Accessing Extra Database Information:
Concurrency Control and Correctness

N. Gehani, K. Ramamritham,
J. Shanmugasundaram, and O. Shmueli

CMPSCI Technical Report 96-16

February 1996

Accessing Extra Database Information: Concurrency Control and Correctness

Narain Gehani
AT&T Bell Laboratories
nhg@research.att.com

Krithi Ramamritham*
Jayavel Shanmugasundaram
University of Massachusetts
{krithi,shan}@cs.umass.edu

Oded Shmueli
Technion - Israel
Institute of Technology
oshmu@cs.technion.ac.il

February 1996

Abstract

Traditional concurrency control theory views transactions in terms of read and write operations on database items. Thus, the effects of accessing non-database entities, such as the system clock or the log, on a transaction's behavior are not explicitly considered. In this paper, we are motivated by a desire to include accesses to such *extra-data* items within the purview of transaction and database correctness. We provide a formal treatment of concurrency control when transactions are allowed access to extra data by discussing the inter-transaction dependencies that are induced when transactions access extra data. We also develop a spectrum of correctness criteria that apply when such transactions are considered. Furthermore, we show that allowing databases to view data which has been traditionally kept hidden from users increases the database functionality and in many cases can lead to improved performance.

*partially supported by the National Science Foundation under grant IRI-9314376.

Contents

1 Introduction **1**

2 Extra Data: Characteristics and Correctness **2**

3 Preliminaries and Definitions **4**
3.1 Operations, Events, and Histories 4
3.2 Conflicts between Operations 5
3.3 Atomic and Non-Atomic Transactions 5
3.4 Serializability 6
3.5 Extra-data Independent Transactions 7

4 Correctness Notions **10**

5 Examples of Extra Data and Associated Correctness **11**
5.1 Extra Data Independent Transactions 11
5.1.1 Work Queues 11
5.1.2 Serializability in Multi-databases 12
5.1.3 A Car Reservation Example 13
5.1.4 Proclamation Locking 13
5.2 Serializable Accesses to the Extended Database 14
5.3 Serializable Accesses to the Database 16
5.4 Application Specific Correctness Notions 17
5.4.1 The Log - Multiple Entries per Transaction 17
5.4.2 The System Clock 18
5.4.3 Page Space Map Tables 19

6 Discussion **19**

1 Introduction

Traditional concurrency control theory views transactions in terms of reads and writes on database items. Our goal in this paper is to examine the concurrency and correctness issues that arise when we take into consideration all the data that is used or can be used by transactions, not just what is in the database. Roughly speaking, "extra data" means useful information that is not part of the enterprise's database schema, but is nevertheless useful and may affect what ultimately appears in the "proper database". Such data lives in the shaded zone between the database and the application software; it is our goal to shed some light on its use and the ensuing implications.

Thus, for example, we are concerned with data that has always been accessed by the transactions but not included traditionally in serializability theory, e.g., the system clock, communication channels, and scratch-pads. We also examine information that can be useful for transactions, such as those that are maintained by lock and recovery managers, e.g., the log and information about transactions waiting for locks (some systems already allow log accesses for tuning and control purposes).

Allowing databases to view such data, namely "extra data", can increase database functionality and can lead to improved performance. For example, if transactions were allowed to access the log, then they could answer queries about the operations performed by previously committed transactions. In this case, the functionality of transactions is enhanced since they access an extra data item (the log). Also, with access to data such as the predicted behavior of other transactions (e.g., their expected execution times or data access patterns) and transaction management information (e.g., the number of transactions waiting to perform operations on a particular object) transactions can be designed for improved performance. For example, knowing how many transactions are waiting for a particular data item may allow a transaction to consider alternatives which are likely to reduce its response time.

We elaborate on the benefits of accessing extra data by (1) showing that there are in fact a number of instances where extra data can be viewed and manipulated as first-class data items; (2) examining the properties of such extra data from the viewpoint of the transactions and their correctness; and (3) providing a formal treatment of concurrency control when transactions are allowed access to extra data.

We use the term *database* with its traditional meaning and use the term *extended database* to refer to the database plus the extra-data objects. We introduce an important new concept, that of *extra-data independent transactions.* Such transactions, intuitively, perform correct database state transitions (in the conventional sense) despite their extra-data accesses. We treat four main correctness requirements: (1) extra data independence of transactions (2) serializability over the extended database (3) serializability only over the database and (4) other application specific correctness criteria. For each of the above, we illustrate its usefulness and flexibility in achieving user's goals. This analysis sheds light on the role, usefulness and pitfalls in using extra data.

In many databases, in accordance with protection and security considerations, transactions are allowed access to data on a need-to-know basis. Clearly, when transactions are allowed access to information that is usually within the purview of the transaction management system the protection and security ramifications of such accesses must also be examined. This, however, is outside the scope of this paper.

The rest of the paper is organized as follows. Examples of extra data and a discussion of their characteristics can be found in Section 2. A brief introduction to the formalism used to describe the properties of extra data and discuss the correctness of transactions using extra data is given in Section 3. The different correctness notions are discussed in Section 4. Section 5 provides details of these correctness notions and illustrates them via concrete examples. Section 6 summarizes the paper and discusses the outstanding issues.

2 Extra Data: Characteristics and Correctness

Extra data can be defined as data that is typically not considered to be part of the database. Extra data can be classified into four categories:

1. Data values modified by some entity outside the database system. The system clock is a prime example of this.
2. Data values modified by the transaction management system in response to some request initiated by a transaction. Information about waiting transactions, concurrency control information such as the serialization graph, and the log are examples here.
3. Data values pertaining to (and modified by) the transactions themselves. Estimates of a transaction's (remaining) execution time and data requirements are examples.
4. Data private to a specific set of transactions. A scratch pad used by a set of cooperating transactions to coordinate their activities is an example of this case.

Figure 1 depicts the components of an extended database. The shaded area denotes the extra data. Note that not all data in the first two categories may be considered as extra data since only those that can be accessed by transactions are considered to be extra data. It is important to note that unlike database items, extra-data items need not be persistent and cannot always be *exclusively* accessed (read and updated) by user transactions.

