

# Specification of a Distributed Database [DRAFT]

Kerry Raymond

Key Centre for Software Technology  
Department of Computer Science  
University of Queensland  
St. Lucia QLD 4067  
AUSTRALIA

Discussion Document No. 15

2nd October 1989\*

## 1 The Database Itself

DATA is the type of all objects in the database.

[*DATA*]

LOBJID is the set of logical object identifiers.

[*LOBJID*]

TRANSID is the set of transaction identifiers.

[*TRANSID*]

---

\*printed December 15, 1989

PHYSDBID is the set of identifiers of physical databases.

[*PHYSDBID*]

Initially I only want to work with a centralized system, so

$\#PHYSDBID = 1$

Databases (in the logical sense of a collection of named data) can be associated with either a physical database or a transaction in execution. DBID is the set of database identifiers.

$DBID \cong physdb \ll PHYSDBID \gg |trans \ll TRANSID \gg$

I'll assume that operations on the databases (both physical and transactional) are serial for any given DBID, so therefore version numbers on the database states can be  $\mathbb{N}$  for the moment.

$VERSION \cong \mathbb{N}$

$InitialVERSION : VERSION$ $precedes : VERSION \leftrightarrow VERSION$
--

$InitialVERSION = 0$ $precedes = \mathbb{N}. <$
--

*MaxVersion*

$vs : \mathbb{P} VERSION$ $vmax : VERSION$
---

$\#vs \geq 1$ $vmax \in vs$ $\forall v : vs - \{vmax\} \bullet$ $vmax \in precedes^+(v)$
---

*NextVersion*

$\Delta v : VERSION$
----------------------

$v' \in precedes(v)$ $\forall ov : precedes(v) - \{v'\} \bullet$ $v' \notin precedes^*(ov)$
---

So we can uniquely identify the state of a database (i.e. a particular set of named values) by DBVERSION.

*DBVERSION*

*dbid : DBID*  
*version : VERSION*

*FirstTVersion*

*t : TRANSID*  
*dbv : DBVERSION*

*dbv.dbid = trans(t)*  
*dbv.version = InitialVersion*

*FirstPVersion*

*p : PHYSDBID*  
*dbv : DBVERSION*

*dbv.dbid = physdb(p)*  
*dbv.version = InitialVersion*

*NextVersion*

$\Delta dbv : DBVERSION$

*dbv'.dbid = dbv.dbid*  
*dbv'.version = dbv'.version + 1*

We can uniquely identify a particular object in a particular database state with OBJVERSION.

*OBJVERSION*

*DBVERSION*  
*lobjid : LOBJID*

An object in a database is therefore just data.

*Value*

*obj* : *OBJVERSION*

*data* : *DATA*

'Returns' the data corresponding to the specified object version.

$\exists dbv : DBVERSION, lobj : LOBJID \bullet$

$dbv.dbid = obj.dbid$

$dbv.version = obj.version$

$lobj = obj.lobjid$

$dbv \in \text{dom } db$

$lobj \in \text{dom } db(dbv).dbobjs$

$data = db(dbv).dbobjs(lobj).data$

ValidObj checks to see if an OBJVERSION exists within the DBSYSTEM

$ValidObj \cong Value \setminus \{data\}$

ExtractObjs is used to make 'copy's of the requested logical objects.

This involves copying their data and saying that they were obtained by copying.

<p><i>CurrentPVersion</i></p> <p><math>p : \text{PHYSDBID}</math>  <math>pdbv : \text{DBVERSION}</math></p> <hr/> <p><math>pdbv \in \text{dom } db</math>  <math>pdbv.dbid = \text{trans}(t)</math>  <math>\forall dbv : \text{dom } db \bullet</math>  <math>dbv.dbid = pdbv.dbid \Rightarrow</math>  <math>pdbv.version \in \text{precedes}^*(dbv.version)</math></p>
<p><i>NextVersion</i></p> <p><math>\Delta dbv : \text{DBVERSION}</math></p> <hr/> <p><math>dbv'.dbid = dbv.dbid</math>  <math>dbv'.version \in \text{precedes}^+(dbv.version)</math>  <math>\neg \exists v : \text{VERSION} \bullet</math>  <math>v \in \text{precedes}^+(dbv.version)</math></p>

## 2 Concurrency Control Examples

In this section, we test drive the model on some common concurrency control algorithms.

First we describe the general form of the specification of a concurrency control algorithm that I will use.

