1991

Transaction Models for Advanced Database Applications

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
91-022
TRANSACTION MODELS FOR ADVANCED
DATABASE APPLICATIONS

Ahmed K. Elmagarmid

CSD-TR-91-022
March 1991
(Revised May 1991)
Transaction Models for Advanced Database Applications

Ahmed K. Elmagarmid

Associate Professor and Executive Director
The Indiana Center for Database Systems
Department of Computer Sciences
Purdue University

The Seventh International Conference on Data Engineering
April 9, 1991

© Ahmed K. Elmagarmid, 1991
Acknowledgements

These tutorial notes are based on the March issue of the IEEE Data Engineering with the same title and edited by the same author. It is also based on an upcoming book with the same title due to come out later this year and published by Morgan Kauffmann.

Thanks are due to Dr. Won Kim of UniSQL for enticing me to organize the special issue, which leads to the book, which in turn lead to this tutorial. Thanks are due to the authors of all those great papers covered herein. Thanks are to my good friend Dr. Amit Sheth, Bellcore, for inviting me to present this tutorial. I would also like to thank all members of the InterBase project. Ai-dong Zhang, Jindong Chen, Jiansan Chen, Weiming Du, Jin Jing, Xu Lu, Yungho Leu, James Mullen and Anne Burns.

Finally, very special thanks go to Yungho Leu, Purdue, for his help in the preparation of these notes.
Introduction

Transactions:

A transaction is a collection of actions that make consistent transformations of system states.

Formally, it is a partial order over the operations that are part of the transactions.
Examples—

Consider an airline reservation example:

Transaction Reservation

begin

input (flight, date, customer_name);

   temp <- Read(flight(date).sold_seats);
   Write(flight(date).sold_seats, temp + 1);
   Write(flight(date).cust_name, customer_name);

Output ("reservation completed")

end. {transaction}
What if there are no free seats?

Transaction Reservation

begin

\textbf{input} (flight, date, customer\_name);

\texttt{temp} <- \texttt{Read(flight(date).sold\_seats)};

\texttt{if temp = flight(date).maximum then begin}

\hspace{1em} \texttt{output(\textquote{\texttt{no free seats}});}

\hspace{1em} \texttt{Abort}

\texttt{end}

\texttt{else begin}

\hspace{1em} \texttt{Write(flight(date).sold\_seats, temp + 1)};

\hspace{1em} \texttt{Write(flight(date).cust\_name, customer\_name)};

\hspace{1em} \texttt{Commit};

\hspace{1em} \texttt{output(\textquote{reservation completed})}

\texttt{end}

end. \quad \{ \text{transaction} \}
Properties of Transactions

Atomicity -
• all or nothing

Consistency -
• a correct transformation

Isolation -
• effects hidden until successful completion

Durability -
• effects survive failure

[Haerder&Reuter ‘83] [ISO TP]
Atomicity

• Either all or none of the transaction’s operations are performed

• If transaction fails, its partial result must be undone

• The activity of preserving the transaction’s atomicity in the presence of transaction aborts due to input errors, or deadlocks is called transaction recovery

• The activity of ensuring atomicity in the presence of system crashes is called crash recovery
Consistency

Internal consistency

• A transaction which executes alone against a consistent database leaves it in a consistent state.

• Transactions do not violate database integrity constraints.
Isolation

Two points:

• Serializability

If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.

• Incomplete results

An incomplete transaction cannot reveal its results to other transactions before its commitment.

Necessary to avoid cascading aborts.
Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures

- Database recovery
Transaction Management Problems

- Semantic Data Control (Integrity Enforcement)
  Consistency

- Concurrency Control
  Consistency, Isolation

- Reliability
  Commit & Recovery -> Atomicity & Durability
Fundamentals

- Simple transactions
- Compensating transactions
- Nested transactions
**Simple Transactions**

Consists of a sequence of *primitive* operations embraced between a `begin` and `end` markers.

```plaintext
begin(Ti)
    read(x)
    read(y)
    write(z := x + y)
end(Ti)
```
Compensating Transactions

- Used to reverse, or compensate for, the effects of an already committed transaction

- It may not always be possible to issue a compensating transaction

  e.g. real action in [Gray '81]
  - a transaction which fires a missile
  - a transaction which writes a check that has been cashed by somebody
**Nested Transactions**

The operations of a transaction can be themselves transactions.

```plaintext
begin(T1)
    read(x)
    write(y)
    begin(T11)
        read(z)
        write(y)
    end(T11)
    write(z := x + y)
end(T1)
```

