NIK'95, November 20-22

Dynamically Cooperating Database Transactions

Ole Jørgen Anfindsen, Telenor Forskning og Utvikling
Instituttveien 23, 2007 Kjeller, E-post: ole.anfindsen@fou.telenor.no

Abstract: *Cooperative transactions are desirable in many application domains. The approach taken in this paper is to generalize the well known concept of isolation levels, thereby enabling a dynamic cooperation without requiring interaction patterns to be predefined. Uncommitted data is considered to be a special case of unreliable data. The quality of uncommitted data is represented by means of reliability indicators. The latter, combined with logic, is used as a language for communication between cooperating transactions.*

1 Introduction and Motivation

Almost any kind of application could, in principle, benefit from using a database management system (DBMS) for its information storage and retrieval. Advantages of DBMSs are well known, and include data independence, data sharing, concurrent access to data in a controlled fashion, recoverability, and more [9, 15]. The transaction management mechanisms of traditional DBMSs were developed to support short, atomic transactions, e.g. for banking and airline reservation applications. However, some application domains have needs that are different. Frequently mentioned examples include CAD and CASE [14, 42]. A major problem is that such applications need support for long-lived transactions (LLTs) that can handle interaction patterns not allowed by ordinary schedulers. In particular, serializability (which requires a schedule of concurrent transactions to be equivalent to some serial schedule [5]) is too restrictive a correctness criterion in an LLT environment [33].

Generally speaking, there are two different motivations for compromising serializability: (1) because one is unwilling to pay the cost in terms of resource consumption and/or delays (such as memory consumption, CPU consumption, and time spent waiting for locks), or (2) because the semantics are unsuitable for the application domain in question. In order to deal with the first case, commercially available DBMSs typically offer isolation levels that permit various degrees of non-serializability. In [2] we argue that our ideas for inter-transaction communication would be useful where serializability is compromised in the interest of performance, in particular that they would represent a vast improvement over a simple *dirty read* approach. However, this paper will focus on the other case, where non-serializable schedules are not only something one is willing to accept for pragmatic reasons but something that is inherently desirable or even necessary. Examples will be taken from the domain of design applications. It is well known that such applications need support for cooperative transactions [41], i.e. they need transaction mechanisms that will allow access to uncommitted data in a controlled fashion. Below we propose mechanisms that enable an updating transaction to share its uncommitted data with other transactions. The problem of allowing multiple transactions to concurrently update a single data item is beyond the scope of this paper.

Our approach to transaction cooperation is based on: (1) a generalization of the isolation level concept, and (2) the use of reliability indicators and multivalued or annotated logic as a language that enables transactions to communicate about the quality of uncommitted data.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our approach to cooperative transactions. Section 4 discusses the most important concepts and techniques that our approach is based on. Section 5 contains our conclusions.

2 Related Work

References [18, 35, 16] all discuss how to relax the serializability requirement in order to improve performance. They introduce the alternative correctness criteria Semantically Consistent Schedules, Multilevel Atomicity, and Relatively Consistent Schedules, respectively.

Several papers have been published that address the needs of CAD, CASE, or other engineering applications. [34] introduces a new transaction model called Flex, which allows flexible control of the transaction isolation granularity, distinguishing between compensatable and non-compensatable subtransactions. [24] introduces Dynamic Validation, an optimistic concurrency control scheme that avoids holding locks for long periods of time. [32] defines a model where users are able to influence the setting and releasing of locks by means of checkin and checkout operations, thus enabling cooperation within groups of transactions. Other related papers include [11, 12, 30, 41, 42, 43, 45,]. The following two references are of particular interest.

[44] generalizes the two-phase locking protocol by introducing altruistic locking which, under certain conditions, enables LLTs to release locks early. This is somewhat similar to our idea of a transactions downgrading its read or write modes to give other transactions access to its data. However, an important difference is that altruistic locking guarantees serializability.

[31] introduces Database Conversations, "an application-independent, tight framework for jointly modifying common data". In their model, each data unit has a binary *conversation flag* associated with it and the two make up an indivisible unit; "the information whether a data unit is uncertain or not, is stored explicitly with the data item itself and not implicitly in some transaction semantics". Conversation flags correspond somewhat to our *reliability indicators* which also form indivisible units with data attributes: Both are used to tell other transactions something about the status of a data attribute, both are examples of the use of meta-data, and both are based on the notion of uncommitted data being uncertain. But while conversation flags have only two values, reliability indicators can have arbitrarily many values. Thus, this paper shares with [31] the basic idea that uncommitted data is unreliable and that transactions should communicate with each other about the status of such data. On the other hand, their language for inter-transaction communication is based on dynamically created *conversation contexts*, while ours is based on reliability indicators combined with multivalued logic.

3 Cooperative Transactions

At the very core of our approach to cooperative transactions is the following observation: uncommitted data contains unreliable information, i.e., it is associated with some level of uncertainty. Thus, dealing with uncommitted data is just a special case of the general problem of dealing with uncertainty. Many papers have been published on this topic. [13] contains "an evolving bibliography of documents on uncertainty and imprecision in information systems" (but apparently unchanged since 1993), and currently gives 352 references. See e.g. [38, 39] for surveys.

According to [39] "most approaches to the modelling of imprecision in databases are based on the theory of fuzzy sets". An alternative approach is to use *probability* distributions, rather than the *possibility* distributions found in fuzzy set theory. Examples of the latter can be found in e.g. [4, 17, 29]. Both in fuzzy and probabilistic approaches, users will have to associate with data attributes numbers in the interval [0, 1] to denote degrees of membership or probabilities, respectively. Choosing numbers that faithfully reflect reality is clearly a non-trivial task, and [4] acknowledges that this "could be a problem in practice because it may be difficult to know the exact probabilities".

