(Expl.1b) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains its own server. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction, where both involve the same accounts. Transactions shall have access to accounts on remote server via remote procedure calls (RPC), a synchronous communication mechanism transparent to the programmer. We assume procedures $withdraw(account, amount)$, $deposit(account, amount)$ and $getBalance(account)$.

A possible interleaving when both transactions are running in parallel.

$Branch1(accountA)$	$Branch2(accountB)$	A	B
		50	50
T_1 : withdraw(A,10)			
T_1 : call(deposit(B,10))			
	T_2 : getBalance(B) 50	50	
	T_2 : call(getBalance(A))		
	T_1 : deposit(B,10)	40	60
T_2 : getBalance(A) 40			
	T_2 : display A+B 90	100	

time ↓

An incorrect balance will be displayed!

(Expl.1c) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains it own servers. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction is started, where both involve the same accounts. Finally assume, that each transaction implements exclusive access to both accounts during execution. Communication is explicitly implemented by exchanging messages between the involved servers.

A possible interleaving when both transactions are running in parallel.

*Branch*1 A	*Branch*2 B	
T_1 : {{lock(A); withdraw(A,10)} ∥ {send {lock(B); deposit(B,10)} to Branch2}}		
	T_2 : {{lock(B); getBalance(B)} ∥ {send {lock(A); getBalance(A)} to Branch1}}	*time* ↓
T_1 : {wait for ACK of deposit at Branch2} T_2 : {wait until lock(A) granted}		
	T_2 : {wait for balance of A} T_1 : {wait until lock(B) granted}	

A deadlock has occured which is difficult to detect!

Consequence of atomicity

- Whenever a transaction has processed a commit action, all its effects are permanent and will survive all failures.
- Whenever a transaction has processed a abort action - respectively is aborted -, all its effects are removed.

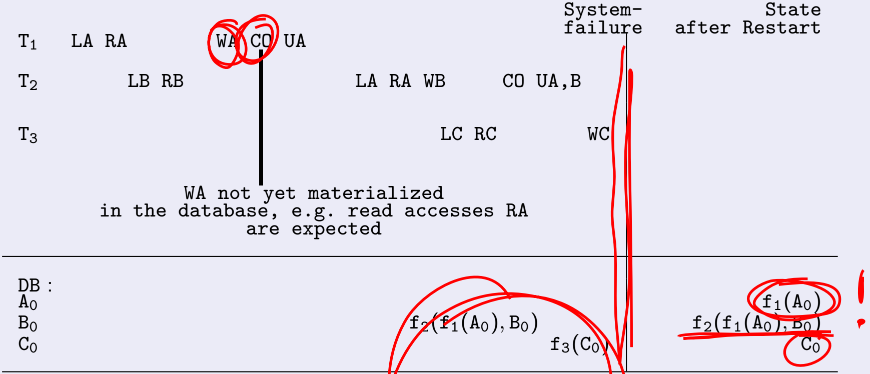
Recovery from system failures: Backwards Restart-Algorithm, logging has to be done on page-level

- $Redone := \emptyset$; $Undone := \emptyset$.
- The log is processed backwards. Let (T, A, A_{old}, A_{new}) the next log-entry to be considered. If $A \notin Redone \cup Undone$:

  Redo: If $(T, Commit)$ has already been found, then process WA with value A_{new} and perform $Redone := Redone \cup \{A\}$.

  Undo: Otherwise perform WA with value A_{old} and perform $Undone := Undone \cup \{A\}$.

Example



System-failure

State after Restart

T_1 LA RA WA CO UA

T_2 LB RB LA RA WB CO UA,B

T_3 LC RC WC

WA not yet materialized in the database, e.g. read accesses RA are expected

DB :

A_0 $f_1(A_0)$

B_0 $f_2(f_1(A_0), B_0)$ $f_2(f_1(A_0), B_0)$

C_0 $f_3(C_0)$ C_0

!

localmemory/ systembuffer :

T_1 $f_1(A_0)$

T_2 $f_2(f_1(A_0), B_0)$

T_3 $f_3(C_0)$

$f_1(A_0)$ $(T_1, A, A_0, f_1(A_0)), \ldots, (T_3, C, C_0)$

Log (reduced):

$(T_1, A, A_0, f_1(A_0)), (T_1, CO), (T_2, B, B_0, f_2(f_1(A_0), B_0)), (T_2, CO), (T_3, C, C_0, f_3(C_0))$

$f_1(f_1(A_0))$

Universität Freiburg
Institut für Informatik
Dr.-Ing. Thomas Hornung

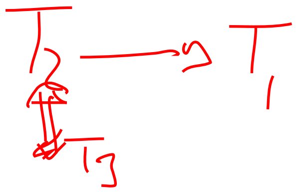
Georges-Köhler Allee, Geb. 51
D-79110 Freiburg
hornungt@informatik.uni-freiburg.de

