# Superdatabases for Composition of Heterogeneous Databases

*Calton Pu*

Department of Computer Science
Columbia University

**Abstract**

*Superdatabases* are designed to compose and extend databases. In particular, superdatabases allow consistent update across heterogeneous databases. The key idea of superdatabase is hierarhical composition of element databases. For global crash recovery, each element database must provide local recovery plus some kind of agreement protocol, such as two-phase commit. For global concurrency control, each element database must have local synchronization with an explicit serial order, such as two-phase locking, timestamps, or optimistic methods. Given element databases satisfying the above requirements, the superdatabase can certify the serializability of global transactions through a concatenation of local serial order. Combined with previous work on heterogeneous databases, including unified query languages and view integration, now we can build heterogeneous databases which are consistent, adaptable, and extensible by construction.

## 1 Introduction

For both efficiency and extensibility, integrated and consistent access to a set of heterogeneous databases is desirable. However, current commercial databases running on mainframes are, by and large, centralized systems. Physical distribution of data in distributed homogeneous databases has been demonstrated in several systems, such as R* [15] and INGRESS/STAR [18]. Nevertheless, the research on integrated heterogeneous databases has been limited to query-only systems.

Good examples of heterogeneous database research on unifying query languages and data view integration are MULTIBASE and MERMAID. MULTIBASE [5,13] is a retrieve-only system, developed at the Computer Corporation of America. Through the functional language DAPLEX, MULTIBASE provides uniform access to heterogeneous and distributed databases. The prototype implemented at CCA supports a CODASYL database and a hierarchical database. The focus of MULTIBASE is on query optimization and reconciliation of data, and consistent updates across databases were not part of their goals. More seriously, no global concurrency control was employed in the retrievals, so inconsistent data (from the global point of view) may be obtained in a query.

MERMAID [3,25] is also a retrieve-only system, but developed at the System Development Corporation (now part of UNISYS). Unlike MULTIBASE, MERMAID supports the relational view of data directly, through a query language, the ARIEL, which is a superset of SQL and QUEL. Another project providing a common query language to access databases using different data models is SIRIUS-DELTA [8].

In contrast to the relative success of research on query processing and optimization over heterogeneous databases, few results have been reported on consistent *update* across heterogeneous databases. To the best of our knowledge, only one paper [10] has discussed the properties of concurrency control mechanisms in heterogeneous database systems.

Our answer to this challenge is the building of *superdatabases*. Unlike earlier works on uniform query access through a single language, our emphasis is on consistent update across heterogeneous databases. A superdatabase is conceptually a hierarchical composition of element databases, which may be centralized, distributed, or other superdatabases.

Update support in homogeneous databases relies on two sets of fundamental techniques: concurrency control and crash recovery. We propose the construction of a superdatabase through hierarchical composition of concurrency control and crash recovery. Many years of research on nested transactions [16,19,22,27] have produced several particular ways to implement nested transactions organized into a hierarchy. Systematic use of hierarchical composition has been used to derive the design and implementation of a nested transaction mechanism in the Eden system [21]. Work reported here applies hierarchical composition to general databases.

In Section 2 we summarize the general architecture of superdatabases. In Section 3 we describe some sufficient conditions to make element databases composable. In Section 4, we explain the design of a superdatabase capable of gluing the element databases together. Section 5 sketches an implementation plan. In Section 6 we summarize related work on many different aspects of heterogeneous databases, comparing and constrasting them with ours. Finally, Section 7 concludes the paper.

## 2 Hierarchical Composition

### 2.1 General Structure

The superdatabase composes element databases hierarchically. In figure 1, $DB_j$ (the leaves) represent different element databases glued together by superdatabases (the internal nodes). A transaction spanning several element databases is called a *supertransaction*. When participating in a supertransaction, the local transaction on each element database is called a subtransaction. For simplicity of presentation, we assume that there is only one subtransaction per element database for each supertransaction. Although this is a standard assumption [10], there are cases (e.g. element databases running strict two-phase locking) in which this assumption is not necessary.



CH2550-2/88/0000/0548$01.00 

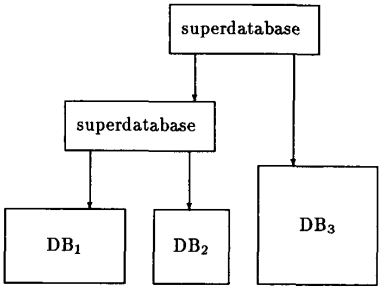


Figure 1: The Structure of Superdatabases

The main reason for the tree-structured organization is to minimize amount of data transfer, both in terms of message size and quantity. Since we will show in section 5.2 that we only need to piggyback a small amount of information on messages already required for distributed commit protocols, this goal has been achieved. Research to distribute the functions of superdatabase is outlined in section 5.4.

We divide this hierarchical composition into two parts. In section 2.2, we summarize the conditions an element database must satisfy, so the superdatabase can handle it. These conditions are described in further detail in section 3. In section 2.3, we outline the design ideas of the superdatabase to connect composable element databases. The design is detailed in section 4.