We have therefore chosen a different approach. Instead of forcing users to *quantify* uncertainty, we allow them to *classify* it. That is, users can denote unreliable data as belonging to one or more of a predefined set of unreliability indicators. This set must be chosen so as to meet the needs of the application.

However, we would like to emphasize the following: Our approach to transaction cooperation is *not* dependent on a particular approach to uncertainty management, we are simply using the one that appears to be most suitable for the task at hand. Likewise, a decision to use unreliability indicators rather than fuzzy or probabilistic approaches, does *not* force one to use Extended Five Valued Logic (X5VL) or some other particular logical framework.

Before we go into the details of the cooperation mechanisms, we describe an example scenario that illustrates the purpose of those mechanisms. The scenario is not intended to be realistic, but gives a rough idea of the problems we are trying to address.

3.1 Example Scenario

Consider an engineer working on the design of some part of a new airplane, e.g. the landing gear, using a CAD system with transactional capabilities. Initially, the design is constantly changing. The engineer probably has a lot of information about the new landing gear she is about to design; she might know that her company has signed a contract with a new supplier of tires, that a decision has been made to use a new hydraulic technology, etc. Whether she copies the landing gear of an existing airplane into her work space to modify it or she chooses to work from scratch, there will be a period of time when her design is unstable and inconsistent. During this period it will make sense to protect the design by ordinary write locks, and her CAD system could automatically put the very lowest reliability rating on the design as a whole.

Sooner or later the design reaches a stage where it can be considered a draft; it is complete, it is consistent, and it meets overall design criteria that were given in advance. This is when the need to cooperate with other designers come into full play. She will need to look at the design of other parts of the airplane, such as the wings, the fuel tanks, the hydraulic system, the electric system, etc. Conversely, other designers will need to look at the landing gear design. A number of modifications to many parts of the airplane is to be expected, many of which will make other modifications necessary. For example, it might be necessary to alter the physical placement of some hydraulic pipes, forcing our engineer to modify her landing gear design accordingly.

Designing an entire airplane is an enormous task, and the work evolves through several stages. Presumably, there will be a company policy in place that specifies various levels of approval that designs can achieve. And e.g. the designer of the overall hydraulic system would be interested in knowing the level of approval achieved by

designs that involve hydraulic components. We propose to represent levels of approval by means of reliability indicators.

Assuming that the many engineers involved in designing an airplane are not always synchronized, i.e., that their designs may be at different levels of approval, one can easily conceive of managers and others wanting to produce reports that help them assess progress. Thus, we think it is a good idea to have a connection between reliability indicators and the query language supported by the system.

3.2 Reliability Indicators

As mentioned above, our approach is based on regarding uncommitted data as unreliable, and we choose to represent the latter by means of reliability indicators. Two important things should be noticed at this point. First, our approach does not say anything about the *semantics* of reliability indicators. Second, our approach does not say anything about the *number* of different reliability indicators. This means that in one database one may define one set of reliability indicators, and assign to each of them a specific meaning. In another database one may define another set of reliability indicators, possibly with completely different semantics. Thus, this approach could be tailored to many different application domains.

Without going into implementation details (which we consider irrelevant here), we are assuming that reliability indicator definitions, along with a description of their semantics, are stored in a meta-database to which users have read access. It is assumed that interpretation of reliability indicator semantics takes place at the application level. In other words, an application or its user must know how to interpret reliability indicators when they are encountered, and which reliability indicators to use when updating the database. If users and applications are aware of the way reliability indicators are being used in their system, they have a *language* by which to communicate when running transactions.

Table 1 shows how meta-attributes can be used. It illustrates several kinds of reliability indicators: Ellen's salary is completely missing, and the reason is that her salary has not yet been decided (possibly because she only holds a trainee position in the R&D department). Nils' salary is 20, but the transaction that entered that value has not yet committed. Olaf currently works in the Marketing department, but the decision is not final. Arne has been out of work for some time, that is why he is not assigned to a department (and presumably that is why the company only pays him a salary of 10). And so forth.

Employee	Salary	Meta-data for Salary	Department	Meta-data for Department
Kari	21		Sales	
Olaf	18		Marketing	Not approved
Birger	25		Toys	
Ellen	-	Not decided	R&D	Trainee
Trude	25		-	Not assigned
Nils	20	Uncommitted	International	
Arne	10	Temporarily	-	Unemployed

Table 1: Use of meta-data in a relational context. An attribute and its meta-attribute is considered an indivisible unit. If no meta-value has been assigned, the corresponding value is reliable. A meta-attribute could occupy e.g. one to four bytes, depending on how many different meta-values are needed. Meta-attribute definitions must be stored in a meta-database (not shown).

The relational model is more suitable than an object-oriented data model when illustrating the use of meta-attributes. However, once the principle is understood, extrapolating to an OO context is straightforward: Encapsulated attributes in an object has meta-attributes, and values returned by methods must be returned together with the corresponding meta-values, which are determined by the method when its return value is calculated using the attributes inside the object (this could happen implicitly or explicitly, by default or by choice of the method implementor; depending on the kind of support for manipulation of reliability indicators offered by the method implementation language).

Ending this section, we briefly comment on how applications can retrieve and manipulate reliability indicators. The simple solution is to retrieve a data value and the corresponding meta-value into separate variables and have the application manipulate those variables separately (this is typically what is done with SQL style nulls).

Alternatively, retrieved values could automatically be annotated with meta-values, which would not be lost during data manipulations (e.g. arithmetic operations) [1]. This would require the application programming language to have the same kind of support for reliability indicators as the DBMS.