Exercises
Distributed Systemes: Part 2
Summer Term 2013
21.6.2013

1. Exercise sheet: Refresh Concurrency Control and Recovery

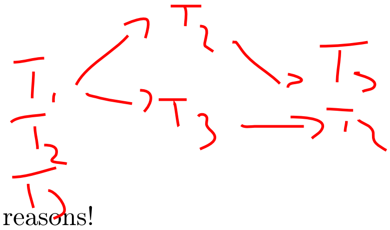
Exercise 1
Consider the following schedules.

S_1: $R_3X\ R_2Y\ W_2Y\ R_1Y\ W_1Y\ R_2X\ W_2X\ R_1X\ W_1X\ W_3Z.$
S_2: $R_3X\ R_2Y\ W_2Y\ R_1Y\ W_1Y\ R_2X\ W_2X\ R_1X\ W_1X\ W_3Y.$
S_3: $R_1Y\ W_1Y\ R_2Y\ W_2Y\ R_2X\ W_2X\ R_3Z\ W_3X\ R_1X\ W_1X.$



For each schedule give its conflict graph. Which schedules are serializable, which are not?

Exercise 2
Assume on a database three transactions are being executed.



a) The transactions are of the form:
$T_1:\ RA\ \ WA$
$T_2:\ RA\ \ WA$
$T_3:\ RA\ \ WA$

(i) How many serial schedules do exist for T_1, T_2, T_3? Give the reasons!

(ii) How many serializable schedules do exist for T_1, T_2, T_3, which are not serial ones? Give the reasons!

b) The transactions are of the form:
$T_1:\ RA\ \ WC$
$T_2:\ RB\ \ WA$
$T_3:\ RC\ \ WD$

(i) How many schedules do exist for T_1, T_2, T_3, which are not serializable? Give the reasons!

(ii) Applying 2-phase-locking, is it possible that all serializable schedules of T_1, T_2, T_3 may occur? Give the reasons!

Exercise 3
(a) Give an example of three transactions, which obey 2PL and have the following properties: (i) When being executed a deadlock may occur. (ii) For each pair of the three transactions and for any execution of such a pair, no deadlock can occur.
(b) Make suggestions for deadlock-free variants of the 2PL-protocol.

Exercise 4
Consider the following schedule.

```
                                             System-
                                             failure
T1  LA RA       WA                               |
T2       LB RB        LD RD WB     CO UD,B       |
T3                              LC RC       WC   |
```

Assume that actions W_1A, W_2B are not materialized in the database, however action W_3C is.

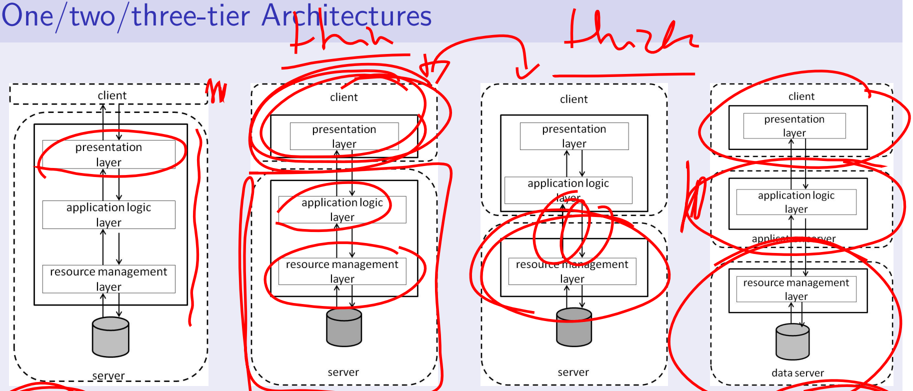
(a) Give the state of the database, the systembuffer and the log file when the system failure occurs.

(b) Describe the operations done when executing the restart algorithm and give the resulting state of the database.

2: Distributed System Architectures

2.1: Client-Server[1]

One/two/three-tier Architectures



one-tier | two-tier (thin/thick client) | three-tier

---

[1] Literature: G. Alonso et al., Web services: Concepts, Architectures and Applications. Springer Verlag 2004.

