Quiz #4
Solution

Dr. Asterios Katsifodimos

Alexander Alexandrov, Jonas Traub

Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

http://www.dima.tu-berlin.de/

(reminder) Parallel Speedup

- The Speedup is defined as: $S_p = {}^{T_1}/_{T_p}$
  - T_1: runtime of the sequential program
  - T_p: runtime of the parallel program on p processors

- Ahmdal's Law: „The maximal speedup is determined by the non-parallelizable part of a program" :
  - $S_{max} = \frac{1}{(1-f)+{}^{f}/_{p}}$ f: fraction of the program that can be parallelized
  - → Ideal speedup: S=p for f=1.0 (linear speedup)
  - However - since usually f<1.0 -, S is bound by a constant ! (e.g. ~10 for f=0.9)
  - → Fixed problems can only be parallelized to a certain degree!

- Gustavson's Law: „More processors are usually added to solve a larger problem in the same time" :
  - $S_{max} = (1-f) + Pf$ P: number of processors AND problem size
  - The larger the problem gets, the better the speedup gets.
  - →A growing problem can be effectively handled using parallelization!

Question 4.1

- What does Amdahl's Law imply?

Linear scale-out can be easily achieved.

Scalability depends on the amount of sequential code.

Scalability depends on the amount of data.

Hardware failures are very likely in massive parallel environments.

Question 4.2

According to the CAP Theorem, distributed systems can only guarantee two out of the following three properties:

- **C**onsistency - All nodes have the same view of the data at all times.
- **A**vailability - All requests sent to the system are answered.
- **P**artition Tolerance - The system maintains its properties even in case of a network partition.

This implies that we can build three different types of database systems CA, CP & AP. Based on this classification, which type of system would you aim for if you were a Bank that wants to develop the software for its ATMs?

Consistency & Partition Tolerance (CP)

Availability & Partition Tolerance (AP)

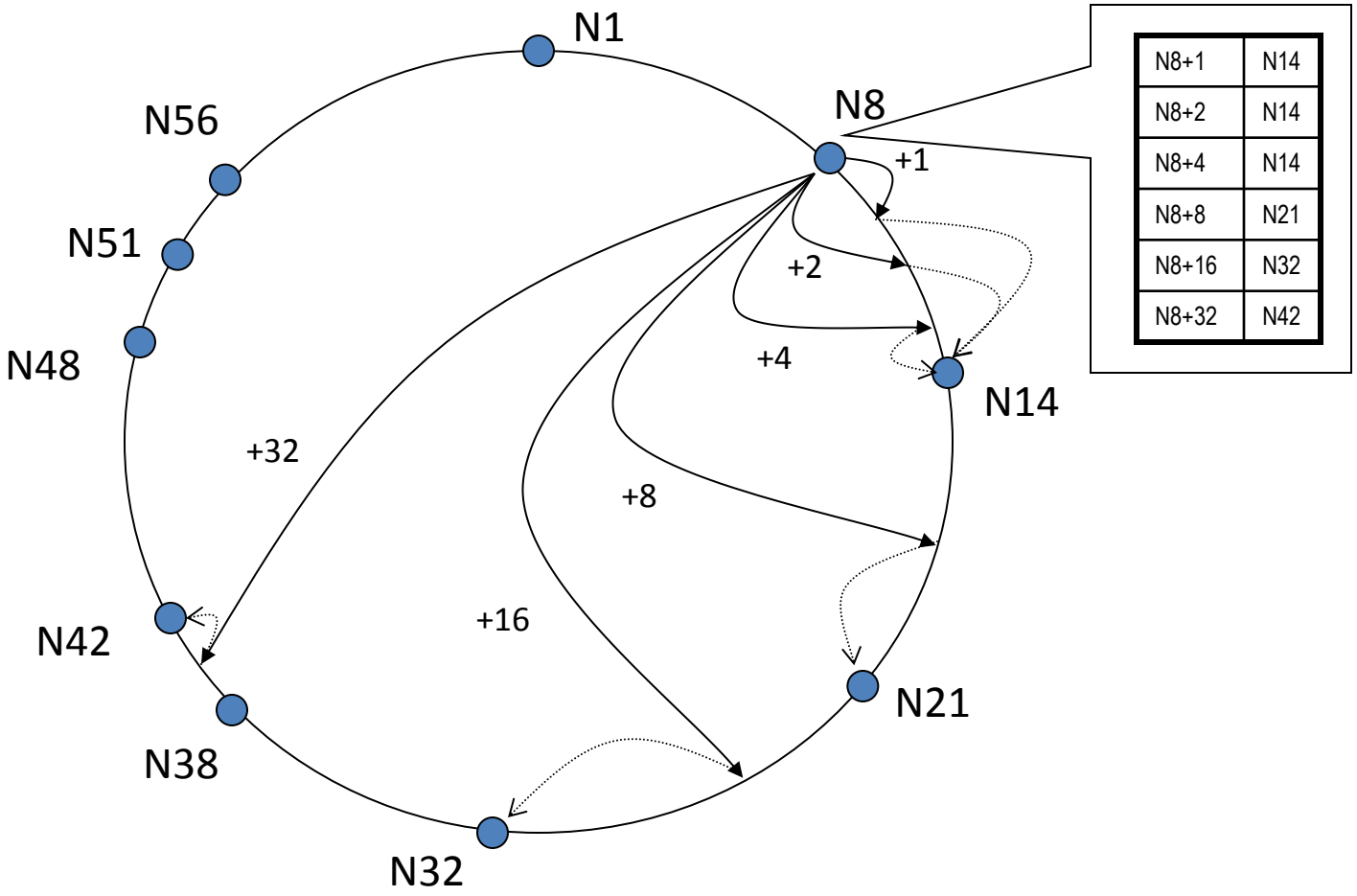
Consistency & Availability (CA)