### 2.2 Composable Element Databases

An element database is said to be *composable* if it satisfies two requirements. The first is on crash recovery: the element database must understand some kind of agreement protocol, for example, two-phase commit. As we will see in section 4.1, this necessary requirement is a consequence of distributed control, not heterogeneity.

The second requirement is on concurrency control: the element database must present an explicit serial ordering of its local transactions. This may seem a serious requirement, demanding extensive modifications on the concurrency control mechanisms on the element databases. Actually, as we will see in section 3.3, all major concurrency control methods (two-phase locking, timestamps, and optimistic concurrency control) provide an easy way to capture the serial order they impose on the transactions. Furthermore, since any agreement protocol implies communication between participants, passing the explicit serial order of subtransactions (local to each element database) may piggyback on these messages, reducing the performance impact of the second requirement.

For consistent updates, these two are the only requirements we make on the element databases. An element database may be centralized, distributed, or as we shall see, another superdatabase. Unfortunately, in practice most centralized databases do not support any kind of agreement protocol. Similarly, most distributed databases do not supply the transaction serial order. Consequently, our results cannot be applied directly to existing databases without modification. Nevertheless, we believe that these relatively mild requirements, once identified, can be feasibly incorporated into current and future database systems. The pay-off is significant: extensibility and accommodation of heterogeneity.

### 2.3 The Superdatabase Glue

We have three design goals for the superdatabase that glue the composable element databases together. The main function of the superdatabase is to support consistent update across heterogeneous element databases.

1. Composition of element databases with many kinds of crash recovery methods.
2. Composition of element databases with many kinds of concurrency control techniques.
3. Recursive composibility; i.e. the superdatabase must satisfy the requirements of an element database.

The realization that we need only an agreement protocol for crash recovery made the first goal easy. The key idea that achieved the second goal is to use the explicit serial ordering of transactions, the common denominator of best known concurrency control methods. The third goal was accomplished through careful design of the agreement protocol and explicit passing of the serial order.

The superdatabse itself does not contain data, which are stored in the element databases. However, it does have to maintain the information to recover from crashes and serialize supertransactions. In Section 4.1, we describe the log management to guard the structure of the supertransaction against crashes. For concurrency control, we describe in section 4.4 the certification of serial order of each subtransaction involved in a supertransaction.

## 3 Composibility Conditions

### 3.1 Declarative Interface for Superdatabase

Since the superdatabase is a general-purpose glue to connect element databases of different construction, the interface to the superdatabase must be declarative and independent of particular implementation methods. It should specify what is needed, instead of what to do. The two requirements on the interface are that it should be simple enough to minimize adaptation effort on existing database systems, and general enough to allow composition of heterogeneous databases.

Currently, we use the following tentative interface, divided into two groups, the *Transaction* group and the *Resource* group.

- `BeginTransaction`( **in:** ParentID, **out:** TID )
- `CommitTransaction`( **in:** TID )
- `AbortTransaction`( **in:** TID )
- `OpenResource`( **in:** TID, ... other parameters ... )
- `CloseResource`( **in:** TID, ... other parameters ... )

The Transaction declarations bracket the extent of the transaction, which starts at `BeginTransaction`, and ends in `CommitTransaction` or `AbortTransaction` depending on the outcome of the transaction. These declarations are standard except for the ParentID parameter in `BeginTransaction` to allow run-time composition.

The Resource declarations correspond to the first and last access of a specific resource, defined as a portion of the database. These declarations are redundant, in that the information in them usually can be deduced from a mechanical analysis of the program. We make these declaration explicit for two reasons. First, we can avoid mentioning actions specific to particular concurrency control methods, for example, lock and unlock. Second, declarations

bracketing the resource access seem to be sufficient for the implementation of superdatabase. Since these declarations may be generated by a pre-processor on the transaction program, we can preserve the compatibility of the superdatabase with the application programs.

### 3.2 Distributed Transaction Commit and Recovery

The usual model of a distributed transaction contains a coordinator and a set of subtransactions. Each subtransaction maintains its local undo/redo information. At transaction commit time, the coordinator organizes some kind of agreement with subtransactions to reach a uniform decision. The two-phase commit protocol is the most commonly used because of its low message overhead. In phase one, the coordinator sends the message "prepare to commit" to the subtransactions, and these vote "yes" or "no". If all votes are "yes", the coordinator enters the phase two, sending the message "committed" to all subtransactions. Otherwise the coordinator decides to abort and sends "aborted" to all subtransactions.

The distributed database system R* [15] provides a tree-structured computation, which refines the above flat coordinator/subtransactions model. Subtransactions in R* are organized in a hierarchy, and the two-phase commit protocol is extended to the tree structure. At each level, the parent transaction serves as the coordinator. During phase one, the root sends the message "prepare to commit" to its children. The message is propagated down the tree, until a leaf subtransaction is reached, when it responds with its vote. At each level, the parent collects the votes; if all its own children voted "yes", then it sends "yes" to the grand-parent. If every subtransaction voted "yes", the root decides to commit and sends the "committed" message, propagated down the tree. Between the sending of its vote and the decision by the root, each child subtransaction remains in the *prepared* state, ready to either undo the transaction if aborted, and to redo the transaction if the child crashed and the root decided to commit.