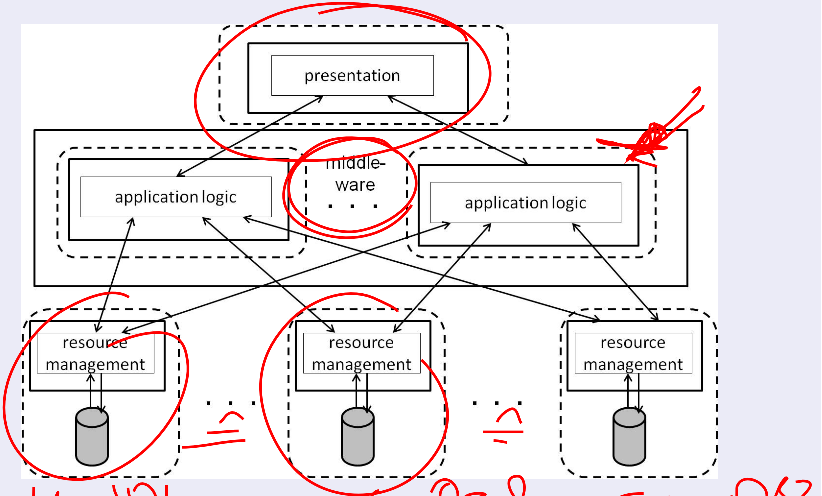
from one-tier to two-tier architecture

- The evolution to 2-tier systems was pushed by the appearence of the PC; now the presentation layer could be physically separated from the application layer.
- The presentation layer no longer takes resources needed by the application layer; it can be tailored for different purposes independently of each other.
- Complexity of the clients range from easily to maintain *thin clients*, offering only minimal functionality, to *thick clients*, offering rich functionality.

three-tier architecture

- How can a client communicate with a server of a different client/server system?
- 3-tiers architectures are mainly intended as integration platforms, where the new additional tier separating clients and servers in the 2-tier setting is also called *middleware*.

three-tier integration architecture: the middleware is the integration engine.



MySQL

Orale

DB2

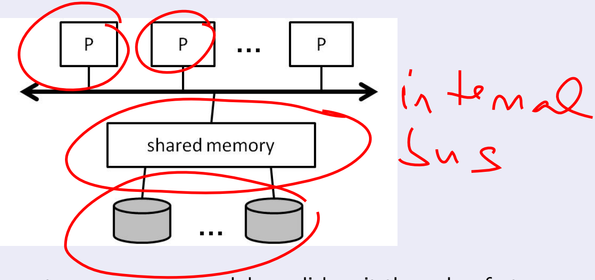
Federated system architecture

2.2: Multiprocessor Architectures

A parallel computer, or multiprocessor, is a special kind of distributed system made of a number of nodes (processors, memories, disks) connected by a very fast network within one or more cabinets in the same room.

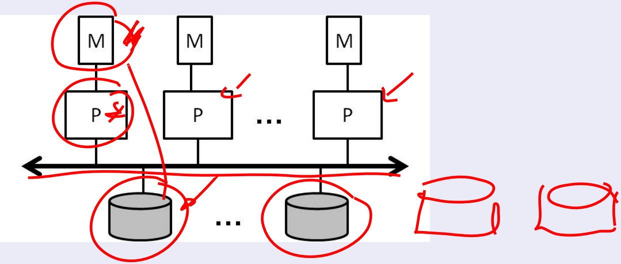
- *High-performance* by parallel data management, query optimization (*inter-query* parallelism to increase throughput and *intra-query* parallelism to decrease response time), load balancing.
- *High-availability* by increased data availability through replication and fault-tolerance through replication of components.
- *Extensibility* by adding processing and storage power to the system with minimal reorganization. Ideal behaviour:
  - *Linear speedup*: Linear increase in performance for a constant database size while the number of nodes (processing and storage power) are increased linearly.
  - *Linear scaleup*: Sustained performace for a linear increase in both database size and number of nodes.

Shared Memory



- Any processor has access to any memory module or disk unit through a fast interconnect. All processors are under the control of a single operating system.
- Simplicity: Meta-information (directories) and control information (e.g. lock tables) can be shared by all processors.
- Load balancing: Easy to be achieved at run-time using the shared-memory by allocating each new task to the least busy processor.
- High cost: Complex hardware required for the interlinking of processors and memory modules or disks.
- Limited extensibility: With faster processors (even with larger caches), conflicting access to shared-memory increases and degrades performance.
- Low availability: A memory fault may affect many processors. Duplex memory with redundant interconnect could be a solution.

