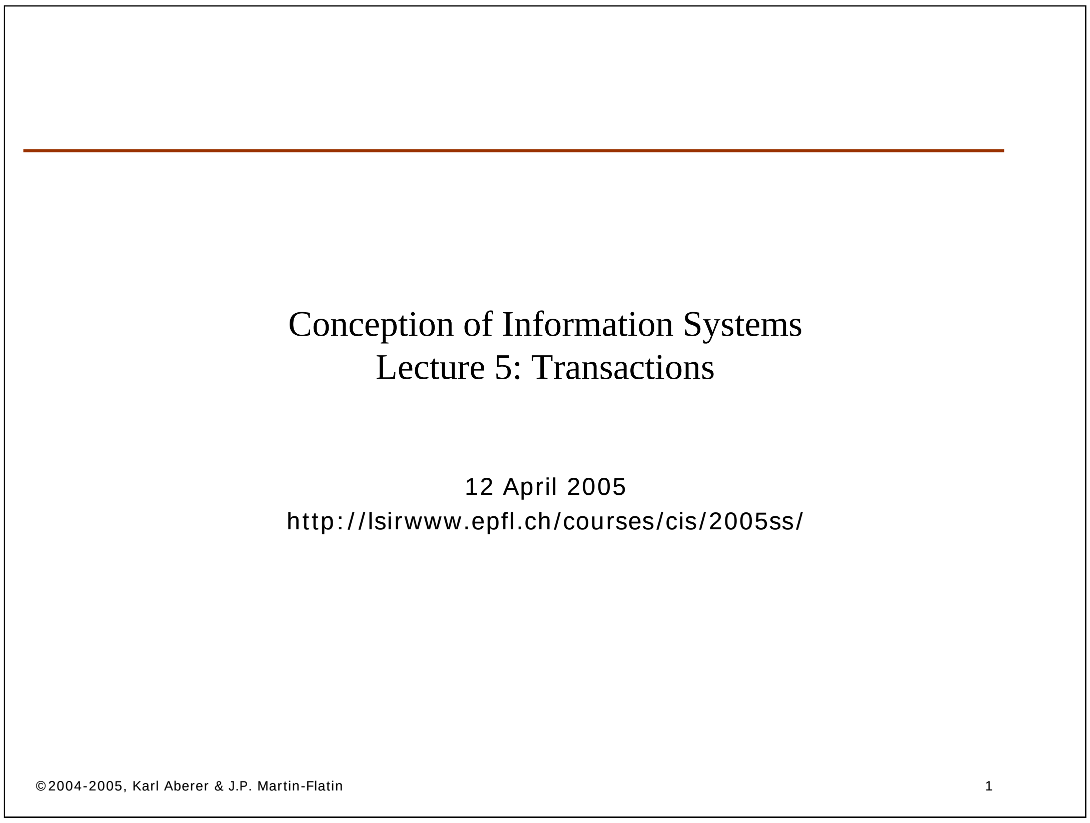
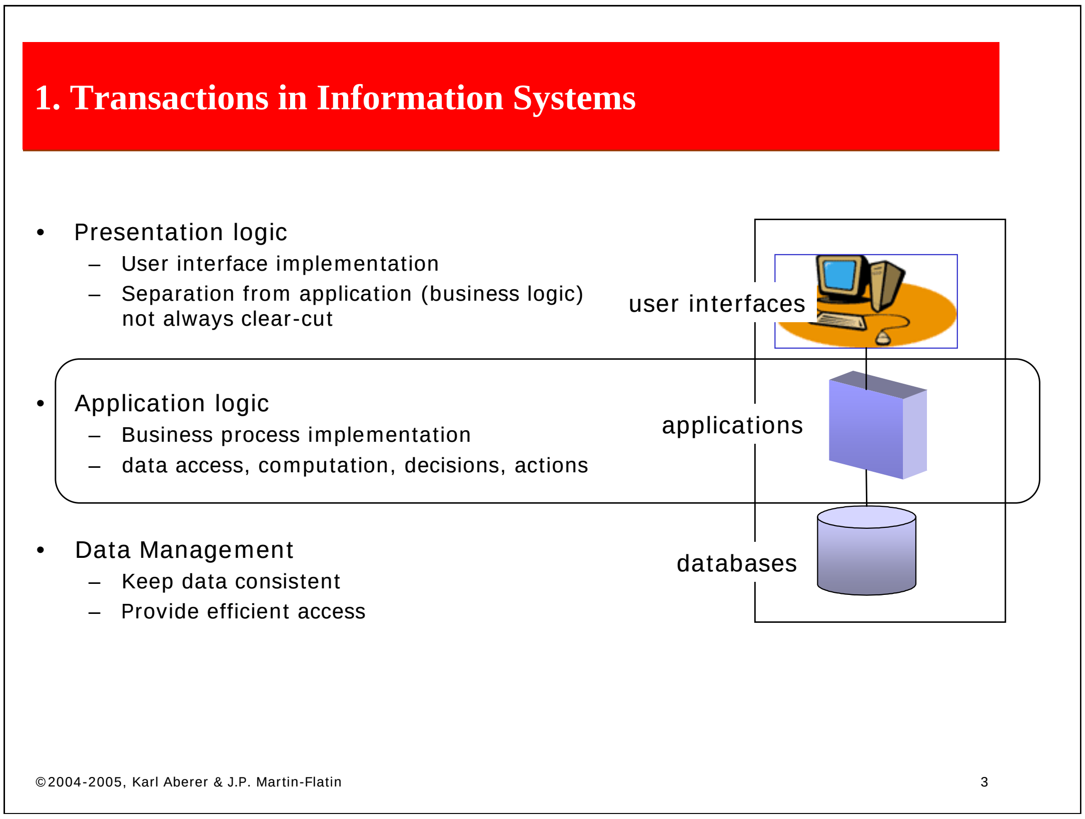


# Conception of Information Systems
# Lecture 5: Transactions

12 April 2005

http://lsirwww.epfl.ch/courses/cis/2005ss/

## Outline



## 1. Transactions in Information Systems

- Presentation logic
  - User interface implementation
  - Separation from application (business logic) not always clear-cut
- Application logic
  - Business process implementation
  - data access, computation, decisions, actions
- Data Management
  - Keep data consistent
  - Provide efficient access

user interfaces
applications
databases

©2004-2005, Karl Aberer & J.P. Martin-Flatin 3

We have already learned about the fundamental three functions in any information system: presentation, application, and data management. This distinction allows to construct information systems in a more systematic way, even if in some cases the distinctions are not always clear-cut: For example, regarding the separation of presentation logic from application logic (business logic) it is not always clear, where a specific functionality belongs to, e.g. checking an input value (business logic or UI ?)

In the previous parts we have already extensively discussed the role of data management, which is the task of the database management system. Database system components provide the necessary services (transaction management, recovery management) to efficiently and consistently manage large amounts of data.

In the following we turn our attention to the applications. Applications implement the "logics" of the business processes (this can be also in other application domains, thus business process is a more general term here). Applications implement certain computations, decisions and actions that are made based upon data. For example, in a banking application

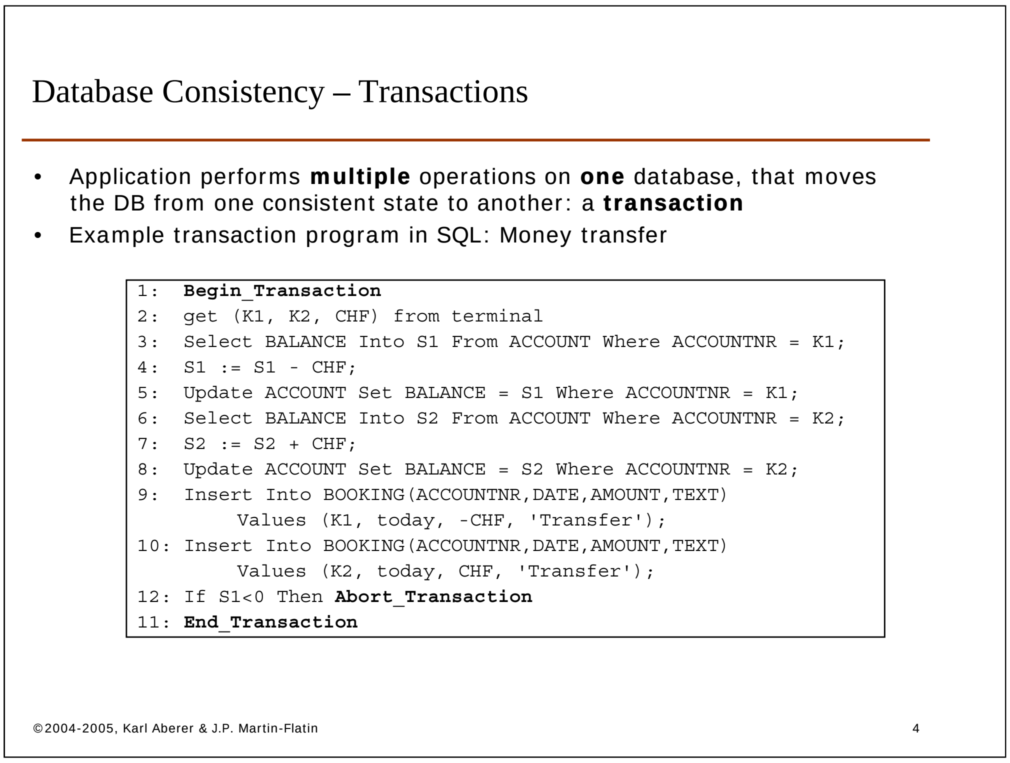
-the data would be the account information

-The computation would be calculating the current amount of money available on all accounts of a customer

-The decision would be whether to grant credit

-The action would be to insert the credit information into the database.

Thus all applications in an information systems access at some point data. However executing such an application typically requires not just a single database access. Therefore we introduce next a basic concept for implementing applications that use persistently stored data: the transaction concept, that supports the consistent execution of multiple actions that are performed against a database and thus forms a basic constituent of any application in an information system.

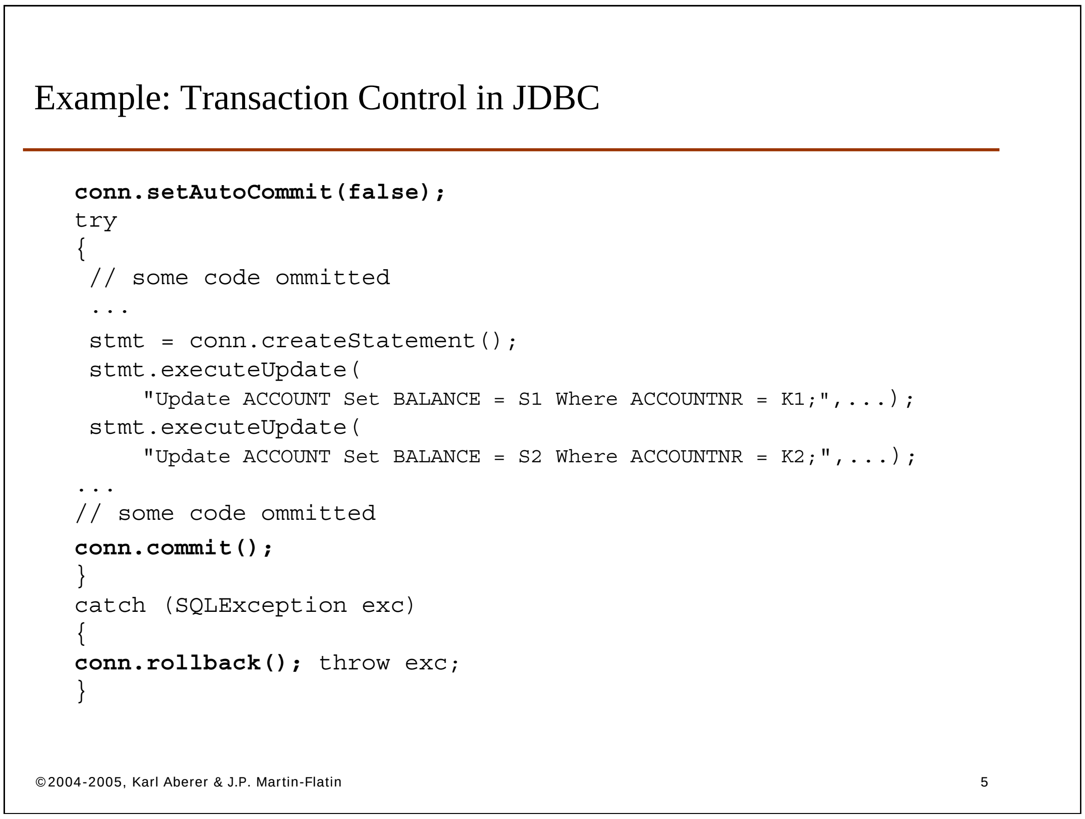


## Database Consistency – Transactions

- Application performs **multiple** operations on **one** database, that moves the DB from one consistent state to another: a **transaction**
- Example transaction program in SQL: Money transfer

```
1:  Begin_Transaction
2:  get (K1, K2, CHF) from terminal
3:  Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4:  S1 := S1 - CHF;
5:  Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6:  Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;
7:  S2 := S2 + CHF;
8:  Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9:  Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
         Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
         Values (K2, today, CHF, 'Transfer');
12: If S1<0 Then Abort_Transaction
11: End_Transaction
```

Typically while accessing a database in an application it occurs that the database involved may be temporarily in an inconsistent state. The example illustrates that point: clearly at the point where money has been taken from one account without putting it on the other account the database is in an inconsistent state, since money "has disappeared". Only at the very beginning of the application and the very end of the application we can be sure that the DB is in a consistent state. This points are marked by the statements Begin Transaction and End Transaction. A transaction is a piece of application code that moves the database from one consistent state to another. Every environment that supports applications that can temporarily violate the integrity of a database should consequently support the transaction concept.



## Example: Transaction Control in JDBC

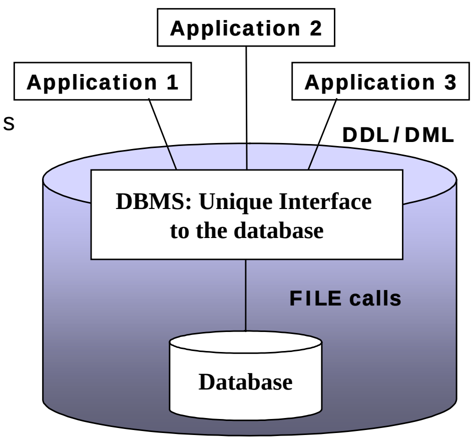
```
conn.setAutoCommit(false);
try
{
 // some code ommitted
 ...
 stmt = conn.createStatement();
 stmt.executeUpdate(
     "Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;",...);
 stmt.executeUpdate(
     "Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;",...);
...
// some code ommitted
conn.commit();
}
catch (SQLException exc)
{
conn.rollback(); throw exc;
}
```

This example shows of how a JAVA application program running at the client would access a relational database via JDBC and exert transaction control.

To understand the transaction commands in JDBC: By default JDBC works in the Auto-Commit-Mode, where after each instruction a commit is executed and a new transaction is started. When this mode is disabled it changes to chained transactions, where only after explicitly issuing commit a new transaction is started.

## Transactions and Database Management Systems

- Coordinates the access by multiple users: **multiple** applications perform **multiple** operations on **one** database: **transactions**
- Goal: Safe execution of application programs on persistent data in the presence of
  - Multiple users (concurrency control - synchronization of parallel programs)
  - Errors (recovery – from hardware and software failures)
- Classical DBMS transaction concept
  - Standard business applications: short transactions, low data volume
  - Transaction manager as component of DBMS



Ensuring that an application that performs **multiple** operations on **one** database does this in a consistent manner is not enough. It may occur that also **multiple** users are accessing the database at the same time, thus transactions that are executed simultaneously need also to be coordinated.