Since heterogeneous databases are distributed by nature, it is necessary that each element database maintains the undo/redo information locally. Since the superdatabase stores only the global information, it has to rely on element databases for local recovery. In addition, it is necessary that each element database understands some kind of agreement protocol, such as the two-phase commit outlined above, three-phase commit, and the various flavors of Byzantine agreements. The following simple example demonstrates that the need for agreement comes from distribution, not heterogeneity.

Consider a distributed transaction $T$ with two subtransactions, $T_1$ and $T_2$. Suppose that $T$ commits if and only if both $T_1$ and $T_2$ commits, and that there is no agreement protocol between $T_1$ and $T_2$ at commit time. Therefore, one of them will decide to commit before the other. As soon as the first one decides to commit, the other crashes, aborting. Consequently, $T$ cannot commit, since one subtransaction aborted. However, $T$ cannot abort either, since the other subtransaction committed. Having shown the necessity of agreement protocol for distributed databases, including the heterogeneous ones, we proceed to compose different concurrency control methods.

### 3.3 Explicit Serialization for Concurrency Control

We assume the element databases maintain serializability of local transactions. The question is whether the superdatabase can maintain global serializability given local serializability. The answer is yes, if the superdatabase certifies that all local serial orders are compatible in a global serial order. One way to implement the superdatabase certification is to require that each element database provide the ordering of its local transactions to the superdatabase. Please note that this assumption provides a sufficient condition for composition of heterogeneous databases, but it is not necessary, since implicit serialization is possible under restricted circumstances (section 5.3). The serial order of each local transaction is represented by an *order-element*, or O-element for short. In Section 4, we shall describe the composition of O-elements for certification. Here, we only discuss how the concurrency control methods produce the O-elements.

First, we consider element databases with two-phase locking concurrency control. Locking says that a lock on a resource must be acquired before it may be accessed. Transactions using two-phase locking acquire all locks in a growing phase, and then release them during a shrink phase, in which no additional locks may be acquired. Eswaran et al. [7] showed that two-phase locking guarantees serializability of transactions because $SHRINK(T_i)$, the timestamp of transaction $T_i$'s lock point, indicates $T_i$'s place in the serialization. We take advantage of this fact and designate $SHRINK(T_i)$ as the O-element for element databases with two-phase locking.

The second most popular concurrency control method uses timestamps for serialization. Since transactions serialized by timestamps have their serialization order explicitly represented in their timestamps, these serve well as O-elements. Timestamp intervals [1] or multidimensional timestamps [14] can be passed as O-elements as well. The important thing is to capture the serialization order of committed local transactions.

As another alternative, optimistic concurrency control methods also provide an explicit serialization order. Kung and Robinson [11] assign a serial transaction number after the write phase, which can be used directly as O-element. Ceri and Owicki [4] proposed a distributed algorithm in which a two-phase commit follows a successful validation. Taking a timestamp from a Lamport-style global clock [12] at that moment will capture the serial order of transactions. Since the write phase has yet to start, all following transactions will have a later timestamp. Similarly, all preceding transactions must have obtained their timestamps before the validation phase has ended.

There is no constraint on the format of the O-element. Each element database may have its own representation. We only require that two O-elements from the same element database be comparable, and that this comparison recover the serialization guaranteed by local concurrency control methods. More formally, let the serialization produced by the concurrency control method be represented by the binary relation *precede* (denoted by $\leq$). We require that O-element($T_1$) $\leq$ O-element($T_2$) if $T_1 \leq T_2$ in the local serialization.

If an element database is centralized, its O-element can be easily obtained as described above. If an element database is distributed in nature, the timestamp will have to come from a global clock to assure total ordering. Ceri and Owicki's distributed optimistic algorithm is an example.

## 4 Algorithms Used in Superdatabase Design

Having established the composibility conditions in the previous section, now we proceed to use them in the superdatabase. For crash recovery, we describe a hierarchical commit protocol and its use in the recovery of supertransactions. For concurrency control, we describe a hierarchical certification algorithm that guarantees serializability given the O-vectors.

### 4.1 Hierarchical Commit

Given that some form of agreement is necessary (section 3.2), the question is whether it is sufficient for hierarchical commit. In R*, two-phase commit implements hierarchical commit. Since the only function of two-phase commit protocol is to reach agreement, and no recovery information is involved, we conclude that any agreement protocol will do. Examples we have mentioned in section 3.2 include three-phase commit and Byzantine agreements. All these agreement protocols have a natural extension to tree-structured computations. The important thing is that for each element database, the superdatabase must understand and use the appropriate protocol. If all element databases use the same protocol, the superdatabase has the obvious role in the hierarchical protocol. Interesting cases arise when element databases support different kinds of agreement protocols. In the discussion below, references on the Byzantine agreements can be found in the several PODC Proceedings; the other protocols are described in the recent book by Bernstein et al. [2].)

