UNIVERSITÄT DES SAARLANDES

Fachrichtung 6.2 – Informatik

Lehrstuhl Prof. Weikum

Transaktionsorientierte Informationssysteme SS 2000

Übung 6 — Beispiellösung

Aufgabe 1:

Discuss how predicate locking can be extended to disjunctive conditions such as queries of the form

```
SELECT Name FROM Emp
WHERE Position='Manager' OR Department='Research'.
```

Also discuss how join queries such as

```
SELECT Emp.Name, Emp.Department FROM Emp, Dept
WHERE Emp.Position='Manager' AND Dept.City='Toronto'
AND Emp.Department=Dept.Department
```

could be (conservatively) handled by predicate locking.

A common way to extend predicate locking to arbitrary boolean expressions (and thereby including discunctions as a special case) is the following: Each boolean expression is converted to its equivalent disjunctive normal form (DNF). For each of the monoms in this DNF, a predicate lock is acquired. Each tuple that satisfies the original expression satisfies at least one of those monoms, so locking the monoms is sufficient to lock all relevant tuples. However, as we split the process of acquiring one 'complex' lock into acquiring several 'simpler' locks, we increase the possibility for deadlocks. (Imagine that two concurrent transactions try to lock an 'overlapping', complex predicate. Both transactions may be able to lock some monoms, but may be blocked by the other before they can acquire locks for all monoms. If they acquired only the complex lock, only one would be blocked.)

The problem with join queries is that its query predicate contains expressions from several tables, not only from one. Because we acquire locks for one table at a time only, we must build a (conservative) approximation about which tuples of each table are hit by a join. For each table, we lock that part of the predicate that describes attributes of this table (like 'Emp.Position='Manager'' in the example), and completely ignore join information. We probably lock too many tuples, but all the relevant ones will be locked. If we don't have any information about the relevant tuples of a table but join attributes, we lock the complete table.

Aufgabe 2:

Consider the following transactions:

$$t_1 = r_1(x)w_1(x)r_1(y)w_1(y)$$

$$t_2 = r_2(x)$$

$$t_3 = r_3(y)w_3(y)$$

Try to decompose t_1 into three pieces s.t. the result is a correct chopping.

It is quite natural to chop t_1 into two pieces, one piece for the operations on each of the data objects x and y. To get three pieces, we have to decompose one of those parts into two new parts consisting of only one operation. The best candidate is obviously the part dealing with x, because the other transactions only read x, so we won't lose consistency in x with this chopping.

Having said this, we consider the chopping of t_1 into $t_11 = r_1(x)$, $t_12 = w_1(x)$, and $t_13 = r_1(y)w_1(y)$. This is indeed a correct chopping, because its chopping graph is acyclic:

(insert graph here)

Aufgabe 3:

Show that if a set of chopped transactions contains an sc cycle, any further chopping of any of the transactions will not render the graph acyclic.

Aufgabe 4:

Discuss how the incremental key-range locking protocol on index keys has to be changed for a unique index. Discuss also how unique and non-unique indexes may be treated differently by the various optimizations sketched in Section 9.5.

If we consider unique indices, key-range locking has to be changed in the following ways:

- No combination of insert and delete operations is commutative. For example, if transaction t_1 inserts a new key into the index, a following insert of the same key will fail due to the uniqueness property of the index; if the two inserts were commuted, the latter would be successful, while the other failed. (If one considered return values, one could allow commuting two failed inserts or two failed deletes, allowing a higher degree of concurrency.)
- A successful delete operation always removes the key from the index, because there is only one associated tuple for this key. It is therefore always necessary to lock the previous key, which could be ommitted with non-unique indices.

- The optimization from Section 9.5.2 (lock key in 'insert' mode) is not correct any more, because two insert operations are no longer commutative.
- The optimization from Section 9.5.3 (lock pairs of key and rowid instead of locking only the key) does not gain any additional concurrency anymore, because there is at most one rowid for each key in the index.