For the safe execution of transaction two things need to be considered:

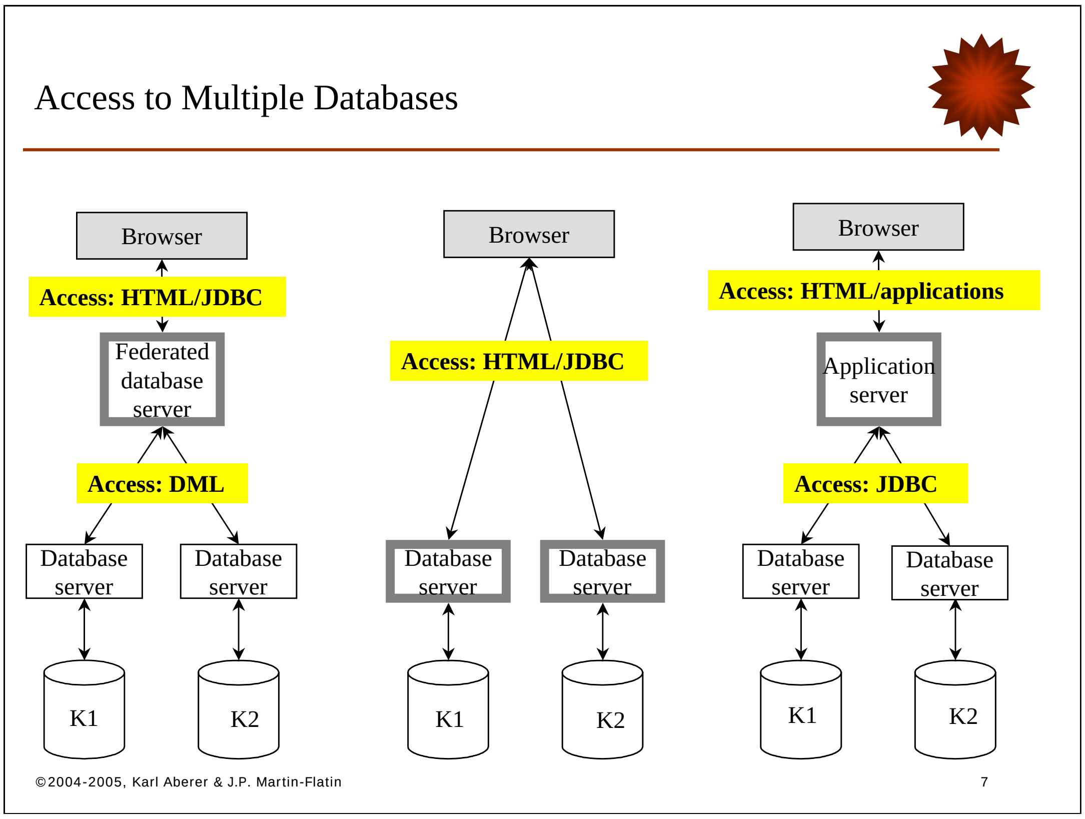
-the synchronization of the accesses of multiple users to the database

-and the recovery from errors in case a transaction is interrupted by any kind of failure (then the effects of the operations of incomplete transactions that have been performed need to be removed from the database – recovery)

Classically this functionality is provided as a component of a database management system. As we will see later the implementation of transaction concepts goes far beyond the scope of DBMS, to other platforms for implementing applications that access databases.

An important remark on the transaction concept as introduced is also in place: it is oriented towards the requirements of classical business applications, with many short transactions that access low data volumes. For other types of applications (e.g. workflows) other concepts have been developed in order to coordinate the access to persistent data sources.

Finally, we have to point out an ambiguity in the graphical notation. Sometimes a "database cylinder" denotes the DBMS that stores and manages the data (colored cylinder in the figure) and in other cases a distinction is made between the data itself, denoted by a cylinder, and the DBMS software, denoted by a rectangle.

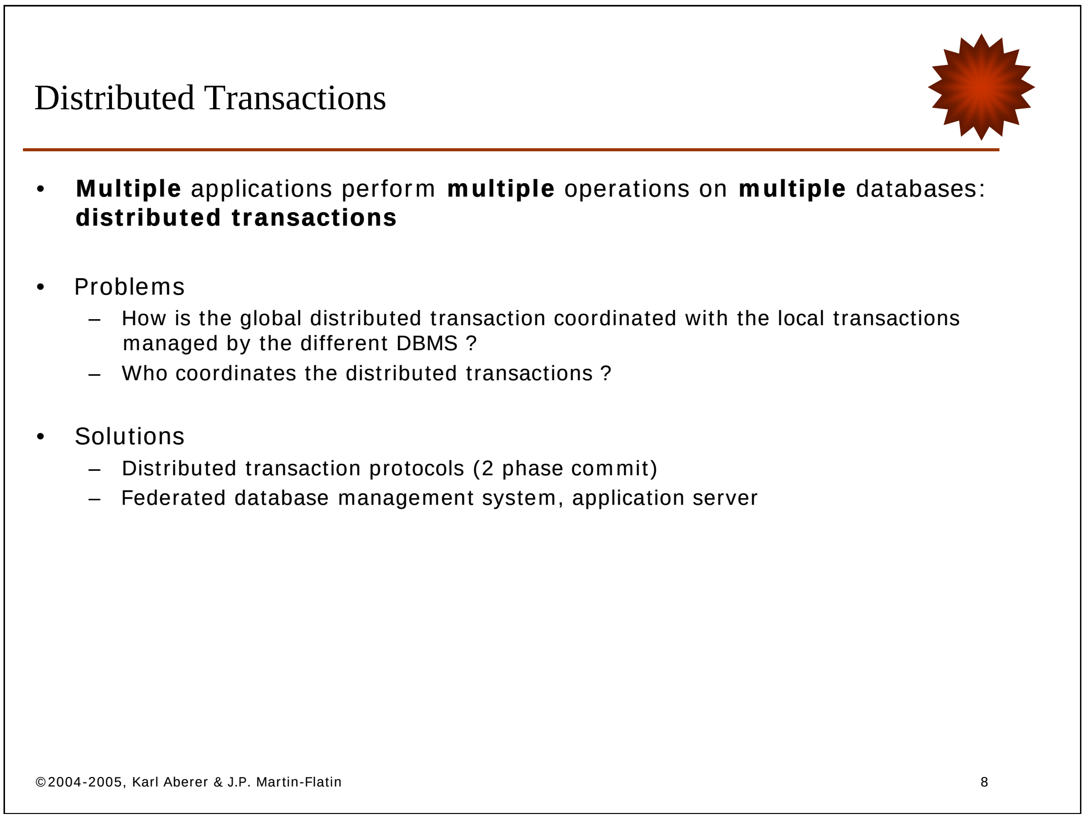


## Access to Multiple Databases

We have earlier studied the problem of accessing multiple databases at the same time. Two solutions have been discussed:

- Access through a federated DBMS. With respect to transactions the problem may occur that within the same transaction multiple databases are accessed at the same time, therefore we have what is called a "distributed transaction"
- Access through a user interface and application programs that are web-enabled. Here no distributed transactions occur since every database is accessed by a separate program.

However, we could easily imagine to use exactly the same mechanism that we used to Web-enable databases, i.e. Web-accessible JAVA applications, that access the DBMS via JDBC, for accessing multiple databases from within the same Java application. Thus we have found a different way to access multiple DBs, by integrating them at the application level. Now we can again have distributed transactions, and they need to be dealt with somewhere. The platform that will be doing this job we call an **application server.** This application server also can run (and in many cases has to run) on a computer (server) that is different from the database servers (remember: JAVA allows DB access over the network).

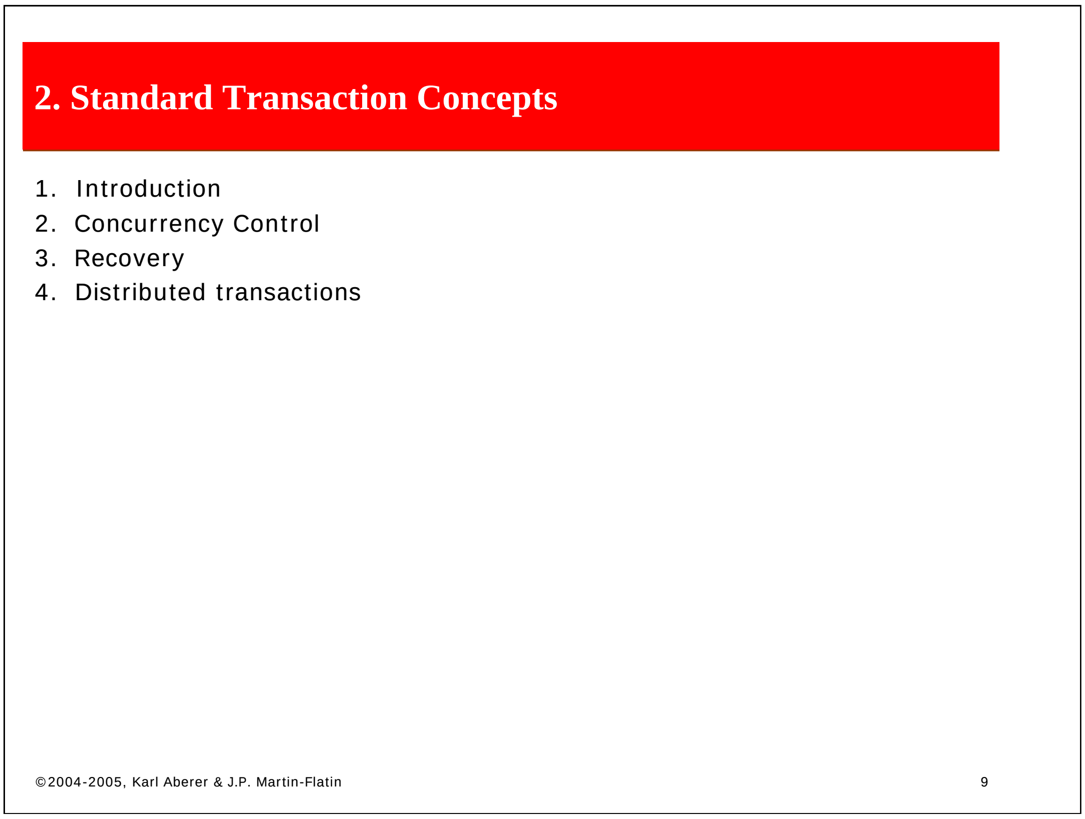


## Distributed Transactions

- **Multiple** applications perform **multiple** operations on **multiple** databases: **distributed transactions**
- Problems
  - How is the global distributed transaction coordinated with the local transactions managed by the different DBMS ?
  - Who coordinates the distributed transactions ?
- Solutions
  - Distributed transaction protocols (2 phase commit)
  - Federated database management system, application server

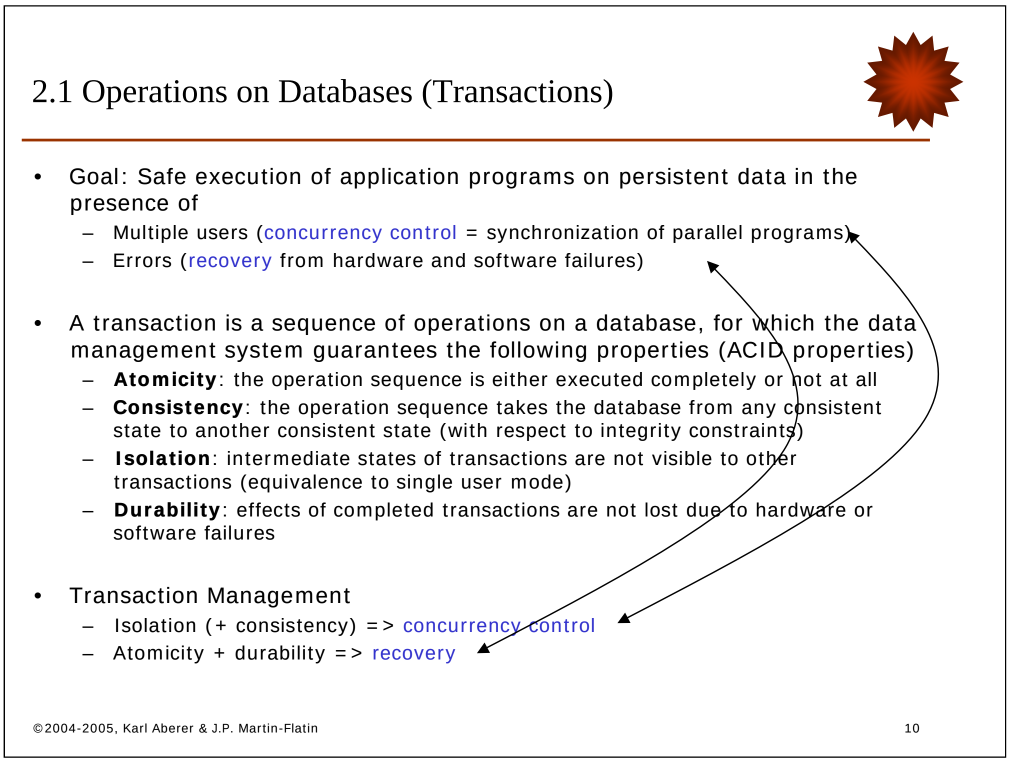
**Multiple** applications perform **multiple** operations on **multiple** databases: **distributed transactions.**

The mechanisms needed to handle distributed transactions extend those that are needed to handle local transactions (e.g. the famous 2PC protocol). Local transactions are typically dealt with by the database servers. Distributed transactions are dealt with a new kind of middleware, so-called transactional middleware. We have already seen two approaches where transactional middleware is implemented: federated DBMS and application servers.



## 2. Standard Transaction Concepts

1. Introduction
2. Concurrency Control
3. Recovery
4. Distributed transactions



## 2.1 Operations on Databases (Transactions)

- Goal: Safe execution of application programs on persistent data in the presence of
  - Multiple users (concurrency control = synchronization of parallel programs)
  - Errors (recovery from hardware and software failures)
- A transaction is a sequence of operations on a database, for which the data management system guarantees the following properties (ACID properties)
  - **Atomicity**: the operation sequence is either executed completely or not at all
  - **Consistency**: the operation sequence takes the database from any consistent state to another consistent state (with respect to integrity constraints)
  - **Isolation**: intermediate states of transactions are not visible to other transactions (equivalence to single user mode)
  - **Durability**: effects of completed transactions are not lost due to hardware or software failures
- Transaction Management
  - Isolation (+ consistency) => concurrency control
  - Atomicity + durability => recovery

10

A transaction takes a DB from one consistent state to another. Thus *atomicity* is the guarantee that even in the presence of errors, the database will not reach an inconsistent state, which means that either all or none of the operations are executed. *Durability* states that once a transaction is completed (=committed), none of the changes will be lost, even if errors occur. Mechanisms are thus required to ensure these properties in case of errors. Doing this is called *recovery*. The failures that recovery has to deal with can be caused by the system or the application. Recovery management makes failures transparent for developers and users.

