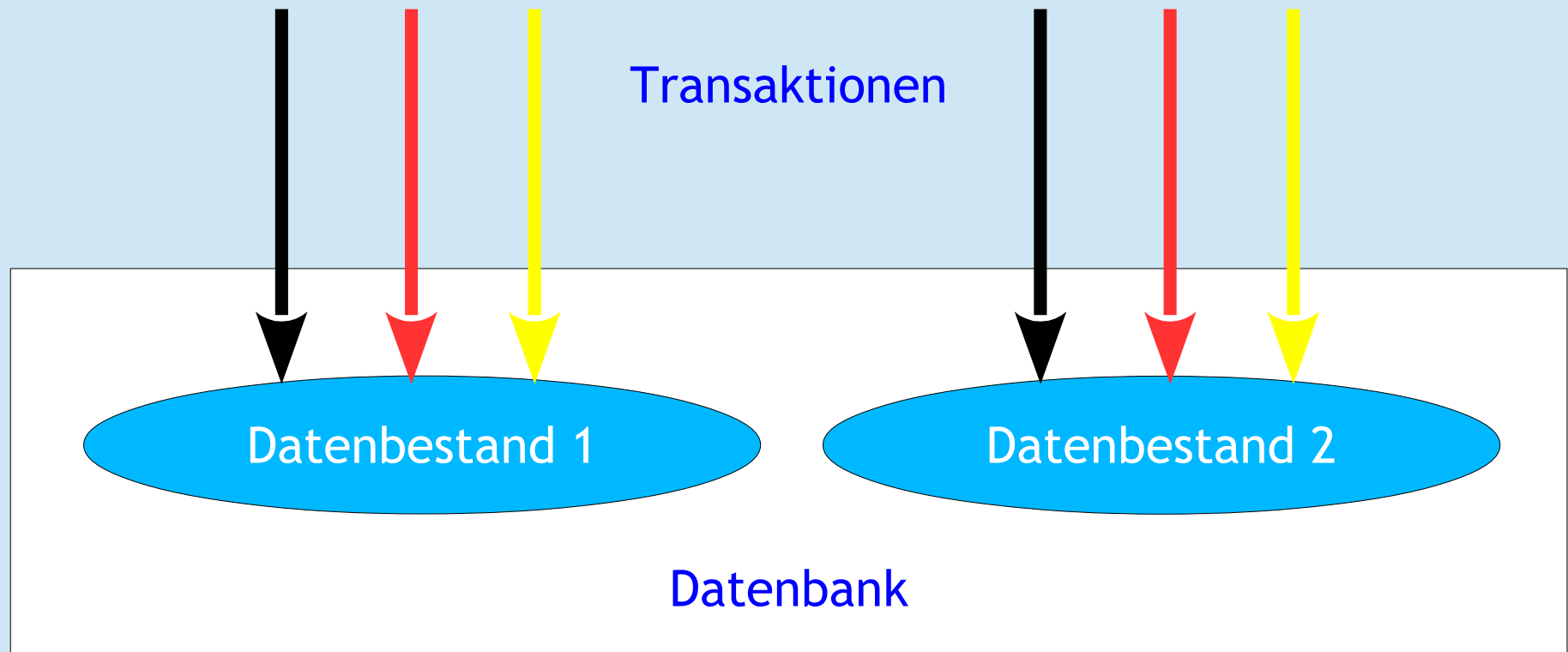
*Serialisierung entspricht in ihrer Wirkung der seriellen Ausführung der Transaktionen.*







*Greifen die Transaktionen auf unterschiedliche Datenbestände zu, dann kann man diese Transaktionen meistens problemlos serialisieren oder parallel ausführen. Greifen sie hingegen auf denselben Datenbestand zu, dann können die Nebenwirkungen bei der Serialisierung vorkommen.*





*Bei der Serialisierung und bei der parallelen Ausführung können folgende Probleme entstehen:*

- *Lost Update;*
- *Dirty Read;*
- *Non-Repeatable Read;*
- *Phantom Read.*

*Man kann diese Probleme durch Isolation der Transaktionen vermeiden. Unter Isolation versteht man verschiedene Sicherheitsstufen (Isolationsstufen) für Transaktionen. Diese Stufen vermeiden die entsprechenden Probleme.*

*Die Isolation ist eigentlich eine der Anforderungen von ACID, wird aber von Datenbanken unterschiedlich umgesetzt (sogar von unterschiedlichen Versionen derselben Datenbank).*