*Nested transaction*
Transaction hierarchy

- TL-transaction
- parent of T12
- child of T1
- ancestors of T11
- descendants of T11
- T11
- T111
- T1111
- T1112
- T12
• TL-transaction must preserve the *ACIDity* properties

• Subtransactions must preserve *Atomicity* and *Isolation* properties

• Consistency is not required

```
  Transfer
    /   \        
   /     \       
Debit    Credit

- *Debit, Credit* need not preserve consistency
- *Transfer* must preserve consistency
```

• Commit of a subtransaction is conditional subject to the fate of its ancestors

- aborting any of its ancestors will undo its effects
- all updates become permanent only when the enclosing TL-transaction commits
• A subtransaction acting like a *fire wall* for failure

- When a subtransaction fails, it parent can still complete its work by
  1. Restarting the subtransaction
  2. Trying other alternatives
  3. Ignoring the failed subtransaction

• Isolation is maintained by employing *locks* 
  *inheritance*

- a subtransaction can acquire an X-mode lock
  if all the subtransactions hold any lock on the same object are its ancestors

- a subtransaction can acquire an S-mode lock if all the subtransactions holding X-mode lock on the same object are its ancestors

- when a child commits, all its locks are inherited by its parent (setting the most restrictive lock of child and parent)

• Parent and child are not isolated while siblings are isolated (safe intra-transaction parallelism)
Advanced Transactions

- Limitations of the traditional transaction model
- Features of the advanced transactions
- Various advanced transactions
Limitations of the Traditional Transactions

• Transactions should not be long-lived
• Transactions cannot be nested
• Transactions are not allowed to fail partially
• Transactions do not support cooperative activities
• Transactions do not support local autonomy
• Transactions do not support user control

[Gray ‘81] [Barghouti & Kaiser ‘91] [Leu ‘89]
Features of Advanced Transactions

Each advanced transaction model extends the traditional transaction concept along the following dimensions:

- Supporting long-lived transactions
- Supporting open-ended activities
- Supporting cooperative activities
- Supporting local autonomy
- Supporting user controlled transaction
- Application-specific transaction manager
- Framework for analyzing transaction models
In the following, we will survey SOME of important advanced transaction models
SAGAS

[Garcia-Molina & Salem '87]

Supporting long-lived transactions

A saga is a collection of relatively independent subtransactions $T_1, T_2, \ldots, T_n$. Associated with subtransactions $T_1, T_2, \ldots, T_{n-1}$ are compensating transactions $C_1, C_2, \ldots, C_{n-1}$.

The system guarantees that either $T_1, T_2, \ldots, T_n$ or $T_1, T_2, \ldots, T_j, C_j, \ldots, C_2, C_1 (j < n)$ is executed.
SAGAS

- Subtransactions of a saga can be interleaved in any way with other (sub)transactions

- When a subtransaction completes, it can commit without waiting for other subtransactions

- When failure occurs, a saga may try to proceed by executing the missing subtransactions (forward recovery); if not possible, it rollbacks the committed subtransactions by issuing compensating transactions
  
  ♦ Subtransactions may not see the same consistent state
  
  $\Rightarrow$ Consistency is compromised

  ♦ Subtransaction can commit when complete
  
  NO commit protocol is needed
  
  $\Rightarrow$ Isolation is reduced to subtransaction level

  ♦ Failure atomicity is required
  
  $\Rightarrow$ Atomicity and Durability are still required
Split Transactions

[Pu, Kaiser & Hutchinson '88]

Supporting open-ended activities

- Open-ended activities like CAD/CAM project, VLSI design and Software development are characterized by:
  
  - uncertain duration (from hours to months)
  
  - uncertain developments (actions unforeseeable at the beginning), and
  
  - interaction with other concurrent activities

- The above often results in long-lived transactions
Split Transactions

• Split the ongoing transaction into two serializable transactions, divide the resources among the resulting transactions

• Major purpose of splitting is to commit one of the resulting transaction to reveal useful results from the original transaction

1. AWriteSet ∩ BWriteSet
   = BWriteLast

2. AReadSet ∩ BWriteSet = ∅

3. BReadSet ∩ AWriteSet
   = ShareSet

• Condition 1, objects in BWriteLast are updated last by B. This prevent A from overwriting B’s output

• Condition 2 guarantees that A will not read from B, Condition 3 says that B is allowed to read from A