Question 4.3

- What is used to achieve (some level of) consistency in parallel operational systems?

Replication
Multi-tenancy
Consensus algorithms
Consistent hashing methods
Vertical partitioning

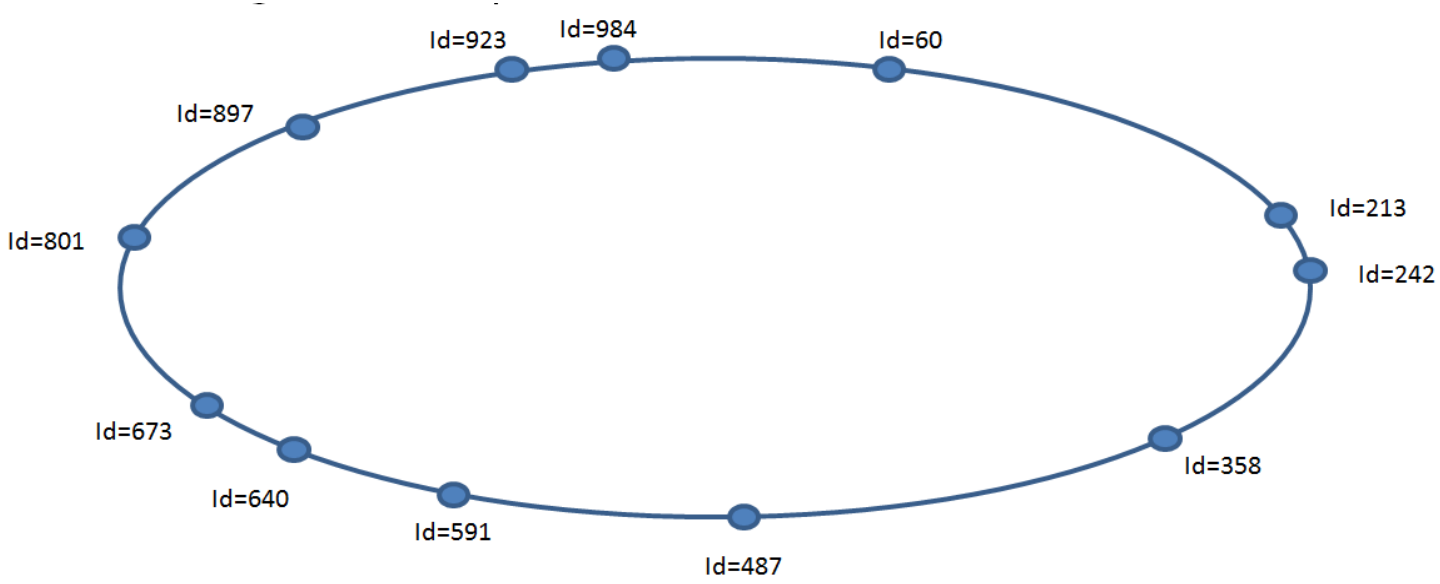
Finger Table: Example



Question 4.4

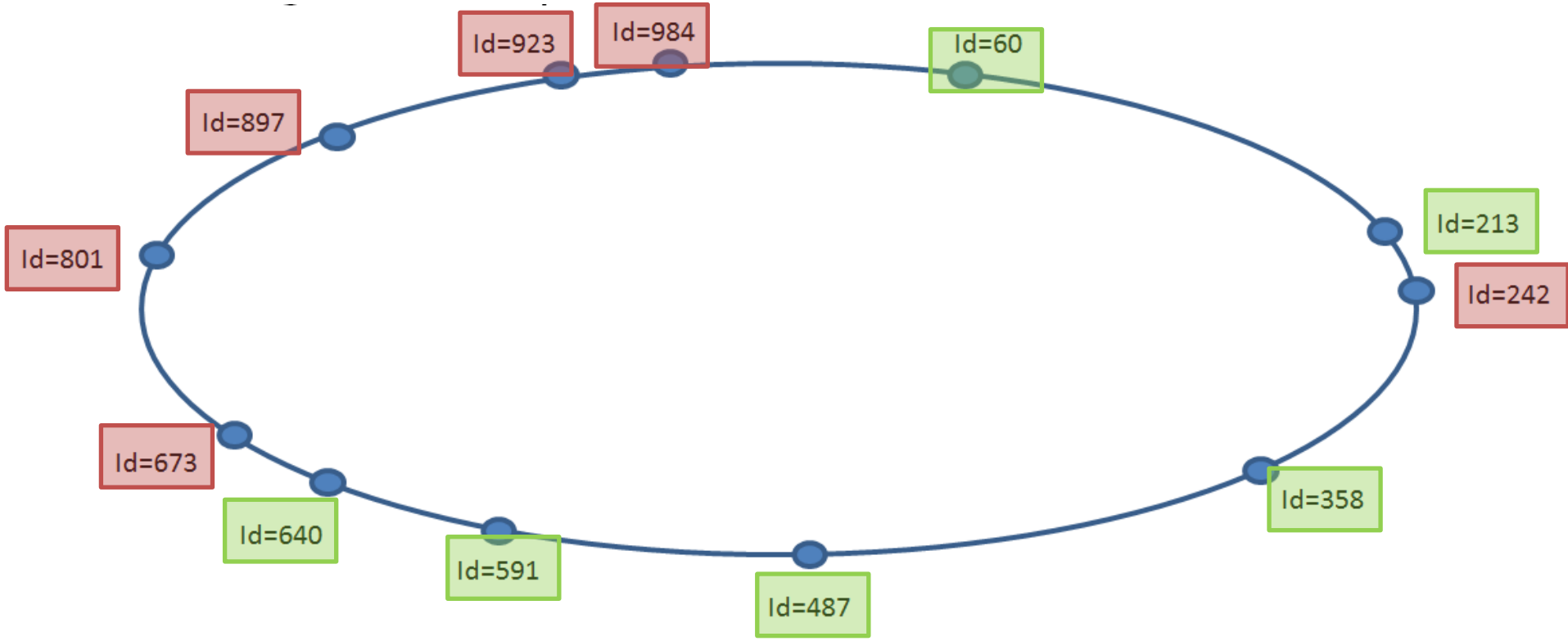
- Given is a distributed HashTable with an architecture similar to Chords:
  - Elements are mapped to a key within the range [0 ...1023].
  - The nodes in the system are organized in a ring.
  - A node in the system has an id within the key-range.
  - Each node holds a fingertable with 8 entries.
  - An element is hosted by the first node whose id is equal or larger than the elements key.