The property of *consistency* is in fact guaranteed by the programmer. But given that transactions are consistent, i.e. take a DB from one consistent state to another, it must be ensured that other transactions that also access the DB see only consistent states. If multiple transactions are executed in parallel, it is not obvious that this is necessarily the case, since one transaction could read information from the database, while another is just performing a change. In order to avoid such problems, the property of *isolation* states that no transaction can see (read) any intermediate state that is caused by the execution of another transaction. Providing this guarantee, and thus preventing inconsistencies in multi-user mode, is the task of *concurrency control*. Concurrency control makes the multi-user mode transparent to developers and users.

In summary, the transaction mechanism guarantees the consistency of databases, provided that application programs are correctly implemented and correct transaction bounds are set.

## Example 1

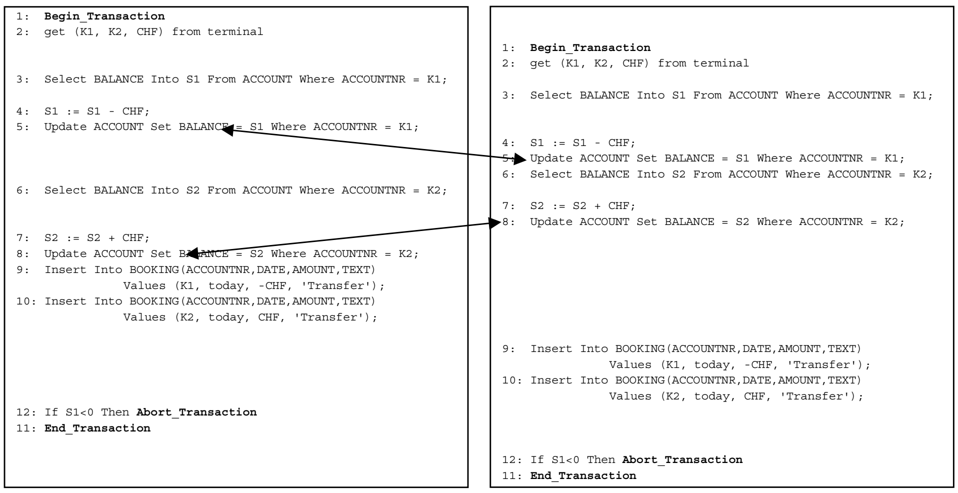
```
1:   Begin_Transaction
2:   get (K1, K2, CHF) from terminal
3:   Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4:   S1 := S1 - CHF;
5:   Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
```

This operation sequence is non-atomic and thus not allowed

```
1:   Begin_Transaction
…
11:  End_Transaction
```

Once the statement End_Transaction is processed the resulting state is durable, i.e. never lost

## Example 2



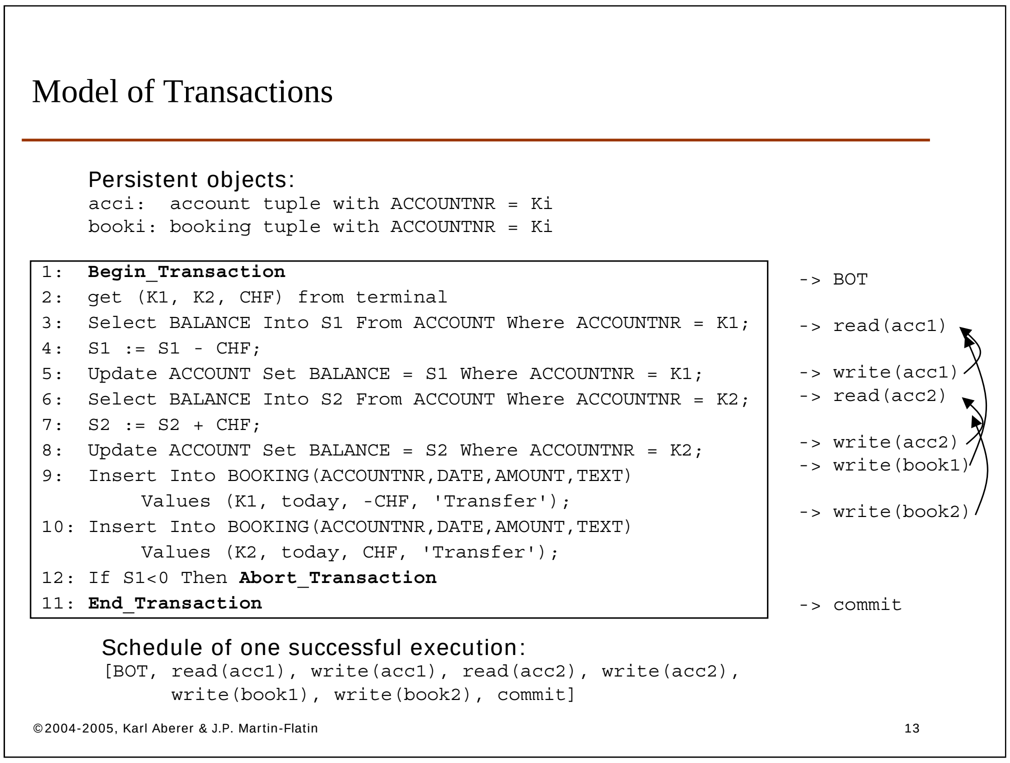
Transaction 1 by Application 1

```
1:  Begin_Transaction
2:  get (K1, K2, CHF) from terminal


3:  Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;

4:  S1 := S1 - CHF;
5:  Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;



6:  Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;


7:  S2 := S2 + CHF;
8:  Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9:  Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
              Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
              Values (K2, today, CHF, 'Transfer');





12: If S1<0 Then Abort_Transaction
11: End_Transaction
```

Transaction 2 by Application 2

```
1:  Begin_Transaction
2:  get (K1, K2, CHF) from terminal

3:  Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;


4:  S1 := S1 - CHF;
5:  Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6:  Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;

7:  S2 := S2 + CHF;
8:  Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;







9:  Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
              Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)
              Values (K2, today, CHF, 'Transfer');



12: If S1<0 Then Abort_Transaction
11: End_Transaction
```

The operations of a transaction may be interleaved, but the transactions must not influence each other



## Model of Transactions

Persistent objects:

```
acci:  account tuple with ACCOUNTNR = Ki
booki: booking tuple with ACCOUNTNR = Ki
```

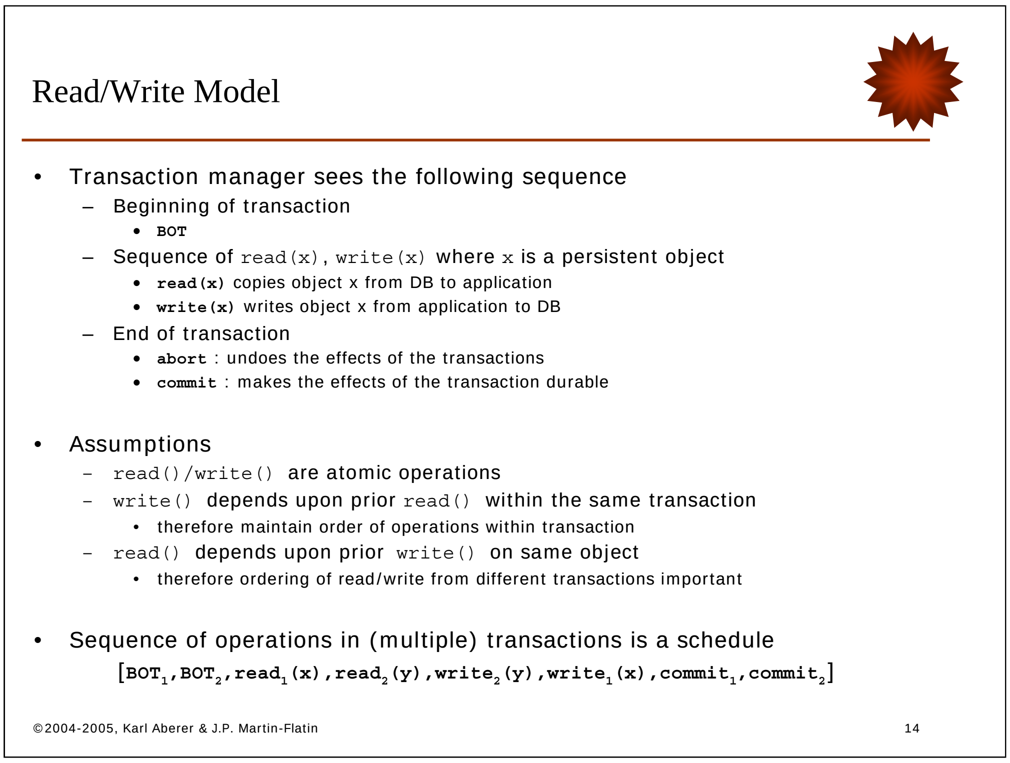
```
1:  Begin_Transaction                                        -> BOT
2:  get (K1, K2, CHF) from terminal
3:  Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1; -> read(acc1)
4:  S1 := S1 - CHF;
5:  Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;    -> write(acc1)
6:  Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2; -> read(acc2)
7:  S2 := S2 + CHF;
8:  Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;    -> write(acc2)
9:  Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)          -> write(book1)
         Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR,DATE,AMOUNT,TEXT)          -> write(book2)
         Values (K2, today, CHF, 'Transfer');
12: If S1<0 Then Abort_Transaction
11: End_Transaction                                          -> commit
```

Schedule of one successful execution:

```
[BOT, read(acc1), write(acc1), read(acc2), write(acc2),
      write(book1), write(book2), commit]
```

©2004-2005, Karl Aberer & J.P. Martin-Flatin 13

This example shows an abstract view of a transaction program using the example program we have introduced earlier.



## Read/Write Model

- Transaction manager sees the following sequence
  - Beginning of transaction
    - **`BOT`**
  - Sequence of `read(x)`, `write(x)` where `x` is a persistent object
    - **`read(x)`** copies object x from DB to application
    - **`write(x)`** writes object x from application to DB
  - End of transaction
    - **`abort`** : undoes the effects of the transactions
    - **`commit`** : makes the effects of the transaction durable
- Assumptions
  - `read()/write()` are atomic operations
  - `write()` depends upon prior `read()` within the same transaction
    - therefore maintain order of operations within transaction
  - `read()` depends upon prior `write()` on same object
    - therefore ordering of read/write from different transactions important
- Sequence of operations in (multiple) transactions is a schedule

  $[\mathbf{BOT_1, BOT_2, read_1(x), read_2(y), write_2(y), write_1(x), commit_1, commit_2}]$

©2004-2005, Karl Aberer & J.P. Martin-Flatin 14

A transaction manager has an abstract view of the application programs: it is not interested into any transient computations, but just into the transaction statements (begin and end of transactions) and accesses to persistent data (reads and writes on the databases).

A transaction can always be normally completed (end of transaction statement or commit) or aborted (abort statement). In the first case the changes are made durable, i.e. they are permanently stored in the database, in the second case the original state before the transaction has started is restored.

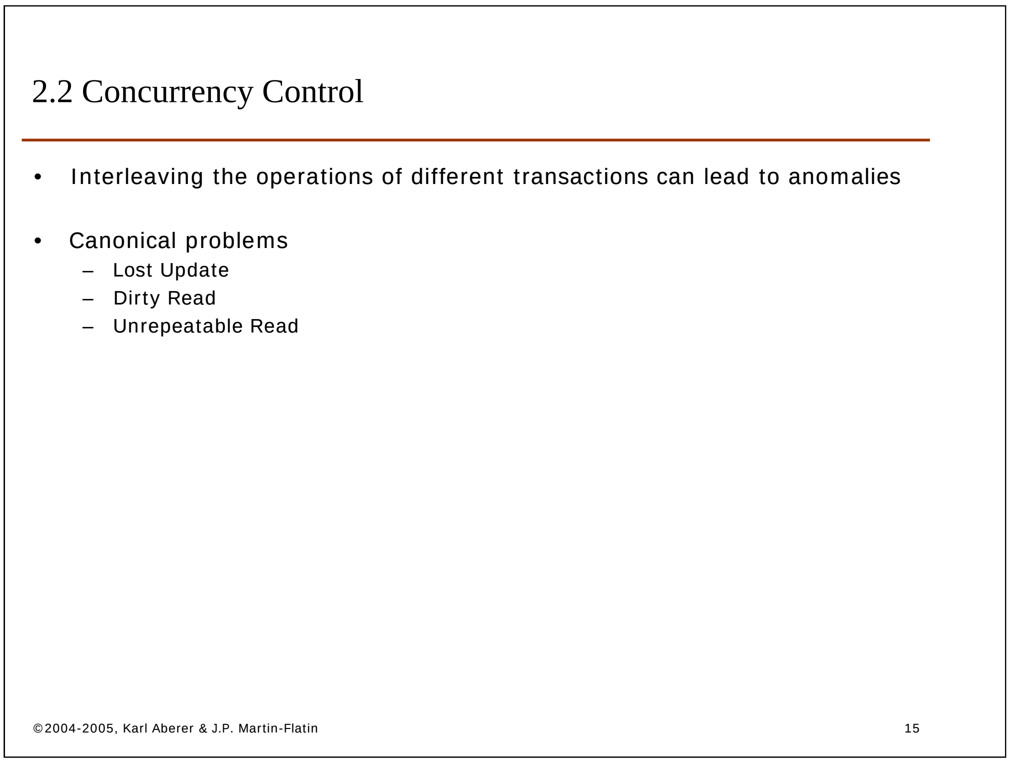
Some assumptions are made on read/write operations.

The granularity of reads/writes can be very different depending on the transaction models: examples are pages of the DBMS storage manager, objects stored in the database, like tuples, relations etc.

write() depends upon prior read() within the transaction since the values read earlier in a transaction can influence values derived by means of a computation and written to the DB

read() depends upon prior write() on same object since the values read in general depend on earlier changes of the object.

In summary, the transaction manager sees a transaction program as a sequence of operations of interest to it, which is called a **schedule**. The example shows a sample schedule of operations that originate from two different transaction programs. Note that it is perfectly possible that the operations from different transactions are interleaved. The question is whether the interleaved execution violates any of the ACID properties (in particular isolation).

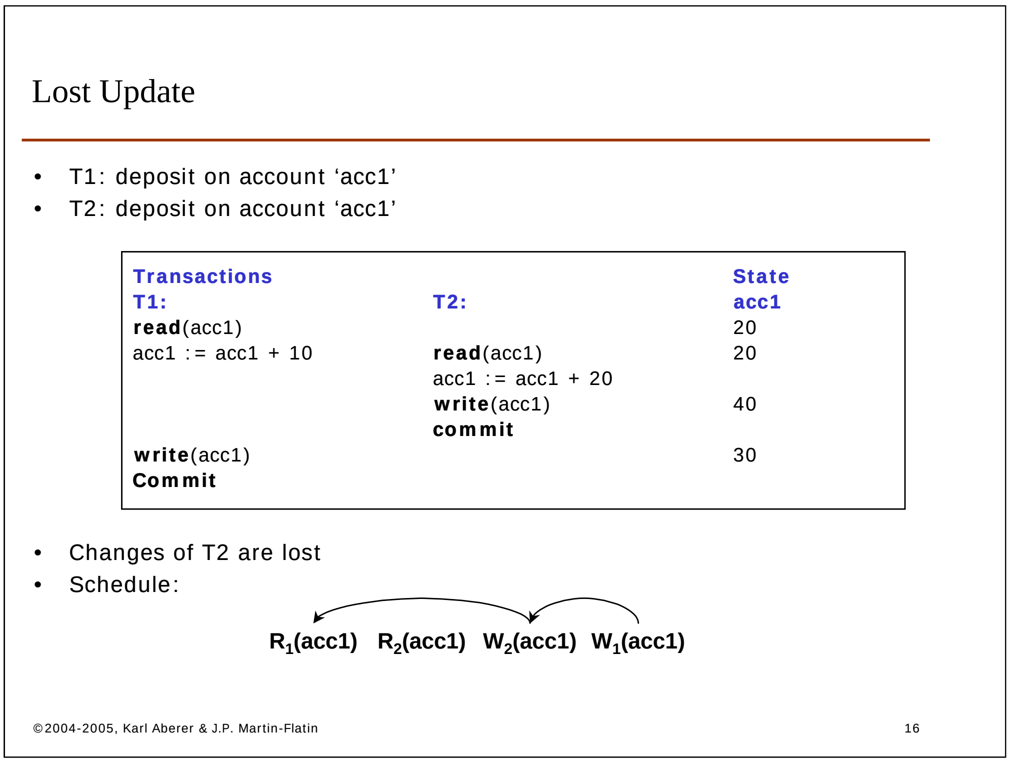


## 2.2 Concurrency Control

- Interleaving the operations of different transactions can lead to anomalies
- Canonical problems
  - Lost Update
  - Dirty Read
  - Unrepeatable Read

15

First we study the methods that are used to guarantee isolation in multi-user mode. If multiple interleaved transactions are not executed properly, undesirable effects may occur. We illustrate these by introducing three canonical examples of problems that may occur with interleaved execution of transactions. Our discussion will be based on the example transaction program for money transfer we have introduced earlier.