• Condition 1, 2, 3 ensure that A is serialized before B

• If both BWriteLast and ShareSet are empty, then A and B can be committed independently; Otherwise A has to commits first and B’s commit depends on A’s Commit
Split Transactions

Advantages:

• *Adaptive recovery*
  \[\rightarrow\] A’s commit will not be affected by B’s abort

• *Reducing Isolation*
  \[\rightarrow\] releasing the committed resources by committing A
Cooperative Transaction Hierarchies

[Nodine and Zdonik '90]

Relaxing Serializability for Cooperative Activities

• **Serializability** is not suitable for cooperative activities (e.g. CAD tools)

• **Application-defined** correctness is desirable
A transaction hierarchy

An operation machine
The label of the operation machine is—

\[ \sigma = \langle M, O, o, P \rangle \]

\( M \in \{\text{any, } m_i, \overline{m_i}\} \) is the TID of some members
where \( \text{any} \) is any member, \( m_i \) identifies some
member \( i \), and \( \overline{m_i} \) is any member except \( m_i \)

\( O \in \{r, w\} \) is an operation, where \( r \) is read and
\( w \) is write

\( o \) is an object identifier

\( P \in \{a, r, q\} \) is a return value, where \( a \) is accept
\( r \) is refuse and \( q \) is queue
Cooperative Transaction Hierarchies

- A transaction group is a task which involves many cooperative transactions.
- A transaction group uses patterns (specified by augmented finite state automata) to specify the allowable sequence of accesses to the data objects.
- The patterns are called protocols.
- An internal protocol specifies the allowable access patterns of its members.
- An external protocol specifies how to interact with its siblings in the transaction hierarchy.
- The cooperative transactions (i.e. designers) can talk or communicate through the database objects; therefore, are able to abide by the protocols.
Groupware Systems

[Ellis '91]

Supporting cooperative activities

• Groupwares are computer based systems which support two or more users to work on a common task

• Groupwares allow users to know and keep track of the activities performed by others; isolation is NOT acceptable

• Concurrency control algorithms do not rely on locking or rollback and can produce non-serializable execution

• Concurrency control algorithms rely on application specific semantic knowledge; operation transformation has been adopted as a method for concurrency control
**Operation transformation**

initial string “xyz”

\[ o_a = \text{insert}[a; 2] \text{ and } o_b = \text{insert}[b; 2] \]

two sites a, b want to insert a character in the same position

site a— \( o_a(“xyz”) = “xayz” \) followed by \( o_b(“xayz”) \)
\[ = “xbayz” \]

site b— \( o_b(“xyz”) = “xbyz” \) followed by \( o_a(“xbyz”) \)
\[ = “xabyz” \]
The results are different.

Solution —

• add one when the concurrent event at the same position is detected.
• assign priority to each site (and its operations), when an operation’s priority is lower that the receiving site, the operation got transformed

\( o_a \) is transformed into \( \text{insert}[a; 3] \) when arrives site b
Interactive Transaction Model

[Lee, Mansfield and Sheth '91]

Supporting cooperative tasks in multimedia telecommunication environment

Cooperative Tasks:

States of Shared Objects
Interactive Transaction Model

• An ITX is a tuple \((ID, \{TX_n\}, ACC)\), where \(ID\) is the identifier, \(\{TX_n\}\) is the set of \(n\) transactions, and \(ACC\) is the acceptable correctness criteria for the ITX.

• The ACC is used to —
  - Specify the acceptable states
  - Specify the execution dependency
  - Specify the commit/abort dependency
  - Specify synchronization requirement for accessing the shared data

   e.g.

   Acceptable states \(\{t1, t2, t3\} = \{s, f, s\}\) or \(\{s, s, s\}\)

   Synchronization can use —

   \textit{Finite state machine, invariants}
Multi-Level Transactions

[Weikum et al]

Relaxing serializability for high concurrency

• Multi-Level Transactions are a variant of nested transactions with a fixed level of nesting

• Nodes in a transaction tree correspond to operations at particular levels of abstraction in a layered system

• The edges in a transaction tree represent the implementation of an operation

• Level-specific conflict relations is exploited to enhance concurrency

• Different levels can apply different concurrency control schemes
Multi-Level Transactions

Example:

- The schedule at level L0 is not conflict serializable
- The schedule at level L1 is serializable, because the two Deposit operations commute
- The conflict at L0 is a pseudo-conflict
Deferred and Decoupled Transactions

[Dayal Hsu and Ladin ‘91]

Generalized Nested Transactions for Active Database

• The execution of a (deferred) subtransaction can be deferred to the end of the transaction

• A subtransaction can start a new Top-Level transaction, called the decoupled transaction, from inside the transaction
Defered and Decoupled Transactions

- A decoupled transaction $T'$ of $T$ is said to be causally dependent on $T$ if $T'$ is serialized after $T$ and $T'$'s commit conditionally depends on $T$. 