*In vielen Datenbanken wird standardmäßig eine Isolationsstufe eingestellt. Die gewünschte (normalerweise – höhere) Isolationsstufe kann zum Anfang der Transaktion eingestellt werden.*

*Die Isolationsstufe betrifft den Schreibvorgang in der Transaktion nicht. Eine Transaktion bekommt immer eine exklusive Sperre für alle Daten, die sie ändern soll. Diese Sperre dauert bis zum Ende der Transaktion, unabhängig von der für diese Transaktion festgelegten Isolationsstufe.*

*Die Isolationsstufe wirkt auf die Lesevorgänge in der Transaktion, wo sie steht. Und nämlich, sie definiert, wie diese Vorgänge von den anderen Transaktionen abhängig sind, z.B. ob die Daten gelesen werden können, die eine andere Transaktion schon modifiziert, aber noch nicht festgeschrieben hat.*



Die von einer Transaktion geänderten Daten werden von einer anderen Transaktion geändert und festgeschrieben, bevor die erste Transaktion diese Daten entweder festgeschrieben oder zurückgesetzt hat.

Transaction 1	Transaction 2
<i>update TabX set V=42;</i>	. . .
. . .	<i>update TabX set V=53;</i>
. . .	<i>commit;</i>
<i>commit;</i>	. . .

Der Update der Tabelle *TabX* in T2 geht verloren. Diese Art von Fehlern kommt in den Datenbanken nicht vor, da die zweite Transaktion auf die Freigabe des Datensatzes durch die erste Transaktion warten wird.



Eine Transaktion liest die Daten aus einer Tabelle in ihren lokalen Bereich ein. Die andere modifiziert diese Daten in der Tabelle und schreibt sie fest. Die erste Transaktion ändert die im Bereich liegenden Daten und schreibt sie zurück in die Tabelle. In diesem Fall verliert die zweite Transaktion ihren Update.

Transaction 1	Transaction 2
<i>select V from TabX into W;</i>	. . .
<i>W := W+1;</i>	<i>update TabX set V=53;</i>
<i>update TabX set V=W;</i>	<i>commit;</i>
<i>commit;</i>	. . .

Der Update der Tabelle TabX in T2 geht verloren. So ein Problem kann durch die Erhöhung der Isolationsstufe vermieden werden.



```

create or replace procedure TrB1
as
    v int;
begin
    set transaction isolation
        level SERIALIZABLE;
    select Masse into v from Z
        where Stern='Sonne';
    APEX_UTIL.PAUSE(3);
    v := v-1;
    update Z set Masse=v
        where Stern='Sonne';
    commit;
end;

begin      -- Block startet
    TrB1; -- zuerst
end;

-- Prozedur läuft einwandfrei
    
```

```

create or replace procedure TrB2
as
    v int;
begin
    set transaction isolation
        level SERIALIZABLE;
    select Masse into v from Z
        where Stern='Sonne';
    APEX_UTIL.PAUSE(3);
    v := v+1;
    update Z set Masse=v
        where Stern='Sonne';
    commit;
end;

begin      -- Block startet
    TrB2; -- unmittelbar danach
end;      -- in weniger als 3s

-- Prozedur bricht ab
    
```



*Fehlerbericht -*

*ORA-08177: can't serialize access for this transaction*

*\*Cause: Encountered data changed by an operation that occurred after the start of this serializable transaction.*

*\*Action: In read/write transactions, retry the intended operation or transaction.*

*In dem Fall hilft die höchste Stufe der Isolation:*

*set transaction isolation level SERIALIZABLE;*

*Sie macht eigentlich aus jeder Transaktion eine atomare Operation, und die zweite (spätere) Prozedur wird abgebrochen.*

*Ohne diese Einstellung führt der gleichzeitige Start beider Prozeduren zu Lost Update. Klar, startet man eine Prozedur deutlich später als die andere, laufen beide Prozeduren einwandfrei ohne Lost Update.*



```
create or replace procedure TrB1
as
    v int;
begin
    set transaction isolation
        level SERIALIZABLE;
    select Masse into v from Z
        where Stern='Sonne';
    APEX_UTIL.PAUSE(3);
    v := v-1;
    update Z set Masse=v
        where Stern='Sonne';
    commit;
end;

begin      -- Block startet
    TrB1; -- zuerst
end;

-- Prozedur bricht ab
-- mit dem selben Fehler
```

```
create or replace procedure TrB3
as
    v int;
    p int;
begin
    select Masse into v from Z
        where Stern='Sonne';
    v := v+1;
    select Masse into p from Z
        where Stern='Sonne';
    dbms_output.put_line(p);
    update Z set Masse=v
        where Stern='Sonne';
    commit;
end;

begin      -- Block startet
    TrB3; -- unmittelbar danach
end;      -- in weniger als 3s

-- =TrB2, keine Isolationsstufe
-- Prozedur läuft durch
```