3.3 Reliability Indicators and Logic

When data values are fully reliable, the result of a scalar comparison like e.g. Salary > 50000 is either true or false, period. However, if a Salary value is known to be unreliable, then one cannot simply return true or false from the above predicate; very loosely speaking, predicate evaluations become as uncertain as the data involved. Considerations like this give rise to a plethora of multivalued logics. A common and extreme case is when data is not only unreliable, but completely missing; SQL style nulls are a well known way of representing missing information, are supported by many DBMSs, and result in three valued logic [6, 8]. [7, 22] advocate the use of two semantically different nulls and corresponding four valued logics. [23] propose a five valued logic. And e.g. the probabilistic approach of [29] gives rise to an infinite number of *degrees of truth* in the interval [0, 1]. Many other logics are defined by papers listed by [13].

The reliability indicator concept [1] has its roots in the null values approach to the problem of missing information [6], and we consider it to be a special case of the ideas advocated by [40]. In order to perform scalar comparisons, evaluate complex predicates, and perform other kinds of manipulations of data values (which may or may not be annotated with reliability indicators) in a well defined manner, we employ the logical framework X5VL. We refer to X5VL as a *framework* since one must define some set of reliability indicators (as well as a set of nulls, if missing information is to be handled) before one can start using it. Since, as mentioned above, one could use some other logical framework than X5VL in this context, no treatment of X5VL is given here. Examples illustrating its use with cooperative transactions are given below. For a description of X5VL, see [1].

3.4 Example Queries

Consider once more the airplane designer scenario and a CAD system based on the X5VL framework. Assume that designs evolve through the approval levels A0, A1, ... A5 Where A0 is used in the initial phase and A5 is the final level of approval. Assume therefore that, corresponding to the approval levels, reliability indicators A0 through A5 have been defined in the meta-database. Also, assume that various parts of a design can be at various levels of approval.

The following examples are meant to illustrate some of the expressive power of X5VL. The syntax used is hypothetical, and should not be interpreted as concrete proposals. We take the liberty of deviating somewhat from the syntax constructs defined in [1].

As in the above scenario, the examples are not meant to be quite realistic. Carefully note that we are not making assumptions about the data model of the CAD system (relational or object-oriented), only about the availability of an SQL like query language.

Example 1:

We want to get an overview of the landing gear design, provided that a certain level of approval has been reached for some key components. We would only want to retrieve information if nuts, bolts, and hydraulics have achieved A4 or higher. The following query may then be used (assuming that DESIGNS identify a set of objects, each of which has attributes NAME, NUTS, BOLTS, and HYDRAULICS):

```
SELECT <whatever>
FROM   DESIGNS
WHERE  NAME = 'Landing gear'
 AND   QUALITY(NUTS, BOLTS,
       HYDRAULICS) IN (A4, A5)
```

QUALITY is a hypothetical operator that simply extracts the values of the reliability indicators of the attributes given to it as parameters. These values are compared to those of the IN list, yielding true or false. If the CAD system offered no X5VL support, and levels of approval were stored in ordinary attributes (NUTS_APPROVAL, BOLTS_APPROVAL, and HYDRAULICS_APPROVAL, say) rather than reliability indicators (which are *meta*-attributes), one would need this kind of a query instead:

```
SELECT <whatever>
FROM   DESIGNS
WHERE  NAME = 'Landing gear'
 AND   NUTS_APPROVAL IN ('A4','A5')
 AND   BOLTS_APPROVAL IN ('A4','A5')
 AND   HYDRAULICS_APPROVAL
       IN ('A4','A5')
```

The point is that the use of reliability indicators enables the system to offer a lot of expressive power while the more ad-hoc approach forces the user to explicitly identify all possible cases. The difference between the two queries becomes greater if the number of attributes and/or reliability indicators is increased.

Example 2:

We would like information about all designs of things that occupy little space, but weigh and cost a lot. Again, the reliability level must be A4 or higher, and the attributes named below are assumed to be present in design objects. The following query may be used:

```
SELECT <whatever>
FROM   DESIGNS
WHERE  TRUTHVALUE(SPACE < 5
 AND   WHEIGHT > 777
 AND   COST > 999) IN QTRUE(A4, A5)
```

The important thing here is the use of the hypothetical TRUTHVALUE operator, which takes as its parameter an arbitrarily complex predicate, followed by the keyword IN and then a truth value specification (soon to be explained). In an X5VL environment predicates that involve annotated values do not evaluate to true or false, instead they evaluate to a member of the set qtrue (quasi-true) or qfalse (quasi-false), respectively. For example, if SPACE contains the value 3 annotated with reliability indicator A4, the predicate SPACE < 5 evaluates to the A4-member of the qtrue set. However, SPACE > 5 would evaluate to the A4-member of the qfalse set. In other words, qtrue values are true-like but weaker than true, and qfalse values are false-like but weaker than false. X5VL defines the result of any combination of true, qtrue, qfalse, and false truth values (as well as a fifth class referred to as maybe (X5VL without the maybe class is X4VL)), and the TRUTHVALUE clause is used to verify that the predicate evaluates to the range of truth values specified by the IN clause. According to the rules of X5VL, if SPACE, WEIGHT, and COST are all annotated with A4 or A5, the above predicate will evaluate to A4, A5, or A4/A5 in the qtrue class, i.e. QTRUE(A4, A5). If at least one of SPACE, WEIGHT, and COST is annotated with something else than A4 or A5, the above predicate will evaluate to something outside of QTRUE(A4, A5), in which case the TRUTHVALUE operator will return false.