Deferred and Decoupled Transactions

Applicatoin in active databases — databases which contains both data and rules—>(event, action) pairs.

• The execution of a transaction can trigger the event of a rule which causes the action part of the rule to be executed

The concept of deferred, decoupled are used to specify when to execute the action:

immediate — immediately after the event occurs

defered — deferred to the end of the transaction

causally dependent — the action is executed as a separate transaction which is causally dependent on the triggering transaction

causally independent — the action is executed as a completely independent transaction
Polytransactions

[Rusinkiewicz and Sheth 91]

For generating related updates that maintains the consistency of Interdependent Data

- Dependency and mutual consistency of an interdependent data is stored as a triple \(<D, C, A>\) in the Interdatabase Dependency Schema (IDS)

- In a \(<D, C, A>\), D specifies what is the related updates for a specific update; C specifies when the related updates should be performed and A is the related updates

- A polytransaction \(T^+\) for a transaction T is created in the following way —

  - Take T as the root of \(T^+\)
  - Check IDS to generate the related updates for T and take them as the children of T
  - For each child Tc of T, calculates the related updates and treat them as the children of Tc
  - Repeat the procedure until no related updates can be generated
• The notion of **coupled and decoupled** transactions can be used to specify the relationship between a parent and a child

• To maintain the interdependent data consistency, the whole (sub)transactions in \( T^+ \) must be executed; however, a decoupled subtransaction can be executed later (after \( T \) commits)
ConTract Model
[Reuter and Wachter '91]

Explicit flow control for non-standard application

- Aiming at non-standard application like office automation, CAD, manufacturing control etc.

- Supporting explicit flow control for long-lived activities

- Important properties:
  - Using invariants for concurrency control
  - Specifying conflict resolution in flow control
  - Computation is forward-recoverable by resuming the execution of a computation (from where it was interrupted) when recovers
  - Externalizing results before the transaction commit; compensating transaction is used to reverse the undesired effects
S-Transactions
[Veijalainen, Eliassen and Tirri '88]
& [Holtkamp '90]

Supporting autonomous banking environments

• S stands for semantic

• Supporting local autonomy

• Isolation of global transaction is not supported; therefore, recovery is based on compensating

• Allowing alternative transactions; the exact execution trace of an s-transaction is non-deterministic

• No explicit flow control is supported (control is decentralized)

• Constructing s-transactions using functional programming paradigm
Flexible Transactions
[Gail Kaiser '90]]

Supporting cooperative work in SDEs

• User-controlled transaction—

A user-controlled transaction starts when a user gives a begin-transaction command to the system. The user may then carry out any number of activities (read and write objects). It is open-ended, i.e., the user does not predeclare all the objects to manipulated at the start of the transaction. The transaction ends when the user gives either a commit-transaction command or an abort-transaction command.
Approaches for supporting cooperation of user-controlled transaction—

— using *commit-serializability*

for *Activity Interaction*

(actually to deal with long-lived transactions)

— using *participation domains*

for *Programmer interaction,*
Commit-serializability —

The set of committed transactions are serializable.

Start out: \( T_1, T_2 \) are executed concurrently

\( T_1 \) splits into \( A, B \)

\( A \) commits, \( T_2 \) reads from \( A \)

\( T_2 \) commits, \( B \) reads from \( T_2 \)

\( B \) commits

End: \( A, B \) and \( T_2 \) are serializable, But not \( T_1 \) and \( T_2 \)

(actually, \( T_1 \) does not exist any more)

- The set of committed transactions is not the same as the orginal set of transactions.

- Commit serializability is implemented by \( split \) & \( join \) operations.
**Participation Domains** —

- A (participation) *domain* is a set of transactions (of some users) that work towards a common goal.

- A transaction is placed in one domain in order to non-serializable share objects with other transactions in the same domain.