To simplify the discussion, we divide the distributed agreement protocols into two groups: symmetric and asymmetric. Symmetric protocols such as Byzantine agreements and decentralized two-phase commit give all participants equal role. In asymmetric protocols, a distinguished coordinator decides the outcome based on information supplied by other participants. For example, in the centralized and linear two-phase commit, as well as the three-phase commit, a coordinator initiates the protocol and decides whether the transaction commits or aborts.

If an element database supports an asymmetric agreement protocol, the superdatabase assumes the role of coordinator with respect to that element database. Notice that the superdatabase may have to act as the coordinator for different protocols. Compared to symmetric protocols, this situation is relatively simple since no information needs to be sent to the participants except for the final decision.

If some element databases employ symmetric protocols, in order to reach agreement each participant needs to send more information to all the others. We have two choices for the superdatabase. First, it can simulate the protocol by translating the information received from "asymmetric" element databases and retransmitting it to the "symmetric" participants. This method makes it easy to prove the correctness of the combined algorithm, but sends unnecessary messages without additional crash resiliency. Second, the superdatabase may act as a *representative* of the "asymmetric" participants, sending the result of the asymmetric protocols in one round of messages. The second method decreases the number of messages, but may increase the response time slightly. These two choices exist also for the communication between "symmetric" participants using different protocols.

In summary, the superdatabase functions both as a coordinator for the asymmetric agreement protocols and as a translator for the symmetric protocols. It collects sufficient information for supertransaction commit, and provides enough information for participants using symmetric protocols to reach their own conclusion that matches the superdatabase's.

### 4.2 Superdatabase Recovery

Since the superdatabase is the coordinator for the element databases during commit, it must record the transaction on stable storage. Otherwise, a crash during the window of vulnerability would hold resources in the element databases indefinitely.

Of the known recovery methods, logging is the best for superdatabase recovery. Since no before-images or after-images need to be saved, versions are of little utility. Conceptually, the superdatabase log is separate from the element database logs, just as the superdatabase itself. In actual implementation, the superdatabase log may be physically interleaved with an element database log, as long as the recovery algorithms can separate them later.

For each transaction, the superdatabase saves the following information on the log:

- Participant subtransactions.
- Parent superdatabase, if any.
- Transaction state (prepared, committed, or aborted).

The superdatabase should remember the participant subtransactions because the transaction does not necessarily abort when the superdatabase crashes. Suppose that the superdatabase crashes, but is brought back online quickly, before its subtransactions have finished. Since the superdatabase performs no computation, the supertransaction may still commit. To carry out two-phase commit after such glitches, the participant subtransactions should be remembered in the log, which is read at restart to reconstruct the superdatabase state before the crash.

The transaction state is written to the log during the agreement protocol. If a transaction was in the active state when the superdatabase crashed, the superdatabase simply waits for (re)transmission of two-phase comm't from the parent. In case it is the root, it (re)starts the two-phase commit. If a transaction was in the prepared state when the superdatabase crashed, the superdatabase inquires the parent about the outcome of the transaction. If the transaction has been committed, the results are retransmitted to the subtransactions.

### 4.3 Concurrency Control: An Example

Consider the following example, in which subtransactions $T_{1.1}$ and $T_{1.2}$ run on element databases $DB_1$ and $DB_2$, respectively.

`BeginTransaction`(Top-level, $T_1$)
  `cobegin`
$DB_1$.`BeginTransaction`($T_1$, $T_{1.1}$) ... *actions* ... `CommitTransaction`($T_{1.1}$)
$DB_2$.`BeginTransaction`($T_1$, $T_{1.2}$) ... *actions* ... `CommitTransaction`($T_{1.2}$)
  `coend`
`CommitTransaction`

If both $DB_1$ and $DB_2$ use *strict* two-phase locking, we have no problem. Since no lock will be released before the commit time, the lock point for all data access in the supertransaction happens when the supertransaction commits at phase two of two-phase commit. Consequently, the supertransaction is two-phase and the superdatabase needs to take no action for concurrency control.

However, if locks may be released before commit agreement, then in the above example $T_{1.1}$ may start releasing locks while $T_{1.2}$ has not reached its lock point. Consequently, the supertransaction $T_1$ may lose its two-phase property and become non-serializable. Although there are other reasons to avoid early lock releases such as cascading of aborts, this case reveals the crucial problem in hierarchical composition of two-phase locking mechanisms: we need to synchronize the lock points of the participating subtransactions. If element databases use strict two-phase locking, the synchronization comes for free at commit time. Otherwise, an explicit synchronization is necessary, which may be pessimistic or optimistic. In the pessimistic case, unlock requests in the element databases are blocked. It is only after all subtransactions have

reached their lock points, indicated by a commit vote or an unlock request, that the superdatabase allows the element database to proceed.