### 2.1 General Format for Concurrency Control Schemas

<p><i>ConcurrencyControl</i></p> <p><math>PDB : \text{PHYSDBID}</math></p>
--

ReadOperation

$t : TRANSID$   
 $lobjs : P LOBJID$

$\exists \Delta tdbv, pdbv : DBVERSION, \Delta tdb, rdb : DBSTATE \bullet$   
    *CurrentPVersion*  
    *ExtractObj*[ $pdbv/dbv, lobjs/lobjs, rdb/dbs$ ]  
    *CurrentTVersion*  
    *ExtractAll*[ $tdbv/dbv, tdb/dbs$ ]  
     $tdb' = tdb \oplus rdb$   
    *NextDBVersion*[ $tdbv/dbv, tdbv'/dbv'$ ]  
    *AddDB*[ $tdbv'/dbv, tdb'/dbs$ ]

WriteOperation

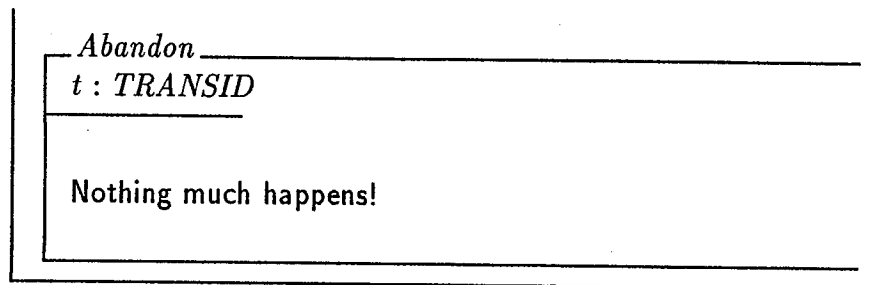
$t : TRANSID$   
 $wdb : DBSTATE$

$\exists \Delta tdbv, \Delta pdbv : DBVERSION, \Delta tdb, \Delta pdb : DBSTATE \bullet$   
    *CurrentTVersion*  
    *ExtractAll*[ $tdbv/dbv, tdb/dbs$ ]  
     $tdb' = tdb \oplus wdb$   
    *NextDBVersion*[ $tdbv/dbv, tdbv'/dbv'$ ]  
    *AddDB*[ $tdbv'/dbv, tdb'/dbs$ ]  
    *CurrentPVersion*  
    *ExtractAll*[ $pdbv/dbv, pdb/dbs$ ]  
     $pdb' = pdb \oplus wdb$   
    *NextDBVersion*[ $pdbv/dbv, pdbv'/dbv'$ ]  
    *NextDBVersion*[ $pdbv/dbv, pdbv'/dbv'$ ]

Commit

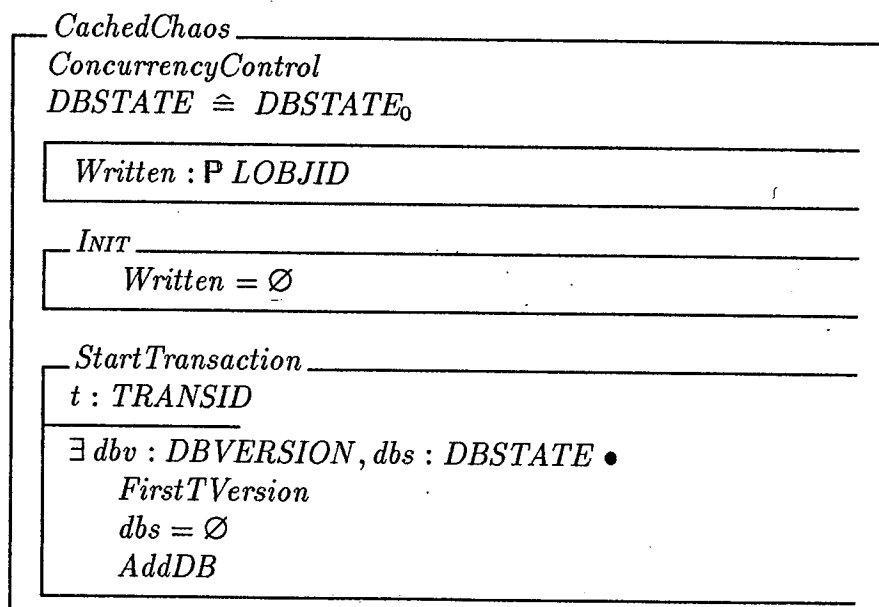
$t : TRANSID$

Nothing much happens!