Clearly, updates to a particular type of extra data will be restricted based on the type. For instance, whereas no transaction can update "system updated" or "transaction-management updated" extra data, a transaction might be allowed to update extra data pertaining to itself and not others. These observations have certain important implications for concurrency control. For instance, while it may be possible to delay the writing of extra-data items belonging to categories (2) and (3) for concurrency control reasons, this is not possible for extra-data items in category (1). Such delays must be carefully evaluated for their effects on other transactions. For instance, if transactions' commitment or abortion could be delayed (for correctness reasons) purely because of their access to extra data, performance can degrade. Thus, it is important to weigh the pros and cons of transactions' access to extra data before their use is permitted in general. Our goal in this paper however is to

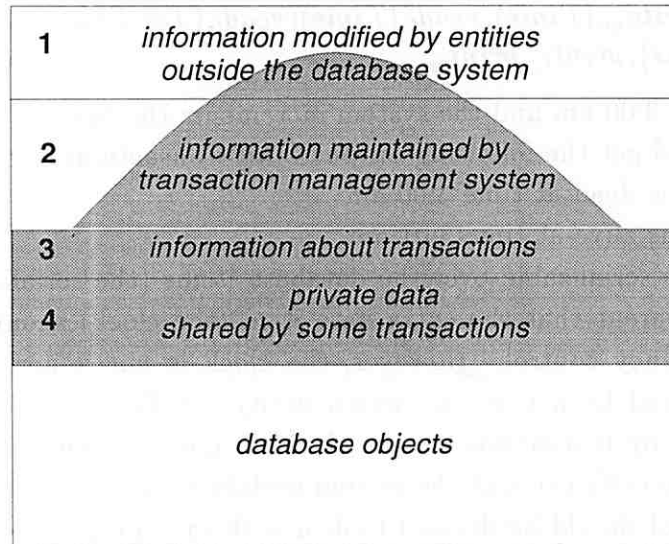

Figure 1: The Extended Database

understand the concurrency control and correctness issues related to such accesses to extra data since in reality transactions *do* access data outside the database.

Both the control flow within a transaction as well as a transaction's data manipulation properties may be modified by the extra-data accesses. What a transaction is allowed to do with the extra data that it reads depends on the correctness requirements imposed on the transactions. For instance, a transaction can use it to optimize its actions – but we may require that the semantics of the transformation performed by the transaction be independent of the resulting behavior. This may require including the extra data items within the scope of concurrency control, as will be discussed later.

More liberal is the case where transactions are allowed to make changes to the database and return values that depend on the values of the extra data. In this case again, it may be necessary to include the extra data items within the scope of concurrency control, as will be evident from the example given below.

Consider two transactions t_1 and t_2 which access the system clock (an extra data item). Both t_1 and t_2 execute the following program.

```
read(Time);
read(Joe_Status);
print ("Joe is" Joe_Status "at time" Time);
```

Thus both transactions t_1 and t_2 read the current time, check Joe's status (whether he is dead or alive) at that time and report it to the user. If we now consider another transaction t_3 which changes *Joe_Status* from "alive" to "dead" (represented as $write_{t_3}(Joe_Status)$ below) and a system event which periodically updates the clock (represented as $write_{sys}(Time)$ below), then the following interleaving of actions is possible.

$read_{t_1}(Time); write_{sys}(Time); read_{t_2}(Time); read_{t_2}(Joe_Status); write_{t_3}(Joe_Status); read_{t_1}(Joe_Status); print_{t_1}; print_{t_2};$

If t_1 read the clock at 3:00 am and the system increments the time by 1 minute during each update, then we would get the following output from transactions t_1 and t_2.

t_1: Joe is dead at time 3:00 am

t_2: Joe is alive at time 3:01 am

The above schedule is serializable over the database items (the serialization order is t_2, t_3, t_1) and it is thus apparent that the extra data item (the clock) should be brought within the scope of concurrency control. However, the clock is not a normal data item in the sense that it is updated by a non-transaction entity. In fact, in the above example, the operations performed by transaction on the clock do not by themselves conflict with each other but conflict *indirectly* through the system update to the clock. Thus new mechanisms for concurrency control should be devised to deal with such properties of extra data. These issues are treated in detail in subsequent sections.

3 Preliminaries and Definitions

In this section we introduce a simple formalism based on the ACTA transaction framework [2]. We also define some of the terms used in the rest of the paper and give formal definitions of certain correctness properties involved when transactions access extra data. A transaction accesses and manipulates objects in the extended database, i.e., objects in the database as well as extra-data objects, by invoking operations specific to individual objects.

3.1 Operations, Events, and Histories

It is assumed that operations are atomic and that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s,p)$ to denote the output produced by operation p, and $state(s,p)$ to denote the state produced after the execution of p.

DEFINITION 3.1: Invocation of an operation on an object is termed an *object event.* The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t. (For simplicity of exposition assume that a transaction does not contain multiple p's. When "what the ob is" is clear, we will simply say p_t. $p_t(val)$ is used to denote op that either takes in val as input or produces val as output.)

DEFINITION 3.2: Committing or aborting a transaction, committing or aborting an operation performed by a transaction are all termed as *transaction management events.* $commit_{t_i}$ and $abort_{t_i}$ denote the commit and abort of transaction t_i respectively. $commit[p_{t_i}[ob]]$ and $abort[p_{t_i}[ob]]$ denote the commit and abort of operation p performed by transaction t_i on object ob, respectively.

DEFINITION 3.3: A *history* is a partially ordered set of events invoked by transactions. Thus, object events and transaction management events are both part of the history $\mathcal{H}$. Transaction management events include committing a transaction, aborting a transaction, committing an operation performed by a transaction and aborting of an operation performed by a transaction. The set of events invoked by a transaction t is a partial order denoting the temporal order in which the related events occur in the history. The transaction's partial order is consistent with the history's partial order.

We write $(\epsilon \in \mathcal{H})$ to indicate that the event ϵ occurs in the history $\mathcal{H}$. $\rightarrow$ denotes precedence ordering in the history $\mathcal{H}$ and $\Rightarrow$ denotes logical implication.

3.2 Conflicts between Operations

DEFINITION 3.4: Let $\mathcal{H}^{(ob)}$ denote the projection of the history with respect to the operations on *ob*. [1] Two operations p and q on an object *ob* *conflict* in a state produced by $\mathcal{H}^{(ob)}$, denoted by *conflict*$(\mathcal{H}^{(ob)}, p, q)$, if

$$\begin{aligned}(state(\mathcal{H}^{(ob)} \circ p,\ q) &\neq state(\mathcal{H}^{(ob)} \circ q,\ p)) \quad \vee \\ (return(\mathcal{H}^{(ob)},\ q) &\neq (return(\mathcal{H}^{(ob)} \circ p,\ q)) \vee \\ (return(\mathcal{H}^{(ob)},\ p) &\neq (return(\mathcal{H}^{(ob)} \circ q,\ p))\end{aligned}$$

($\circ$ denotes functional composition.) Thus, two operations conflict if their effects on the state of an object are not independent of their execution order (first clause) or their return values are not independent of their execution order (second and third clauses). Operations p and q are said to *conflict* on object *ob*, denoted $conflict(ob, p, q)$ if there exists a history $\mathcal{H}$ such that *conflict*$(\mathcal{H}^{(ob)}, p, q)$. The notion of conflict will be used to define different types of correctness when transactions access data as well as extra data concurrently.

