UNIVERSITÄT DES SAARLANDES

Fachrichtung 6.2 – Informatik

Lehrstuhl Prof. Weikum

Transaktionsorientierte Informationssysteme SS 2000

Übung 10

Aufgabe 1:

Consider the algorithm for object-model recovery. Assume that the undo information and the redo information for an action are recorded in separate log entries, rather than always combining them into one log entry. What is the impact on the algorithm? In particular, what are the implications for the order in which log entries must be created and the forcing of the log buffer? Sketch the modified algorithm.

Aufgabe 2:

Consider the DB Cache method (see Exercise 9.2), applied to subtransactions, for implementing the L0 recovery of the simple 2-level recovery algorithm. When adopted in a straightforward way, this method would force after-images to the stable log, coined the "safe", upon each subcommit. Discuss under which conditions these forced log I/Os can be avoided. As a hint, assume that a set of after-images can be written to the safe atomically and analyze what happens when the forcing of a subtransaction's after-images is deferred until the transaction's commit. In particular, analyze how to deal with subtransactions, belonging to the same or different transactions, whose writesets have pages in common. How do such "dependencies" between subtransactions affect the deferral of forced log I/Os?

Aufgabe 3:

Consider the 2-level action history given in the following table, with operations on records (*store*, *modify*) and index keys (*insert*, *delete*); for the latter operations, the first parameter denotes a key and the second parameter the RID (i.e., address) of a stored record. Assume that there is a system crash right after the last action. Determine the necessary logging actions during normal operation and the recovery actions during restart, by completing the table. Use the enhanced 2-level recovery algorithm with a single log from Section 14.4.

Sequence number: action	Cached changes [PageNo:] [SeqNo]	Stable changes [PageNo:] [SeqNo]	Log entry added [LogSeqNo: action] [NextUndoSeqNo]
1: begin(t1)			
2: modify(x,t1)			
3: subbegin(t11)			
4: write(p,t11)			
5: begin(t2)			
6: store(y,t2)			
7: subbegin(t21)			
8: write (q,t21)			
9: write(r,t21)			
10: subcommit(t21)			
11: write(r,t11)			
12: subcommit(t11)			
13: delete(a,@x,t1)			
14: subbegin(t12)			
15: write(l,t12)			
18: subcommit(t12)			
19: insert(f,@x,t1)			
20: subbegin(t13)			
21: write(l,t13)			
22: write(k,t13)			
23: write(n,t13)			
24: subcommit(t13)			
25: begin(t3)			
26: store(z,t3)			
27: subbegin(t31)			
28: write(q,t31)			
29: write(r,t31)			
30: subcommit(t31)			
31: insert(h,@z,t3)			
32: subbegin(t32)			
33: write(k,t32)			
34: subcommit(t32)			
35: commit(t3)			
36: insert(b,@y,t2)			
37: subbegin(t22)			
38: write(l,t22)			

Aufgabe 4:

Reconsider the 2-level action history from Exercise 10.3. It contains operations on records (*store*, *modify*) and index keys (*insert*, *delete*); for the latter operations, the first parameter denotes a key and the second parameter the RID (i.e., address) of a stored record. Assume that the $insert(f, @x, t1)$ operation initiates a split of leaf page l, creating the new leaf page k and posting the split to the parent node n. Discuss to what extent logical log entries for the higher-level operations are feasible for redo purposes. (For undo such log entries are needed anyway.) Can we avoid creating physiological log entries at all? Which flush-order dependencies need to be observed for the execution?

Sequence number: action	Cached changes [PageNo:] [SeqNo]	Stable changes [PageNo:] [SeqNo]	Log entry added [LogSeqNo: action] [NextUndoSeqNo]
1: begin(t1)			
2: modify(x,t1)			
3: subbegin(t11)			
4: write(p,t11)			
5: begin(t2)			
6: store(y,t2)			
7: subbegin(t21)			
8: write (q,t21)			
9: write(r,t21)			
10: subcommit(t21)			
11: write(r,t11)			
12: subcommit(t11)			
13: delete(a,@x,t1)			
14: subbegin(t12)			
15: write(l,t12)			
18: subcommit(t12)			
19: insert(f,@x,t1)			
20: subbegin(t13)			
21: write(l,t13)			
22: write(k,t13)			
23: write(n,t13)			
24: subcommit(t13)			
25: begin(t3)			
26: store(z,t3)			
27: subbegin(t31)			
28: write(q,t31)			
29: write(r,t31)			
30: subcommit(t31)			
31: insert(h,@z,t3)			
32: subbegin(t32)			
33: write(k,t32)			
34: subcommit(t32)			
35: commit(t3)			
36: insert(b,@y,t2)			
37: subbegin(t22)			
38: write(l,t22)			

Aufgabe 5:

Give all log entries and their proper *NextUndoSeqNo* backward chains for the nested-transaction scenario of Figure 15.5. Assume that all of $t11$ is executed sequentially in the sense that a new subtransaction begins only after its previously initiated siblings are terminated, and that $t12$ is spawned after the termination of $t11$. Within $t12$ assume that all subtransactions are spawned asynchronously in separate threads; so at the end of the scenario all subtransactions of $t12$ are simultaneously active. Describe the necessary steps to abort subtransaction $t12$.

Aufgabe 6:

Develop pseudocode for the lock conflict test of new transactions that are admitted during the redo pass of a restart, assuming that the OldestUndoLSN and the DirtyPages data structure has been determined by the analysis pass and serves as an approximative test for “uncritical” pages that require neither redo nor undo steps. Assume that page locking is used, but no explicit locks are re-acquired during the restart to keep the overhead low.

Aufgabe 7:

Design a scenario, with concrete page numbers, transaction identifiers, log sequence numbers, etc., that shows the need for synchronizing the local log sequence numbers created at different servers of a data-sharing cluster. Construct anomalies that would arise if the global sequence numbers merely were local numbers padded with server identifiers.

Aufgabe 8:

Develop pseudocode for the crash recovery in a data-sharing cluster, based on the algorithms outlined in Section 15.6. Assume that all undo-related log entries for the same transaction are confined to a single "private" log, but redo log entries for the same page may be dispersed across multiple logs. Further assume that pages can be directly transferred between the caches of different servers, without having to be flushed to disk. Distinguish two cases:

a. recovery from a single server crash, where one of the other servers takes over the failed server's role, and
b. recovery from a software failure that brings down the entire cluster (which is unlikely, but still possible), assuming that none of the disks is damaged.