## Lost Update

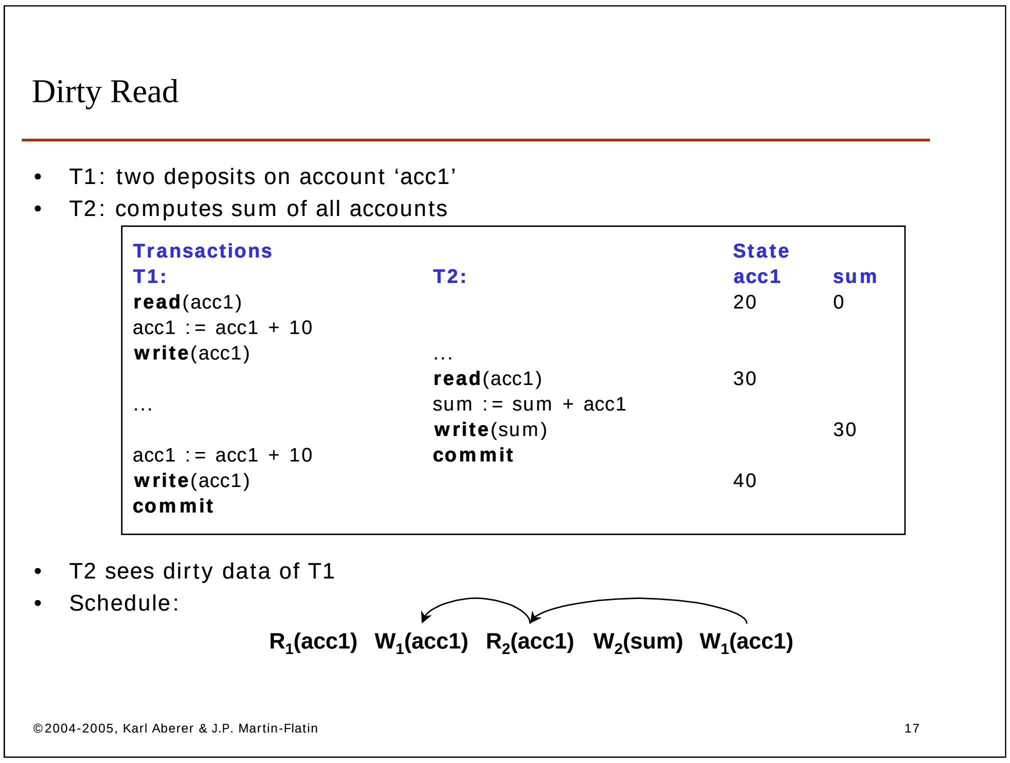
- T1: deposit on account ‘acc1’
- T2: deposit on account ‘acc1’

| **Transactions** | | **State** |
|---|---|---|
| **T1:** | **T2:** | **acc1** |
| **read**(acc1) | | 20 |
| acc1 := acc1 + 10 | **read**(acc1) | 20 |
| | acc1 := acc1 + 20 | |
| | **write**(acc1) | 40 |
| | **commit** | |
| **write**(acc1) | | 30 |
| **Commit** | | |

- Changes of T2 are lost
- Schedule:

$$\begin{tikzcd}
R_1(acc1) & R_2(acc1) & W_2(acc1) \arrow[ll, bend right] & W_1(acc1) \arrow[l, bend right]
\end{tikzcd}$$

The first problem is called lost update. It occurs when two transactions are interleaved, such that one transactions reads an object, then the other transaction writes a value to this object and commits and then the first transaction writes a value to the object, based on the earlier read access. This is illustrated in the example. Since the first transaction updates the object based on a value it has read before the second transaction wrote on it, it eliminates the effects of the second transaction. With a (semantically) correct execution acc1 would first be increased by 10 and then by 20 (or vice versa) and have a final value of 50, which is not the case with the above execution sequence of operations.



## Dirty Read

- T1: two deposits on account 'acc1'
- T2: computes sum of all accounts

| Transactions | | State | |
|---|---|---|---|
| **T1:** | **T2:** | **acc1** | **sum** |
| **read**(acc1) | | 20 | 0 |
| acc1 := acc1 + 10 | | | |
| **write**(acc1) | … | | |
| | **read**(acc1) | 30 | |
| … | sum := sum + acc1 | | |
| | **write**(sum) | | 30 |
| acc1 := acc1 + 10 | **commit** | | |
| **write**(acc1) | | 40 | |
| **commit** | | | |

- T2 sees dirty data of T1
- Schedule:

$$\begin{tikzcd}
\mathbf{R_1(acc1)} & \mathbf{W_1(acc1)} & \mathbf{R_2(acc1)} \arrow[l, bend right] & \mathbf{W_2(sum)} & \mathbf{W_1(acc1)} \arrow[ll, bend right]
\end{tikzcd}$$

Another problem occurs when one transaction reads an object while another transaction performs multiple writes on it, as shown in the example. In this way, the first transaction can read data values that do not correspond to a consistent database state: they correspond neither to the state before the execution of the second transaction, nor to the state after the execution of the second transaction.

## Unrepeatable Read

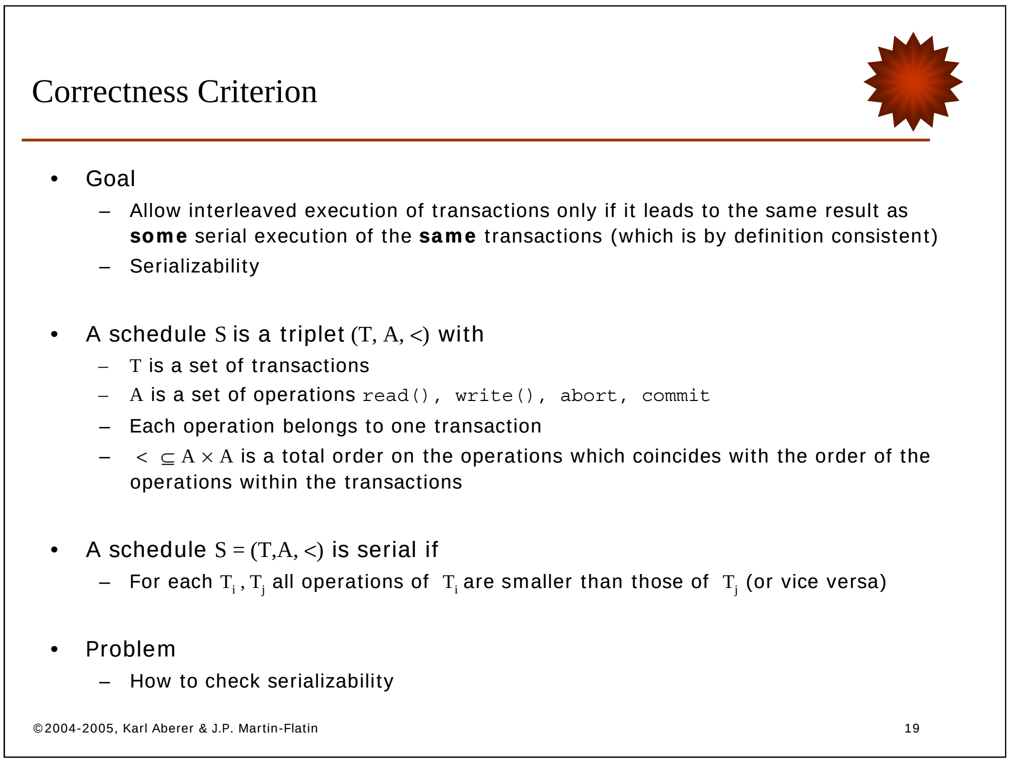
- T1: multiple read from account ‘acc1’
- T2: deposit on account ‘acc1’

| Transactions | | State |
|---|---|---|
| **T1:** | **T2:** | **acc1** |
| **read**(acc1) | | 20 |
| … | **read**(acc1) | 30 |
| | acc1 := acc1 + 20 | |
| | **write**(acc1) | 50 |
| | **commit** | |
| **read**(acc1) | | |
| sum := sum + acc1 | | |
| **write**(sum) | | 60 |
| **Commit** | | |

- T2 reads different values for acc1
- Schedule:

$$\begin{tikzcd} R_1(acc1) & R_2(acc1) & W_2(acc1) \arrow[ll, bend right] & R_1(acc1) \arrow[l, bend right] & W_1(sum) \end{tikzcd}$$

Another problem is the converse of the previous one. If one transaction performs multiple reads on the same object, and a second transaction performs in between a write on the object, it may happen that the first transaction reads two different values for the objects, even though it did not perform an update on the object. This is clearly an inconsistent behavior, since we do not expect data values to change by themselves.



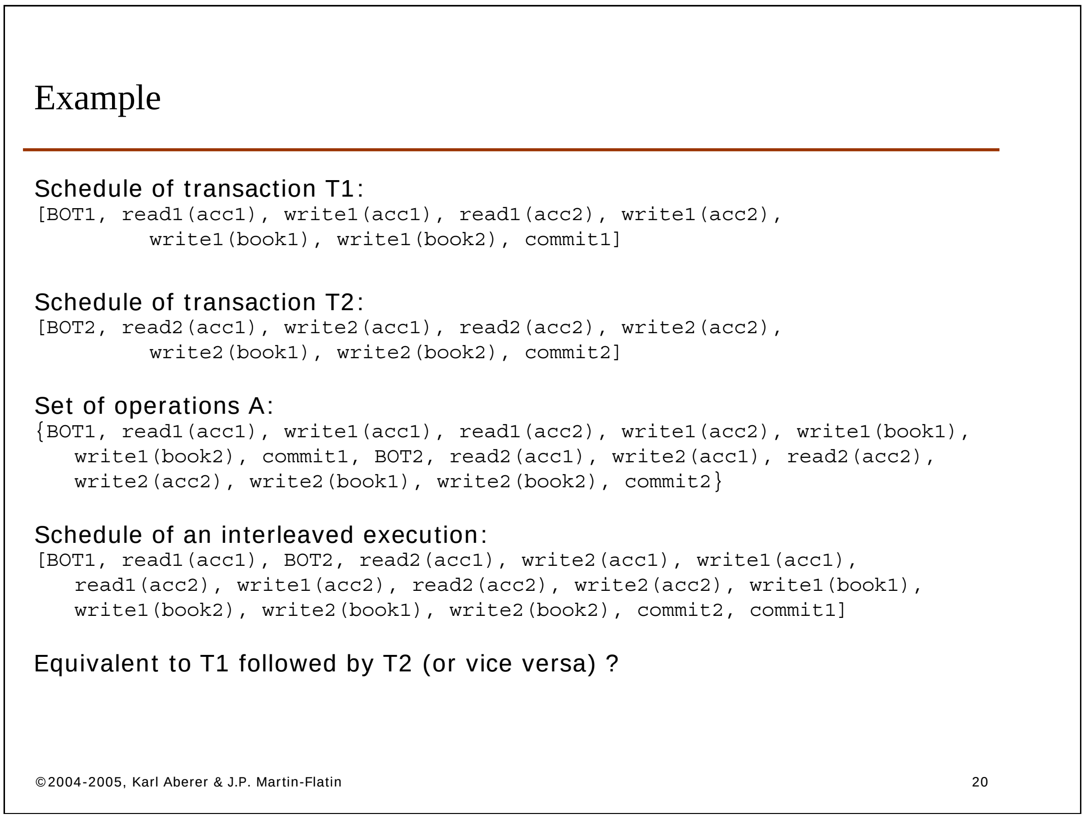
## Correctness Criterion

- Goal
  - Allow interleaved execution of transactions only if it leads to the same result as **some** serial execution of the **same** transactions (which is by definition consistent)
  - Serializability
- A schedule S is a triplet (T, A, <) with
  - T is a set of transactions
  - A is a set of operations `read(), write(), abort, commit`
  - Each operation belongs to one transaction
  - $< \subseteq A \times A$ is a total order on the operations which coincides with the order of the operations within the transactions
- A schedule S = (T,A, <) is serial if
  - For each $T_i$, $T_j$ all operations of $T_i$ are smaller than those of $T_j$ (or vice versa)
- Problem
  - How to check serializability

We do not want to prohibit interleaved executions of transactions completely (which would obviously avoid the aforementioned problems) for performance reasons. If we would allow only what is called *strictly serial executions*, it could happen that all the clients that are performing a transaction at the same time would have to wait till the current transaction has completed in order to proceed. Imagine what would happen if all the customers of a bank had to wait, while withdrawing money at an ATM, just because someone in front of an ATM has forgotten his password!

Therefore we have to find out which interleaved transaction executions or transaction schedules to admit, since not all of them are possible. The approach to answering this question is the following: allow an interleaved execution if it is equivalent to some serial execution of the same transactions (note the word "some": no specific order is required). This criterion for a schedule to be valid is called *serializability*.

The problem is now: How to decide serializability and what does "being equivalent to a serial schedule" mean?