3.3 Atomic and Non-Atomic Transactions

Traditional databases deal predominantly with atomic transactions. Thus, when a transaction commits, all the operations performed by the transaction commit and when a transaction aborts, all the operations performed by the transaction abort.

However, as we saw in the previous section, when transactions modify the extra data items too, we have to take into account the fact that the execution of a transaction may not be atomic over the extended database. While the failure recovery semantics of extra data items accessed by transactions are usually comparable to that of database items (for example, updates to the page space map tables, which are extra data items, are normally logged [9], see Section 5.4.3), the recovery of extra data items in the event of transaction aborts may

[1] $\mathcal{H}^{(ob)} = p_1 \circ p_2 \circ \ldots \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations. Thus, a state s of an object produced by a sequence of operations equals the state produced by applying the history $\mathcal{H}^{(ob)}$ corresponding to the sequence of operations on the object's initial state s_0($s = state(s_0, \mathcal{H}^{(ob)})$). For brevity, we will use $\mathcal{H}^{(ob)}$ to denote the state of an object produced by $\mathcal{H}^{(ob)}$, implicitly assuming initial state s_0.

not be possible. The reason for this is twofold. In some cases, when a transaction aborts, we may require that the operations performed by (or due to) the transaction on the extra data items be persistent (for example, when the log is an extra data item). In other cases, the extra data item may also be modified by entities outside the transaction management system. Thus, it is possible that certain operations on extra data items are committed even when the invoking transaction aborts. This notion is defined formally in [5].

3.4 Serializability

In traditional databases, serializability and, in particular, *conflict serializability*, is the well-accepted criterion for correctness. We first define serializability formally since it forms the basis for the correctness notions discussed here.

Let $\mathcal{D}$ be the set of database items and let $\mathcal{C}$ be the conflict (binary) relation on transactions in $\mathcal{T}$, where $\mathcal{T}$ is the set of all transactions in the history $\mathcal{H}$.

DEFINITION 3.5: $\forall\, t_i, t_j \in \mathcal{T}, t_i \neq t_j,$

$$(t_i \;\; \mathcal{C} \;\; t_j) \text{ if } \exists ob \in \mathcal{D}\; \exists p, q\; (conflict(ob, p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge (commit_{t_i} \in \mathcal{H}) \wedge (commit_{t_j} \in \mathcal{H}))$$

The $\mathcal{C}$ relation captures the fact that two committed transactions have invoked conflicting operations on the same object and the order in which they have invoked the conflicting operations. Consequently, the $\mathcal{C}$ relation captures direct conflicts between committed transactions in a history which affect their serialization order. The fact that a serialization order is acyclic is stated by requiring that there be no cycles in the $\mathcal{C}$ relation. All this is formalized below.

DEFINITION 3.6: $\mathcal{H}$, the history of events relating to transactions in $\mathcal{T}$, is *(conflict) serializable* iff $\forall t \in \mathcal{T}, \neg(t \;\; \mathcal{C}^* \;\; t)$ where $\mathcal{C}^*$ is the transitive-closure of $\mathcal{C}$.

Suppose t_j has done a write on an object and then t_i does a read on the same object. Then, $(t_j \; \mathcal{C} \; t_i)$. Also, if we desire failure atomicity, then if t_j aborts then t_i must also abort. Thus, abort dependencies [2] between transactions may also form due to conflicting operations.

However, when we extend the notion of serializability to the extended database, we have to take into account the fact that the execution of a transaction may not be atomic over the extended database. Thus, even the committed operations invoked by aborted transactions will be involved in determining the serialization ordering.

Let $\mathcal{E}$ be the set of extended database items and let $\mathcal{C}$ be the (binary) conflict relation on transactions in $\mathcal{T}$, where $\mathcal{T}$ is the set of all transactions in the history H.

DEFINITION 3.7: $\forall\, t_i, t_j \in \mathcal{T}, t_i \neq t_j,$

$$(t_i \;\; \mathcal{C} \;\; t_j) \text{ if } \exists ob \in \mathcal{E}\; \exists p, q\; (conflict(ob, p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge (commit[p_{t_i}[ob]] \in \mathcal{H}) \wedge (commit[q_{t_j}[ob]] \in H)).$$

The $\mathcal{C}$ relation, in this case, captures the ordering in which two transactions invoked (or caused to invoke) two conflicting committed operations on the same object. The definition of serializability remains the same, where acyclicity is ensured with respect to this $\mathcal{C}$ relation.

As mentioned earlier, an application may desire correctness properties that are weaker than serializability when an extended database is used. A discussion of such correctness properties can be found in [10].

For the purposes of this paper, given an extended database, we can consider (1) serializability of accesses to data items in the extended database (2) serializability of accesses to data items in the database. That is, in (2), a cycle of $\mathcal{C}$ relationships formed by accesses to the database as well as by accesses to extra data will be ignored since only the set of C relationships induced by the items in the database need be acyclic.

3.5 Extra-data Independent Transactions

The reading and writing of extra data may in principle affect both the operations that transaction T performs while executing as well as its flow of control. In this section, we formalize the notion of when transactions are "dependent" on extra data items. We show that by restricting the set of allowable histories, transactions can be made extra data independent. This restriction on allowable histories is specified by a set of axioms which induce $\mathcal{C}$ relationships between transactions.

Let $\mathcal{K}$ be a set of axioms which specify the conditions under which $\mathcal{C}$ relationships are induced between transactions. For example, the set $\mathcal{K}$ could contain the axiom which specifies when $\mathcal{C}$ relationships are introduced due to conflicts (see section 3.2). In general, the axioms may involve both database and extra data items and hence $\mathcal{C}$ relationships may be induced between transactions due to extra data accesses also.

Definition 3.8: A history $\mathcal{H}$ relating to transactions in $\mathcal{T}$ is said to be *serializable with respect to a set of axioms* $\mathcal{K}$ if $\forall t \in \mathcal{T}, \neg(t\mathcal{C}^*t)$, where $\mathcal{C}$ is the relation induced between transactions in $\mathcal{T}$ due to the axioms in $\mathcal{K}$.

Consider a set $\mathcal{K}$ which contains the axioms that introduce $\mathcal{C}$ relationships between transactions when they perform conflicting operations. Then, any history serializable with respect to the set of axioms $\mathcal{K}$ will also be conflict serializable because the set of $\mathcal{C}$ relationships induced due to the latter is a subset of the $\mathcal{C}$ relationships induced due to the former.

Definition 3.9: A *conventional* transaction is one that does not access extra-data items.