Alternatively, the synchronization may be optimistic. The subtransactions may be allowed to run independently, without preventive synchronization. Since two-phase locking provides dynamic atomicity [26], the subtransactions from two different supertransactions may interleave in a non-serializable manner. To check the serializability of all subtransactions, we use the explicit serialization order of two-phase locking, captured by the O-elements.

### 4.4 Hierarchical Certification with O-vectors

The main problem that the superdatabase has to detect is when subtransactions from different element databases were serialized in different ways. In our example, this happens when a second transaction $T_2$ with the same subtransactions produces the ordering: O-element($T_{1.1}$) $\leq$ O-element($T_{2.1}$) and O-element($T_{2.2}$) $\leq$ O-element($T_{1.2}$).

To prevent this kind of disagreement from happening, we define an order-vector (O-vector) as the concatenation of all O-elements of the supertransaction. In the example, O-vector($T_1$) is (O-element($T_{1.1}$), O-element($T_{1.2}$)). The order induced on O-vectors by the O-elements is defined strictly: O-vector($T_1$) $\leq$ O-vector($T_2$) if and only if for all element database $j$, O-element($T_{1.j}$) $\leq$ O-element($T_{2.j}$). If a supertransaction is not running on all element databases, we use a wild-card O-element, denoted by $*$ (star), to fill in for the missing element databases. Since its order does not matter, by definition, O-element(any) $\leq *$, and, $* \leq$ O-element(any).

From this definition, if O-vector($T_1$) $\leq$ O-vector($T_2$) then all subtransactions are serialized in the same order, ordering the supertransactions. Therefore, we can serialize the supertransactions by checking the O-elements of a committing supertransaction against the history of all committed supertransactions. If the new O-vector can find a place in the total order, it may commit.

The comparison with all committed supertransactions may be expensive, both in terms of storage and processing. Fortunately, it is not necessary to compare the O-vector with all committed supertransactions. Since a transaction trying to commit cannot be serialized in the ancient history, it is sufficient to certify the transaction with a reasonably "recent history" of committed supertransactions.

The part of the serialization history we have to look at is limited by the oldest active transaction in each element database. Suppose we are certifying an O-vector whose subtransactions are older than the currently oldest active transaction on all element databases. Comparing this O-vector to the history of all committed supertransactions, we may not be able to certify this O-vector because of some other older transaction, in which case it must be aborted. Alternatively, we may find a place in the serialization history for the O-vector. Once we find such an O-vector($T_0$) preceding all active subtransactions, it must precede the O-vectors of all serializable supertransactions that have yet to commit.

The above claim follows from the observation that any subsequent O-vector must have one component preceded by the corresponding component in $T_0$. (The component that was active when $T_0$ was certified.) Consequently, either the new O-vector cannot be serialzed with respect to $T_0$ and is aborted, or all its components are preceded by $T_0$, QED. From this claim, in the certification process we need only to compare the new O-vector with $T_0$ and O-vectors more recent than $T_0$. Therefore, the O-vectors preceding $T_0$ are not necessary and can be released.

From the composition point of view, the key observation is that the certification based on O-vectors is independent of particular concurrency control methods used by the element databases. Therefore, a superdatabase can combine two-phase locking, timestamps, and optimistic concurrency control methods in any way. As long as we can make the serialization in element databases explicit, the superdatabase can certify the serializability of supertransactions.

More importantly, the certification gives the superdatabase itself an explicit serial order (the O-vector) allowing it to be recursively composed as an element database. Thus we have found a way to hierarchically compose database concurrency control, maintaining serializability at each level.

The certification method is optimistic, in the sense that it allows the element databases to run to completion and then certifies the serial ordering. In particular, the O-vector is constructed only after the subtransactions have committed. Since some concurrency control techniques (such as time-interval based and optimistic) decide the transaction ordering only at the transaction commit time, it is difficult for the superdatabase to impose an ordering during subtransaction execution. In other words, the superdatabase has to be as optimistic as its element databases.

## 5 Implementation and Performance

### 5.1 Superdatabase

Although in principle a superdatabase must check the serializability of all subtransactions, there are important cases that permit some optimization. As we have observed in section 4.3, if all element databases use strict two-phase locking, the lock points of the subtransactions are synchronized by the agreement protocol, and no certification will be necessary. However, the certification algorithm should be used if simple two-phase locking, timestamps, or optimistic concurrency control is introduced into the superdatabase.

In crash recovery, since in practice all distributed databases use two-phase commit, the introduction of more sophisticated agreement protocols into the superdatabase will await their use in the element database first.

Currently, several groups of students are implementing parts (query translation and execution, concurrency control and storage management) of an element database and a prototype superdatabase. Another prototype based on the Camelot transaction system [24] is under way. Camelot runs on top of Mach, a Berkeley/Unix-compatible operating system. Transaction functions supported by Camelot include a full nested transactions mechanism, fast and reliable logging, and many utility packages.