## Example

Schedule of transaction T1:
```
[BOT1, read1(acc1), write1(acc1), read1(acc2), write1(acc2),
         write1(book1), write1(book2), commit1]
```

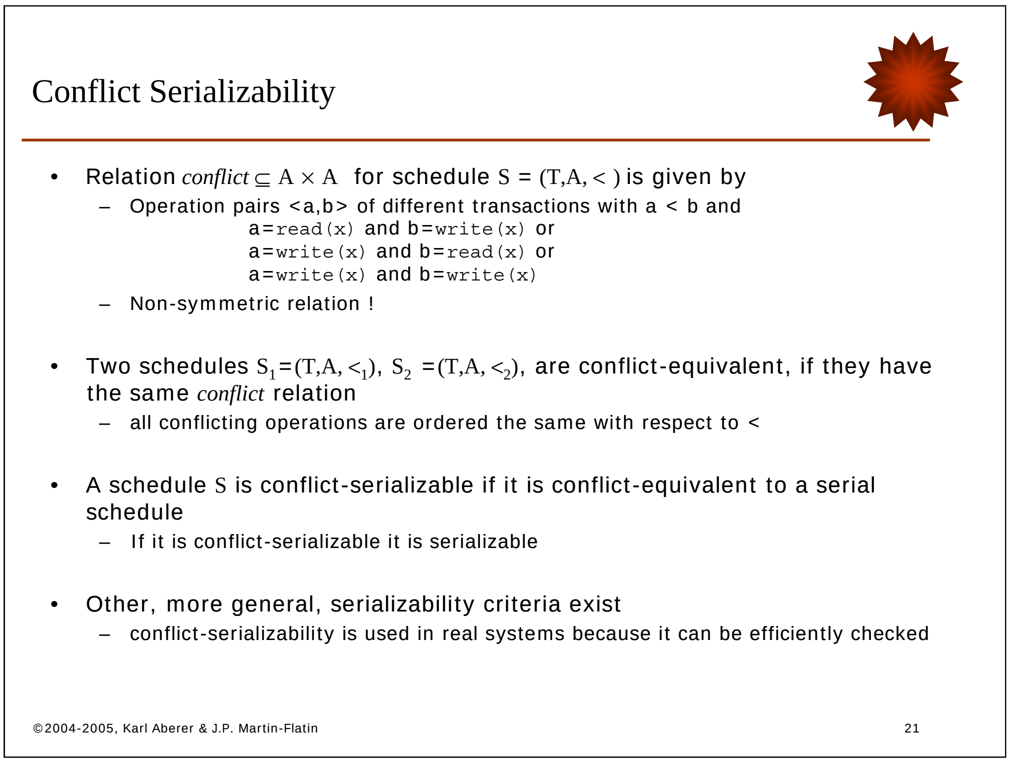
Schedule of transaction T2:
```
[BOT2, read2(acc1), write2(acc1), read2(acc2), write2(acc2),
         write2(book1), write2(book2), commit2]
```

Set of operations A:
```
{BOT1, read1(acc1), write1(acc1), read1(acc2), write1(acc2), write1(book1),
   write1(book2), commit1, BOT2, read2(acc1), write2(acc1), read2(acc2),
   write2(acc2), write2(book1), write2(book2), commit2}
```

Schedule of an interleaved execution:
```
[BOT1, read1(acc1), BOT2, read2(acc1), write2(acc1), write1(acc1),
   read1(acc2), write1(acc2), read2(acc2), write2(acc2), write1(book1),
   write1(book2), write2(book1), write2(book2), commit2, commit1]
```

Equivalent to T1 followed by T2 (or vice versa) ?

First we illustrate the notions of schedule, operation set, interleaved execution for our running example. Note that also in an interleaved execution of transactions the order of operations from the same transaction must not be changed.

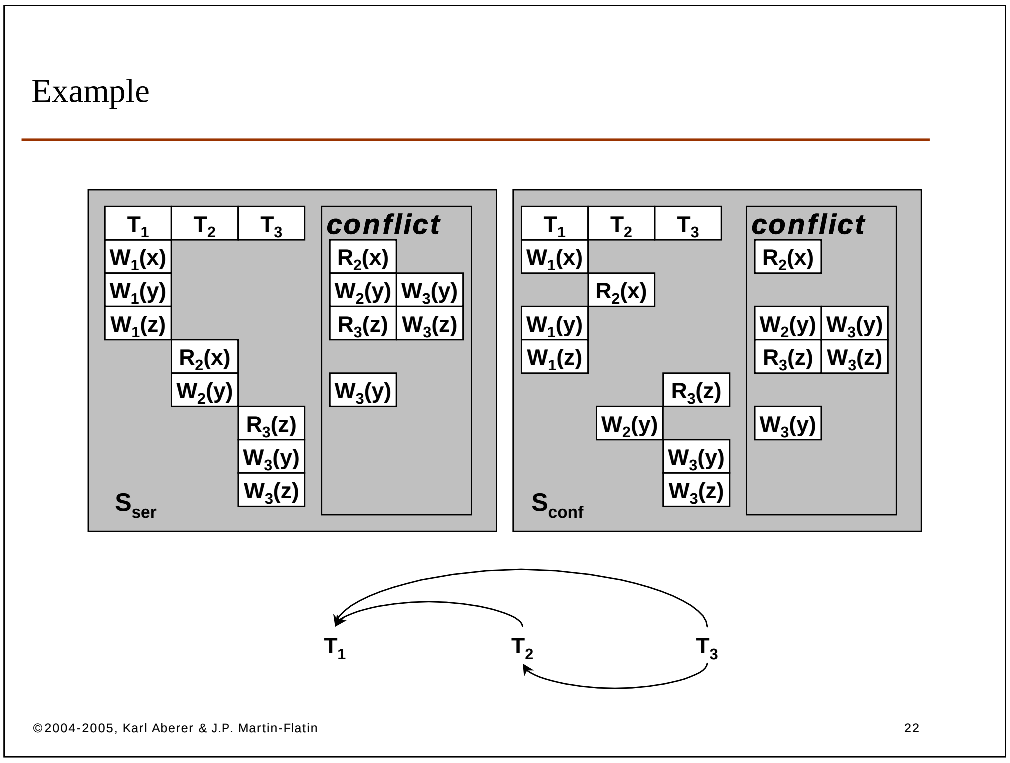


## Conflict Serializability

- Relation *conflict* $\subseteq A \times A$ for schedule $S = (T,A,<)$ is given by
  - Operation pairs <a,b> of different transactions with a < b and
    a=`read(x)` and b=`write(x)` or
    a=`write(x)` and b=`read(x)` or
    a=`write(x)` and b=`write(x)`
  - Non-symmetric relation !
- Two schedules $S_1=(T,A,<_1)$, $S_2=(T,A,<_2)$, are conflict-equivalent, if they have the same *conflict* relation
  - all conflicting operations are ordered the same with respect to <
- A schedule S is conflict-serializable if it is conflict-equivalent to a serial schedule
  - If it is conflict-serializable it is serializable
- Other, more general, serializability criteria exist
  - conflict-serializability is used in real systems because it can be efficiently checked

©2004-2005, Karl Aberer & J.P. Martin-Flatin 21

There exist different definitions on serializability, e.g. expressing that a schedule is equivalent to a serial schedule if the outcome of the two schedules is the same (view-serializability). However, these definitions are algorithmically hard to check. Therefore in practice a more restrictive, but more direct definition of serializability is used, conflict serializability. The idea is fairly simple: one considers operation pairs in a schedule and whether the operations in these pairs can be exchanged within the schedule. For example, this can be done if both operations are read-only or if the operations access different objects. In the other cases, listed above, the operations are said to be in conflict. Now, if there exists a serial schedule that has the same conflicting pairs of operations as a non-serial schedule, the non-serial schedule is said to be conflict-serializable. Intuitively, the operations of the non-serial schedule can be brought into the same order as those of the serial schedule by only exchanging the order of non-conflicting operations.



## Example

| $T_1$ | $T_2$ | $T_3$ | ***conflict*** | |
|---|---|---|---|---|
| $W_1(x)$ | | | $R_2(x)$ | |
| $W_1(y)$ | | | $W_2(y)$ | $W_3(y)$ |
| $W_1(z)$ | | | $R_3(z)$ | $W_3(z)$ |
| | $R_2(x)$ | | | |
| | $W_2(y)$ | | $W_3(y)$ | |
| | | $R_3(z)$ | | |
| | | $W_3(y)$ | | |
| | | $W_3(z)$ | | |

$S_{ser}$

| $T_1$ | $T_2$ | $T_3$ | ***conflict*** | |
|---|---|---|---|---|
| $W_1(x)$ | | | $R_2(x)$ | |
| | $R_2(x)$ | | | |
| $W_1(y)$ | | | $W_2(y)$ | $W_3(y)$ |
| $W_1(z)$ | | | $R_3(z)$ | $W_3(z)$ |
| | | $R_3(z)$ | | |
| | $W_2(y)$ | | $W_3(y)$ | |
| | | $W_3(y)$ | | |
| | | $W_3(z)$ | | |

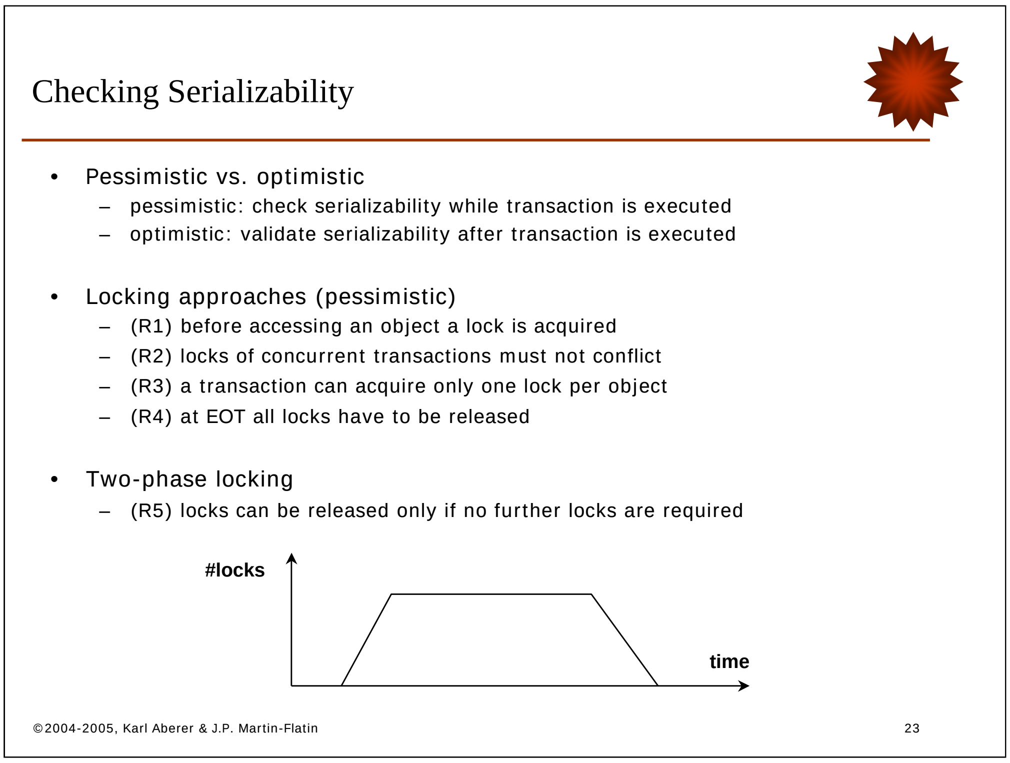
$S_{conf}$

$$\begin{tikzcd}
T_1 & T_2 \arrow[l, bend right] & T_3 \arrow[ll, bend right] \arrow[l, bend left]
\end{tikzcd}$$

This example illustrates conflict-serializability: we have three transactions accessing objects x, y, and z. The first schedule (left hand side) is serial. Under heading "Conflict", we denote for each operation of the transaction which are the conflicting operations in the other transactions that follow. That is, these are all the operations executed later and accessing the same object, where the operation pair is R/W, W/R or W/W.

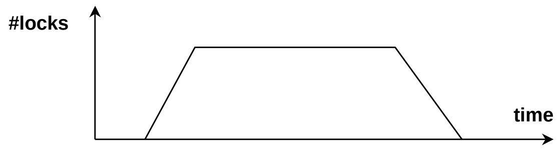
The second schedule (right hand side) is non-serial. Again we list all the conflicts. By comparing the two conflict relations, we observe that exactly the same conflicts occur and therefore the right schedule is serializable.

An algorithmic way to determine serializability for a given schedule is by investigating the so-called *conflict graph*. The nodes of the conflict graph are the transactions. We add a directed edge whenever there exists a conflict relation among two operations from the two corresponding transactions (remember: the conflict relation is asymmetric, therefore the edge is directed). If the resulting graph is acyclic, the schedule is serializable, because we can bring the transactions into a serial order such that the conflict graph is the same, and therefore the same conflicts hold.



## Checking Serializability

- Pessimistic vs. optimistic
  - pessimistic: check serializability while transaction is executed
  - optimistic: validate serializability after transaction is executed
- Locking approaches (pessimistic)
  - (R1) before accessing an object a lock is acquired
  - (R2) locks of concurrent transactions must not conflict
  - (R3) a transaction can acquire only one lock per object
  - (R4) at EOT all locks have to be released
- Two-phase locking
  - (R5) locks can be released only if no further locks are required



23

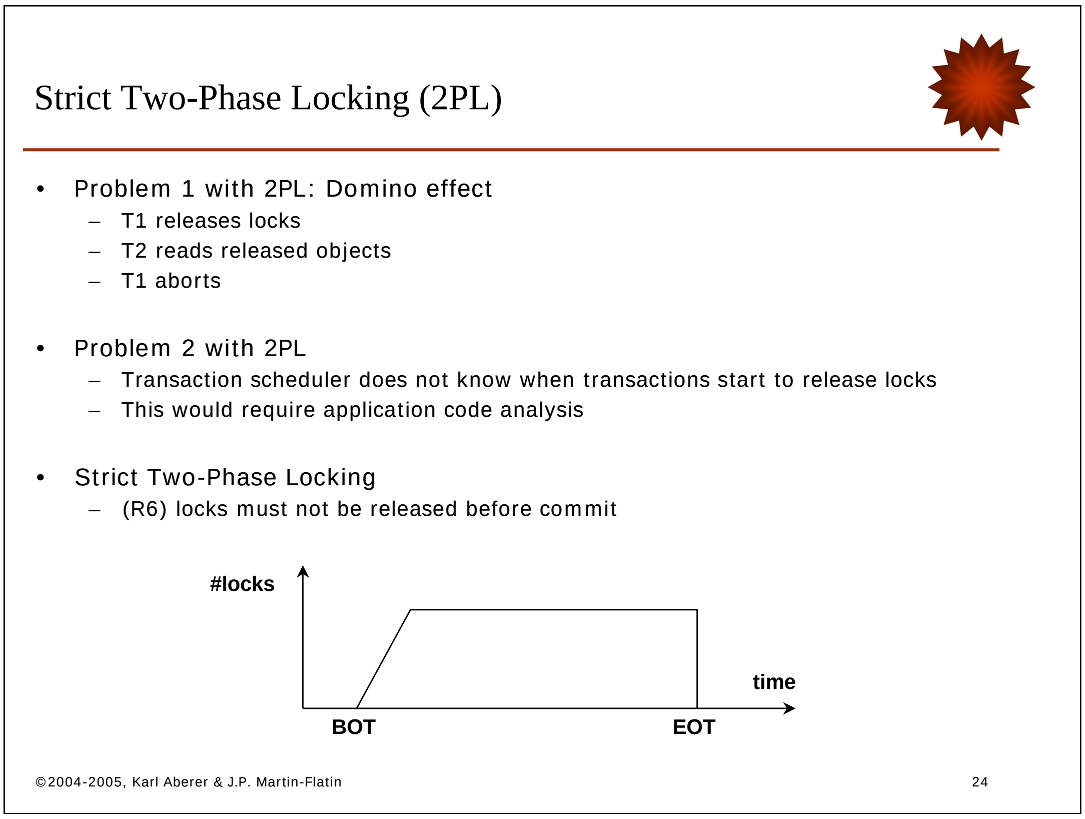
For checking conflict serializability, we can normally not use the approach described before using a conflict graph, because the schedule is not known a priori. Therefore in practice different approaches are used: one distinguishes between pessimistic and optimistic approaches.

The pessimistic approaches try to avoid any non-serializable schedules from the very beginning. The idea is to use locks on objects when a transactions accesses them in order to avoid accesses of other transactions that could create conflicts.

The optimistic approaches validate after the transactions have been executed whether the execution schedule was serial. One possible implementation of an optimistic approach is to assign to each transaction a timestamp at its start, and each transaction creates new object versions when writing. When the transactions are ended, the serializability is checked and if not given the transaction is aborted and the new versions are abandoned.