Aufgabe 5:

Incremental key-range locking based on read and write locks on index keys, as explained in Section 9.3, does not allow two insert operations in the same, previously unoccupied key interval to proceed concurrently. For example, in the index of Figure 9.5, an insert(27) would block a later insert(28) operation, although none of these keys are initially present and the two operations should be considered as commutative. Discuss whether the resulting blocking of one of the two operations can be safely avoided, and how this could be accomplished. Also discuss the analogous situation for delete operations, and the combination of insert and delete operations.

We don't consider any of the optimizations discussed in Section 9.5 of the script. Then, every insert operation has to lock the key previous to the inserted key (25 in the example), thus the first transaction blocks the second one. To avoid this problem, we introduce a new 'insert' lock mode in which we lock the previous key. An insert lock is compatible with other insert locks, but not with read or write locks.

Aufgabe 6:

Consider a situation where a node p in a B+-tree is not split-safe, but has a split-safe descendant q, with both p and q on the traversal path of an insert operation. Discuss how the lock-coupling technique should handle this situation to minimize lock contention.

Because node q is split-safe, this insert operation won't split q. Therefore, we can safely release the lock on all its predecessors.

Aufgabe 7:

Assume that the leaf nodes of a B+-tree index are linked together bidirectionally (rather than only in forward direction). This may be useful to support range scans in descending key order, which in turn is of interest for returning qualifying records in descending sort order (as requested, for example, by a decision-support query). Discuss the consequences of this bi-directional linkage with regard to the lock-coupling and the link techniques for the page-level concurrency control. If possible, generalize these techniques to accommodate the new feature.

Aufgabe 8:

Consider the following B+-tree index on the attribute AccountNumber of an Accounts table. Assume that all tree nodes have a space capacity to hold up to four entries. Which locks need to be requested for the execution of the following transaction, assuming the incremental key-range locking at the access layer, lock coupling at the page layer?

```
begin of transaction;
   select count(*) from Accounts where AccountNumber
          between 11 and 25;
   insert into Accounts (AccountNumber,...) values (27,...);
commit transaction;
```

search(11):
read lock r
read lock p1
unlock r
read lock q3
read lock q2 // determine previous key
key-range lock 9 // previous key
unlock p1
unlock q3
unlock q2
return 12,q3

next(12,q3,25):
read lock q3
key range lock 12
unlock q3
return 13,q3

next(13,q3,25):
read lock q3
key range lock 13
read lock q4
unlock q3
unlock q4
return 15,q4

next(15,q4,25):
read lock q4
key range lock 15
unlock q4
return 16,q4

next(16,q4,25):
read lock q4
key range lock 16
unlock q4
return 17,q4

next(17,q4,25):
read lock q4
key range lock 17
unlock q4
return 20,q4

next(20,q4,25):
read lock q4
key range lock 20
read lock q5
unlock q4
unlock q5
return 22,q5

next(22,q5,25):
read lock q5
key range lock 22
unlock q5
return 24,q4

next(24,q5,25):
read lock q5
key range lock 24
read lock q6
unlock q5
unlock q6
return no_more_keys,nil

insert(27):
write lock r
write lock p2
unlock r
write lock q6
read lock q5 // get previous key
key range lock 24
key range lock 27
unlock q5
unlock q6

unlock p2

commit:
unlock all key range locks

Aufgabe 9:

Discuss to what extend lock escalation may incur deadlocks or livelocks.

See the solution for excercise 7.1.

Aufgabe 10:

Give examples for schedules that fall into the following five isolation-level classes (but not into the next larger, surrounding class): not even read uncommitted, read uncommitted, read committed, repeatable read, serializability.

See the solution for excercise 7.3.

Aufgabe 11:

Give an example for a snapshot-isolated history violating the consistency of the persistent data. Explain what kinds of inconsistencies can arise.

See the solution for excercise 7.4.

Aufgabe 12:

Design a concurrency control protocol that can guarantee snapshot isolation (and can indeed generate all snapshot-isolated histories).

See the solution for excercise 7.5.