Taking advantage of Camelot's Unix compatibility, we intend to adopt existing distributed databases running on Unix, for example, the public domain INGRES. In this case, we need to add both O-element passing and two-phase commit. Another candidate element database is the one mentioned above being implemented by project students on top of the Synthesis operating system [20].

### 5.2 Run-Time Cost

We have argued in Section 4.4 that the certification process is limited to the recent history of committed supertransactions. Since the certification occurs in a central location (the superdatabase), and is limited by the recent history, the message overhead is small. Postponing the question of distributing superdatabase to the next section, we turn our attention to possible sources of delay in the element databases or communications.

In the element databases, we require only that the serial order of transactions be made explicit. With some concurrency control methods, such as timestamps, this is trivial. If the element database is centralized, then the cost of taking a timestamp is also low. However, if the element database is a distributed database with internal concurrency control, then a global clock will be necessary to capture the serial order. Fortunately, the maintenance of a global clock is independent of the number of transactions, and therefore can be amortized.

Finally, the additional piece of information that the superdatabase requires from the element databases is the O-element. Since we have demonstrated the necessity of an agreement protocol for recovery purposes, at least one message must be exchanged between the superdatabase and each element database at commit time. The certification occurs only at commit time, so the subtransaction serial order information can piggyback on the commit vote message. Therefore, the superdatabase does not introduce any extra message overhead during transaction processing.

### 5.3 Transaction Concurrency

The superdatabase design using O-vectors in section 4 is minimal in the sense that it receives only the explicit serialization order from the element databases. Consequently, supertransactions that are in reality unrelated, but apparently conflict due to their serialization order, will be aborted.

Fortunately we have found methods to increase concurrency in the superdatabase by taking into account the particular information provided by each concurrency control method. Two examples are two-phase locking and timestamps.

In the first place, element databases using strict two-phase locking do not have to participate in the certification. Since they hold their locks, and their lock points are synchronized by the hierarchical commit protocol, they are serialized with respect to each other and all other component transactions. This observation applies even to the minimal design.

Second, we can avoid unnecessary aborts involving element databases using general two-phase locking concurrency control. All it takes is an agreement protocol to synchronize the lock point of participating component transactions. If a supertransaction has several component transactions under general two-phase locking, it could use two-phase agreement once to synchronize the lock points, and a second time to commit the supertransaction. However, we have to be careful and take into account the ordering of these component transactions with respect to other component transactions synchronized through different concurrency control methods.

Third, timestamp-based element databases could provide the superdatabase with additional information. For example, time-interval based concurrency control methods would allow the superdatabase to serialize some transactions that would have been aborted in the minimal design.

Finally, we observe that serializability is itself more restrictive than optimal scheduling. We use serializability as the best trade-off in overhead and number of transactions unnecessarily aborted. Similarly, in the design of supertransactions, we strive for a good trade-off between run-time overhead and the additional restriction on concurrency.

### 5.4 Symmetric Distribution

As we have seen in previous sections, hierarchical organization of superdatabases results in low message overhead. However, the main disadvantage of the hierarchical structure is its centralized organization. Shutting down any of the internal nodes will isolate part of the tree. More concretely, if any node running a superdatabase crashes, all element databases connected to that superdatabase will remain inaccessible.

We are investigating two research directions to distribute the functions of superdatabase, which consists of participation in agreement protocols for recovery and serialization certification for concurrency control. On the recovery side, any node can assume the different roles in different agreement protocols, so distributing crash recovery seems relatively straightforward. The situation is more complicated for concurrency control.

First, we can replicate the superdatabase nodes, resulting in higher message overhead to keep the replicas consistent. Simple replication comes close to being the "brute force" method to distributed functions in a distributed system. In principle, just about any program or data can be distributed this way, provided that they are kept consistent. Unfortunately, consistent replication is expensive and this approach then loses the low-overhead advantage of hierarchical superdatabase.

Second, we can circulate the concurrency control certification information among several sites. This approach is similar to the work by Ceri and Owicki [4] in distributing the optimistic concurrency control certification algorithm. Again, higher message overhead will be necessary. Perhaps the hierarchical organization with low overhead functions best under normal situations, and a distributed algorithm should be added if more fault-tolerance is desired in the heterogeneous database.

## 6 Comparisons

### 6.1 Crash Recovery

The hierarchical commit algorithm described in section 4.1 is a direct descendent of distributed commit protocols such as R* [15] and commit protocols for nested transactions [21]. Our conclusion is that heterogeneity does not introduce additional difficulty, compared to homogeneous distributed databases.

Gligor and Luckenbaugh [9] have described the recovery problem in heterogeneous databases. Using a terminlogy based on two-phase commit protocol, they suggested that the prepared state may be necessary for any recovery algorithm. Since we know that the window of vulnerability always exists in distributed commit, and that the prepared state of two-phase commit corresponds to the window of vulnerability, we have confirmed their conjecture. In addition, our work shows that any agreement protocol will do, not just two-phase commit.