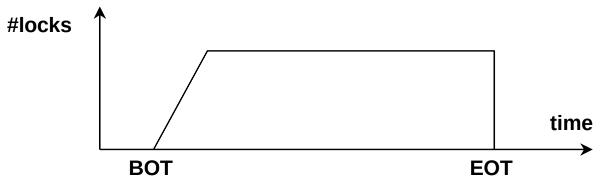
On the slide, a more detailed description of the basic locking approach is given. Rules R1-R4 implement the basic locking principle. In practice, not all locks are of the same type; we distinguish at least between locks for reading and writing, so that when one transaction maintains a read lock other transactions can still read the value of the object (but not write on it). This exploits idempotence of read accesses.

Two-phase locking imposes a further requirement on the acquisition and release of locks: locks can only be released if no further locks are required. It can be proved that together with this condition, the resulting schedules are always conflict-serializable.



## Strict Two-Phase Locking (2PL)

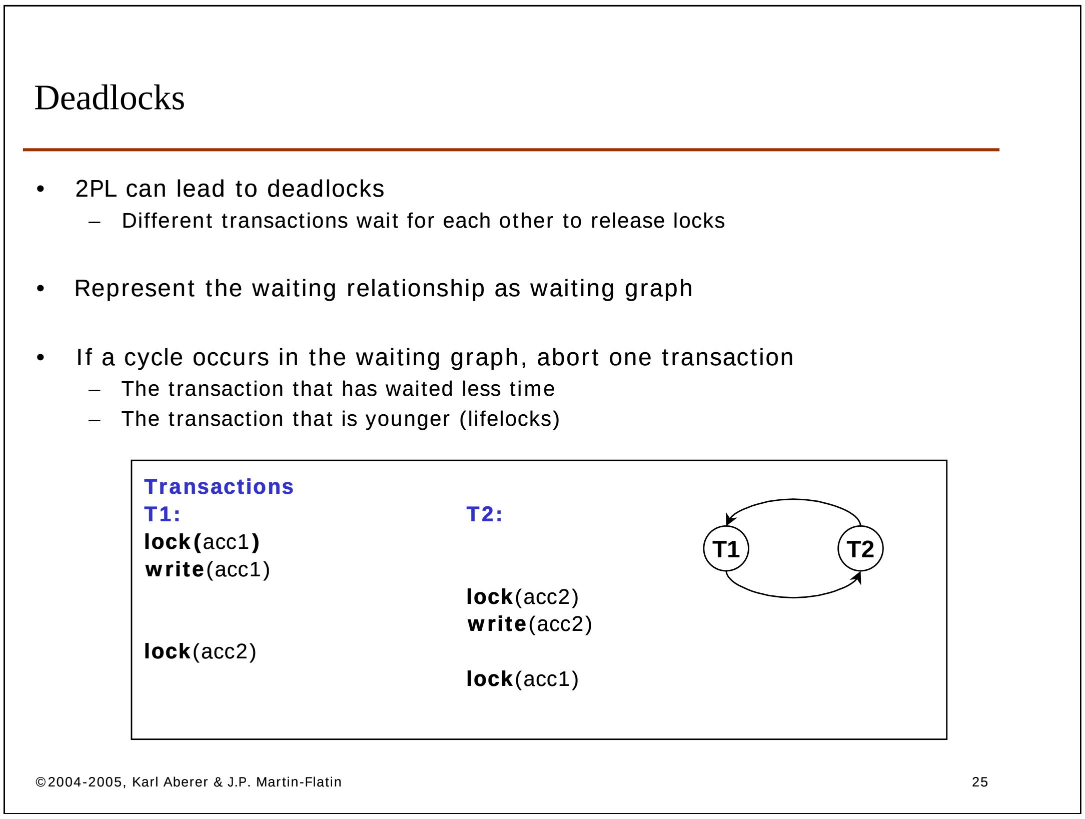
- Problem 1 with 2PL: Domino effect
  - T1 releases locks
  - T2 reads released objects
  - T1 aborts
- Problem 2 with 2PL
  - Transaction scheduler does not know when transactions start to release locks
  - This would require application code analysis
- Strict Two-Phase Locking
  - (R6) locks must not be released before commit



With the locking protocol described so far, we still have practical problems. The first one is the so-called *Domino effect*, which leads to *cascading aborts*. It can happen that a transaction releases a lock before it ends, and in the meantime another transaction acquires the lock on the corresponding object. Now it may happen that after this the first transaction aborts, and consequently also the second transaction has to be aborted as its results are based on a non-committed database state.

The second problem is that a transaction scheduler does not know in practice when the shrinking phase can start. This would require to know all future operations that are to be performed within the transaction, and therefore some code analysis or additional specification by the programmer would be required.

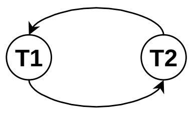
Therefore, in practice, a stricter protocol is used for locking: the strict two-phase locking protocol, which allows locks to be released only when the transaction commits.



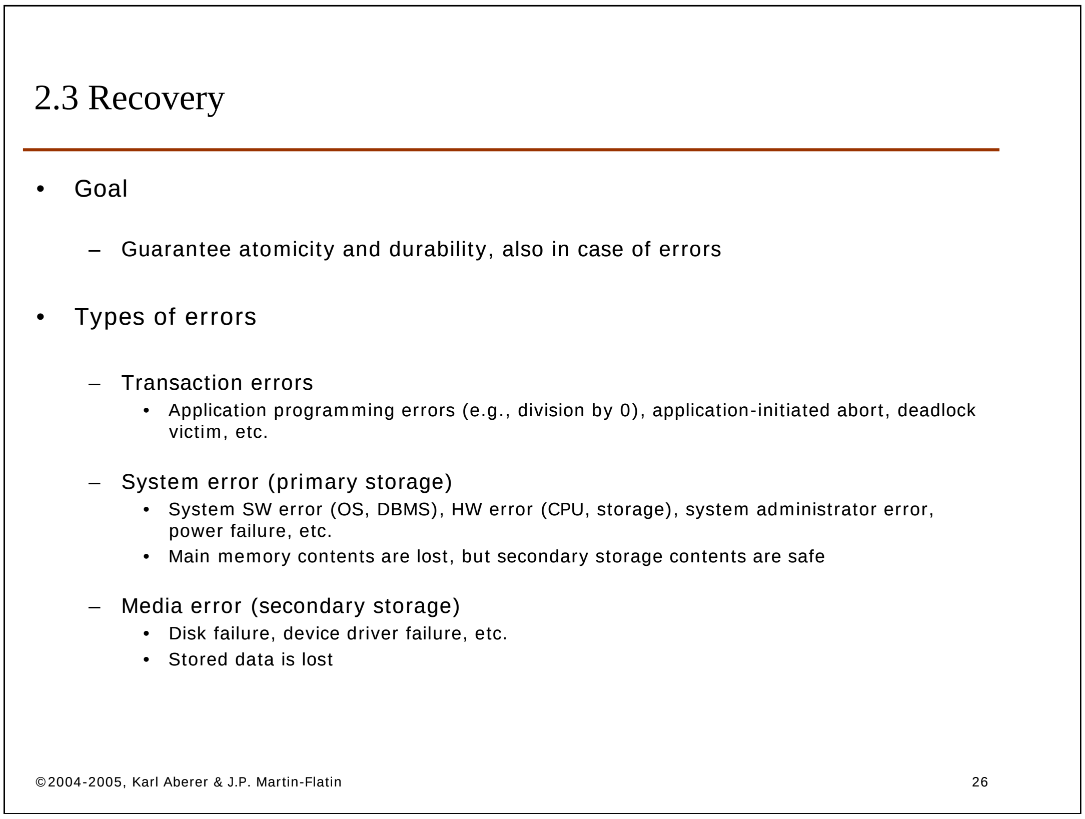
## Deadlocks

- 2PL can lead to deadlocks
  - Different transactions wait for each other to release locks
- Represent the waiting relationship as waiting graph
- If a cycle occurs in the waiting graph, abort one transaction
  - The transaction that has waited less time
  - The transaction that is younger (lifelocks)

```
Transactions
T1:                          T2:
lock(acc1)
write(acc1)
                             lock(acc2)
                             write(acc2)
lock(acc2)
                             lock(acc1)
```



When transactions acquire locks, it may happen that they run into a deadlock situation: transaction 1 requires a lock that is held by transaction 2 in order to proceed, and vice versa for transaction 2. Therefore a transaction scheduler has to maintain a waiting graph, where it keeps the information which transaction is waiting for a lock on an object that is held by another transaction. If the graph contains cycles, a deadlock situation occurs. One way to resolve this is to abort one of the two transactions in the cycle. Deciding which transaction to abort is one of the specificities of different transaction scheduling methods.



## 2.3 Recovery

- Goal
  - Guarantee atomicity and durability, also in case of errors
- Types of errors
  - Transaction errors
    - Application programming errors (e.g., division by 0), application-initiated abort, deadlock victim, etc.
  - System error (primary storage)
    - System SW error (OS, DBMS), HW error (CPU, storage), system administrator error, power failure, etc.
    - Main memory contents are lost, but secondary storage contents are safe
  - Media error (secondary storage)
    - Disk failure, device driver failure, etc.
    - Stored data is lost

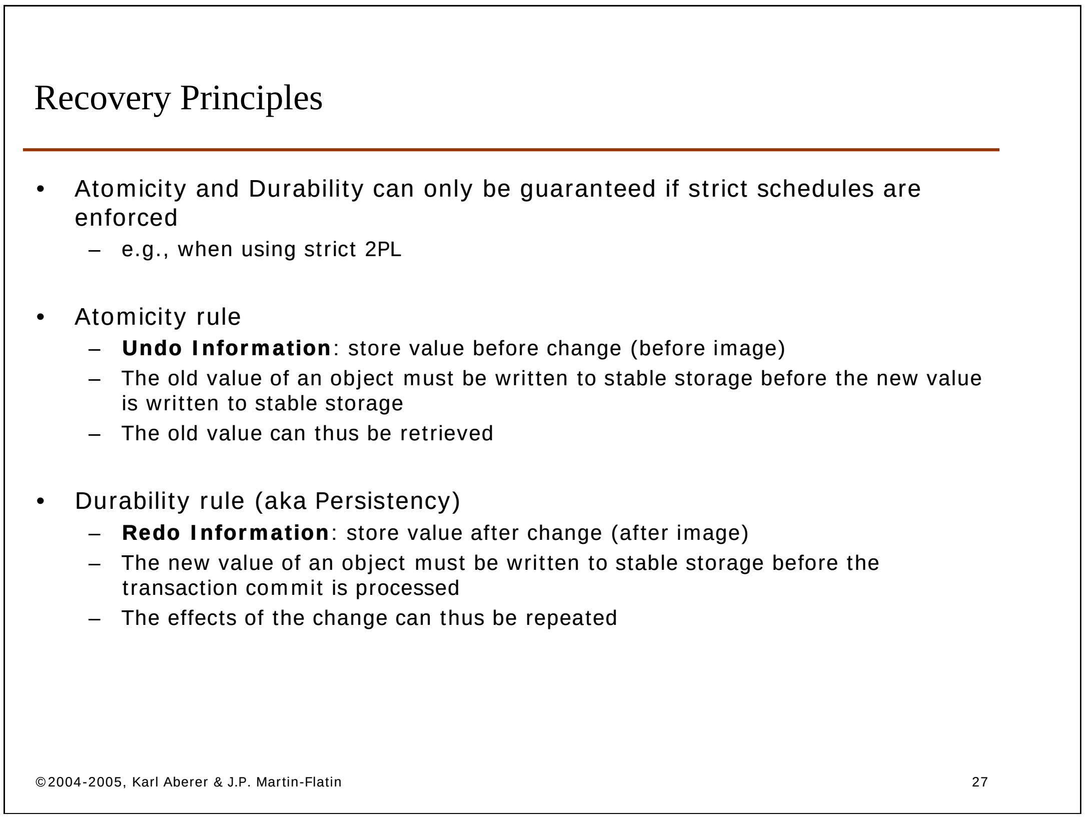
26

After dealing with the issues of synchronizing multi-user access and supporting isolation, we turn our attention now to the problem of guaranteeing atomicity and durability even in case of errors. In an information system errors of different kinds may occur, ranging from transaction errors, over system errors, to media errors. They are distinguished according to which part of the system is affected and thus which actions have to be taken in order to recover from the error.

With transaction errors, the situation is comparably simple because the only thing that is affected is the transaction program itself. Since the transaction manager makes durable only transactions that are committed, these problems are dealt with by processing the transaction abort.

With system errors, the main memory contents are lost, which can create more problems because information that is relevant for the correct processing of transactions may have been kept in the main memory (e.g., data that was written to the database buffer but not yet to the stable database). This problem and how to deal with it will be subsequently the focus of our discussion.

With media errors, the contents of the secondary storage are also lost. In such a case, the database state has to be recovered from archive copies (e.g., tapes).



## Recovery Principles

- Atomicity and Durability can only be guaranteed if strict schedules are enforced
  - e.g., when using strict 2PL
- Atomicity rule
  - **Undo Information**: store value before change (before image)
  - The old value of an object must be written to stable storage before the new value is written to stable storage
  - The old value can thus be retrieved
- Durability rule (aka Persistency)
  - **Redo Information**: store value after change (after image)
  - The new value of an object must be written to stable storage before the transaction commit is processed
  - The effects of the change can thus be repeated

27

In order to enable the kind of recovery described before (in case of system errors), certain things are required. A theoretical result says that atomicity and durability can only be guaranteed (together with isolation) if strict schedules are enforced, which is the case for instance when using strict 2PL. This is another reason for using this locking protocol in practice.

Given that, certain rules need to be obeyed to ensure that the undo and redo operations as described before can actually be performed. With respect to atomicity (i.e., execute all operations of a transaction or none), before a value is written to stable storage (database), the old value (the before image) must be stored in stable storage (in some other place). This makes it possible to always undo the effect of the write operation.

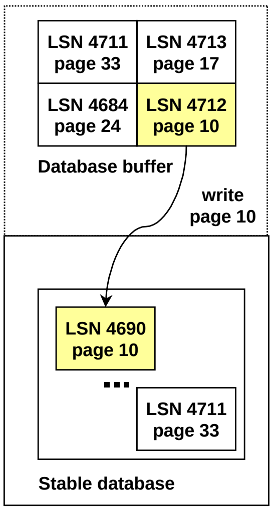
Similarly, for durability, in order to ensure that a committed change cannot be lost, the new value of an object that is committed (after image) needs to be stored in stable storage before committing the transaction. Thus the effects of the change can always be repeated.