- Given the following node setup:



Question 4.4

- Node 60 receives a request for the element with the key 613.
  Check all nodes that process the request, i.e. all nodes which are involved in routing the request.
- Element with key 613 is hosted on node 640
- Fingertable with 8 entries: Jump width: $2^0 = 1$ to $2^7 = 128$



Review: Parallel RDBMs

- Pros:
  - Usually very good and consistent performance.
  - Flexible and proven interface (SQL).
- Cons:
  - Scaling is rather limited (10s of nodes).
  - Does not work well in heterogeneous clusters.
  - Not very Fault-Tolerant.

Review: MapReduce

- Pros:
  - Very fault-tolerant and automatic load-balancing.
  - Operates well in heterogeneous clusters.
  - Flexible parallel programming interface (map / reduce)
- Cons:
  - Writing map/reduce jobs is more complicated than writing SQL queries.
  - Performance depends largely on the skill of the programmer.

Question 4.6

- Which of the following statements about MapReduce and parallel database management systems (PDBMS) are true.

- For each correct answer you get 0.5 points, for each incorrect you loose 0.5. The minimum amount of points for this question is 0.

Compared to PDBMS, MapReduce is better in recovering from hardware failures.

Resizing a MapReduce cluster is easier than resizing a PDBMS.

MapReduce processes data more efficiently than PDBMS.

MapReduce and PDBMS utilize horizontal data partitioning.

PDBMS do not support data parallel query processing.

PDBMS use more efficient data access techniques than MapReduce.

MapReduce systems perform more advanced optimizations than PDBMSs.

MapReduce has superior transaction handling than PDBMS.

Reminder: Data Partitioning

- How to partition the data into disjoint sets?

- **Round robin:** Each set gets a tuple in a round, all sets have guaranteed equal amount of tuples, no apparent relationship between tuples in one set.

- **Hash Partitioned:** Define a set of partitioning columns. Generate a hash value over those columns to decide the target set. All tuples with equal values in the partitioning columns are in the same set.

- **Range Partitioned:** Define a set of partitioning columns and split the domain of those columns into ranges. The range determines the target set. All tuples on one set are in the same range.

Question 4.7

- Please check all correct statements concerning partitioning methods below.

- You will get 1 points for each correct answer and -1 for each incorrect.
  The minimum amount of points for this question is 0.

Round-robin partitioning handles data skew the best.

Range partitioning provides the most benefits but requires most information on the data to achieve good load balancing.

Round-robin partitioning can speed up the evaluation of group-by clauses.

Hash partitioning can speed up the evaluation of joins and range predicates.

Question 4.8

- Assume you have an application that provides a certain service. There are no competitors for that service and infinitively much demand for it. Furthermore, your application is CPU-bound.

- Assume you can buy a machine with infinitely many CPU cores. Each core costs exactly $100 when bought and $25 per month for power and maintenance. Running on a machine with a single core your application earns $1000 per month.

- Your developers found out that your application spends 65% within parallelizable instructions during its execution. The remaining 35% must be executed sequentially.

- You want to earn as much money as possible in one year. How many cores would you run your application on?