### 6.2 Concurrency Control

Gligor and Popescu-Zeletin [10] studied concurrency control in heterogeneous databases with emphasis on deadlock detection. Through an example, they showed that there exist some deadlocks which escape hierarchical distributed deadlock detection algorithms. Consequently, either we employ some deadlock avoidance mechanism such as time-outs, or we must pass local dependency information to global deadlock detection algorithms. They also specified five conditions which should be satisfied by any concurrency control mechanisms for heterogeneous databases.

Their first condition says that all local concurrency control (of component databases) must provide local synchronization atomicity. We also make this assumption. Their second condition says that all local concurrency control must preserve the relative order of execution determined by the global transaction manager. This corresponds to a pessimistic approach. In contrast, the superda-

tabase certifies the serializability after the execution in an optimistic manner. Their third condition says that each site can run only one subtransaction. Although we also make this assumption for simplicity, we are working to relax this restriction. Their fourth condition says that the global transaction manager must be able to identify objects referenced by all subtransactions. Using explicit serialization order in O-elements, we have eliminated the need to check object references. Finally, their fifth condition refers to global deadlock detection. Deadlocks remain a problem for further research.

Elmagarmid and Leu [6] have studied the use of a centralized optimistic concurrency control to validate each subtransaction based on its readset and writeset. Readset and writeset of subtransactions are sent to the Global Data Manager for validation at global transaction commit. Their approach allows more concurrency between transactions since their validation is sophisticated. In compensation, the superdatabase requires a much smaller amount of data transfer for concurrency control and the work necessary for validation is simple.

### 6.3 Partial Integration

In contrast to our "strongly consistent" database composition, significant work has been done based on weaker consistency constraints. Two examples of this approach are MRDSM [17] and ADMS± [23]. Being developed at INRIA, the prototype multidatabase system MRDSM provides a relational interface to independent databases. Instead of global schemas, special "dependency schemas" define interdatabase relationships. Since they avoid integration by design, no consistent updates are included in MRDSM.

ADMS± takes advantage of current hardware advances to integrate a mainframe database (ADMS+) with workstation databases (ADMS−) downloaded from the mainframe. Since each user typically uses only a portion of the database, local queries on ADMS− data are very efficient. Updates occur only on ADMS+ and they are incrementally propagated to ADMS− databases offline. In summary, ADMS± can be seen as a systematic decomposition of a centralized database.

### 6.4 Other Issues

Deadlock detection is non-trivial for a hierarchical approach. Simple examples have been exhibited in which distributed deadlocks cannot be detected in a hierarchical way [10]. More work on deadlock detection and avoidance will be necessary to determine the advantages and disadvantages of each. Since time-out mechanisms are necessary for network communications, it seems reasonable to use it to avoid deadlocks in distributed systems connected through superdatabases.

## 7 Conclusion

We have described the design of superdatabases and the algorithms used to compose consistent databases out of both homogeneous and heterogeneous elements. There are four good characteristics in the superdatabase approach to building heterogeneous databases.

First, superdatabases guarantee the atomicity of global updates across the element databases. This atomicity includes both reliability atomicity through an agreement protocol, such as two-phase commit, and concurrency atomicity through the certification of serialization provided by the element databases.

Second, the design of superdatabase is adaptable to a variety of crash recovery methods and concurrency control techniques used in the element databases. We have established the necessity for an agreement protocol for supertransaction commit. However, the protocol is independent of particular crash recovery methods used to undo and redo local transactions in the element databases. We have also shown that as long as the element databases use concurrency control methods which easily supply an explicit serial order of their transactions, they can be included under the superdatabase.

Third, databases built with superdatabases are extensible by construction. Element databases may be added or deleted without changing the superdatabase. In additon, many interesting applications can take advantage of the extensibility. For example, a replicated database can be constructed by connecting two identical element databases with a superdatabase. Another example is that given a database X, satisfying the requirements of section 3 for crash recovery and concurrency control, a superdatabase delivers the distributed version of X.

Fourth, transactions local to element databases run independently of the superdatabase, which intervenes only when needed for synchronization or recovery of supertransactions across different element databases. In other words, the additional overhead introduced by the indirection through superdatabase is paid only by the direct users of its services. The only interference happens when a component transaction of a supertransaction conflicts with a local transaction.

Even though we described the serialization of supertransactions using O-vectors, the hierarchical approach admits other methods that explore the properties of particular concurrency control methods. For example, using an agreement to synchronize lock points of two-phase locking elements databases and distributing global timestamps to timestamp-based element databases are techniques that may improve the concurrency in the superdatabase.

Global deadlock detection and resolution remains a research challenge, since it is immune to hierarchical approaches. Observing that the time-out mechanism is inherent in distributed systems, we expect it to be useful in avoiding deadlocks.

Many years of research on heterogeneous databases have achieved impressive and substantial progress, especially in query language translation and view integration. We hope the combination of our results with previous work on heterogeneous databases will produce superdatabases which are consistent, adaptable, and extensible.