Das vorige Beispiel zeigt genau, wie die Isolationsstufe **SERIALIZABLE** wirkt: Obwohl die zweite Prozedur nach der ersten gestartet wurde, sie war schneller durch und konnte problemlos die Daten ändern. Die erste Prozedur war langsamer und hat durch eigene Isolationsstufe **SERIALIZABLE** erkannt, dass die Daten seit eigenem Start geändert wurden, und hat deswegen die Ausführung abgebrochen. Ein Lost Update für die zweite Prozedur wurde somit verhindert.

Ist die Isolationsstufe **SERIALIZABLE** eingestellt, so merkt sich die Transaktion die Daten, die zum Start dieser Transaktion (nicht der Abfrage!) festgeschrieben wurden, und arbeitet mit diesen Daten. Ändert eine andere Transaktion (egal welche Isolationsstufe) inzwischen diese Daten, dann bricht die erste Transaktion ab.



*Dieses Problem tritt auf, wenn eine Transaktion die Daten liest, die eine andere Transaktion geändert, aber noch nicht festgeschrieben hat.*

Transaction 1	Transaction 2
. . .	<code>update TabX set V=42;</code>
<code>select V from TabX;</code>	. . .
. . .	<code>rollback;</code> – – oder <code>commit;</code>

*Wert von V in T1 ist ungültig wegen ROLLBACK in T2.*

*Die Standard-Isolationsstufe (und gleichzeitig auch die niedrigste) in Oracle ist **READ COMMITTED**. Diese Stufe verhindert Dirty Read. Einige Datenbanken erlauben Dirty Read mit **READ UNCOMMITTED**;*



```

create or replace procedure TrC1
as
    v int;
begin
    APEX_UTIL.PAUSE(5);
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
end;

begin      -- Block startet
    TrC1; -- zuerst
end;

/* Die Standard-Isolationsstufe
(und gleichzeitig auch die
niedrigste) in Oracle ist READ
COMMITTED, deswegen ist Dirty
Read hier nicht möglich. Dirty
Read ist nur möglich, wenn hier
set transaction isolation
    level READ UNCOMMITTED; */
    
```

```

create or replace procedure TrC2
as
    v int;
begin
    update Z set Masse=42
        where Stern='Sonne';
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
    APEX_UTIL.PAUSE(10);
    rollback;
    -- commit;
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
end;

begin      -- Block startet
    TrC2; -- unmittelbar danach
end;
    
```



Dieses Problem passiert, wenn eine Transaktion zwei mal dieselben Daten liest und unterschiedliche Ergebnisse bekommt.

Transaction 1	Transaction 2
<code>select V from TabX;</code>	<code>. . .</code>
<code>. . .</code>	<code>update TabX set V=42;</code>
<code>. . .</code>	<code>commit;</code>
<code>select V from TabX;</code>	<code>. . .</code>

Man sieht unterschiedliche Ergebnisse von `SELECT` in `T1`, obwohl sich nichts in `T1` ändert.

Die Standard-Isolationsstufe in Oracle lässt dieses Problem zu, die Abhilfe hier leistet nur die Isolationsstufe `SERIALIZABLE`.



```
create or replace procedure TrD1
as
    v int;
begin
    set transaction isolation
        level SERIALIZABLE;
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
    APEX_UTIL.PAUSE(6);
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
end;

begin      -- Block startet
    TrD1; -- zuerst
end;

/* Die Standard-Isolationsstufe
bringt in Oracle unterschiedliche
Ergebnisse */
```

```
create or replace procedure TrD2
as
    v int;
begin
    APEX_UTIL.PAUSE(2);
    update Z set Masse=42
        where Stern='Sonne';
    commit;
    select Masse into v from Z
        where Stern='Sonne';
    dbms_output.put_line(v);
end;

begin      -- Block startet
    TrD2; -- unmittelbar danach
end;
```



Dieses Problem ähnelt sich dem von Non-Repeatable Read. Eine Transaktion berechnet zwei mal die Anzahl der Zeilen in einer Tabelle, und die andere inzwischen fügt hinzu oder löscht einige Zeilen in dieser Tabelle.