- Transactions in the same domain are not serializable (the cooperating users apply semantic to resolve inconsistency)

- A transaction in a domain has to be serialized with respect to all transactions not in the domain.
**Tool Kit Approach**

[Unland and Schlageter '91]

*An environment which supporting application-specific transaction manager*

- Different environments may have incompatible requirements for transaction processing

**Example:**

Banking environments emphasizing on *Isolation* while *Isolation* is not acceptable for cooperative environments

- Allows strict isolation of (sub)transactions and non-serializable cooperative (sub)transactions in one transaction hierarchy
Approach

1. By using lock protocols which allow transactions to exchange data or to release data at an earlier point of time

2. By offering lock modes which facilitate a higher degree of concurrent work on data (exploiting application-specific semantics)
Earlier release of data—

For nested transactions, locks are only released after the (sub)transactions are (conditionally) committed. Earlier release may cause problems.

- $T_{12}$ conditionally commits and releases its locks
- $T_{12}$'s locks are inherited by $T_5$
- If $T_5$ releases locks earlier then it may not be able to inherits $T_{12}$'s lock when $T_{12}$ commits
- locks inheritance is needed for proper synchronization
To be able to earlier release of locks for a sub-transaction, a child uses stepwise transfer to check out its ancestors' data, and check in data to its' ancestors.

e.g., T12 needs O from the parent of T3

O is checked out by T3 from the parent of T3
T5 checks out O from T3
T12 checks out O from T5

check-out operation must obey the concurrency control at each level

- Two-stage control-sphere—

Parent transaction is only responsible for the coordination and execution of the work (therefore subtransactions) on its level. Different spheres can use different transaction types (cc, and recovery)
ACTA

[Chrysanthis and Ramamritham '90]

A framework for analyzing transaction models

• ACTA means actions in Latin

• Major goals —
  - to capture the semantics of complex transactions
  - to reason about the concurrency and recovery properties of complex transactions

• Approach —
  - Modeling the effects of transactions on each other
  - Modeling the effects of transactions on the objects that they access

Effect on transactions —

Commit-dependency:

A ↼ B  A cannot commit until transaction B either commits or aborts

Abort-dependency:

A → B  If transaction B aborts A should also abort
Effects of transactions on objects —

By applying the notion of delegation

View set — objects potentially accessible to T
Access set — objects already accessed by T

In general, when T aborts, all objects in access set are restored; when T commits, objects in access set is made persistent (i.e., changes are effected)

delegator
delegation

remove objects from the access set of the delegator put them to the access set of the delegatee
• By delegation, $T_1$ give up some of its objects to $T_2$

• $T_2$ can then access the partial results of $T_1$ and is then responsible for finalizing the effects of the delegated objects for $T_1$

• Delegation is useful for modeling the cooperative transactions

Summary

Effects

on transactions

commit dependency abort dependency

on objects

view set & access set specification delegation
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Long-lived</th>
<th>Open-ended</th>
<th>Cooperative</th>
<th>Autonomy</th>
<th>User-control</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAGAS</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Split</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cooperative</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Groupware</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interactive</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Multi-level</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferred</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConTract</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Flex</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flexible</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool Kit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Note:  
1. Multi-level trans. aiming at high concurrency, level specific cc, rv  
2. Tool Kit for assembling transaction models  
3. ACTA for analyzing transaction models  
4. Polytransactions for handling interdependent data
InterBase Approach

- Flexible Transactions
- Quasi-Serializability
- Interdependent Data
Flexible Transactions

- Motivations
- Properties
- Model
- Implementations
Motivations

• Global transactions in MDBSs tend to be long-lived
  - LDBSs delay the subtransactions (execution autonomy)
  - LDBSs located in different time zone such that subtransactions cannot be executed at the same time
  - Some involving local systems are slow (design autonomy)

• Global transaction is subjected to failure
  - Communication failure (communication autonomy)
  - Deny subtransaction execution (execution autonomy)
  - unreliable local systems (design autonomy or heterogeneity)
Properties

• Supporting multiple equivalent goals
  (relaxing atomicity)
• Supporting controlled isolation
  (relaxing isolation)
• Supporting external dependencies
  (Flexible execution control)
Multiple equivalent goals

Function replication

Travel Agent:

Buy-a-ticket from:  
- airline A
- airline B

Rent-a-car from: car_rental X
- hotel 1
- hotel 2
- hotel 3

Reserve-a-room from:
- hotel 2
- hotel 3

Equivalent goals:

\[(A, X, 1), (A, X, 2), (A, X, 3)\]
\[(B, X, 1), (B, X, 2), (B, X, 3)\]
We may want to have some preference for combinations.