## Error Handling

- Operation of a DBMS
  - Database buffer in primary storage (volatile storage)
  - Database in secondary storage (stable storage)

*Handling the different error types:*

- Transaction errors
  - Abort the transaction programs while DBMS is running
- System errors
  - Reboot required (crash recovery)
  - **Undo Phase**: all changes of incomplete transactions that are in stable storage are undone
  - **Redo Phase** : all changes of completed transactions that are only in volatile storage are redone
- Media errors
  - Archive Recovery: recreate database state by starting from
  - an archive copy
  - **Redo Phase**: all changes of completed transactions since the creation of the last archive copy are redone

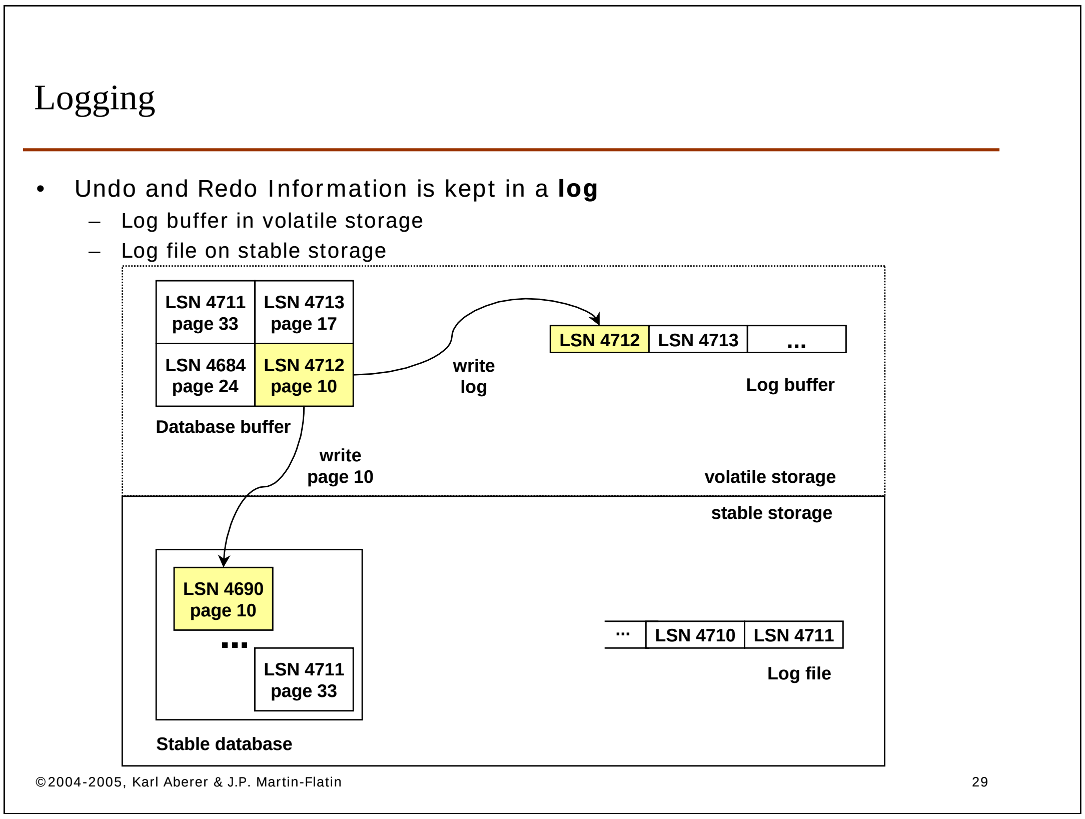


For understanding of how errors are handled it is important to understand the operation of a DBMS. The most important observation is that a DBMS maintains data both in the stable storage and in the primary storage (database buffer, for efficiency reasons). Since different types of errors affect these different stores differently a DBMS treats for recovery each of these system parts differently.

Handling transaction errors is straightforward since we exploit the existing mechanism of aborting a transaction that is supported by the transaction processing.

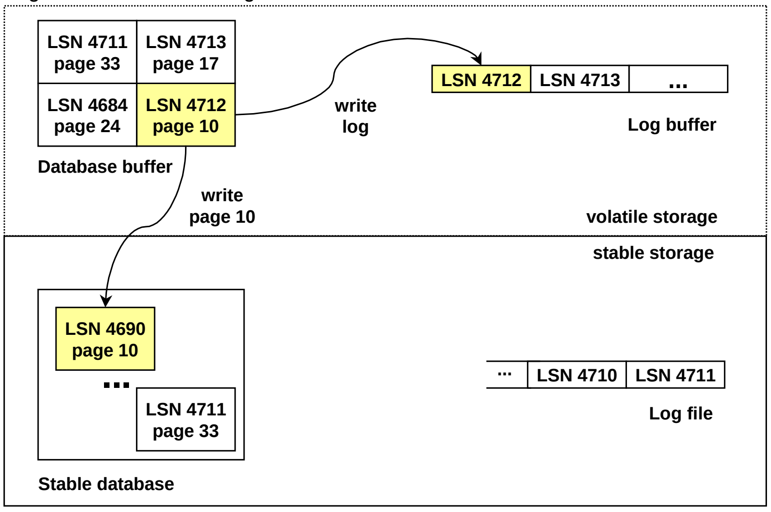
With system errors, the computer system is rebooted. In order to bring the database into a consistent state, the recovery manager has to correct inconsistencies among the current actual state of the stable database and the correct state of the DBMS at the time where the error has occured. These inconsistencies occur as a consequence of the loss of main memory that can contain part of the database state. Two things may happen: first, transactions that have not committed already might have written their results into the stable database. These effects have to be corrected by undoing the changes. Second, transactions may have committed, but not yet written their results into the stable storage. Durability says that the changes of all committed transactions have to be persistent, and therefore these changes must be introduced into the stable database in a redo phase.

For media failures and archive recovery the situation is insofar simpler, as changes of incomplete transactions to the stable storage are lost anyway, and therefore no undo phase is required when restoring the database state from an archive copy (of course the redo phase will be substantially more costly)



## Logging

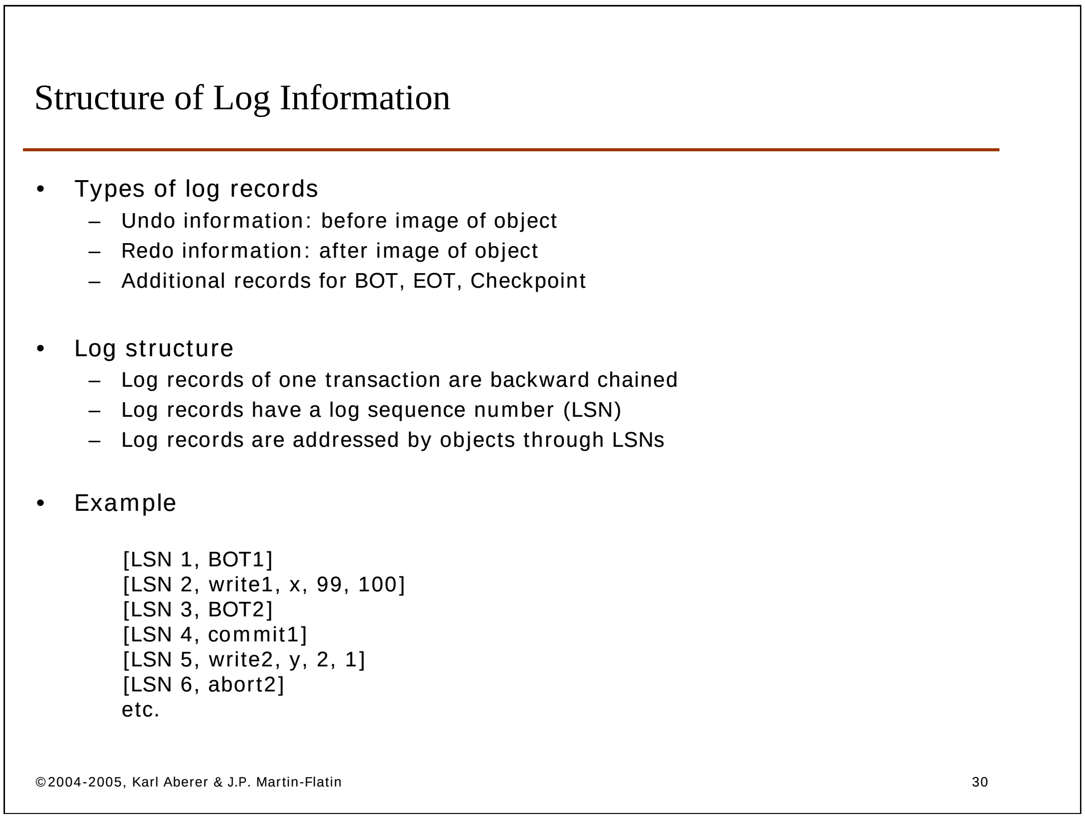
- Undo and Redo Information is kept in a **log**
  - Log buffer in volatile storage
  - Log file on stable storage



29

The question is now where the before and after images are stored. A standard solution to this is the use of logs. In addition to the database in the stable storage and the database buffer in the volatile storage (on the left side of the figure) the recovery manager maintains a log file on the stable storage and a log buffer in the volatile storage. The log buffer is used to record any activity that is done on the database. For example, when writing to a database page the following is happening:

The page is loaded from the stable database in the database buffer and the write operation is performed on the page. In parallel a log entry with the before and after image is put into the log buffer. Now before the changed page would be written to stable storage, one can write the log entry to the log file, and thus ensure that a before image is available for undo. Similarly the buffer is used for commits. We will discuss this in more detail in the following.



## Structure of Log Information

- Types of log records
  - Undo information: before image of object
  - Redo information: after image of object
  - Additional records for BOT, EOT, Checkpoint

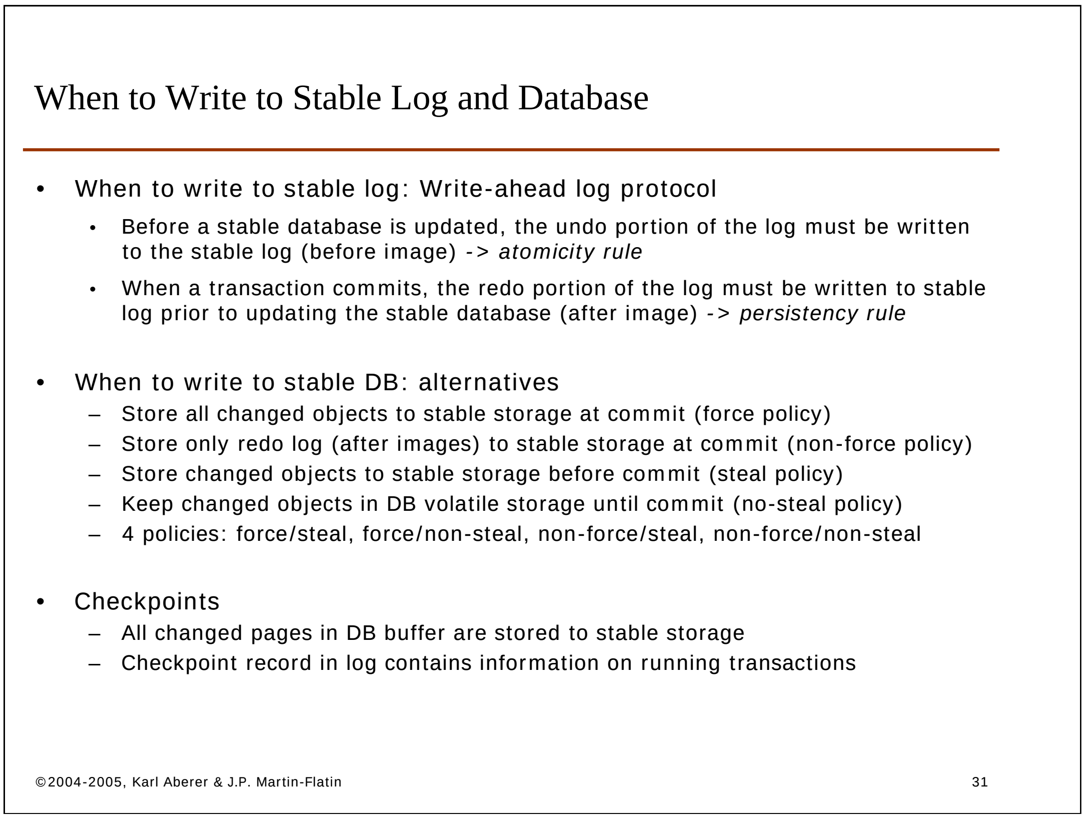
- Log structure
  - Log records of one transaction are backward chained
  - Log records have a log sequence number (LSN)
  - Log records are addressed by objects through LSNs

- Example

```
[LSN 1, BOT1]
[LSN 2, write1, x, 99, 100]
[LSN 3, BOT2]
[LSN 4, commit1]
[LSN 5, write2, y, 2, 1]
[LSN 6, abort2]
etc.
```

The log entries either log write operations and keep then the before and after image of the object or log other transaction-relevant operations, such as begin and end of transactions or the creation of checkpoints. The log entries are backward chained, such that the logs can be sequentially traversed, and each log entry has a unique identifier, the so-called log sequence number.

The example shows a few possible log entries. The first field is the log sequence number. For write log entries, the second field indicates which is the transaction, the third field references the object, the fourth field is the before image and the fifth field is the after image (this is of course a highly simplified presentation, for the sake of clarity).



## When to Write to Stable Log and Database

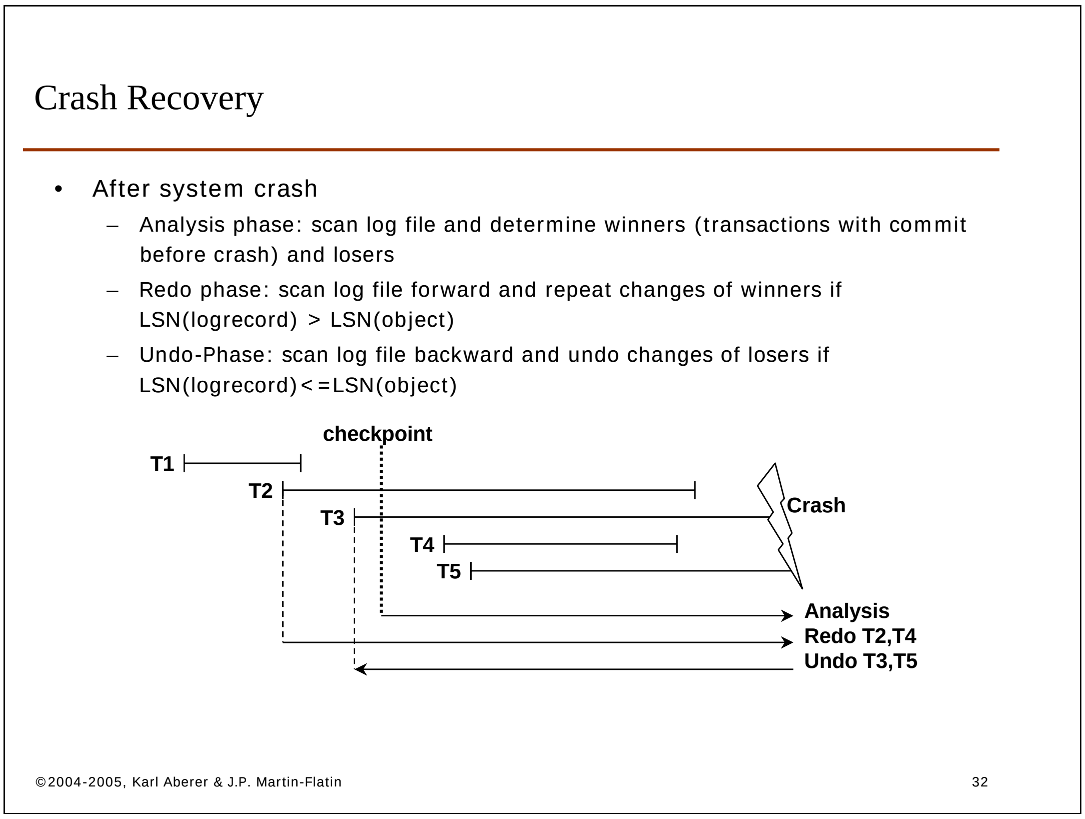
- When to write to stable log: Write-ahead log protocol
  - Before a stable database is updated, the undo portion of the log must be written to the stable log (before image) -> *atomicity rule*
  - When a transaction commits, the redo portion of the log must be written to stable log prior to updating the stable database (after image) -> *persistency rule*
- When to write to stable DB: alternatives
  - Store all changed objects to stable storage at commit (force policy)
  - Store only redo log (after images) to stable storage at commit (non-force policy)
  - Store changed objects to stable storage before commit (steal policy)
  - Keep changed objects in DB volatile storage until commit (no-steal policy)
  - 4 policies: force/steal, force/non-steal, non-force/steal, non-force/non-steal
- Checkpoints
  - All changed pages in DB buffer are stored to stable storage
  - Checkpoint record in log contains information on running transactions

Now that we have the possibility to log information, the atomicity and durability rules can be translated into the *write-ahead log protocol*, which states that (i) the before images are always written to the log before an update is written to the stable database, and (ii) after images are always written to the log before a transaction is committed.

