

# The Xymphonic Transaction Model: a technical summary

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

Last revision: 28 June 2002.

Database management systems (DBMSs) are software systems that control information storage and retrieval. DBMSs are popular, to a large part because they provide control, discipline, and predictability where one would otherwise have all sorts of ad-hoc solutions to the challenges involved in managing information. These nice properties of DBMSs have their roots in two important concepts known as *data* models and *transaction* models, respectively. While a data model describes the *structure* of a database, a transaction model describes the *behavior* of a database system. This memo is only concerned with the latter.

The kind of transactions that have been used in DBMSs for the past twenty to thirty years, are often referred to as ACID-transactions (the acronym ACID is explained below) or, simply, classical transactions.

While classical transactions are great for many kinds of applications, they turn out to be counterproductive for others. This is where the xymphonic transaction model comes in, and this memo tries to explain some of the technical issues for readers who are not database or transaction experts. If you are interested in less technical aspects of xymphonic collaboration, e.g. its business potential, more information is available upon request.

## A brief introduction to ACID transactions

When a user manipulates or interacts with a database, his or her actions will normally happen as part of a *transaction*. The point of using transactions in the first place, is to prevent things from going wrong. Many things could go wrong if transactions were not used; e.g. a user could insert, delete, or change something in the database, and have the effects of his or her work nullified by another user (the so-called lost update problem); or a user may retrieve a piece of data thinking that its value is reliable, while in reality that value is in the process of being changed by another user, and therefore may not be reliable at all (the so-called dirty read problem).

Any decent database management system (DBMS) will support transactions (typically by means of locking), and the behavior of such transactions is captured by the acronym ACID, which stands for Atomic, Consistent, Isolated, and Durable. There is a lot to say about ACID transactions (a 16 page tutorial is given in chapter 2 of the PhD thesis that defines the xymphonic model (available upon request)), but the bottom line is that they prevent almost any problem you can think of, save those caused by floods, fires, earthquakes, or other major

disasters. Therefore, everyone agrees that ACID transactions are great for banking, airline reservations, and other classical applications of database technology.

## The shortcoming of ACID transactions

However, there is also broad consensus that ACID transactions are unsuitable for a number of other application domains, the common denominator of which is typically the need for long-lasting transactions, i.e. transactions whose duration is measured in hours, days, or months, rather than seconds or minutes.

First, ACID transactions employ a very strong notion of *conflict*, which causes one transaction to block another whenever at least one of them tries to perform a write operation and they both access the same data item, resulting in the second transaction having to wait until the first one completes. Waiting for a few seconds may only be a minor problem, but waiting for hours or days is usually unacceptable.

Second, the I in ACID prescribes total isolation between transactions. Such isolation is known as *serializability*, since transactions behave *as if* they executed serially (i.e., as if they had the database all to themselves). This is just what you want in classical database applications, but is counterproductive for users who need to share information with each other or participate in some form of collaborative work. Therefore, something more general and flexible than the ACID transaction model is required.

## Generalizing from ACID to ACCID

This is where the xymphonic model makes a contribution. Rather than insisting on unconditional isolation between transactions, the xymphonic model allows the degree of isolation to be customized, and introduces the notion of *conditional* isolation (not to be confused with the SQL standard's concept of *isolation level*, which is discussed on pages 14 - 16 of the xymphonic PhD thesis, and which may easily be combined with the xymphonic model). The isolation property is generalized in two different ways; one way that enables readers and writers to share data, and another that enables writers to perform collaborative work on a set of data items.

The first of these generalizations can be implemented by means of standard locking techniques where locks have user-defined parameter values associated with them, and the second can be implemented by allowing locks to be applied recursively. The formal basis for all of this is provided by generalizing the definition of serializability, or *conflict serializability* to be precise. The data sharing stuff is captured by *conditional conflict serializability* (CCSR - defined on pages 29 - 30 in the xymphonic PhD thesis), and the collaborative work stuff is captured by *nested conflict serializability* (NCSR - defined on page 49 in the xymphonic PhD thesis). CCSR and NCSR can be used independently of each other, or in combination.

The result is a much more general transaction model with ACCID rather than ACID properties. That is, transactions are still Atomic, Consistent, and Durable, but the (unconditional) Isolation demanded by the ACID model has been replaced by *Conditional Isolation*, hence the extra C in ACCID. Simply put, this means that users are now free to choose how much concurrency should be allowed. If they want the good, old ACID properties, the xymphonic model will provide that by default. If the concurrency thus provided is inadequate, they can explore the new possibilities that ACCID transactions open up to them. Those who just need a little extra concurrency can stay ashore and merely dip their toes in the ACCID waters, while those with different requirements may want to go for a swim; the xymphonic model does not force users to handle any more complexity than just

what is needed for their particular application. In other words, transactional behavior can be tailored to meet the needs of individual applications (which is why this model is labeled "application-oriented"). Another important aspect of the xymphonic model is that ACID and ACCID transactions can co-exist, without the former influencing the results of the latter (although long-lasting ACCID transactions can cause ACID transactions to wait for locks, thus degrading their performance).

### **Potential business cases**

These are some areas where ACID transactions tend to provide inadequate concurrency:

- All sorts of CAD and CASE systems
- Document handling systems
- Collaborative editing and multimedia conferencing
- Data warehouses that need to be continuously updated
- Workflow management systems
- Virtual reality systems

### **Conclusions**

Many application domains need more concurrency than ACID transactions can provide, and the xymphonic model meets this need. The model is based on a straightforward mathematical generalization of classical transaction theory, and so has a sound formal foundation. Still its greatest strengths are probably *simplicity* and *power*. The xymphonic model is simple to implement and use, yet powerful enough to meet the needs of a large class of applications.