For example, we may have—

Equivalent goals:

\[(A, X, 1), (A, X, 2), (B, X, 3)\]
Controlled Isolation

Global Trans. -> 2PC
  Vote and prepared (if complete)

Isolated sphere
  non-compensatable subtransactions

non-isolated sphere
  compensatable subtransactions

Vote and commit (if complete)
Commit Protocol for Controlled Isolation

When completes—

- a non-compensatable subtransaction votes yes and enters a prepared state
- a compensatable subtransaction votes yes and commits

When global decides to commit—
non-compensatable subtransactions commit

When global decides to abort—
non-compensatable subtransactions rollback committed subtransactions are compensated on-going subtransaction are aborted
Supporting External Dependencies

- specify the execution condition of (sub)transactions based on some external events.

For Example:

— specify when a transaction should be scheduled
  • Subtransaction should be executed in between office hours
  • Global transaction should be finished within a day

— specify the acceptable condition for a subtransaction using some cost function.

  • The cost of order-a-ticket should be less than $200
**Formal model**

A global transaction is a 5-tuple

$$(ST, PP, EP, A, V)$$

- $ST$ set of subtransactions
- $PP$ precedence predicate
- $EP$ external predicate
- $A$ acceptable set
- $V$ value function
Execution state $x_i$ of subtransaction $t_i$

\[ x_i = \begin{cases} 
N & t_i \text{ is not yet issued} \\
E & t_i \text{ executing} \\
S & t_i \text{ successfully finished} \\
F & t_i \text{ failed} 
\end{cases} \]
Execution Dependencies

Success dependency \( pp_j := (x_i = S) \)

\( t_j \) has to wait until \( t_i \) succeeds

Failure dependency \( pp_j := (x_i = F) \)

\( pp_j := (x_i = F) \)

\( t_j \) has to wait until \( t_i \) fails
Example

A flexible transaction for travel agent

t_1  buy ticket from airline-A

(t_1 fails)

t_2  buy ticket from airline-B if  t_1 fails

(t_3 fails)

(t_3 fails)

t_3  rent a car from rental

(t_4 and t_5 fail)

t_4  reserve a room from hotel-1 if  t_5 fails

(t_6 fails)

(t_6 fails)

t_5  reserve a room from hotel-2

(t_6 fails)

t_6  reserve a room from hotel-3 if  t_4 and t_5 fail

Ticket can not be refunded and can be purchased only from 8AM to 5PM

Transaction should finish within 12 hours but no later than 24 hours
\[ ST = \{ t_1(\text{NC}), t_2(\text{NC}), t_3(\text{C}), t_4(\text{C}), t_5(\text{C}), t_6(\text{C}) \} \]

\[
\begin{align*}
\text{PP}: \\
pp_1 &:= \text{true} \\
pp_2 &:= (x_1 = F) \\
pp_3 &:= (x_1 = S) \lor (x_2 = S) \\
pp_4 &:= (x_3 = S) \land (x_5 = F') \\
pp_5 &:= (x_3 = S) \\
pp_6 &:= (x_3 = S) \land (x_4 = F') \land (x_5 = F')
\end{align*}
\]

\[
\begin{align*}
\text{TP}: \\
tp_1 &:= \text{between}(08:*.*.*, 17:*.*.*) \\
tp_2 &:= \text{between}(08:*.*.*, 17:*.*.*) \\
tp_3 &:= * \\
tp_4 &:= * \\
tp_5 &:= * \\
tp_6 &:= *
\end{align*}
\]

\[
A = \{ (S, N, S, N, S, N), (S, N, S, S, F, N), \\
(S, N, S, F, F, S), (F, S, S, N, S, N), \\
(F, S, S, F, F, S), (F, S, S, F, F, S) \}
\]

\[
v(t) = \begin{cases} 
1 & \text{if } t \leq 12 \text{ hours} \\
0.5 & \text{if } 12 < t \leq 24 \text{ hours} \\
0 & \text{otherwise}
\end{cases}
\]
Implementations

We are exploring two approaches for controlling the execution of flexible transactions

**Predicate Transition Nets Approach**

• Map a flexible transaction into a PTN, use the PTN as a data structure to control the execution of the flexible transaction

**Parallel Prolog Approach**

• Extending Prolog to support concurrent constructs

• Using the extended Prolog to compose the flexible transactions

Both approaches are being implemented in InterBase.
Quasi-Serializability

- Concurrency control problem in MDBSs
- Serializability vs. Quasi-Serializability
- Maintaining Quasi-Serializability
- Implementation
Concurrence Control Problem

Transaction processing model [Gligor et al '86]

*Global concurrency control is required to maintain the consistency of the MDBS*
Global Serializability

A global execution is serializable if it is equivalent to a serial one in which transactions are executed sequentially.