There is some flexibility, however, as to when to write the data from the database buffer to the stable database. This step is not necessarily connected to the fact that a transaction commits, and having additional flexibility at this point can greatly increase the efficiency of a DBMS.

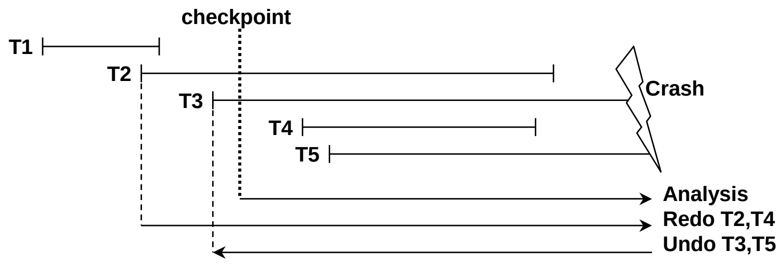
The first distinction to be made is whether the writing of the data is forced by the commit (force/non-force); the second distinction is whether the buffer manager can write to the stable database ahead of committing the transaction (steal/no-steal). Any combination of the policies is possible.

In addition, we also have a special form of writing to the database, known as *checkpointing*. At a well-defined point in time, *all* changed buffer pages are written to the database. This can save a lot of effort when it is later required to restore the state of the database in case of error.



## Crash Recovery

- After system crash
  - Analysis phase: scan log file and determine winners (transactions with commit before crash) and losers
  - Redo phase: scan log file forward and repeat changes of winners if LSN(logrecord) > LSN(object)
  - Undo-Phase: scan log file backward and undo changes of losers if LSN(logrecord)<=LSN(object)

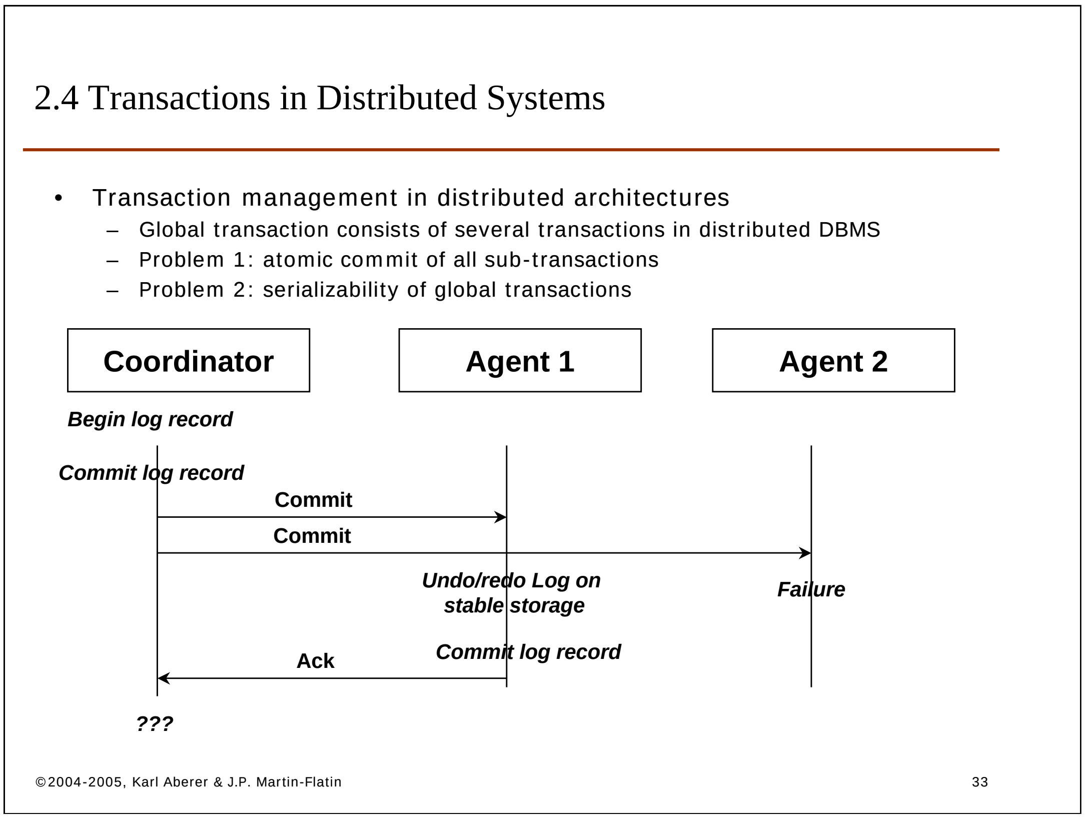


Having taken all the measures to ensure recoverability, there remains the question how to actually perform recovery. This procedure has to analyze the state of the log file in order to determine which changes need to be applied to the stable database in order to reach the state that it had at the time of the crash. For doing so a specific order of the processing is required.

First, in an analysis phase it is determined which transactions where already committed at the time of the crash and which not. This can be done by scanning the log file for end of transaction entries for those transactions that appear in it (with writes – others are not of relevance). This also needs only to be done for log entries after the last checkpoint has been saved (this is a reason why checkpointing saves work during recovery).

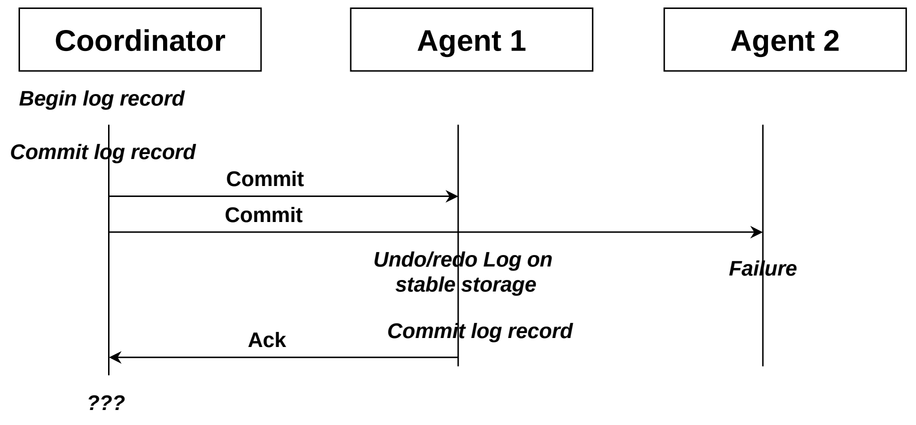
Second, in the redo phase the changes of winners are redone (written to stable storage), if needed. The redo is always necessary if the LSN of the log record is larger than the one of the entry in the database, which means that the change has not been written to the stable storage yet. This has to be done in the order the updates have been performed on the database originally.

Third in an undo phase, changes of losers that have been already written to the stable storage need to be undone. This is always required if the LSN of the log record is smaller or equal than the LSN of the object in the database, which means that the update of the log record has been written to the database. This has to be done in the inverse order the changes have been written to the stable database.



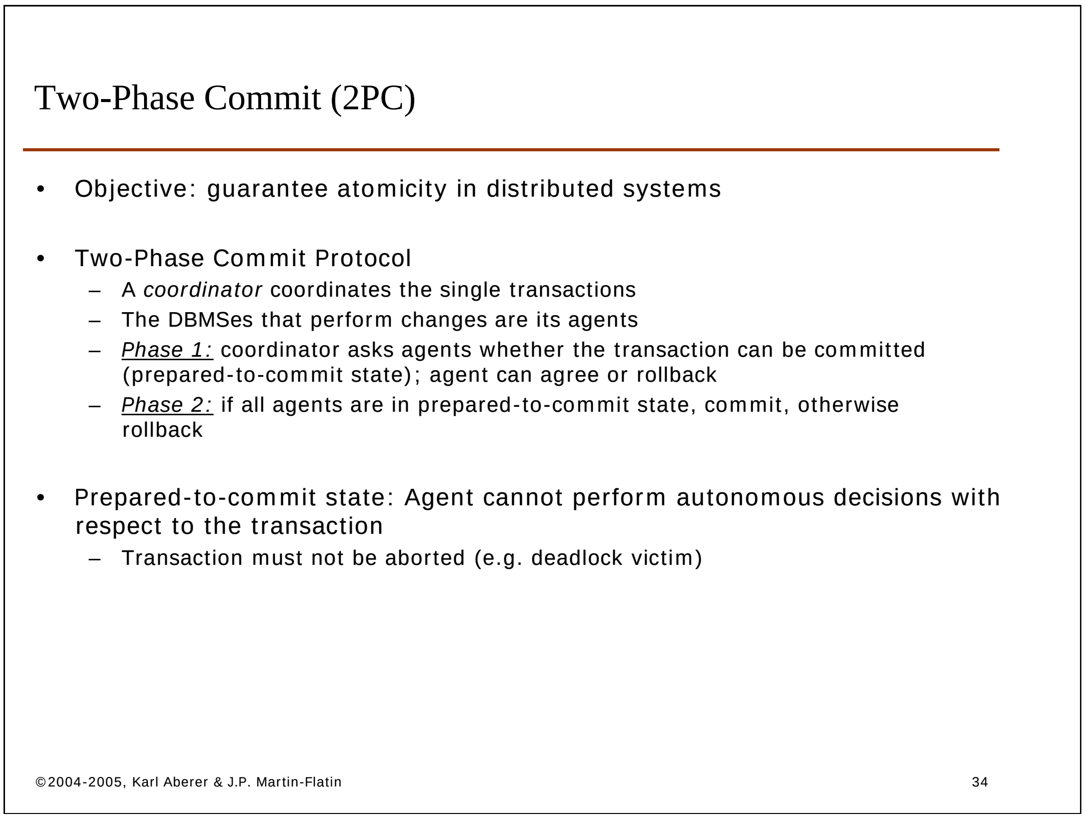
## 2.4 Transactions in Distributed Systems

- Transaction management in distributed architectures
  - Global transaction consists of several transactions in distributed DBMS
  - Problem 1: atomic commit of all sub-transactions
  - Problem 2: serializability of global transactions



In a distributed environment we encounter accesses to databases that are issued by some coordinator and are executed at different sites (agent 1 and 2) which are managed independently (autonomously). Also for distributed transactions the ACID properties have to be guaranteed, i.e. atomic commitment (and recoverability) and serializability (and synchronization) are to be supported. In the following we will just look at the first problem, atomicity. For the second problem a possible solution is given in the appendix.

What is the problem when executing a distributed transaction with respect to atomicity when the transaction control is performed by different agents autonomously ? Assume a case where a coordinator, which is accessing multiple databases, issues a transaction that is executed at two agents. Assume after all the write operations have been executed as specified by the global transaction the coordinator would like to commit the transaction. The coordinator would write a commit log record in its own log and send a message to the agents that they should also commit the transactions. Agent 1 would in response also commit its local transaction, by writing the log information and a commit record to the stable storage. Thus the changes of the global transaction that affect the data at agent 1 are made durable, and can no longer by undone. Agent 1 also informs the coordinator that it has done so. However, assume agent 2 fails and does not respond at all. Now the coordinator has a big problem. It cannot undo the transaction, which would be the only possibility since it is not able to complete all operations of the transaction due to failure of agent 2, since agent 1 has already committed the transaction. Thus as a result the transaction is not executed atomically.



## Two-Phase Commit (2PC)

- Objective: guarantee atomicity in distributed systems
- Two-Phase Commit Protocol
  - A *coordinator* coordinates the single transactions
  - The DBMSes that perform changes are its agents
  - *Phase 1:* coordinator asks agents whether the transaction can be committed (prepared-to-commit state); agent can agree or rollback
  - *Phase 2:* if all agents are in prepared-to-commit state, commit, otherwise rollback
- Prepared-to-commit state: Agent cannot perform autonomous decisions with respect to the transaction
  - Transaction must not be aborted (e.g. deadlock victim)

34

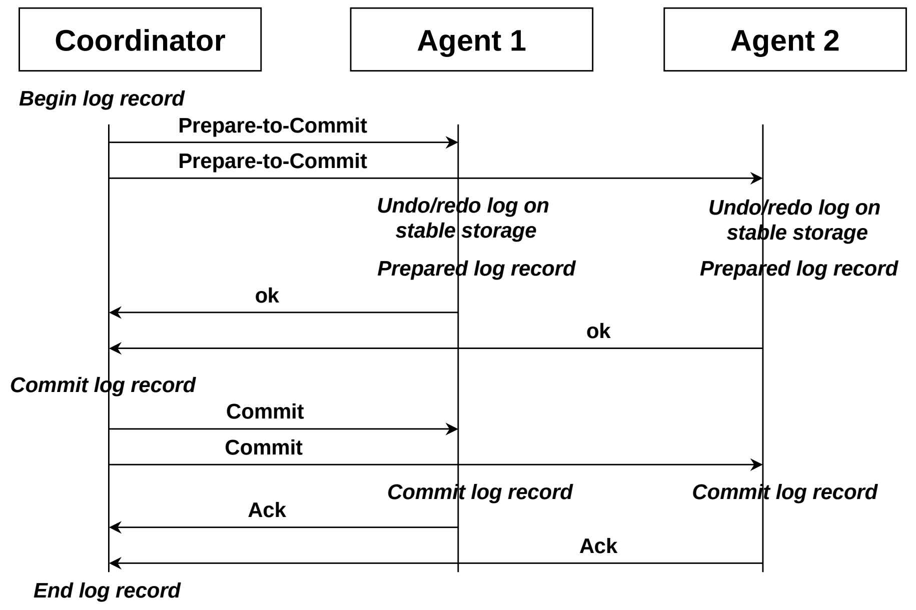
The solution to the problem described above is the 2PC protocol. It introduces a safeguard for the coordinator that ensures that it can always undo a transaction in case of failure. The protocol requires however that the agents cooperate, thus they have to restrict their autonomy.

The protocol proceeds in two phases. In the first phase, when intending to commit, the coordinator asks all participating agents whether they are ready to commit (prepared-to-commit). "Being ready to commit" means that the agent can either commit the transaction or rollback (undo the effects of the transaction). This is achieved by writing the necessary log information on stable storage. In this phase, an agent can choose to answer that it wishes to abort its transaction, in which case the coordinator must abort the global transaction.

After all agents have confirmed that they are prepared to commit, the coordinator (which is the only one to know about this fact) sends a message to all agents that they should commit. At this stage, the agents have no more choice and must commit their local transactions. This is not a problem because they have now taken measures to ensure their ability to commit also in case of failure.

The second phase limits the autonomy of the agents, since after having confirmed that they are prepared to commit, they can no longer autonomously decide on the execution of the transaction. For example, they are not allowed to abort the transaction because they chose it as a deadlock victim within their local transaction management system.

## Two-Phase Commit



35

This figure gives an overview of the messages exchanged and log records written to stable storage in the 2PC protocol in case of a successful execution of the commit processing.

## References

- Books
  - Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1992.
  - P.A. Bernstein, E. Newcomer: Principles of Transaction Processing, Morgan Kauffman, 1997.
  - G. Weikum, G. Vossen: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control, Morgan Kauffman, 2001.