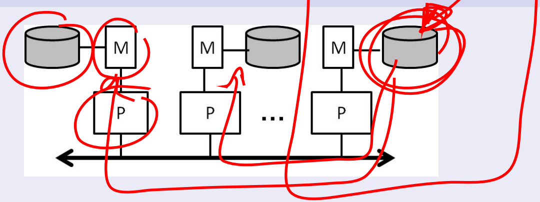
Shared-Disk



- Any processor has access to any disk unit through the interconnect but exclusive access to its main memory. Each processor can access database pages on the shared disks and cache them into its own memory.
- Low cost: Standard bus technology can be used for the interconnect.
- High extensibility, load balancing: Easy to add new disks.
- Availability: Memory faults are isolated from other nodes.
- Easy migration: No reorganization on disks necessary.
- High complexity: Distributed database system protocols are required.
- Cache consistency: Incurs high communication overhead.
- Performance: Access to shared disks is a potential bottleneck.

// Map Reduce

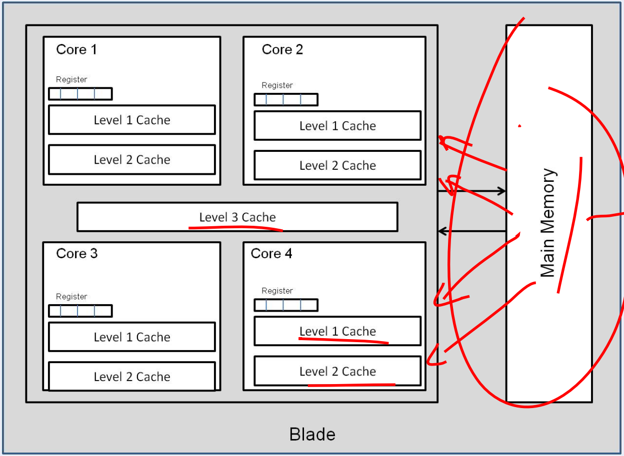
Shared-Nothing



- Each processor has exclusive access to its main memory and disks. Each node can be viewed as a local site in a distributed database. Using a fast interconnect it is possible to accomodate large numbers of nodes.
- Low cost: No special interconnect required.
- High extensibility: Easy to add new disks. Linear scaleup and linear speedup possible to achieve.
- High availability: Replicating data on multiple nodes.
- High complexity: Distributed database system protocols are required for a large number of nodes.
- Load balancing: Depends on data location and not actual load of the system.

Multicore Architecture

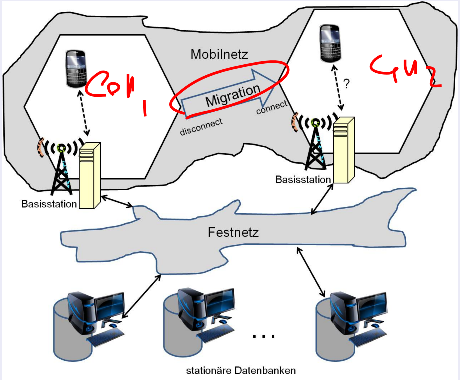
Consider a blade with 2 TB main memory and up to 64 cores. With 25 of such blades the enterprise data of the largest companies in the world can be hold and processed.



SAP
HANA
2TB

- Shared-nothing architecture among blades and shared-memory inside a blade.
- Cache coherency becomes critical.

2.3: Mobility Architectures



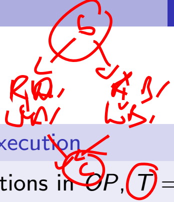
- Mobile devices with their local database may be temporarily disconnected.
- Stationary databases may be disconnected, respectively may be continuously updated.
- Consistent global states cannot be guaranteed, in general - undo of local operations may become necessary.

3: Transaction Model

Page Model

- All operations on data will be eventually mapped into read and write operations on pages.
- To study the concurrent execution of transactions it is sufficient to inspect the interleavings of the resulting page operations.
- Independently whether a page resides in cache memory or resides on disk, read and write are considered as indivisible.

$\{(RA, WA), (RB, WB)\}$



$\{RA, WA, RB, WB\}$

Parallelism as prerequisite for distributed execution

A transaction T is a partial order $<^1$ of actions in OP, $T = (OP, <)$, where OP is a finite set of T's actions RX and WX, where X is a data item.

Moreover, $< \subseteq OP \times OP$ is a partial order on OP which fulfills the following properties:

- Each data item is read and written by T at most once.
- If p is a read action and q is a write actions of T and both access the same data item, then $p < q$.

Complete transaction

We call a transaction *complete*, if its first action is begin b and its last action either is commit c or abort a.