### 2.3 Cached Chaos

Cached Chaos is the concurrency control that reads from its transaction database rather than the physical database where possible, and always writes to its transaction database. Commit writes the updated values to the physical database. Abandon does nothing to the physical database. Therefore Cached Chaos is a safer database than Chaos, in that abandoned transactions do not affect the database.



*ReadOperation*

$t : TRANSID$   
 $robjs : \mathbb{P} LOBJID$

$\exists \Delta tdbv, pdbv : DBVERSION, \Delta tdb, rdb : DBSTATE, mrobjs : \mathbb{P} LOBJID \bullet$

*CurrentTVersion*

*ExtractAll*[ $tdbv/dbv, tdb/dbs$ ]

$mrobjs \text{ dom } tdb.dbojs - robjs$

$mrobjs \neq \emptyset \Rightarrow$

*CurrentPVersion*

*ExtractObj*[ $pdbv/dbv, mrobjs/lobjs, rdb/dbs$ ]

$tdb' = tdb \oplus rdb$

*NextDBVersion*[ $tdbv/dbv, tdbv'/dbv'$ ]

*AddDB*[ $tdbv'/dbv, tdb'/dbs$ ]

*WriteOperation*

$t : TRANSID$   
 $wdb : DBSTATE$

$\exists \Delta tdbv : DBVERSION, \Delta tdb : DBSTATE \bullet$

*CurrentTVersion*

*ExtractAll*[ $tdbv/dbv, tdb/dbs$ ]

$tdb' = tdb \oplus wdb$

*NextDBVersion*[ $tdbv/dbv, tdbv'/dbv'$ ]

*AddDB*[ $tdbv'/dbv, tdb'/dbs$ ]

$Written' = Written \cup \text{dom } wdb.dbojs$

<p><i>Commit</i></p> <hr/> <p><math>t : TRANSID</math></p> <hr/> <p><math>\exists tdbv, \Delta pdbv : DBVERSION, wdbs, \Delta pdbs : DBSTATE \bullet</math>  <i>CurrentTVersion</i>  <i>ExtractObjs</i>[<math>tdbv/dbv, Written/lobjs, wdbs/dbs</math>]  <i>CurrentPVersion</i>  <i>ExtractAll</i>[<math>pdbv/dbv, pdbs/dbs</math>]  <math>pdb's' = pdbs \oplus wdbs</math>  <i>NextDBVersion</i>[<math>pdbv/dbv, pdbv'/dbv'</math>]  <i>AddDB</i>[<math>pdbv'/dbv, pdbs'/dbs</math>]</p>
<p><i>Abandon</i></p> <hr/> <p><math>t : TRANSID</math></p> <hr/> <p>Nothing much happens!</p>

## 2.4 Serial Execution

In Serial Execution, the transactions are executed one at a time. Serial Execution is therefore just the same as Chaos but without the dangerous concurrency.

<p><i>SerialExecution</i></p> <hr/> <p><i>CachedChaos</i></p> <hr/> <p><math>\#Ongoing \leq 1</math></p>
--

If we are interested in a particular sequence of transactions being executed, then we need to keep a track of the order that they start.

<p><i>ParticularSerialExecution</i></p> <hr/> <p><i>SerialExecution</i></p>
---

*serial* : seq *TRANSID*

*INIT*

*serial* =  $\langle \rangle$

*StartTransaction*

*serial'* = *serial*  $\frown$   $\langle t \rangle$

## 2.5 Serializability

Now that we understand serial execution, what is serializability? Well, we need a notion of a 'final state of the database'.

*Serializability*[*CC*]

*cc* : *historyCC*

$\exists se : \text{historySerialExecution} \bullet$

$cc(1).db = se(1).db$

The initial databases are the same

$\exists cldb : CC, sedb : DBSYSTEM \bullet$

$cldb = cc(\#cc).db$

$sedb = se(\#se).db$

$cldb.Ongoing = sedb.Ongoing = \emptyset$

Both databases are in a 'final state'

$cldb.Committed = sedb.Committed$

... and involve the same committed transactions

$\exists ccv, sev : DBVERSION; ccs, ses : DBSTATE, pdb : PHYSDBID \bullet$

$pdb = cc.PDB = se.PDB$

$cldb.CurrentPVersion[pdb/p, ccv/pdbv]$

$sedb.CurrentPVersion[pdb/p, sev/pdbv]$

$cldb.ExtractAll[ccv/dbv, ccs/dbs]$

$sedb.ExtractAll[sev/dbv, ses/dbs]$

$ccs = ses$

... and produce the same final database state

## 2.6 Reader-Writer Rules

We will deal with the usual reader/writer rules, writers exclude both readers and writers, and multiple readers are OK. In addition, we allow a single process to hold the exclusive right to read-and-write (which excludes all other processes).