If the CAD system supported cooperative transactions and reliability indicators, but had not integrated the latter with X5VL or some other logic, one would have less expressive power. The above query would have to be written as:

```
SELECT <whatever>
FROM   DESIGNS
WHERE  SPACE < 5
 AND   SPACE_APPROVAL IN ('A4','A5')
 AND   WEIGHT > 777
 AND   WEIGHT_APPROVAL IN ('A4','A5')
 AND   COST > 999
 AND   COST_APPROVAL IN ('A4','A5')
```

Again, the difference between the two approaches would have been more dramatic had the query involved more attributes and/or reliability indicators.

Remarks

The above examples make the point that representing data quality in meta-attributes enables compact and elegant query expressions. Another important point is that representing data quality by extra attributes causes schemas to double in size; a considerable disadvantage for complex applications with large schemas.

4 Underlying Concepts and Techniques

4.1 Isolation Levels

A common way to specify concurrency properties of transactions in DBMSs is isolation levels. Our approach to cooperative transactions may be seen as generalizing isolation levels. This concept was first introduced in [25] under the name *degrees of consistency* (see [26] for a textbook treatment, or [3] for a tutorial style treatment). Although [25] introduced isolation levels in a locking context, they may be considered implementation independent and defined without reference to locking or any other mechanism by which the desired concurrency properties are achieved. This is the case in the SQL92 standard [36] which defines the different isolation levels only by means of various phenomena that they prohibit (for a more readable treatment of SQL92, see e.g. [10, 37]). Traditional isolation levels include:

a. *Read uncommitted* (RU, also known as uncommitted read) allows an application to read both committed and uncommitted data.

b. *Read committed* (RC, also known as committed read) allows an application to read committed data only.

c. *Cursor stability* (CS) allows an application to read committed data only and guarantees that an object will not change as long as a cursor is positioned on it.

d. *Repeatable read* (RR) allows an application to read committed data only and guarantees that read data will not change until the transaction commits or aborts (thus, a read that is repeated will return the original row unchanged). RR will not prevent the so called phantom phenomenon, i.e., when a cursor is reopened an object not present the previous time may appear.

e. *Transaction consistency* (TC, also known as serializable) guarantees that the transaction has a consistent view of the database (as if no other transactions were active). We prefer to refer to this isolation level as TC rather than SR (serializable) because (1) a TC transaction may be part of a non-serializable schedule where other transactions use lower levels of isolation, and (2) because TC is a more intuitively appealing name in a nested transactions environment (see [2] for a discussion of dynamic usage of isolation levels in a nested transaction context).

Except for CS, the other isolation levels are defined in the SQL92 standard. In [2, 3] an additional isolation level called *Query Consistency* (QC), which falls between CS and RR, is defined. QC allows an application to read committed data only, and guarantees that all objects accessed in a single query is consistent, i.e., all objects accessed by one query is from a single, committed state of the database. Implementing QC by means of locking is straightforward; all read locks must be kept until the query is completed (until the cursor is closed).

Note that traditional isolation levels do not in any way influence the handling of write-write or write-read dependencies, only read-write dependencies. This can be seen e.g. by analyzing the locking protocols used to implement various isolation levels: The protocols for write locks are always two-phase in that no write locks are released before the transaction terminates (i.e. commits or aborts), irrespective of isolation level. The protocols for read locks on the other hand, are directly influenced by the isolation level. Only with TC (and certain implementations of RR, like e.g. in the DB2 Family which actually provides full TC support under the name RR) are read lock protocols two-phase. With less restrictive isolation levels read locks may be released *before* the transaction terminates, and new ones may still be acquired. Simply put, traditional isolation levels, when implemented by means of locking, determine the *duration* of read locks, and nothing else.

Below we generalize the isolation level concept in such a way that the handling of write-read dependencies is influenced too. This is achieved through the introduction of new read and write modes, which give rise to *protection levels* and *share levels*, respectively. Protection levels that only use traditional read modes, coincide with traditional isolation levels.

4.2 Access Modes

An important concept when dealing with concurrent use of databases, is that of *access mode*. There are two basic access modes; read and write. When a database object is accessed in read mode, the agent in question can only perform read operations on that object. Two or more transactions may access a given object concurrently, provided each uses read mode. When a database object is accessed in write mode, the transaction in question can perform both read and write operations on that object. More specifically, write mode enables reading, deleting, inserting, and modifying objects. If an object is accessed in write mode by one transaction, no other transaction can access that object in either read or write mode.

In addition to these two basic access modes, many DBMSs support the following three: browse, update, and exclusive. Browse mode enables a transaction to read an object even if some other transaction is currently accessing it in write mode. Thus, when using browse mode, transactions have no guarantee of a consistent view of the

database, since there is a risk that they will read uncommitted data. The use of browse mode is often denoted *read through* or *dirty read*, and is used with isolation level RU. Update mode is like read mode with the added semantics that the transaction in question may at any time request a promotion from update mode to write mode. If an object is accessed in update mode by one transaction, other transactions may access that object in read mode, but no other transaction can access that object in either update or write mode. Update mode and the corresponding locks are introduced to prevent single-object deadlock [26]. If an object is accessed in exclusive mode by one transaction, no other transaction may access that object, irrespective of access mode. Exclusive mode is necessary when even browsers must be kept away, e.g. when a table is to be dropped from a relational database.