Definition 3.10: A set of transactions $\mathcal{T}$ is said to be *extra-data independent* with respect to a set of axioms $\mathcal{K}$ if for each transaction (program) t in $\mathcal{T}$, there exists a conventional transaction (program) t' (let $\mathcal{T}'$ be the set of all t') such that for all legal histories $\mathcal{H}$ involving instances of some subset of transactions in $\mathcal{T}$ and serializable with respect to the set of axioms $\mathcal{K}$,

1. the transformation performed on the database is the same as that performed by some serial history $\mathcal{H}'$ consisting of instances of transactions in $\mathcal{T}'$.
2. there is a surjection mapping each instance of transaction t in $\mathcal{H}$ to its corresponding t' in $\mathcal{H}'$.

3. the values (if any) returned by each t and its corresponding t' (as determined by the surjective mapping) to the invoker are the same[2].

4. each t' satisfies the integrity constraints on the database (i.e., each t', when run by itself, performs a correct database state transformation).

Observe that if t is extra-data independent then, when applied to a database state s, it yields exactly what t' yields when applied to s (here the history consists of a single transaction). If t is extra-data independent, as far as the serializability of the changes to the database is concerned, we can think of the actual transaction t as a "surrogate" for t'.

In the above definition, we have assumed that the correctness criterion for the conventional transactions (t' s) is serializability. If we desire a weaker (stronger) correctness criterion for the conventional transactions, then the axioms in $\mathcal{K}$ with respect to which extra data independence is defined will also be weakened (strengthened).

Henceforth, we refer to any t' according to the above definition for t, as $equiv(t)$. (In general there may be a number of such t's, for our purposes they are all equivalent.) If a set of transactions $\mathcal{S}$ is not extra data independent with respect to a set of axioms $\mathcal{K}$, then it is said to be *extra-data dependent* with respect to the set of axioms $\mathcal{K}$[3].

Let us consider a simple example to clarify the notion of extra-data independence. Suppose a transaction is invoked to make a single car rental reservation. It chooses among car companies based on the anticipated delay due to waits. The transaction uses extra data as control information to find the path with the shortest waiting time:

```
1.  trans {
2.       car_rental avis = get_oid("avis");
3.       car_rental hertz = get_oid("hertz");
4.       if (num_waiting(avis) > num_waiting(hertz))
5.            hertz->reserve(...);
6.       else
7.            avis->reserve(...);
8   }
```

This is an example of a transaction that is *not* extra-data independent (with respect to conflict serializability over the database): either an Avis car or a Hertz car is reserved.

Consider a transaction invoked on behalf of another user who intends to reserve an Avis car in Los Angeles and a Hertz car in Houston. If we replace lines 5-7 by the following, we accomplish this while attempting to minimize the waiting time. The operations to reserve the Avis car and the Hertz car are assumed to be commutative.

[2] The return value of a transaction t will be a function of the values returned by the operations on objects invoked by t.

[3] In the rest of the paper, when the set $\mathcal{K}$ is sufficiently clear from the context, we will omit the phrase "with respect to to the set of axioms $\mathcal{K}$". Also, for the sake of clarity, when the set $\mathcal{K}$ corresponds to the set of axioms for determining conflicts over the database, the phrase "with respect to the set of axioms $\mathcal{K}$" will be replaced by "with respect to conflict serializability over the database".

```
6.          hertz->reserve(...); avis->reserve(...);
7.      else
8.          avis->reserve(...); hertz->reserve(...);
```

This transaction *is* an extra-data independent transaction (with respect to conflict serializability over the database) which is equivalent to the conventional transaction obtained by replacing lines 4-7 by the following.

```
hertz->reserve(...); avis->reserve(...);
```

Consider a serializable history (with respect to some set of axioms K) where all transactions are extra-data independent. In hypothetically running the original transactions serially, extra data may be different or may not exist, which could lead to a change of control flow and/or values written. But, substituting $equiv(t)$ for each t isolates its effect on the database state, from other facets as embodied in extra data.

Consider a transaction (program) t written in a programming language with expressive power equivalent to that of Turing Machines. It is then undecidable whether an $equiv(t)$ exists. So, we cannot delegate the task of verifying the existence of $equiv(T)$ to an automatic tool. An alternative is to have the user supply $equiv(T)$ and an equivalence proof (with respect to a set of transactions $\mathcal{S}$ and a set of axioms $\mathcal{K}$). This is similar to having the user certify a transaction as doing a "correct" state transformation in the traditional formulation of concurrency control theory.

We now discuss some practical aspects of constructing a set of extra-data independent transactions. Let $\mathcal{D}$ denote the set of database items and $\mathcal{E}$ denote the set of extra data items. Also, let R_t and W_t denote the read and write sets, respectively, of transaction t which, we assume, are determined at compile time. Consider a set $\mathcal{S}$ of transactions already known to be extra data independent with respect to a set of axioms $\mathcal{K}$, where all histories serializable with respect to $\mathcal{K}$ are also conflict serializable over the database (that is, the set of axioms in $\mathcal{K}$ specify a criterion at least as strong as conflict serializability over the database). A transaction t_i can be added to the set $\mathcal{S}$ such that the set $\mathcal{S} \cup \{t_i\}$ is extra data independent with respect to to the set of axioms $\mathcal{K}$, if t_i does not return any value and at least one of the following conditions is true:

- $W_{t_i} \subseteq \mathcal{E} \wedge \forall t \in \mathcal{S}(R_t \cap W_{t_i} = \phi)$. The transaction writes to only the extra data items and thus cannot have any effect on the state of the database (in fact, if this condition is satisfied, the transaction t is extra data independent irrespective of whether the criterion specified by the axioms in $\mathcal{K}$ is stronger or weaker than conflict serializability over the database). In this case, $equiv(t_i)$ is the null transaction.

- $R_{t_i} \subseteq \mathcal{D} \wedge \forall t \in \mathcal{S}(R_t \cap W_{t_i} = \phi)$. The transaction reads the values of only the database items and thus cannot be affected by the values of extra data. In this case, $equiv(t_i)$ will be t_i with all accesses to the extra data removed. Given no further information about the transaction, we require the axioms in $\mathcal{K}$ to specify a criterion which is at least

as strong as serializability over the database since $equiv(t_i)$ should be substitutable for t_i in some serial order (see definition 3.10).

There are cases when a transaction t_i can be added to $\mathcal{S}$ (with the same restriction on $\mathcal{K}$) while preserving extra data independence even though it violates both of the above conditions. For example, if transaction t_i is written in an imperative programming language, then using data flow analysis, we can verify that extra-data information does not propagate into the database. Then, the only effect extra data may have on t_i is manifested through flow of control. If we can verify that t_i has the same effect on the database regardless of the flow of control, then $\mathcal{S} \cup \{t_i\}$ is extra data independent with respect to the set of axioms $\mathcal{K}$. This, for example, is the case with the second car rental example.

4 Correctness Notions

It should be obvious by now that different types of correctness notions can be applied in the context of extra data. In this section, we discuss several such correctness notions. The next section illustrates them with concrete examples of extra data.