Question 4.8

- Cost for a CPU over the year: $100+12*$25 = $400
- Income per scale-up and year: 12*$1000 = $12000

- Scale-up (Amdahl's Law): S = 1 / (seqCode+ parCode / #cores)
  S = 1 / (0.35 + 0.65 / #cores)

- Earnings per year:
  $12000 * S - #cores * $400

#cores	Scale-up (S)	Earnings

Question 4.8

scale.nb

```
Gain = Income - Cost;
Income = Income0 * scale;
scale = 1 / (seqCode + parCode / cpu);
parCode = 1 - seqCode;
Cost = Cost0 * cpu;
cpuopt = Solve[D[Gain, cpu] == 0, cpu]
```

$$\left\{\left\{\text{cpu} \to \frac{-\sqrt{\text{Cost0 Income0 seqCode}^2 - \text{Cost0 Income0 seqCode}^3} + \text{Cost0 seqCode}^2 - \text{Cost0 seqCode}}{\text{Cost0 seqCode}^2}\right\},\right.$$

$$\left.\left\{\text{cpu} \to \frac{\sqrt{\text{Cost0 Income0 seqCode}^2 - \text{Cost0 Income0 seqCode}^3} + \text{Cost0 seqCode}^2 - \text{Cost0 seqCode}}{\text{Cost0 seqCode}^2}\right\}\right\}$$

```
N[Income0] = 12 * 1000;
N[Cost0] = 100 + 25 * 12;
N[seqCode] = .35;

N@cpuopt
Round@cpu /. N@cpuopt
```

{{cpu → −14.4739}, {cpu → 10.7597}}

{−14, 11}

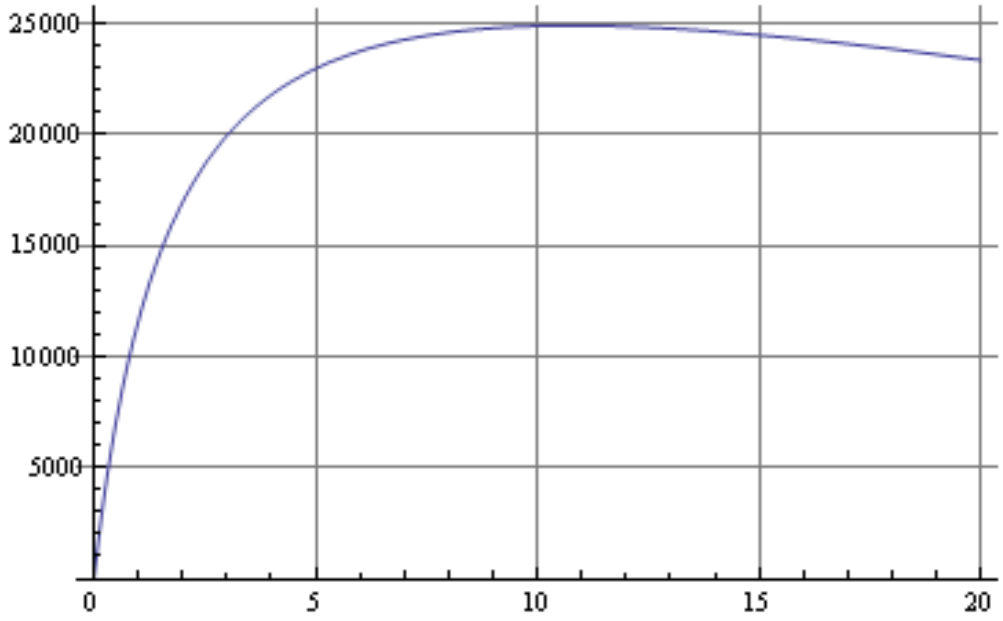
100%

Question 4.8

scale.nb

{−14, 11}

```
Plot[Gain /. cpu → x, {x, 0, 20}, PlotRange → {0, All},
 GridLines → Automatic]
```



100%

Question 4.9

- Given are two relations R(r1,r2,r3) and S(s1,s2,s3).
  The tuples of R have a width of 1.024B, the tuples of S have a width of 64B.
  The cardinalities of R and S are: |R| = 10.000, |S| = 1.000.000.

- R and S are stored on a shared-nothing parallel DBMS with 20 nodes.
  R is horizontally partitioned on r1. S is round-robin partitioned.