## References


[1] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser.
Dynamic timestamp allocation for transactions in database systems.
In H. J. Schneider, editor, *Distributed Data Bases*, North-Holland, 1982.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman.
*Concurrency Control and Recovery in Database Systems.*
Addison-Wesley Publishing Company, first edition, 1987.

[3] D. Brill, M. Templeton, and D. Yu.
Distributed query processing strategies in MERMAID, a frontend to data management systems.
In *Proceedings of the First International Conference on Data Engineering*, 1984.

[4] S. Ceri and S. Owicki.
On the use of optimistic methods for concurrency control in distributed databases.
In *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Lawrence Berkeley Laboratory, University of California, Berkeley, February 1982.

[5] U. Dayal.
Processing queries over generalization hierarchies in a multidatabase system.
In *Proceedings of the Ninth International Conference on Very Large Data Bases*, October-November 1983.

[6] A. Elmagarmid and Y. Leu.
An optimistic concurrency control algorithm for heterogeneous distributed database systems.
*Data Engineering Bulletin*, 10(3):26–32, September 1987.

[7] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notions of consistency and predicate locks in a database system.
*Communications of ACM*, 19(11):624–633, November 1976.

[8] A. Ferrier and C. Stangret.
Heterogeneity in the distributed database management system SIRIUS-DELTA.
In *Proceedings of the Eighth International Conference on Very Large Data Bases*, Mexico City, September 1983.

[9] V. Gligor and G.L. Luckenbaugh.
Interconnecting heterogeneneous database management systems.
*Computer*, 17(1):33–43, January 1984.

[10] V. Gligor and R. Popescu-Zeletin.
Concurrency control issues in distributed heterogeneous database management systems.
In F.A. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*, pages 43–56, North Holland Publishing Company, 1985.
Proceedings of the International Symposium on Distributed Data Sharing Systems.

[11] H. T. Kung and John T. Robinson.
On optimistic methods for concurrency control.
*Transactions on Database Systems*, 6(2):213–226, June 1981.

[12] L. Lamport.
Time, clocks and ordering of events in a distributed system.
*Communications of ACM*, 21(7):558–565, July 1978.

[13] T. Landers and R.L. Rosenberg.
An overview of MULTIBASE.
In H.J. Schneider, editor, *Distributed Data Bases*, North Holland Publishing Company, September 1982.
Proceedings of the Second International Symposium on Distributed Data Bases.

[14] P.J. Leu and B. Bhargava.
Multidimensional timestamp protocols for concurrency control.
In *Proceedings of the Second International Conference on Data Engineering*, pages 482–489, Los Angeles, February 1986.

[15] B. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost.
Computation and communication in R*: a distributed database manager.
*ACM Transactions on Computer Systems*, 2(1):24–38, February 1984.

[16] B.H. Liskov and R.W. Scheifler.
Guardians and Actions: linguistic support for robust, distributed programs.
In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 7–19, January 1982.

[17] W. Litwin and A. Abdellatif.
Multidatabase interoperability.
*Computer*, 19(12):10–18, December 1986.

[18] R. McCord.
INGRES/STAR: a distributed heterogeneous relational DBMS.
Vendor Presentation in SIGMOD, May 1987.

[19] J.E.B. Moss.
*Nested Transactions: An Approach to Reliable Distributed Computing.*
PhD thesis, Massachusetts Institute of Technology, April 1981.

[20] C. Pu, H. Massalin, J. Ioannidis, and P. Metzger.
*The Synthesis System.*
Technical Report CUCS-259-87, Department of Computer Science, Columbia University, February 1987.

[21] Calton Pu.
*Replication and Nested Transactions in the Eden Distributed System.*
PhD thesis, Department of Computer Science, University of Washington, 1986.

[22] D.P. Reed.
*Naming and Synchronization in a Decentralized Computer System.*
PhD thesis, Massachusetts Institute of Technology, September 1978.

[23] N. Roussopoulos and H. Kang.
Principles and techniques in the design of ADMS±.
*Computer*, 19(12):19–25, December 1986.

[24] A. Spector, J.J. Bloch, D.S. Daniels, D. Duchamp, R.P. Draves, Eppinger J.L., S.G. Menees, and D.S. Thompson.
*The Camelot Project.*
Technical Report CMU-CS-86-166, Computer Science Department, Carnegie-Mellon University, December 1986.

[25] M. Templeton, D. Brill, A. Hwang, I. Kameny, and E. Lund.
An overview of the MERMAID system – a frontend to heterogeneous databases.
In *Proceedings of EASCON 1983*, pages 387–402, IEEE/Computer Society, 1983.

[26] W.E. Weihl.
*Specification and Implementation of Atomic Data Types.*
PhD thesis, Massachusetts Institute of Technology, March 1984.
Tech.Report MIT/LCS/TR-314.

[27] M. Weinstein, T. Page, B. Livezey, and G. Popek.
Transactions and synchronization in a distributed operating systems.
In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 115–126, ACM/SIGOPS, December 1985.