1. Ensure extra data independence of transactions. Given our definitions in Section 3.5, this implies that we should define an appropriate set of axioms $\mathcal{K}$ with respect to which transactions are extra data independent. Further, we have to ensure that the C relationships induced between transactions due to the axioms in $\mathcal{K}$ form an acyclic relation. This correctness criterion assures view serializability[1] (though not necessarily conflict serializability) of the conventional transactions which are equivalent to the transactions which access extra data. This is because, as per definition 3.10, we only require that the net transformations performed on the database by transactions which access extra data is the same as the net transformations performed by some serial execution of the conventional transactions.

2. Achieve serializability of accesses to the extended database in the presence of extra data dependent transactions. We need to ensure that the C relationships due to conflicts over accesses to the extended database is acyclic. Since transactions need not be extra data independent, transactions may produce results that reflect the fact that they accessed extra data.

3. Achieve serializability of accesses to (just) the database in the presence of extra data dependent transactions. Changes to the database may depend on the transactions' accesses to extra data. Intuitively, by serializability with respect to just the database data, we mean that if transactions are run serially and whenever an extra-data access is performed in this serial execution the same values as in the actual execution are "magically" supplied, then transactions will produce the same final database state as in the actual execution. Since the extra-data conflicts are not considered in determining which transactions conflict, transactions may produce database changes that are affected by extra-data accesses. Also, this correctness criterion *does not* guarantee consistency of the extended database (assuming consistent transactions) since the extra data items can be accessed in a non-serializable fashion.

4. Achieve some application specific correctness criterion in the presence of extra data dependent transactions. Some applications may desire correctness criteria in which extra data items must be brought within the scope of concurrency control but in which requiring serializability over the extended database would result in a loss of performance (through loss of concurrency). In this case, application dependent correctness criteria which are weaker than serializability over the extended database are specified and ensured.

5 Examples of Extra Data and Associated Correctness

In this section we discuss in detail each of the correctness notions introduced in the last section, providing concrete examples of extra data for which these correctness notions are applicable.

5.1 Extra Data Independent Transactions

In this section, we discuss four examples in which we require the extra data independence of transactions.

5.1.1 Work Queues

Consider a transaction t' which performs operations op_1, op_2, and op_3, in sequence, on each data item in a set D. The transactions may access database items not in D, indicated by '...' in the code below.

```
trans t' {
    for all d ∈ D {op1(d); ... }
    for all d ∈ D {op2(d); ... }
    for all d ∈ D {op3(d); ... }
}
```

We now show how t' can be implemented as a transaction t which uses extra-data items in such a way that it has the potential for better performance in a parallel environment. That is, t' will be $equiv(t)$.

We proceed to construct t as follows: t is a nested transaction which consists of three (sub)transactions t_1, t_2 and t_3. Each t_i performs the equivalent of the ith for loop in the text for t'. t_1, t_2 and t_3 operate in a pipelined fashion with respect to the data items in D, i.e., after t_1 does op_1 on a data item d, t_2 does op_2 on d and then t_3 does op_3 on d.

Proper control of the transactions' actions is achieved via work queues q_1 and q_2, which are extra-data items of category 4 (see figure 1). q_1 has two operations defined on it: *insert*1 and *remove*1. Similarly for q_2. A *remove* operation on a queue blocks if the queue is empty. t_1 inserts the id of the data item on which it just completed op_1 into q_1. t_2 removes the id in the front of queue q_1 and after doing op_2, it inserts the id into q_2. t_3 removes an id from q_2 and performs op_3 on the corresponding data item. When q_1 and q_2 are empty and t_1, t_2, and t_3 have completed their ongoing operations, they all commit together. If any of them aborts, all abort. The programs for the (sub)transactions are given below.

```
trans t1 {                trans t2 {                  trans t3 {
    for all d ∈ D {           P = D;                      P = D;
        op1(d);               while P ≠ φ {               while P ≠ φ {
        insert1(d);           remove1(d);                 remove2(d);
        ...                   op2(d);                     op3(d);
    }                         insert2(d);                 P = P − {d};
}                             P = P − {d};                ...
                              ...                         }
                              }                       }
                          }
```

Since each *remove* operation blocks if the queue on which the operation is performed is empty, we have

$\forall d \in D(insert1(d) \rightarrow remove1(d))$

$\forall d \in D(insert2(d) \rightarrow remove2(d))$.

Since we want the operations op_1, op_2 and op_3 to be performed in sequence on an object in D, we define the following $\mathcal{C}$ relationships.

$(t_1\ \mathcal{C}\ t_2)$ if $\exists d(insert1_{t_1}(d) \rightarrow remove1_{t_2}(d))$.

$(t_2\ \mathcal{C}\ t_3)$ if $\exists d(insert2_{t_2}(d) \rightarrow remove2_{t_3}(d))$.

The above axioms along with the axiom for conflicts over the database items (see 3.2) make up the set $\mathcal{K}$ with respect to which extra data independence is achieved. Thus transaction t, consisting of t_1, t_2, and t_3, has the same effect as t' on D.

What this example shows is that it is possible to realize a given transaction in such a way that even though the implementation uses extra-data items, its behavior with respect to the rest of the transactions and the effect on the database will be the same as the original transaction. The motivation here is to rewrite the transactions in a way that allows us to execute components of the transactions in parallel thereby improving the performance of the system. The queues allow the transaction components to synchronize their activities so as to achieve the desired functionality.

5.1.2 Serializability in Multi-databases

Another example of extra-data independent transactions occurs in the multi-database concurrency control scheme proposed in [3]. Here, to ensure the serializability of transactions that access multiple (autonomous) database sites, the following scheme is used. Every site has a special "ticket" that all global transactions that visit the database at that site are expected to read and write. In this case, the ticket is the extra-data item since only transactions visiting that site can access it. Here we require the extra data independence of transactions because the execution of the transactions accessing the ticket should correspond to some serial order of transactions not accessing the ticket. This is achieved by defining a set of axioms $\mathcal{K}$ which achieve conflict serializability over the extended database.