*TwoPhaseLocking*

*CachedChaos*

*readers, writers* : *LOBJID*  $\leftrightarrow$  *TRANSID*

$\forall l$  : *LOBJID* •

$readers(l) \cup writers(l) \subseteq Ongoing$

$\exists t$  : *TRANSID* •  $readers(l) = writers(l) = \{t\}$

$\forall readers(l) = \emptyset \wedge \#writers(l) \leq 1$

$\forall writers(l) = \emptyset$

*Initially*

$\forall l$  : *LOBJID* •

$readers(l) = writers(l) = \emptyset$

This is actually derivable from being a subset of Ongoing

*ReadOperation*

$t$  : *TRANSID*

$robjs$  :  $\mathbb{P}$  *LOBJID*

$\forall l$  : *LOBJID* •

$l \in robjs \Rightarrow$

$readers'(l) = readers(l) \cup \{t\}$

$l \notin robjs \Rightarrow$

$readers'(l) = readers(l)$

*WriteOperation*

$t$  : *TRANSID*

$wdbs$  : *DBSTATE*

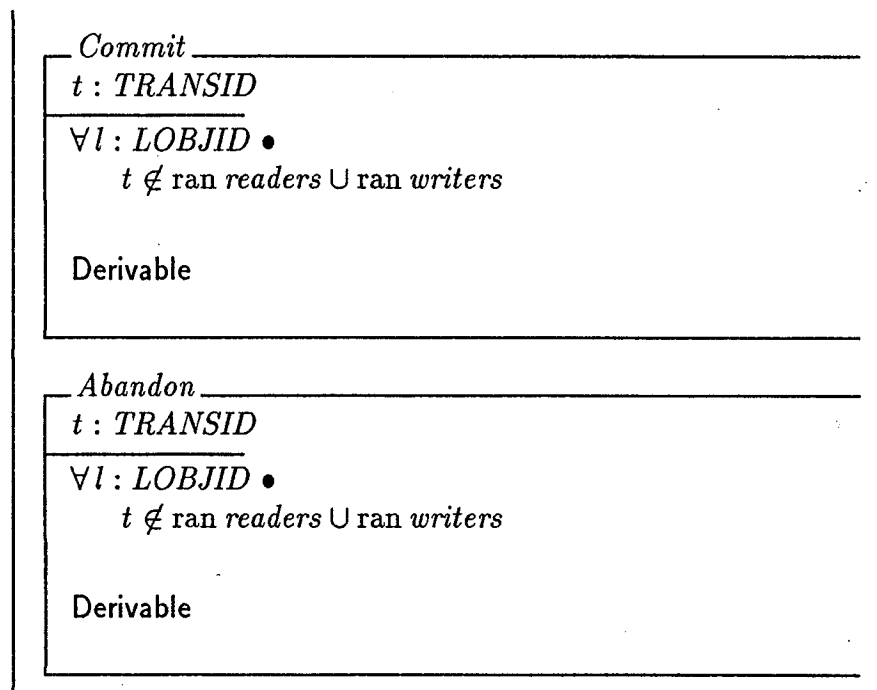
$\forall l$  : *LOBJID* •

$l \in \text{dom } wdbs \Rightarrow$

$writers'(l) = writers(l) \cup \{t\}$

$l \notin \text{dom } wdbs \Rightarrow$

$writers'(l) = writers(l)$



### 3 Transactions

Having thought long and hard about it, a transaction is best viewed as the sequence of interactions between the 'User' which makes the specific requests for operations and the 'DBMS' which actually performs those operations. I use 'user' and 'dbms' to characterize the sides of the interface rather than suggest that they represent the ultimate end-user or the entire DBMS.

A transaction is a sequence of intermingled requests from the user and replies from the dbms (which I will call events).

$$ENTITY \cong USER|DBMS$$

The possible operations that the user may request are START transaction, READ, WRITE, COMMIT and ABANDON (where the user requests that the transaction is aborted). The dbms will eventually reply to each of these, indicating if the operation succeeded (i.e. took place, or failed). The dbms may abort a transaction whenever it pleases.

