Enhancing Concurrency in Layered Systems

Gerhard Weikum
Technical University of Darmstadt
Alexanderstr. 24, D-6100 Darmstadt
West Germany

Abstract
To enhance concurrency in a layered system architecture, a multi-level transaction approach is proposed. Low-level operations issued by some higher level action are treated as a layer-specific subtransaction. Thus, according to the rule of level-by-level serializability, semantic knowledge about application-specific actions can be utilized, and still single high-level actions may be interleaved on a deeper layer. In the case of a nested two-phase locking protocol this means that low-level locks are released prematurely and only semantic locks are held until the end of an application transaction. Due to this characteristic of subtransactions, transaction aborts have to be performed by compensating low-level changes through inverse high-level operations rather than backing them out on the page level. Besides developing implementation guidelines for a multi-level concurrency control protocol, various mechanisms to cope with hot spots are discussed within the presented framework.

1. Towards a Low-Conflict Concurrency Control Framework
Much work has been devoted to the issue of database system concurrency control during the last ten years, both in practice and theory. While the overall goal of the developed solutions is to reduce contention on data items, particular efforts have been made to cope with so-called "hot spots", i.e. high-traffic data elements like reservation counters in an airline booking system, or index root pages internally. The following non-exhaustive list tries to summarize most of the proposed techniques:

1) Eliminate hot spots by designing applications carefully, e.g. avoid explicitly storing aggregate data like branch totals unless queries are very frequent [St86].

2) Divide hot spot elements into several subelements [St86].

3) Choose a finer lock granularity, e.g. lock records or even fields instead of pages [An85, MLC87].

4) Treat access path data different from primary data, e.g. lock index "subpages" [CLSW84] or single index keys [DLPS85].

5) Apply special protocols to specifically organized data structures [SG86] like B⁺-trees [BS77] or hash tables [El183].

6) Utilize advance knowledge about access patterns of transactions and apply a non-two-phase locking protocol [SK80, MFKS85].

7) Make use of multiple versions allowing readers to access (an old version of) a data item while it is updated concurrently [CFLNR82, Ba83, BG83].
(8) Apply an optimistic concurrency control scheme [KR81], i.e. perform validation tests instead of locking data [GK85].

(9) Sacrifice serializability for concurrency
   (a) by tolerating a lower "degree of consistency" [GLPT76, GW82],
   (b) by specifying "semantically consistent" interleavings of actions [Ga83, Ly83].

(10) Take into account the semantics of application operations
   (a) by introducing operation-specific lock modes [BGL83, Ko83, SS84], e.g. for incrementing a counter field, or even
   (b) by applying state-based commutativity [BBGLS83], i.e. examining the value (or even the set of potential values) of an object when a conflict test is performed [Reu82, GK85, O'N86].

(11) Split long transactions into several (independent) transactions [GS84]. In particular, avoid transactions which span terminal I/Os.

While it seems that a well-designed mixture of these techniques is quite appropriate to handle hot spots in traditional high-performance transaction processing like in banking applications or reservation systems, and concurrency control may perhaps no longer be a crucial problem there [cf. St86], upcoming database applications like CAD/CAM, office filing and large-scale knowledge-based systems which deal with complex objects and have more sophisticated operational characteristics will probably suffer from a lack of adequate transaction management methods. It has been brought up that, in view of a whole bunch of proposed concurrency control algorithms, there may be no point in inventing yet another one [St83, Pa83], and we share this opinion at least to some extent. However, what we feel is needed is a framework how to make the right use of this large variety of methods. Just the length of our above list indicates that we are still far from this point of having a (minimal) set of well-understood mechanisms which can be combined with each other. Actually, this is not surprising since most of the techniques in our list have evolved within a particular class of applications and thus may be tailored more or less to particular data and load characteristics. In order to generalize their usage, a thorough analysis of underlying assumptions and inherent properties is necessary to fully understand why performance is gained by some mechanism.

This paper aims at this goal of a unifying framework, or rather should be considered as a first attempt on some aspects of it, in that it presents basic principles of so-called multi-level transactions, the idea of which has been around for quite some time in the notions of "open nested transactions" [Tr83, cf. also Gr81b] and "spheres of control" [Da78]. Besides discussing some of the mentioned techniques within that framework, the main emphasis of the paper is on implementation guidelines for multi-level concurrency control strategies. A special protocol called "nested two-phase locking" has been implemented within a simulation testbed and is currently realized as part of the Darmstadt database system (DASDBS), a layered kernel system [SW86, PSSWD87] supposed to support advanced "object-oriented" [DD86] applications.

The paper is organized as follows: Section 2 presents the basic idea of multi-level trans-