5.1.3 A Car Reservation Example

Consider a modified version of the car rental transaction from Section 3.5. A transaction t_i first reads the data items to determine the number of free Hertz cars and Avis cars respectively and reserves one Hertz car and one Avis car in different orders depending on which has a larger number of free cars.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    if (num_available(avis) > num_available(hertz))
        hertz->reserve(...);
        avis->reserve(...);
    else
        avis->reserve(...);
        hertz->reserve(...);
}
```

In this case, it is not necessary to have an exact number of the number of cars available and an approximate value would do. Thus, to improve the performance of the above transaction, we could have a transaction t_k which periodically reads the number of cars available and writes them to extra data items *num_avis* and *num_hertz*. Then transaction t_i could be written as a transaction t_j which reads from the extra data items rather than accessing the corresponding database item, thus reducing conflicts. The transactions t_j and t_k are extra data independent (with respect to serializability over the database) in which $equiv(t_j)$ is t_i and $equiv(t_k)$ is a null transaction. Thus, the functionality of the transactions is not changed and hence the consistency of the database is ensured.

This example illustrates how the use of the concept of extra data independence can be used to allow lower degrees of isolation in concurrency control without affecting the consistency of the database or the functionality of transactions in any way. If we feel that a transaction t_1 does not require exact values of a data item X, then we could prove that it is indeed not necessary as follows: Create a hypothetical transaction t_3 which periodically accesses the value of X and writes it into an extra data item x. Now t_1 could be modified to read the value x instead of X (call this modified transaction t_2). Now t_1 does not require exact values of the data item X iff t_2 and t_3 are extra data independent (with respect to serializability over the database).

5.1.4 Proclamation Locking

Consider the following example in which 2PL is used for concurrency control. Suppose a transaction t' holds a write lock on a data item and another transaction t desires to read that data item. Under 2PL, t would be made to wait. However, suppose t' gives t (or any other transaction) an indication of what the possible values it might write are, then t might be able to proceed with its computations using this information, thus increasing the degree of concurrency. This is the idea underlying proclamations [7]. t' proclaims the set of possible values that may be written so that transactions such as t may be able to proceed without waiting. These proclamation data items can be treated as extra data items with the required

correctness criterion being the extra data independence of transactions. Further details of this example appear in [5].

5.2 Serializable Accesses to the Extended Database

Here we remove the extra-data independence requirement imposed on transactions, but require that the transactions be serializable with respect to the extended database. For concreteness, we consider the database log as an example of extra data which can be accessed by transactions in the course of their execution [4]. Both committing and aborting transactions write log records. Transactions can also read the log to perform queries. We require serializability of all the data items in the extended database, in this case, the database plus the log.

Let us assume that the commitment or abortion[4] of a transaction results in the writing of a single log item[5] containing all the relevant information about the transaction. The operation of writing this log item is committed irrespective of whether the invoking transaction commits or aborts. Conceptually, the log can be considered as a linear object that grows in one direction. Each item in the log has an id (its LSN; i.e., log sequence number).

$append_{t_i}(k)$ denotes the appending of a log item pertaining to transaction t_i; when the operation completes, the id of the appended log item is returned in k. The value of k is one larger than the id of the previously appended log item.

$read_{t_i}(k)$ denotes the read operation on log item k by transaction t_i; this item should already exist in the log for the read to be successful. Otherwise, the read fails and the transaction which invoked the read aborts. k is given as an input to the operation *read*.

$last(k)$ can be used to determine the id of the last item in the log. This id will be returned in k when *last* completes.

Here are the correctness requirements imposed on read and last operations:

- $read_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j \ (append_{t_i}(k) \rightarrow read_{t_j}(k))$

  This states that t_j reads log record k after the write of record k by some t_i.

- $last_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j \ (append_{t_i}(k) \rightarrow last_{t_j}(k) \wedge \nexists t_m, l (append_{t_i}(k) \rightarrow append_{t_m}(l) \wedge append_{t_m}(l) \rightarrow last_{t_j}(k)))$

  This states that if the *last* operation executed by t_j returns k, then the kth record should have been written by some transaction t_i and no other transaction has written since.

Since we require conflict serializability over accesses to the log items, C relationships are induced due to conflicting operations on the log. Specifically, $\forall \ t_i, t_j \ t_i \neq t_j \ \ (t_i \ C \ t_j)$ if:

---

[4]Transactions can abort for one of many reasons, including unilateral aborts.

[5]In intention list based transaction processing systems, a single log record is usually written for each transaction. The more general case in which many log records may be written for a single transaction is treated in section 5.4.1.

1. $\exists\ k, k'\ (append_{t_i}(k) \rightarrow append_{t_j}(k'))$

   Since both aborting transactions and committing transactions write to the log, transactions append to the log in the same order in which they commit or abort. Also, every transaction, once it begins execution, will commit or abort. Thus, when a transaction t_i appends an entry into the log, a transaction t_j that has not yet committed or aborted, i.e., t_j is a transaction in progress, will write its log entry after t_i's entry. Since t_j updates the log after t_i updates it, $(t_i\ \mathcal{C}\ t_j)$, that is, t_j must appear after t_i in the serialization order.

2. $\exists\ k \exists i \geq 0\ (append_{t_i}(k) \rightarrow read_{t_j}(k+i))$

   This states that if t_j reads log record $k+i, i \geq 0$, then t_j should occur later in the serialization order than the transaction t_j which caused the log record k to be written.

3. $\exists\ k\ (append_{t_i}(k) \rightarrow last_{t_j}(k))$

   This states that if t_j, via the *last* operation, "knows" that the last log record to be written was the kth record (written by transaction t_i) then transaction t_i should precede t_j in the serialization order.

4. $\exists k, k'\ (last_{t_i}(k) \rightarrow append_{t_j}(k'))$

   A transaction t_j that has not yet committed or aborted when a transaction t_i performs the *last* operation will write its log entry after t_i executes *last*. That is, t_i observes the queue before t_j updates it and hence t_j will have to follow t_i in the $\mathcal{C}$ ordering.

Since we require serializability over the extended database, we need to consider the $\mathcal{C}$ relationships that result not only from the invocation of the operations on the database objects but also from the invocation of operations on the log. The modifications that a transaction t performs on the log (through the *append* operations) are persistent irrespective of whether t commits or aborts (thus the *append* operation belongs to the set $\mathcal{P}$ as defined in section 3.3). Serializability is achieved by ensuring that the $\mathcal{C}$ relation defined on committed *operations* is acyclic. This is the safety-related correctness property that must be satisfied usually, and is still the case when accesses to the log are considered. The practical implication of the above axioms on transaction management is the following: When an operation is performed on the log, the system must note the $\mathcal{C}$ relationships induced by the operation, in light of the above axioms, and must ensure that the $\mathcal{C}$ relation is acyclic.

Note that in the case of the log, as long as the history resulting from concurrent transaction executions is serializable, correctness is considered to be preserved. The log is viewed like any other data item and is considered as being part of the database.

Traditionally, if there are cycles in the $\mathcal{C}$ relation, not all transactions involved in the cycle can commit, i.e., cycles are broken by transaction aborts. But with transactions being able to access the log, we must worry about the liveness of the transactions because aborting and committing transactions that access the log form $\mathcal{C}$ relationships which may create

cycles which cannot be broken. The proof of the fact that the liveness of transactions is indeed ensured in this case appears in [5].

When other extra-data objects are accessed by transactions and serializability remains the correctness criterion, the semantics of these other objects should be specified just as we dealt with the log and then we must show that safety and liveness properties of transactions are kept intact.

It is important to realize that when transactions are allowed access to the log and serializability remains the correctness criterion, it may delay the commitment or abortion of other transactions. Specifically, because of axiom 4, once a transaction t_i performs *last*, no other transaction can commit or abort until t_i commits. Such a delay can affect performance. However, if one were to relax the correctness criterion, say by requiring something akin to the lower degrees of isolation of traditional databases, then this negative impact can be reduced or eliminated. Specifically, one could opt for non-repeatable reads of the log. We return to this issue in Section 5.4.

5.3 Serializable Accesses to the Database

In this section, we give some examples of cases where we require serializability over only the database in the presence of extra data dependent transactions.

Consider a modified version of the first car rental transaction from Section 3.5. Here, the available credit is updated where the amount depends on the car rental company chosen.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    if (num_waiting(avis) > num_waiting(hertz))
        hertz->reserve(...);
        master_card->credit_reserve(100);
    else
        avis->reserve(...);
        master_card->credit_reserve(75);
}
```

