The Xymphonic Transaction Model:

Technical Summary for Experts

Dr. Ole Jørgen Anfindsen, founder & CEO
Xymphonic Systems AS
P. O. Box 50
2027 Kjeller Technology Park, Norway.
ole.anfindsen@xymphonic.com

Last revision: 28 June 2002.

At the very heart of the xymphonic model are two generalizations of serializability or, more precisely, two generalizations of conflict serializability (CSR). And even though CSR may not always be fully supported in commercial systems, it serves an important function as a yardstick against which those systems are measured. CSR is (theoretically, at least) perfect for many classical database applications, but is usually unsuitable for long-lasting transactions (LLTs). The xymphonic model defines two new correctness criteria that enable much greater concurrency while still having CSR as a special case.

The first new correctness criterion is called conditional conflict serializability (CCSR). It enables the isolation between transactions to be made conditional, in contrast to classical transactions where isolation is unconditional. The other new correctness criterion is nested conflict serializability (NCSR), which enables collaborative work involving two or more transactions. Let's look at each of them in turn.

Conditional conflict serializability (CCSR)

Good, old CSR is based on the notion that two operations should always be regarded as conflicting if at least one of them is a write operation. In CCSR this is reduced to a default from which applications may choose to deviate. Applications can do this by associating with their read or write operations one or more predefined parameter values. Let A and B be sets of such parameter values, and let W(A) and R(B) denote write and read operations with A and B, respectively, associated with them. If W(A) and R(B) are issued concurrently by two different transactions on the same data item, then the two operations will conflict with each other unless A is a subset of B.

To make this more concrete, assume the set of available parameter values is {good, medium, bad}. Then e.g. W(bad) and R(good) would conflict, while W(good) and R(medium, good) would not. The idea is that a writer using W(good) signals to any potential reader that the reliability or quality of the W-locked data is "good". Conversely, a reader using R(medium, good) thereby tells the system that he/she is willing to read data that is of "good or medium" quality/reliability. Carefully note that while application programmers need to understand the semantics of the parameter values that they use, the transaction manager does not.

The classical Serializability Theorem says that a transaction history is CSR if and only if its conflict graph (also known as a serialization graph) is acyclic. The xymphonic model substitutes the unconditional definition of conflict between readers and writers with a conditional one. Using this new definition of conflict, we get the Generalized Serializability Theorem, which says that a transaction history is CCSR if and only if its conditional conflict graph is acyclic. Using this new theorem it is easy to show that CCSR can be implemented (or enforced, if you will) by means of parameterized two-phase locking, just as CSR can be implemented by means of ordinary two-phase locking (2PL).

It is not enough for a transaction history just to be serializable; it must also have certain properties having to do with recovery, viz. recoverability (RC), avoidance of cascading aborts (ACA), strictness (ST), and, in practice, rigorousness (RG). It turns out that RC, ACA, ST, and RG *modulo* CCSR can be defined, and that both CCSR and RG can be ensured if rigorous, parameterized 2PL is used. This is why CCSR, unlike many other approaches to collaborative transactions, can afford its users automatic recovery. In other words, CCSR does not depend on recovery by means of compensation, which would have been a considerable burden on the users.

CCSR enables any level of concurrency from serializability to dirty reads. However, in the latter case CCSR has a great advantage over plain dirty reads in that readers will always be informed (by means of the parameter values) of the quality, reliability, maturity, or degree of completeness of the data they retrieve. Carefully note that a database administrator (or some similar person) can define as few or as many parameter values in the meta-database as is needed by the applications running in his/her system. Also note that an application that is only interested in a subset of all the parameter values defined in the system, can simply disregard the others; the usage of these by other applications need not influence the first application in any way. Another very attractive property of CCSR is that classical and xymphonic transactions can co-exist in the same system without either group being aware of the other (except that any transaction that is long-lasting, whether it is classical or xymphonic, can of course block others and thus influence their response times).

Nested conflict serializability (NCSR)

In the following, the term sphere of control (SOC) will be used to denote a set of resources over which an agent has a certain level of control. A database is an important example of a SOC, and the agent in charge of that SOC is the DBMS in question. An active transaction has been given access to the database SOC (DBSOC) by the DBMS, and as the transaction executes its locks dynamically establish a new SOC within the DBSOC. The set of data items locked by a transaction is a SOC over which that transaction has control until it aborts or commits. This SOC may be seen as consisting of a read SOC and a write SOC, established by means of read and write locks, respectively. The write SOC (WSOC) of a transaction is of particular interest to us in this context, because it turns out to be a special case of a DBSOC; it is in a sense a single-user database.

What if we allowed a (single-user) WSOC to be turned into a (multi-user) DBSOC? We could then allow other users to execute transactions within this dynamically created DBSOC on behalf of its owner, i.e. on behalf of the transaction that created the DBSOC in question. We would then need concurrency control within such a DBSOC, and the natural solution is to use locking here as well. This means that new WSOCs will be established at this level, enabling the creation of a next level DBSOC, and so forth; giving rise to the term nested database, or simply *xymphony*. If we enforce conflict serializability as the correctness criterion at all levels of databases, i.e., in all xymphonies, it is only natural that this should be referred to as nested conflict serializability (NCSR).

In other words, NCSR generalizes CSR by allowing the sequence of operations within a WSOC to be serializable rather than serial; thus applying CSR recursively. This simple idea allows arbitrarily complex patterns of collaboration involving two or more users.

For example, consider the designers Alice and Bob working on the same project. If Bob discovers that he needs some help from Alice, he could turn a WSOC of his into a xymphony and indicate to the system that Alice should be given read and write access. Then the two designers can start new transactions to work within this newly created xymphony. As long as they do so, their transactions will be prevented from interfering with each other by the standard locking protocol. When Alice completes her work, she will commit to the xymphony, not to the global database (as transactions started at the global level always do). This means that she is effectively committing her work to Bob (since he is the xymphony owner), and the system could easily be set up such that he can inspect the results thus committed to him, and then accept or (partially) reject it. In this way Bob never gives up transactional control over the design objects for which he is responsible.

It is easily shown that NCSR can be implemented by using 2PL in each DBSOC (the global database as well as all xymphonies at all levels of nesting). This means that the transaction history generated in each DBSOC will be CSR. In particular, the transaction history for the global database will be CSR, meaning that from the point of view of the DBMS there is no difference between NCSR and CSR; both kinds of schedulers can produce exactly the same results. One could say that NCSR is a more elaborate way of producing serializable histories; it allows users to collaborate, but confines all the complexity of their collaboration to the xymphonies in which they work.

Combining CCSR and NCSR

We further generalize the nested database concept by allowing transaction histories within xymphonies to be CCSR rather than CSR, thus providing even more flexibility for users who need to collaborate. The combination of CCSR and NCSR is referred to as Nested Conditional Conflict Serializability, or NCCSR.

Conclusions

The xymphonic model allows application-specific handling of read-write and write-read conflicts by means of CCSR, enabling sharing of data among transactions. And the xymphonic model allows application-specific handling of write-write conflicts by means of NCSR, enabling collaborative work. NCSR and CCSR can be combined into NCCSR, which can be implemented by means of moderate changes to 2PL locking schedulers. NCCSR is easy to use, and analysis shows that it will have a low run-time overhead. NCCSR has not been tailored to any specific application domain, and can be used to generate a large class of xymphonic transaction interaction patterns. The bottom line is that the xymphonic model is both powerful and simple.