Thus, there are five traditional access modes. We will refer to these five access modes frequently in the following, and will therefore denote each of them by a single letter:

B - browse
R - read
U - update
W - write
X - exclusive

We will refer to B, R, and U as read modes, and to W and X as write modes. Later, we will introduce new read and write modes. The relationship between the five access modes that was described above, is nicely captured by the following compatibility matrix:

	B	R	U	W	X
B	*	*	*	*	
R	*	*	*		
U	*	*			
W	*				
X					

Asterisks indicate compatibility. For example, R is compatible with B, R, and U; W is compatible only with B; and X is not compatible with any other access mode.

4.3 Implementation of Access Modes

Implementing the five access modes by means of locks is fairly straightforward (actually, many authors choose not to distinguish between locks and access modes at all). However, there is a need to support different lock granularities [25]. Typical granularities could be tuple/object, page, table/class, index, file, area/tablespace, and database. Loosely speaking, the granularities of choice will correspond to the resource hierarchy at hand. If a given transaction is to use page locks, say, there will be a need for three page lock modes with the following lock compatibility matrix:

	R	U	W
R	*	*	
U	*		
W			

In this paper we use *symmetric* compatibility matrices only but see [26] for a discussion of why one might want to make them *asymmetric*, and see [2] for a proposal of *priority locking* that avoids the dilemma of symmetric/asymmetric matrices. As can be seen, there is a one-to-one correspondence between this matrix and a *subset* of the access mode compatibility matrix (it will soon be explained why a subset suffices). There is, by the way, not a universally agreed upon convention for lock names. However, R locks are often called S (for share), and W locks are often called X (for exclusive). We prefer to denote read locks by R instead of S, since we later introduce locks that are more shareable than these. And since write locks are not truly exclusive, we think W is better than X (some people find it amusing that certain vendors have to refer to the truly exclusive lock (see below) as e.g. *super*-exclusive).

Unless it is required that all transactions use the same lock granularity, one must be able to coordinate concurrent transactions that request locks at different levels in the resource hierarchy. Except that U locks were apparently first described by [27], the solution given in [25] and implemented in a large number of commercial systems, is as follows: a transaction will request R, U, and W locks at some level of choice, it will not request any locks at lower levels, but it will request *intent* locks at all higher levels. (Simplifications of this procedure by means of clever tricks with low-level physical mechanisms will not be considered here. Also, it is possible for a transaction to use different lock granularities for different statements, but this is not significant for the discussion at hand.) Two basic intent locks are needed; IR indicates an intent to request R locks at some lower level, and IW indicates

an intent to request W locks at some lower level. No IU lock is needed, because a transaction that intends to request U locks at some lower level also has an implicit intent to request W locks (otherwise there would be no need for U locks in the first place). Thus, such a transaction must use IW. There turns out to be need for one more lock, RIW, which is a combination of R and IW; it provides R-access to the entire resource in question (e.g. a class extension) while also enabling the transaction to request U and W locks at some lower level (e.g. page or object). The complete lock compatibility matrix looks like this:

	B	IR	R	U	IW	RIW	W	X
B	*	*	*	*	*	*	*	
IR	*	*	*	*	*	*		
R	*	*	*	*				
U	*	*	*					
IW	*	*			*			
RIW	*	*						
W	*							
X								

The necessity of an exclusive access mode (and hence an X lock to implement it) was briefly outlined above. Note that the RIW row is identical to the intersection of the R and IW rows (and since the matrix is symmetric, the same applies to the RIW, R, and IW columns). In practice, B locks, and hence X locks, are not used at the lowest levels such as tuple/object or page (since that would - partially at least - have defeated the purpose of using isolation level RU in the first place). For example, in a relational DBMS an RU transaction will typically request a B lock at the table level (and all levels above), and then proceed without requesting any locks on pages or tuples (however, it will request some sort of low-cost, short duration locks (known as latches) on pages or tuples to ensure atomicity of individual read operations).

In order to figure out which locks are compatible with which, simply consider the definitions of the access modes involved. For example, the R lock is compatible with the other read locks (B, IR, and U), but not compatible with any of the write locks (IW, RIW, W, and X). Or consider the IW lock; obviously compatible with B, compatible with IR since potential conflicts between IW- and IR-transactions are resolved at some lower level, not compatible with R and U since these locks preclude writers, compatible with IW since potential conflicts between two IW-transactions are resolved at some lower level, and finally IW is of course not compatible with W and X locks. By reasoning like this the entire matrix follows.

Some vendors denote browse locks with IN, for intent none. Just like denoting read locks with S, this seems to reflect the perspective of implementors rather than users. Since the work presented here is part of a project whose goal is to define and describe Application-Oriented Transaction Management (AOTM), we prefer things to be the other way around, i.e., we think names should be chosen with a user perspective in mind.

We conclude this section with some brief examples (assuming for now an SQL environment). For an RR transaction that wants to scan an entire table, it is probably a good idea (for the DBMS component that makes the decision) to request an R lock on the table in question, and then IR locks on the tablespace and database. If the same transaction used isolation level CS instead of RR, R locks on pages and IR locks on the levels above might be a better idea. If a CS transaction wants to scan (parts of) a table with an updatable cursor, it may use U locks (which will be promoted to W locks as needed) on pages and IW locks at the levels above. If, for some reason, the latter transaction requests an R lock on the table that it already has an IW lock on, that IW lock will be promoted to an RIW lock (IW locks at higher levels remain unaffected). It may be worth noting that (even though the name does not indicate so) an IW lock on a given level enables its holder to request R, U, and W locks (not just W locks) on lower levels. This is quite natural since W locks are stronger than the other two.