The application writer may not care from which company the car is rented as long as exactly one car is rented (even though the credit reservation amounts, 75 or 100, depend on the extra data). Thus, in this case, we do not require extra data independence of transactions and are only interested in the serializable execution (with respect to database items) of transactions. Hence, we only need to consider the C relationships induced due to conflicts on database items and achieve acyclicity over this relation. This ensures that the database is always in a consistent state as far as applications are concerned.

For another example, consider two transactions t_i and t_j that access disjoint parts of the database but what one transaction accesses is dependent on what the other accesses. The transactions communicate via communication channels that can be modeled as extra-data items which are read and written by the transactions. That is, t_i writes into the channel the ids of the data items it has accessed and t_j reads these, and vice versa. Here, we do not

require the transactions to be extra data independent (which they are not) but just require serializability over the database. There is a subtle difference here from serializability in that *if* we look at the resulting serial schedule and rerun the programs corresponding to t_i and t_j we will not necessarily get the same overall state changes because of accesses to the extra data and the extra-data dependence of t_i and t_j. However, if we rerun t_i and t_j so that the values returned by the operations on the extra data items are the same as when they originally ran, then the rerun will produce the same database state as the original schedule.

5.4 Application Specific Correctness Notions

Thus far, we have worked with serializability as the basic correctness criterion for transactions accessing extra data. But, as we alluded to at several places, it might be appropriate to relax serializability, as has been suggested even for transactions accessing just the database. The added motivation for this in the context of extra-data access is that access to extra-data items, such as the log, which lie in the processing path of every transaction must be allowed with minimal or no impact on performance. Since relaxing correctness requirements is one way to achieve this, serious consideration must be given to it.

Let us now consider some weakened isolation requirements [6] that have been suggested and adopted in practice for transactions accessing the database. In the context of read/write objects, degree-2 isolation ignores conflicts resulting from a read followed by a write. Such a requirement leads to lack of repeatable reads. Degree-1 isolation ignores, in addition, conflicts resulting from a write followed by a read. This permits the read of an object, writes on which have not yet committed, without forming a C relationship between the writing and the reading transaction. Degree-0 isolation ignores all dependencies. Let us consider some examples applying these ideas to extra data.

5.4.1 The Log - Multiple Entries per Transaction

In normal practice, the log is implemented in such a way that the operations performed by transactions are logged separately and so there may exist multiple log records for a single transaction. Let us consider such a log which can be accessed by transaction through the following operations:

$append(Info)$ writes the information in $Info$ as the next log record.

$read_first(t_j, Info)$ returns the first log record written by t_j in $Info$.

$read_next(t_j, Info)$ returns the next log record (i.e., the log record after the log record read by the previous $read_next(t_j, _)$ or $read_first(t_j, _)$ operation) written by t_j.

$read_begin(t_j, Info)$ returns the first log record in the log and the transaction t_j which wrote the log record.

$read_succ(t_j, Info)$ returns the next log record (i.e., the log record immediately after the log record read by the previous *read_begin* or *read_succ* operation) in the log and the transaction t_j which wrote the log record.

A *read* operation fails if there is no log record that can be returned.

With the operations on the log defined as above, if we were to require serializability, then

it would have a negative impact on the degree of concurrency allowed since the operations performed by different transactions have to be logged without any interleaving. In this context, it would seem that an acceptable correctness criterion would be that all transactions which read a log record written by a transaction t should be serialized after t. This condition is formally specified below.

In a history $\mathcal{H}$, $(t_i \; \mathcal{C} \; t_j)$, $i \neq j$, if

- $\exists Info(read_first_{t_j}(t_i, Info) \in \mathcal{H})$
- $\exists Info(read_begin_{t_j}(t_i, Info) \in \mathcal{H})$
- $\exists Info(read_succ_{t_j}(t_i, Info) \in \mathcal{H})$

Besides the $\mathcal{C}$ relationships induced by the above axioms, the $\mathcal{C}$ relationships induced due to conflicts on the database items must also be taken into account. We require the acyclicity of the relation involving these $\mathcal{C}$ relationships on the committed operations as our correctness criterion. Liveness of transactions with this correctness criterion can be shown [5].

5.4.2 The System Clock

Suppose transaction t_i accesses the system maintained current time. A subsequent update of the current time by the system clock will not affect t_i if degree-2 isolation is in effect. More generally, suppose we want transactions to possess the temporal causality property which requires that if two transactions read the clock in a certain order with an intervening system update to the clock, their serialization should reflect the order of the reads. We are also not concerned about the extra data dependence of transactions. In this case, the correctness criterion can be formalized in terms of $\mathcal{C}$ relationships as follows:

$t_i \mathcal{C} t_j$, $i \neq j$, if $(read_{t_i} \rightarrow write_{sys}) \wedge (write_{sys} \rightarrow read_{t_j})$

Here the reads and writes refer to the operations on the clock, where the clock is updated by the system, a non-transactional entity. In addition to these $\mathcal{C}$ relationships, those arising due to conflict on the database items are also considered and acyclicity of this $\mathcal{C}$ relation is ensured over all committed operations. It is interesting to note that this correctness criterion rectifies the anomaly found in section 2 dealing with Joe's status. However, this criterion is weaker than serializability in the sense that we allow for non-repeatable reads to the clock.

Another example in which clocks are accessed as extra data items occurs in the context of real-time databases [11]. In this case, transactions have a correctness requirement which is stronger than traditional serializability in the sense that we have the additional requirement that if a transaction t_i commits, it should commit before its deadline, *deadline*(t_i). This requirement is formally stated below:

$$commit_{t_i} \in \mathcal{H} \Rightarrow (commit_{t_i} \in \mathcal{H}^{deadline(t_i)})$$

where $\mathcal{H}^t$ is the prefix of the history $\mathcal{H}$ until time t. In order to satisfy this requirement, the transaction management system accesses the clock (an extra data item) just before the transaction wants to commit. In case its deadline has been met, the transaction commits, otherwise it aborts. As per our definition in section 3.5, such transactions are extra data

independent, where $equiv(t)$ of a real-time transaction t is a transaction which attempts to commit regardless of whether its deadline has been met or not. Thus, accessing the clock in the case of real-time transactions does not violate the consistency of the database and hence the clock need not be brought within the scope of concurrency control.