Transaction 1	Transaction 2
<code>select count(*) from TabX;</code>	. . .
. . .	<code>insert into TabX ...;</code>
. . .	<code>commit;</code>
<code>select count(*) from TabX;</code>	. . .

Man bekommt unterschiedliche Ergebnisse von `SELECT` in `T1` wegen des zusätzlichen Datensatzes in `T2`, obwohl sich nichts in `T1` ändert.

Die Isolationsstufe `SERIALIZABLE` löst dieses Problem in Oracle ebenfalls.



```

create or replace procedure TrE1
as
  v int;
begin
  set transaction isolation
    level SERIALIZABLE;
  select count(*) into v from Z;
  dbms_output.put_line(v);
  APEX_UTIL.PAUSE(6);
  select count(*) into v from Z;
  dbms_output.put_line(v);
end;

begin      -- Block startet
  TrD1;    -- zuerst
end;

/* Die Standard-Isolationsstufe
bringt in Oracle unterschiedliche
Ergebnisse */
    
```

```

create or replace procedure TrE2
as
  v int;
begin
  APEX_UTIL.PAUSE(2);
  select count(*) into v from Z;
  dbms_output.put_line(v);
  insert into Z values ('AZ',7);
  commit;
  select count(*) into v from Z;
  dbms_output.put_line(v);
end;

begin      -- Block startet
  TrD2;    -- unmittelbar danach
end;
    
```



*Eine niedrigere Isolationsstufe erlaubt den gleichzeitigen Zugriff auf die Daten, aber erhöht somit die möglichen Nebenwirkungen für Benutzer, wie z.B. Lost Updates oder Dirty Reads.*

*Eine höhere Isolationsstufe vermindert die Probleme, die Benutzer möglicherweise bekommen. Sie verbraucht aber mehr Systemressourcen und erhöht die Wahrscheinlichkeit, dass eine Transaktion eine andere blockiert.*

*Bei der Auswahl einer Isolationsstufe muss man deren Overhead und die Integritätsanforderungen der Anwendung abschätzen.*





Allgemein – verschiedene Stufen der Sicherheit für Transaktion (Isolation) können u.U. in den Datenbanken eingestellt werden, um diese Probleme zu vermeiden (laut Standard ANSI/ISO SQL92):

- *Read Uncommitted* – die minimale Standard-Stufe in einigen Datenbanken; die Daten aktueller Transaktion können von anderen Transaktionen gelesen werden, auch wenn sie noch nicht von der aktuellen Transaktion festgeschrieben wurden (Dirty Read ist erlaubt); die Stufe fehlt in Oracle.
- *Read Committed* – die Stufe vermeidet Dirty Read; die Lesesperren werden kurz vor dem Lesen gesetzt und danach aufgehoben; jede Abfrage einer anderen Transaktion greift nur auf die Daten, die vor Beginn dieser Abfrage von der aktuellen Transaktion festgeschrieben wurden.



- *Repeatable Read* – die Stufe vermeidet zusätzlich zu der vorigen Stufe noch den *Non-Repeatable Read*; die Sperren werden auf die Dauer der gesamten Transaktion gesetzt; die Stufe fehlt in Oracle.
- *Serializable* – die Stufe vermeidet zusätzlich zu der vorigen noch *Phantom Read* und *Lost Update*, sie ist die höchste restriktive Stufe und gewährleistet eine Umgebung, in der neben dieser Transaktion keine weiteren Transaktionen auf Daten schreibend zugreifen.

*Der SQL-Standard schreibt diese Isolierung der Transaktionen vor, aber die konkreten Datenbanken unterstützen sie unterschiedlich.*



Der Scheduler (Programm) steuert, wie Operationen aus verschiedenen Transaktionen relativ zu einander ausgeführt werden.

Es gibt folgende Strategien für Synchronisation:

- Pessimistisch. Der Scheduler verzögert die Ausführung der eingehenden Operationen der Transaktionen. Sind mehrere Operationen angekommen, legt der Scheduler eine optimale Reihenfolge der Operationen fest.
- Optimistisch. Der Scheduler schickt die ankommenden Operationen sofort zur Ausführung.

Es gibt demzufolge die zwei Verfahren für Synchronisation:

- Sperrbasierte Synchronisation (wird oft eingesetzt).
- Zeitstempelbasierte Synchronisation (wird selten eingesetzt).