4.4 New Access Modes

We are now ready to extend the concepts described above. We have seen that RU transactions can read uncommitted data. Thus, reading in browse mode could be thought of as a *very* primitive kind of cooperation between transactions. It is primitive because (1) there is - in general - no knowledge of who is cooperating with whom, (2) browsers don't get to know whether data they read is committed or not, and (3) whenever uncommitted data is encountered, the browsers receive no information about the quality or status of the data. Thus, for applications that need the data sharing kind of cooperation (design and engineering applications are classical examples), using RU transactions is an unsatisfactory solution.

As a first step away from the blind cooperation provided by RU, we introduce a write mode that indicates willingness to share information with readers. For historic reasons (see above) we will not refer to this access mode (nor its corresponding lock) as S, but denote it by F for friendly. The idea is that a friendly writer indicates to other transactions that it is *relatively safe* to read its data. This could be so simply because statistical analysis shows that

some transactions hardly ever abort. Or in e.g. a design environment one may reach a point in the process where the design has stabilized sufficiently that other transactions should be given read access. Admittedly, this alone would not be much to shout about. But combined with the use of reliability indicators and logic as a language for inter-transaction communication, we believe it is a powerful approach to transaction cooperation.

Obviously, many readers need the strong protection offered by read mode R, so the introduction of write mode F should not influence R readers. Instead, we introduce a read mote that corresponds to F, and call it L for low protection. Thus, the use of L mode indicates willingness to read data that belongs to F writers.

The resulting access mode compatibility matrix is a trivial extension of the old one:

	B	L	R	U	F	W	X
B	*	*	*	*	*	*	
L	*	*	*	*	*		
R	*	*	*	*			
U	*	*	*				
F	*	*					
W	*						
X							

Remarks

The introduction of new access modes does not necessarily stop here. It might be desirable to provide more fine granularity control by introducing multiple F and L modes, e.g. F1, F2, ... , and L1, L2, etc. Further, one could imagine read modes that have *conditions* associated with them. Such conditions could be implicitly given by the query performing the read operations (i.e., by the predicates of the query), or explicitly given by extra predicates. We refer to these read modes as Q for query and P for predicate, respectively. Likewise, a write mode that accepts restrictions imposed by the just mentioned read modes, referred to as C for constrained, can be introduced. Loosely speaking, the conditional read modes create a continuum that fills the gap between L and R, while C creates a continuum of write modes that are weaker than F.

Access modes Q, P, and C are not important concerning the understanding of our approach to cooperative transactions. However, they are briefly mentioned here partly to indicate the versatility of our approach, and partly to answer questions like the following: What if a transaction is willing (or even wants) to read uncommitted data of a certain quality, but still needs to scan data more than once, how can such requirements be combined? That is, if an RR transaction scans part of the database in read mode L for the second time, would it not be possible that the data in question could have been deleted since the first scan, or modified in such a way that it no longer satisfied the search criteria? And yes, the latter scenario is possible unless precautions are taken. One approach could be to put restrictions on the allowable locking protocols, such as e.g. prohibiting deletions to be performed with F locks (probably a very good idea in and of itself, but unfortunately it solves only part of this problem). Another approach could be to provide readers with needs of the kind mentioned above with conditional read modes. Conditional access modes (under somewhat different names), as well as the locks needed to implement them, are discussed in [2].

4.5 Implementation of New Access Modes

The introduction of two new access modes, makes it necessary to introduce seven new locks (assuming all the time that locking is the chosen implementation method); two new basic locks (L and F), two new intent locks (IL and IF), and three new read-intent combinations (LIF, RIF, and LIW). The lock compatibility matrix now becomes as shown in Table 2.

We do not consider the increasing number of lock modes a serious run-time overhead problem. Held locks reside in (more or less) dynamically allocated storage blocks, each the size of several bytes (identifying the held resource, the holding transaction, lock duration, lock mode, and more). With 15 instead of 8 lock modes, one needs 4 instead of 3 bits to represent the lock mode. Even with hundreds of new lock modes, space requirements per lock would grow less than a byte.

Remarks

A vendor that were to implement generalized isolation levels would probably want to give careful consideration to details of locking protocols. For example, one may want to make sure certain operations are always covered by W or X locks, irrespective of the isolation level of the transaction in question. A discussion of locking protocols is considered beyond the scope of this paper.

	B	IL	IR	L	R	U	IF	IW	LIF	RIF	LIW	RIW	F	W	X
B	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
IL	*	*	*	*	*	*	*	*	*	*	*	*	*		
IR	*	*	*	*	*	*	*	*	*	*	*	*			
L	*	*	*	*	*	*	*		*	*			*		
R	*	*	*	*	*	*									
U	*	*	*	*	*										
IF	*	*	*	*			*	*	*		*				
IW	*	*	*				*	*							
LIF	*	*	*	*			*		*						
RIF	*	*	*	*											
LIW	*	*	*				*								
RIW	*	*	*												
F	*	*		*											
W	*														
X															

Table 2: Support for additional access modes L and F result in this lock compatibility matrix.

Likewise, this paper does not allow for a discussion of recovery considerations. However, one should keep in mind that LLTs in general need more versatile recovery mechanisms than traditional transactions. For example, in the case of power or media failure it simply is not acceptable to completely undo a transaction that has been running for several weeks; such transactions must be brought back where they left off before the crash (or at least close to that state).

4.6 Generalized Isolation Levels

As already mentioned, traditional isolation levels deal with read-write dependencies only, and in terms of locking this means varying the *duration* of read locks. The introduction of access modes L and F modifies the way both read-write and write-read dependencies are handled. We use *protection level* to denote an application's use of a read mode and its duration. We use *share level* to denote an application's use of write modes. The combination of share and protection levels is referred to as *concurrency levels*.