Also, in real-time databases, if we know that a transaction is highly unlikely to meet its deadline it is better to abort the transaction early. This can be achieved by examining the wait-for-graph. Under degree-1 isolation, a transaction will be allowed to access the wait-for-graph without forming any additional C relationships.

5.4.3 Page Space Map Tables

Page space map tables [9] are commonly used in databases to indicate the location of free space in pages. These tables are normally accessed by transactions, though they are not considered as database items and brought within the scope of correctness and concurrency control. Here, we identify the correctness criterion relevant to this extra data item and the impact this criterion has on concurrency control.

The following operations are defined on page space map tables for freeing and reserving space in pages: *reserve(page,begin_offset,end_offset)* reserves the area of the page *page* from *begin_offset* to *end_offset*. *free(page,begin_offset,end_offset)* frees the area of the page *page* from *begin_offset* to *end_offset*.

Extra data independence of transactions accessing the page space map tables is not required since the state of the database is not dependent on the physical location of the database items. However, in order to provide for transaction roll back (in case of transaction abort), we may need to specify that a transaction can reclaim the space it freed (in the event of it aborting) so that its changes can be undone easily [9]. Thus, we require that if a transaction t_i uses up the space freed by transaction t_j, then t_i should abort whenever t_j aborts (so that the space freed by t_j is now free). Thus t_i has an abort dependency on t_j.

$$t_i \; \mathcal{AD} \; t_j, i \neq j, \text{ if } \exists page, eo1, bo1, eo2, bo2(free(page, bo1, eo1) \rightarrow reserve(page, bo2, eo2)) \wedge (eo1 \geq bo2) \wedge (bo1 \leq eo2)$$

We note that there are no C relationships induced due to accesses to the extra data item (since we do not require extra data independence) but instead, an abort dependency is induced. In normal implementations of the page space map tables, a transaction t_j can use the space freed by a transaction t_i only if t_i has committed [9]. Thus, this induced abort dependency is taken care of and only serializability over accesses to the database items need be ensured.

6 Discussion

Recently, there have been many extensions to the classical work on concurrency control. One extends and elaborates the structure of data items, viewing them as abstract data type objects, thus exploiting the semantics of the operations for better concurrency control. Another relaxes the serializability correctness criterion by imposing instead specific constraints on acceptable schedules [8]. The work reported herein bears resemblance to these extensions

in that, technically, we also view extra data as objects with arbitrary operations defined on them and impose some restrictions on acceptable schedules. However, these are by-products of our main interest, that of enlarging the set of transaction accessible data to include structures that are traditionally either (a) hidden within, and are internal to, the database system itself or (b) local to a set of transactions. We have examined the consequences of such accesses and in doing so we are forced to use extensions to the traditional concurrency control setting. Specifically, our goal in this paper was two-fold:

- To illustrate via detailed examples that allowing transactions to access extra data not only improves the functionality of transactions but also has many performance benefits.

  We considered extra-data items that occur in typical database systems such as the log, the clock, and the concurrency control information. We also showed that other types of extra data, such as proclaimed values, and queues used for coordinating pipelined and cooperating transactions prove very useful for structuring transactions in order to improve performance.

- To investigate, in detail, the correctness issues that arise when transactions are allowed access to extra data.

  We saw that while traditional serializability can continue to be the mainstay of correctness, one needs to also consider extra-data independence of transactions. To precisely capture the interactions due to extra-data access, we axiomatized the operations on extra-data items to determine the serialization ordering requirements induced by extra-data access. These helped characterize the different types of correctness issues that must be considered. In this regard, we studied a variety of correctness notions.

The techniques presented in this paper are useful both for the transaction programmer as well as the database system implementor. The transaction programmer is concerned with correctness issues which arise when transactions access data items which are not part of the database (for example, the system clock or the log) and the concurrency control mechanisms required to achieve them. The database systems implementor, on the other hand, is concerned with the issues which arise when extra data is used to improve the performance of transactions (for example, proclamation data items or work queues) or when transactions access extra data items (for example, page space map tables) which are hidden at the transaction programmer level.

Allowing transactions to access extra-data improves transaction functionality. On the other hand, as we discussed at several points in the paper, performance consequences can be either positive or negative depending on the properties of the data. One implication is that extra-data access must be allowed only if the consequences are not detrimental to performance, and if they are and yet extra data must be accessed, one must apply the least restrictive correctness criterion that fits the needs.

Some of the practical implications for the mechanisms used for transaction processing remain to be investigated. For instance, the transaction processing system must keep track of

C relationships that are induced when transactions access not just the database but also the extra data. Also, *it* must ensure that operations are scheduled in such a way that acyclicity of C relations is ensured.

References

[1] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proceedings of the seventeenth International Conference on Very Large Databases*, pages 103–112, 1991.

[3] D. Georgakopoulos, M. Rusinkiewicz and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the IEEE Seventh International Conference on Data Engineering*, 1991.

[4] N. Gehani and O. Shmueli. The LOG as Part of the Database. *Bell Laboratories Technical Memorandum*, 1992.

[5] N. Gehani, K. Ramamritham, J. Shanmugasundaram and O. Shmueli. Accessing Extra Database Information: Concurrency Control and Correctness. Technical Report 1996-016, University of Massachusetts, Amherst, Massachusetts, 1996.

[6] J.N. Gray and A. Reuter. Transaction Processing: Techniques and Concepts. Morgan-Kaufman, 1992.

[7] H.V. Jagadish and O. Shmueli. A Proclamation-Based Model for Cooperating Transactions. In *Proceedings of the eighteenth International Conference on Very Large Databases*, pages 265–276, 1992.

[8] H. F. Korth and G. Speegle. Formal Models of Correctness without Serializability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–386, 1988.

[9] C. Mohan and D. Haderle. Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking. In *Proc. 4th International Conference on Extending Database Technology*, pages 1 – 13, March 1994.

[10] Ramamritham, K. and Chrysanthis, P. K. "A Taxonomy of Correctness Criteria in Database Applications", *VLDB (Very Large Data Bases) Journal*, Vol. 5, No. 1, Jan, 1996, pp. 85-97.

[11] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, Vol. 1, No 2, 1993, pp. 199- 226.

[12] M. R. Stonebraker. Hypothetical Data Bases as Views. *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp 224-229, May 1981.