Idee der sperrbasierten Synchronisation ist:

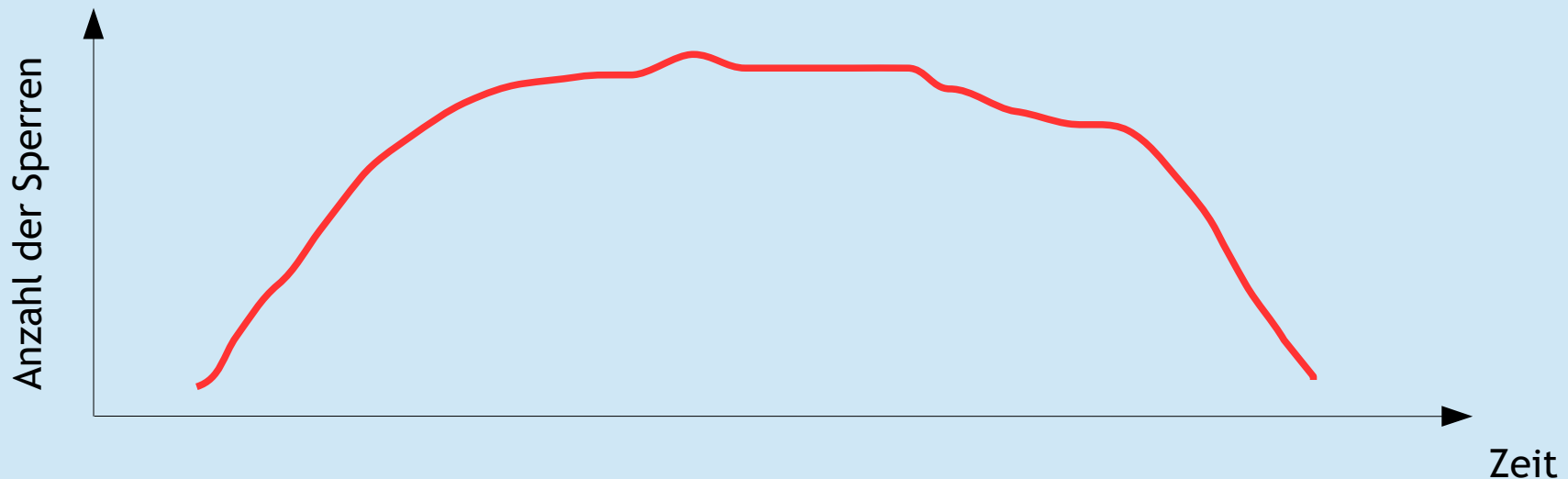
- Jedes Datenobjekt (Tabelle, Datensatz, Attribut, Index, View) hat eine Sperre.
- Jedes Objekt vor Benutzung muss gesperrt werden.
- Transaktion fordert eine Sperre vor dem Zugriff an. Ist Objekt schon von einer anderen Transaktion gesperrt, muss sie warten.
- Nur eine Transaktion kann die Sperre halten.
- Bei Lesesperre (shared lock) ist es den Operationen aus anderen Transaktionen erlaubt, die Daten zu lesen.
- Bei Schreibsperre (exclusive lock) ist es den Operationen aus anderen Transaktionen verboten, die Daten zu lesen oder zu ändern.
- Endet die Transaktion, so muss sie alle Sperren aufheben.

Alle obigen Probleme außer Phantom Read werden damit gelöst.



Bei der sperrbasierten Synchronisation gibt es zwei Phasen (2PL-Protocol):

- *Wachstumsphase – Viele Sperren werden angefordert, aber wenige freigegeben. Die sicherste Variante des Protokolls sieht vor, dass die Sperren nur gesetzt und nicht aufgehoben werden.*
- *Minderungsphase (Schrumpfphase) – Viele Sperren werden freigegeben, und wenige angefordert. Die sicherste Variante des Protokolls sieht vor, dass die Sperren nur aufgehoben und nicht gesetzt werden.*





Idee der zeitstempelbasierten Synchronisation ist:

- Jede Transaktion bekommt einen eindeutigen Zeitstempel.
- Scheduler benutzt diesen Zeitstempel, um die konkurrierenden Operationen unter einander zu ordnen.
- Zu jedem Datenobjekt wird der Zeitstempel der letzten auf ihm ausgeführten Operation gespeichert.
- Kommt eine Operation mit einem früheren Zeitstempel als der des Objektes, so wird sie abgewiesen, somit auch die entsprechende Transaktion.
- Transaktionen müssen laut der Zeitstempelreihenfolge beendet werden.
- Alle obigen Probleme außer Phantom Read werden gelöst.