Below is a list of the different protection levels that result from combining traditional isolation levels (including query consistency, see above) with read modes L and R. Since locking is the standard implementation technique for isolation levels, we briefly indicate for each protection level the read lock duration and protocol:

1 read uncommitted - RU (no low level read locks at all, only B locks at higher levels, and latches at the lowest level to ensure atomicity of individual read operations)

2 read with low-protection read - RL ("zero duration" L locks)

3 read committed - RC ("zero duration" R locks)

4 low-protection cursor stability - LCS (CS protocol with L locks)

5 cursor stability - CS (CS protocol with R locks)

6 low-protection query consistency - LQC (query duration L locks)

7 query consistency - QC (query duration R locks)

8 low-protection repeatable reads - LRR (RR protocol with L locks)

9 repeatable reads - RR (RR protocol with R locks)

10 low-protection transaction consistency - LTC (TC protocol with L locks)

11 transaction consistency - TC (TC protocol with R locks)

Assuming that write modes can be specified independently for insert, delete, and update operations, there are 8 possible share levels (not all of which are equally useful). Combining this with the 11 protection levels shown

above, gives 88 concurrency levels. With additional read and write modes this number will increase rapidly. This should not be a problem, since users who don't want or need hundreds of concurrency levels to choose from can stick to the good old isolation levels with which they are familiar (this is so because traditional isolation levels are special cases of concurrency levels). For other users it means flexibility to tailor concurrency level usage to specific application needs.

Concurrency levels have no influence on write lock durations; write locks are never released until termination (otherwise one would not be able to guarantee recoverability [5]).

4.7 General Remarks

It is possible for a piece of data to simultaneously be unreliable in more than one way. For example, a geographical information system may store satellite photos that have different resolutions, were taken at different angles, and have different ages; these are three dimensions of unreliability. A piece of data may be inaccurate because it is an estimate, but if it has been written by a not yet committed transaction, that adds another dimension to its unreliability. Adding features to X5VL to support multidimensional unreliability is straightforward [2]. Thus, X5VL may be used not only as a language for communication between cooperating transactions, but also to integrate this with the ability to deal with missing and (several kinds of) unreliable information in general.

We believe the generality of our approach is an advantage in several ways. Consider e.g. this pragmatic argument. The DBMS vendors' primary concern is to make money, therefore their development is driven by market demand. Because the engineering community is only a small fraction of the DBMS market, their needs have traditionally not been at the top of the priority lists of DBMS vendors. Thus, we believe it is an advantage that the ideas presented in this paper are not limited to use in engineering applications only, but have a number of potential uses in business applications as well [1, 2]. We are e.g. pleased to see that Version 4 of DB2 for MVS will add limited support for an AOTM feature [28].

5 Conclusions

Our approach to cooperative transactions and data sharing generalizes a well understood and widely used concept, isolation levels. Therefore, we believe it would be relatively simple to implement and use. The use of reliability indicators and logic (X5VL) as a language for inter-transaction communication enables cooperation to be dynamic, rather than having to be predefined and static. It is an advantage that our approach can be integrated with mechanisms for dealing with unreliable and missing information in general, thereby unifying important aspects of data manipulation and transaction management. It is also an advantage that both the X5VL and AOTM concepts are relevant in a number of application domains.

Acknowledgments

In 1993 and 1994 I spent a year as a guest researcher at GTE Laboratories Inc., and was exposed to the Transaction Specification and Management Environment (TSME) [19, 20, 21]. The AOTM ideas were conceived during that year and inspired by the TSME. I am particularly indebted to Dimitrios Georgakopoulos and Mark Hornick who helped me write the technical reports that this paper is based on and with whom I had numerous fruitful discussions. Many thanks to my PhD advisor, professor Oddvar Risnes, UNIK Center for Technology at Kjeller, who read an early draft of this paper and provided many useful suggestions. Special thanks to professor Ragnar Normann, University of Oslo, for several years of friendship and support.

References

[1] Anfindsen, O J, Georgakopoulos, D, Normann, R. 1994. *Extended three and five value logics for reasoning in the presence of missing and unreliable information.* Kjeller, Norwegian Telecom Research (technical report TF R 9/94).

[2] Anfindsen, O J, Hornick, M. 1994. *Application-Oriented Transaction Management.* Kjeller, Norwegian Telecom Research (technical report TF R 24/94).

[3] Anfindsen, O J, Hornick, M. 1994. Isolation levels in relational database management systems, *Telektronikk*, 90, (4), 71-72.

[4] Barbara, D, Garcia-Molina, H, Porter, D. 1990. A probabilistic relational data model. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology - EDBT'90*. Venice, Italy.

[5] Bernstein, P A, Hadzilacos, V, Goodman, N. 1987. *Concurrency control and recovery in database systems.* Reading, Mass., Addison-Wesley.

[6] Codd, E F. 1979. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4, (4), 397-434.

[7] Codd, E F. 1990. *The relational model for database management: version 2.* Reading, Mass., Addison-Wesley.

[8] Date, C J. 1982. Null values in database management. In: *Proceedings of 2nd British National Conference on Databases.* Bristol, England.

[9] Date, C J. 1990. *An introduction to database systems, volume 1* (fifth edition). Reading, Mass., Addison-Wesley.

[10] Date, C J, Darwen, H. 1993. *A guide to the SQL standard*. Reading, Mass., Addison-Wesley.

[11] Dayal, U, Hsu, M, Ladin, R. 1990. Organizing long-running activities with triggers and transactions. In: *Proceedings of ACM SIGMOD International Conference on Management of Data*. 204-214.