- Given is the following SQL query:

```
SELECT * FROM R, S
   WHERE R.r1 = S.s1
     AND R.r2 < X;
```

- The optimizer chooses between a partition-based and a broadcast join strategy. The strategy that ships the least data over the network is chosen. The amount of shipped data depends on the local predicate (R.r2 < X).

- Compute the selectivity (s) for which the broadcast strategy ships the same amount of data as any partition-based strategy.

Question 4.9

- Broadcast Join: Smaller side is broadcasted
  - S = 64B * 1.000.000 = 64MB
  - R = 1024B * 10.000 * s < 10MB -> R is broadcasted
  - Costs: (20-1) * 1.024B * 10.000 * s
- Repartition Join: R already partitioned on the join key, S must be re-partitioned towards R
  - Costs: (20-1)/20 * 64B * 1.000.000
- Solve equation:
  - 19 * 1.024B * 10.000 * s = 19/20 * 64B * 100.000
  - ⇔ s = 1/20 * 64/1024 * 1.000.000/10.000
  - ⇔ s = 1/20 * 2^-4 * 100
  - ⇔ s = 0.3125

Question 4.10

- Assume a cluster with 1000 homogeneous machines. The probability of a machine to fail during the execution of a certain job is 0.0005. If a machine fails job fails.

- Compute the probability that the job will fail during execution.

- Fail Probability: $p_f = 0.0005$
- Machine Count: $n = 1000$

- Job Fail Probability: $p_{jf} = 1 - (1 - p_f)^n$

$$= 1 - (1 - 0.0005)^{1000}$$

$$= 0.3935$$

Question 4.11

- Which aspects become more complicated in logging if log-records are written to a buffer manager instead directly to disk?

- Check all correct answers. Each correct answer gives you 1 point, each incorrect -1. The minimum amount of points is 0.

- [x] Handling of concurrent transactions.
- [ ] Applying changes in case of a recovery.
- [ ] Accesses by the query processor to the buffer manager.
- [x] Synchronization of writing log-records and table data.

Question 4.11

- Why are log records send to a buffer manager and not directly written to disk?

Improves fault tolerance of logging.

Improves disk accesses.

Supports the transaction manager in isolating transactions.

Eases recovery from failures.

Question 4.13

- What's the purpose of checkpoints for logging?

Guarantee isolation of transactions.

Reduce the number of log-records to process during recovery.

Improve the performance of logging.

Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Redo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Reminder: Undo vs. Redo Logging

- Problem with Undo-Logging: „Commit“ not until having written to disc
  - Potentially high I/O costs

- Undo log: allowed to “commit” when all changes are *on disk*
  - COMMIT is written to log when all values are on disk
  - Log holds old values
  - Incomplete transactions are rolled back
  - Complete transactions are ignored

- Redo log: allowed to commit when changes are *on log*
  - COMMIT is written to log before any value is written to disk
  - Log holds new values
  - Incomplete transactions are ignored
  - Complete transactions are repeated

Reminder: Rules of Undo Logging

- U1: If transaction T changes element X, <T, X, v> must have been written on disc BEFORE writing new value of X on disc
- U2: If transaction T commits , <COMMIT T> can only be written into log after all changed elements have been written to disk.
- Writing to disk is carried out in the following order:
  1. Write log record for changed elements
  2. Write elements to disk
  3. Write COMMIT log record
     - 1. and 2. separately for each element!

Reminder: Recovery via Undo Logging

- Problem
  - *Uncommitted* transactions made changes to database
- Solution: Undo recovery
  - Process complete log file backwards from end to start
    - „Chronologically backwards“
  - Most recent values first
- When walking back
  - Memorize all transactions with COMMIT or ABORT
  - When update record <T, X, v>
    - If a COMMIT or ABORT exists for T: Do nothing
    - Otherwise: Write v on X
- At the end
  - Write <ABORT X> for all uncommitted transactions into log
  - FLUSH LOG