$OPTYPE \cong START|READ|WRITE|COMMIT|ABANDON|ABORT$

Naturally READs and WRITEs will need to include details of which object and what values in requests and replies as appropriate.

Replies will need to specify the request which initiated them. This is necessary as the dbms may be able to be instructed to do many things concurrently. An event identifier, EVID, is used to uniquely identify an event w.r.t. a particular transaction.

[EVID]

Operations at the database must inform the dbms if they succeeded or not. E.g. suppose a READ request cannot take place because it references a logical object which does not exist within the database. Another example is COMMIT failing because some consistency constraint is violated. Apart from having to pass this information to the user, the dbms itself must be aware of this as the non-reading/non-committal of data is significant for concurrency control purposes. However such failures do not automatically cause the transaction to abort.

Hence we regard our response as either ABORT or something that the user can make sense of.

[USERRESPONSE]

$RESPONSE \cong ABORT|UserResponse \ll USERRESPONSE \gg$

So lets define our events

*Event*

<p><i>entity</i> : ENTITY <i>optype</i> : OPTYPE <i>replying</i> : EVID <i>response</i> : RESPONSE <i>objid</i> : OBJID <i>data</i> : DATA</p>
--

We have a whole bunch of constraints about what sort of operations use what fields and which ones have them NULL. It's all rather messy and hopefully some neater way will be found (polymorphism perhaps?). Anyhow I'll try and summarize the request-response pairs.

- user start: , dbms: replying response
- user read: objid, dbms: replying response data
- user write: objid data, dbms: replying response
- user commit: , dbms: replying response
- user abort : , dbms : replying response
- dbms abort:

Now we've defined what we mean by an event, we can move on to what we mean by a transaction. This is a sequence of events, although it is possible to group some events as they appear to have simultaneously, e.g. a user may be able to request a read of X and a read of Y together. I'm not considering a partial order here, as the interface of user and dbms should be visible as a sequence based on the notion of a transaction in execution at a particular site. If later we consider the remote execution of subtransactions, then we'll need partial orders.

In the general case, a transaction should look like 'user start', 'dbms : ok', a bunch of read/write requests and responses, followed by 'user commit', 'dbms: ok'.

The intention is generally that request event be followed (not necessarily as the next event) by exactly one reply event.

Not surprisingly, there should be at most one ABORT and it should be the last event of a transaction. (Do user-initiated ABORTs get replied to?) Likewise 'dbms: ok (commit)' should be the last event in the successful case. However there are many other little questions ...

Can read/write/commit/abort be requested by the user before the dbms ok's the start?

Can the user request commit/abort while read/write requests remain unreplied?

Can the dbms OK/abort the commit when read/write requests remain unreplied?

Can the dbms abort the transaction before 'user start'?

Does a response mean anything in the the context of a user-initiated abandon?

*Transaction*

*trans* : seq P EVID

$$\Sigma_{i:1..#trans} \#trans(i) = \# \cup_{i:1..#trans} trans(i)$$

All the EVIDs are unique

and all the lists of constraints above are maintained

and perhaps some progress rules ... bring on the temporal logic

*Initially*

*trans* =  $\langle \rangle$

Or perhaps we should start with 'user start' in *trans*

*UserEvents*

$\Delta(trans)$

*userevents* : P 1Event

check that *userevents* really are 'user' events and that event constraints are maintained

$$trans' = trans \hat{\ } userevents$$

*DBMSEvents*

Similar to *UserEvents*, with obvious constraints like replies must pertain to requests etc.

### 3.1 Many Transactions

Clearly we are interested in not just one transaction, but many. We will uniquely identify these transactions with TRANSID. There may exist some ordering constraints between transactions, i.e. that at some higher level, transaction2 is not started until transaction1 has completed. So some sort of partial order on the transactions is needed.

(Aside. The formal Z 'partial order' type includes reflexive properties, my definition (based on the mighty Street & Wallis) does not!)

The actual event sequence associated with all transactions consists of all of the events of the individual transaction event sequences (each occurring exactly once). The overall event sequence is in fact a partial order. Naturally the ordering within individual event sequences must still be observed, as must be any ordering among the transactions as a whole.

If we know that the dbms is a single 'funnel' to the database itself, then all of the actual database operations are in strict sequence BUT we only consider requests and replies in our interface, not the actual database operations.