Quasi-Serializability

A global execution is quasi serializable if

1. Each local execution is serializable; and

2. It is equivalent to a quasi serial execution in which global transactions are executed sequentially.
Serializability vs quasi-serializability

Serializability Approach

No distinction between local and global transactions

- Transaction level consistency for both local and global transactions.

- Partial precedence relation between local transactions at different sites.

Difficult to ensure if LDBSs are autonomous

- Serialization order is incompatible with execution order.

- Information about and control over both local and global transactions are required.
Serializability vs quasi-serializability

Quasi Serializability Approach

Distinction between local and global transactions

- Separation of interactions among global transactions from those among local xacts.
- Interactions among global transactions are controlled by scheduling.
- Local transactions at different sites do not affect each other in many applications.
- Interactions among local transactions are indirect and can be controlled explicitly.
Maintaining Quasi-Serializability

Concurrency Control Based on QSR

- Controlling both submissions of and interactions between transactions as well as other aspects of execution.
- Scheduling global transactions
- Preventing undesirable remote interactions


Maintaining Quasi-Serializability

Scheduler for Quasi Serializable Executions

- Global transactions are grouped such that no two transactions in a group interleave at more than one site
- Transaction in a group interact with each other in a partial order
- Transactions in a group are executed concurrently
- Transactions in different groups are executed separately
Scheduling Global Transactions -- Example

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>T4</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>2</td>
</tr>
</tbody>
</table>
Implementation
Interdependent Data

[Sheth, Leu and Elmagarmid ‘91]
[Sheth and Rusinkiewicz ‘90]
[Rusinkiewicz and Sheth ‘91]

• Motivation
• Interdependent data vs replicated data
• Our solution
Motivation

- Big companies use many databases stored on multiple heterogeneous and autonomous systems.
- Many data of the companies are duplicated (or inter-related) in more than one databases which are control led by different systems.
- The stewardship of the interrelated data is not assigned; therefore, inconsistency of the interrelated data are not handled “automatically”.
- Currently, inconsistency is resolved by human, which is costly and inefficient.

There is a need that mutual consistency of the interrelated data be recognized and maintained by the system.
**Definition**

Interdependent Data (ID data) is a set of interrelated data (members) which are characterized by the dependency and mutual consistency requirements.
Dependency

Structure dependency

replicated data
vertical and horizontal partitions
value or existential constraints

Control dependency

derived data
primary-secondary copies
independently updatable


**Mutual Consistency**

Specify when the members of an interdependent should be converged

*Time aspect (C_t(D)) — examples*

@5pm Friday

every 2 hours

*Data state (C_d(D)) — examples*

10% change on the data value

*Operation (C_o(D)) — examples*

no more than three updates

before executing a specific transaction
Combination of the three aspects is allowed

For example:

\[ CON(D) \iff ((C_t(D) = 2 \text{ hours}) \land (C_o(D) = 2)) \]

means that if 2 hours have elapsed since the last consistency action (which makes members consistent) and within this period more than 2 updates have been performed on the members of \( D \)
Interdependent data vs replicated data

Purpose is different —
Replicated data are induced to allow high reliability and availability of data in distributed databases while the interdependent data typically exists in multidatabase systems.

Control is different —
A replicated database system has total control over all copies of a replicated data while the members of an interdependent data are owned (and therefore controlled) by different systems.
In general, interdependent data represents a relaxation of replicated data along three dimensions —

- **Control** — members of an ID data are owned by different systems

- **Dependency** — ID data allow complex structural dependency and control dependency

- **Consistency** — ID data allow user specified mutual consistency
Our Solution

- Based on update through current copy
• ID read-only transactions are allowed to access lagging members

• Current copy is used for enforcing consistency action

\begin{center}
\begin{tikzpicture}
  \node (current) at (0,0) {current copy};
  \node (copy) at (2,0) {copy to};
  \node (consistent) at (4,0) {consistent};

  \draw[->] (current) -- (copy);
  \draw[->] (copy) -- (consistent);
\end{tikzpicture}
\end{center}
**Update Control**

- Distinguish **ID update transactions** from **ID read-only transactions** and then impose different control.

- **Mark primitives** are used for controlling access to ID data.

- **Global locks** on the ID data are used to implement the Mark primitives.
Controlling Mutual Consistency

• **Lazy enforcer**— we allow the ID data to remain inconsistent (i.e., consistency constraint is violated) until it is used by the outside world.