Reminder: Undo Checkpointing Recovery

- Reading from back first <END CKPT> or first <START CKPT …>?
  - First <END CKPT>
    - Recovery only till next <START CKPT>
  - First <START CKPT T1,…, Tk>
    - Means that we had a system fault during checkpointing
    - T1, …, Tk are the single active transaction at this time
    - Recovery more backwards, but only till starting of the earliest transactions of T1, …, Tk
      - » Fast through appropriate pointer structure inside log file

Questions 4.16 – 6.17

- Given is the following Undo-Log:

```
01. <T19, A, 10>
02. <COMMIT T18>
03. <T19, B, "ABC">
04. <START T20>
05. <ABORT T19>
06. <START CKPT (T17,T20)>
07. <T20, A, 20>
08. <COMMIT T17>
09. <T20, C, 100>
10. <START T21>
11. <T21, B, "UMS">
12. <COMMIT T20>
13. <END CKPT>
14. <START T22>
15. <T22, D, "GND">
16. <T21, A, 12>
17. <T22, E, 42>
18. <COMMIT T22>
```

- Q 6.3:
  From which line must be recovered?

  Line 6

- Q 6.4:
  What is done with the Action in line 9?
  - ☐ Nothing
  - ☐ Redone
  - ☐ Undone

- Q 6.5:
  What is done with the Action in line 11?
  - ☐ Nothing
  - ☐ Redone
  - ☐ Undone

Reminder: Rules of Redo Logging

- Log record <T, X, v>
  - For database element X, transaction T has written new value v

- Redo Rule („write-ahead logging“ rule)
  - R1: Before any db element X changed by T is written to disc, all log records of T and COMMIT T must be written to log

- Writing to disk is carried out in the following order:
  - Write update log records on disc
  - Write COMMIT log record on disc
  - Write changed database elements on disc

Reminder: Recovery via Redo Logging

- Observation
  - If no COMMIT in log, elements on disc are untouched
    - There is no need to recover them
  - => Incomplete transactions can be ignored
- Committed transactions are a problem
  - It is not clear, which changes are on disk right now
  - But: Log records hold all information about that

- Process
  - Identify committed transactions
  - Read log data from the beginning to the end (chronological)
    - For each update record <T, X, v>
    - If T is not committed: Ignore
    - If T is committed: Write v as element X
  - For each uncommitted transaction write <ABORT T> into log
  - Flush log

Reminder: Redo Checkpointing Recovery

- As in undo logging,
  see whether the last checkpoint record in log is a START or an END:

- <END CKPT>
  - All transactions that are committed before <START CKPT (T1, …, Tk)> are on disc
  - T1,…, Tk **and** all transactions that have been started after START are unsafe
    - Even when COMMIT
  - It is sufficient to consider the earliest <START Ti>
    - Backward-linked log is helpful

- <START CKPT (T1, …, Ti)>
  - System fault occurred during checkpointing
  - Even committed transactions before this point are unsafe
  - Backward oriented search to next <END CKPT> and then continue backwards to the appropriate <START CKPT (S1, …, Sj)>
  - Then redo of all transactions, which have been committed after the START and redo of Si

Questions 4.14 – 4.15

- Given is the following Redo-Log:

```
01. <T19, A, 10>
02. <COMMIT T18>
03. <T19, B, "ABC">
04. <START T20>
05. <COMMIT T19>
06. <T20, A, 20>
07. <COMMIT T17>
08. <T20, C, 100>
09. <START T21>
10. <T21, B, "UMS">
11. <START CKPT (T20,T21)>
12. <COMMIT T20>
13. <START T22>
14. <T22, D, "GND">
15. <T21, A, 12>
16. <COMMIT 21>
17. <END CKPT>
18. <T22, E, 42>
```

- Q 6.6:
  From which line must be recovered?

  Line 4

- Q 6.7:
  What is done with the Action in line 08?
  - ☐ Nothing
  - ☐ Redone
  - ☐ Undone

- Q 6.8:
  What is done with the Action in line 14?
  - ☐ Nothing
  - ☐ Redone
  - ☐ Undone