$G = (V, E)$

$e \subseteq E \times E$

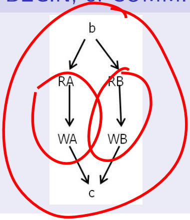
$p = RA \quad q = WA$

$RA < WA \Rightarrow (RA, WA)$

---

[1] A binary relation is a partial order , if it is reflexive, antisymmetric and transitive.

A parallel debit/credit transaction. *b*: BEGIN; *c*: COMMIT.

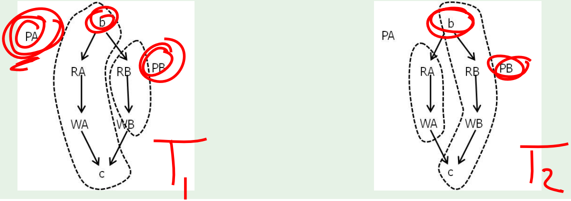


$OP = \{RA, RB, WA, WB\}$

$< = \{(RA, WA), (RB, WB)\}$

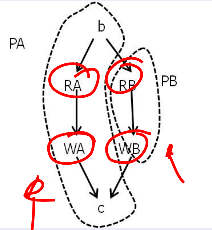
When transactions are depicted as directed graphs, we omit transitive edges.

Two parallel debit/credit transactions, each prepared for parallel execution.

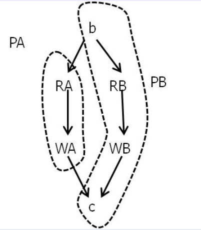


⟹ Definition of a schedule? Definition of serializability?

Two parallel debit/credit transactions, each prepared for parallel execution.



Transaction T_1



Transaction T_2

Locally observable schedules of the two transactions when executed in parallel by CPU PA and CPU PB.

(i) $PA: R_1A\ W_1A\ R_2A\ W_2A$
$PB: R_1B\ W_1B\ R_2B\ W_2B$

(ii) $PA: R_1A\ W_1A\ R_2A\ W_2A$
$PB: R_2B\ W_2B\ R_1B\ W_1B$

On each CPU in both cases the local schedules are serializable - however, globally, in the second case the transactions are not executed in a serializable manner!

Histories and schedules

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a (finite) set of complete transactions, where for each T_i we have $T_i = (OP_i, <_i)$.

A *history* of $\mathcal{T}$ is a pair $S = (OP_S, <_S)$, where

- $OP_S = \cup_{i=1}^{n} OP_i$ and $<_S$ a partial order on OP_S such that $<_S \supseteq \cup_{i=1}^{n} <_i$.
- Let $p, q \in OP_S$, where p and q belong to distinct transactions, however access the same data object. If p or q is a write action, then either $p <_S q$ or $q <_S p$; we say, p and q are in *conflict*; if $p <_S q$ and p and q are in conflict, we write $(p, q) \in \mathit{conf}(S)$.

A *schedule* of $\mathcal{T}$ is a prefix of a history.[2]

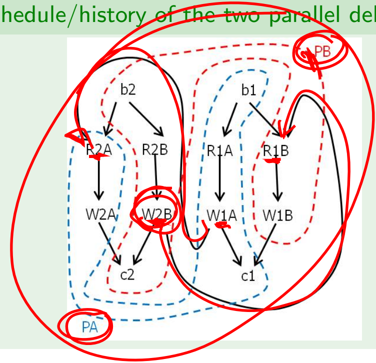
Conflict graph

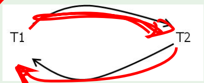
The conflict graph of a schedule S is given as $G(S) = (V, E)$, where V is the set of transactions in S and the set of edges E is given by the conflicts in S: $T_i \rightarrow T_j \in E$, iff there are conflicting actions $p \in OP_i$, $q \in OP_j$ and $p <_S q$.

---

[2] A partial order $L' = (A', <')$ is a prefix of a partial order $L = (A, <)$, if $A' \subseteq A$, $<' \subseteq <$, for all $a, b \in A'$: $a <' b$ if $a < b$, and for all $p \in A, q \in A'$: $p < q \Rightarrow p <' q$.

A schedule/history of the two parallel debit/credit transactions.





The schedule is not serializable as its conflict graph is cyclic.

Serializability

- A schedule $S = (OP_S, <_S)$ is *serial*, if for any two transactions T_1, T_2 appearing in S, $<_S$ orders all actions of T_1 before all actions of T_2, or vice versa.
- A schedule is called (conflict-)serializable,[3] if there exists a (conflict-)equivalent serial schedule over the same set of transactions.
- A schedule $S = (OP_S, <_S)$ is serializable, iff its conflict graph is acyclic.

---

[3] We consider only conflict-serializability and therefore talk about serializability in the sequel, for short.