[12] Dayal, U, Meichun, H, Ladin, R. 1991. A transactional model for long-running activities. In: *Proceedings of the 17th VLDB Conference.* Barcelona, Spain.

[13] Dyreson, C E. 1993. *A bibliography on uncertainty management in information systems.* Department of Computer Science, University of Arizona (ftp from cs.arizona.edu).

[14] Elmagarmid, A K (ed.). 1992. *Database transaction models for advanced applications.* San Mateo, Calif., Morgan Kaufmann Publishers.

[15] Elmasri, R, Navathe, S B. 1989. *Fundamentals of database systems*. Redwood City, Calif., Benjamin/Cummings.

[16] Farrag, A A, Özsu, M T. 1989. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14, (4), 503-525.

[17] Feng, Z, Jia, Y, Miller, M. 1991. Null values in relational DBMS. In *Proceedings of the 2nd Australian Database & Information Systems Conference*. Australia.

[18] Garcia-Molina, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8, (2), 186-213.

[19] Georgakopoulos, D, Hornick, M, Krychniak, P. 1993. An Environment for the Specification and Management of Extended Transactions in DOMS. In *Proceedings of the Third International Workshop on RIDE-IMS'93*, Vienna, Austria.

[20] Georgakopoulos, D, Hornick, M, Manola, F, et al. 1993. An Extended Transaction Environment for Workflows in Distributed Object Computing. *Data Engineering*, 16, (2).

[21] Georgakopoulos, D, Hornick, M, Krychniak, P, Manola, F. 1994. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proceedings of the 10th Int. Conf. on Data Engineering*, Houston, Texas.

[22] Gessert, G H. 1990. Four valued logic for relational database systems. *ACM SIGMOD Record*, 19, (1), 29-35.

[23] Gottlob, G, Zicari, R. 1988. Closed world databases opened through null values. In: *Proceedings of the 14th VLDB Conference.* Los Angeles, California.

[24] Goyal, P, Narayanan, T S, Sadri, F. 1993. Concurrency control for object bases. *Information Systems*, 18, (3), 167-180.

[25] Gray, J N, Lorie, R A, Putzolu, G R, Traiger, L I. 1976. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of IFIP Working Conference on Modelling of Data Base Management Systems.* Freudenstadt, Germany. 695-723. Also in *Modelling in Data Base Management Systems*, G M Nijssen, Ed., Elsevier North-Holland, 1976, 365-395.

[26] Gray, J, Reuter, A. 1993. *Transaction processing: concepts and techniques.* San Mateo, Calif., Morgan Kaufmann Publishers.

[27] Gray, J. 1980. *Experience with the System R lock manager.* IBM San Jose Research Laboratory, internal memo.

[28] Hoover, C, Rizner, J, Yevich, R. 1994. DB2 Version 4: our dreams come true? *IDUG Solutions Journal*, 1, (2).

[29] Jia, Y, Feng, Z, Miller, M. 1992. A multivalued approach to handle nulls in RDB. In *Future Database'92, Proceedings of the Second Far-East Workshop on Future Database Systems.* Kyoto, Japan, 71-76.

[30] Kim, W, Lorie, R, McNabb, D, Plouffe, W. 1984. A transaction mechanism for engineering design databases. In: *Proceedings of the 10th VLDB Conference*. Singapore, 355-362.

[31] Kirsche, T, Lenz, R, Ruf, T, Wedekind, H. 1994. Cooperative problem solving using database conversations. In *Proceedings of the 10th Int. Conf. on Data Engineering*, Houston, Texas, 134-143.

[32] Klahold, P, Schlageter, G, Unland, R, Wilkes, W. 1985. A transaction model supporting complex applications in integrated information systems. In: *Proceedings of ACM SIGMOD International Conference on Management of Data*. 388-401.

[33] Korth, H F, Speegle, G. 1994. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19, (3), 492-535.

[34] Leu, Y, Elmagarmid A K, Boudriga, N. 1992. Specification and execution of transactions for advanced database applications. *Information Systems*, 17, (2), 171-183.

[35] Lynch, N. 1983. Multilevel atomicity - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8, (4), 484-502.

[36] Melton, J (ed.). 1992. *Information processing systems - database language SQL 2*. ISO/IEC 9075 : 1992.

[37] Melton, J, Simon, A R. 1993. *Understanding the new SQL: a complete guide*. San Mateo, Calif., Morgan Kaufmann Publishers.

[38] Motro, A. 1990. Accommodating imprecision in database systems: issues and solutions. *ACM SIGMOD Record*, 19, (4).

[39] Motro, A. 1990. Imprecision and incompleteness in relational databases: survey. *Information and Software Technology*, 32, (9).

[40] Motro, A. 1992. Annotating answers with their properties. *ACM SIGMOD Record*, 21, (1).

[41] Nodine, M, Ramaswamy, S, Zdonik, S. 1992. A cooperative transaction model for design databases. In: *Database transaction models for advanced applications.* Ed. A Elmagarmid. San Mateo, Calif., Morgan Kaufmann Publishers.

[42] Nodine, M, Zdonik, S. 1992. Cooperative transaction hierarchies: Transaction support for design applications. *VLDB Journal*, 1, (1), 41-80.

[43] Rauft, M A, Rehm, S, Dittrich, K R. 1990. How to share work on shared objects in design databases. In: *Proceedings of the International Conference on Data Engineering*, 575-583.

[44] Salem, K, Garcia-Molina, H, Shands, J. 1994. Altruistic locking. *ACM Transactions on Database Systems*, 19, (1), 117-165.

[45] Skarra, A H. 1991. A model of concurrent cooperating transactions in an object-oriented database. *Lecture Notes in Computer Science 492*, 352-368.