• Locking requests are used to trigger the evaluation of the consistency constraint.

• When inconsistency is detected, consistency action is invoked to enforce the mutual consistency.
**Efficient Copy**

*Copy by applying missing updates —*

- Version number start at 0 when mutual consistency is enforced
- Check the local version number against the largest version number in the list to determine the missing update
- Applying the missing update to make local member current

![Diagram showing version numbers and updates]

*missing updates are updates 4 and 5*

<table>
<thead>
<tr>
<th>version number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

*Update List*

- update_1
- update_2

*local version number at site k*
# Publications in InterBase

Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907

## I. Conference Publications

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>W. Du and A. Elmagarmid</td>
<td>QSR: A Correctness Criterion For Global Concurrency Control in InterBase, VLDB'89</td>
<td>[DE89]</td>
</tr>
<tr>
<td>1989</td>
<td>W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann</td>
<td>Effects of Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems, DKSME'89</td>
<td>[DELO89]</td>
</tr>
<tr>
<td>1989</td>
<td>Y. Leu and A. Elmagarmid</td>
<td>A Hierarchical Approach to Concurrency Control for Multidatabases, DPDS'90</td>
<td>[LE90]</td>
</tr>
<tr>
<td>1990</td>
<td>A. Elmagarmid and W. Du</td>
<td>A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems, DE'90</td>
<td>[ED90c]</td>
</tr>
<tr>
<td>1990</td>
<td>A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz</td>
<td>A Multidatabase Transaction Model for InterBase, VLDB'90</td>
<td>[ELLR90]</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
<td>Ref.</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>1990</td>
<td>M. Rusinkiewicz and A. Elmagarmid and Y. Leu and W. Litwin</td>
<td>Extending the Transaction Model to Capture More Meaning, SIGMOD Record</td>
<td>[RELL90]</td>
</tr>
<tr>
<td>1991</td>
<td>Y. Leu</td>
<td>Composing Multidatabase Applications using Flex Transactions, IEEE DE</td>
<td>[Leu91]</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
<td>Ref.</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>1989</td>
<td>A. Elmagarmid and W. Du</td>
<td>Supporting Value Dependency for Nested Transactions in InterBase, CS-TR</td>
<td>[ED89]</td>
</tr>
<tr>
<td>1989</td>
<td>W. Du, A. Elmagarmid, and Marek Rusinkiewicz</td>
<td>Transaction Accessibility and Integrity Issues in Heterogeneous Distributed Database Systems, CS-TR</td>
<td>[DER89]</td>
</tr>
<tr>
<td>1990</td>
<td>W. Du and A. Elmagarmid and W. Kim</td>
<td>Maintaining Quasi Serializability in HDDBSs, SERC-TR, Appear to DE'91</td>
<td>[DEK90b]</td>
</tr>
<tr>
<td>1990</td>
<td>W. Du and A. Elmagarmid and W. Kim</td>
<td>Effects of Local Autonomy on Heterogeneous Distributed Database Systems, MCC-TR, Submitted to CACM</td>
<td>[DEK90a]</td>
</tr>
<tr>
<td>1990</td>
<td>A. Elmagarmid and W. Du</td>
<td>Preserving Data Integrity in HDDBSs Using Quasi Serializable Executions, SERC-TR</td>
<td>[ED90d]</td>
</tr>
<tr>
<td>1990</td>
<td>A. Elmagarmid, W. Du and Y. Leu</td>
<td>Global Concurrency Control in Heterogeneous Distributed Database Systems, CS-TR, Submitted to ACM Comp. Surv.</td>
<td>[EDL90]</td>
</tr>
</tbody>
</table>
### III. Technical Reports (Continued)

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>Y. Leu, A. Elmagarmid, and M. Rusinkiewicz</td>
<td>An Extended Transaction Model For Multidatabase Systems, SERC-TR</td>
<td>[LER90]</td>
</tr>
<tr>
<td>1991</td>
<td>A. Sheth, Y. Leu, and A. Elmagarmid</td>
<td>Maintaining Consistency of Interdependent Data In Multidatabase Systems, CS-TR, Submitted to VLDB'91</td>
<td>[SLE91]</td>
</tr>
<tr>
<td>1991</td>
<td>A. Elmagarmid, J. Jing, and W. Kim</td>
<td>Global Commitment in Multidatabase Systems, CS-TR, Submitted to VLDB'91</td>
<td>[EJK91]</td>
</tr>
</tbody>
</table>

### References


References


