[27] C.H. Papadimitriou.
Serializability of concurrent updates.
Journal of ACM, 26(4):631–653, October 1979.

[28] C. Pu and A. Leff.
Replica control in distributed systems: An asynchronous approach.
Technical Report CUCS-053-90, Department of Computer Science, Columbia University, December 1990.
Submitted to SIGMOD.

[29] Calton Pu.
Superdatabases for composition of heterogeneous databases.
In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 150–157. IEEE Press, 1989.
Also appeared in Proceedings of Fourth International Conference on Data Engineering, 1988, Los Angeles.

[30] K.V.S. Ramarao.
Commitment in a partitioned distributed database.
In *Proceedings of 1988 ACM SIGMOD Conference on Management of Data*, pages 371–378, May 1988.

[31] A. Sheth and P. Krishnamurthy.
Redundant data management in Bellcore and BCC databases.
Technical Report TM-STS-015011/1, Bell Communications Research, December 1989.

[32] A. Sheth and M. Rusinkiewicz.
Management of interdepenent data: Specifying dependency and consistency requirements.
In *Proceedings of the Workshop on Management of Replicated Data*, pages 133–136, Houston, November 1990.

[33] G. Wiederhold and X. Qian.
Modeling asynchrony in distributed databases.
In *Proceedings of the Third International Conference on Data Engineering*, pages 246–250, February 1987.

[34] G. Wiederhold and X. Qian.
Consistency control of replicated data in federated databases.
In *Proceedings of the Workshop on Management of Replicated Data*, pages 130–132, Houston, November 1990.

Persistent transmission and unilateral commit.
In *Proceedings of Workshop on Muldatabases and Semantic Interoperability*, pages 48–52, November 1990.

[17] M. Hsu and A. Silberschatz.
Unilateral commit: A new paradigm for reliable distributed transaction processing.
In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, February 1990.

[18] S. Jajodia and C. Meadows.
Mutual consistency in decentralized distributed systems.
In *Proceedings of the Third International Conference on Data Engineering*, pages 396–404, February 1987.

[19] D.R. Jefferson.
Virtual time.
ACM Transactions on Programming Languages and Systems, 7(3):404–425, July 1985.

[20] Boris Kogan and Hector Garcia-Molina.
Update propagation in Bakunin data networks.
In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 13–26. ACM, August 1987.

[21] H. Korth, E. Levy, and A. Silberschatz.
Compensation transactions: A new recovery paradigm.
In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[22] H. Korth, E. Levy, and A. Silberschatz.
Optimistic commit in multidatabase systems.
In *Proceedings of Workshop on Muldatabases and Semantic Interoperability*, pages 65–69, November 1990.

[23] H.F. Korth and G.D. Speegle.
Formal model of correctness without serializability.
In *Proceedings of 1988 ACM SIGMOD Conference on Management of Data*, pages 379–386, May 1988.

[24] C. Mohan, D. Haderle, B. Lindsay, H. Piradesh, and P. Schwarz.
ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.
ACM Transactions on Database Systems, 1991.
To Appear. Also IBM Report RJ 6649, Revised 11/90.

[25] J.E.B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing, volume 1 of *Information Systems.*
The MIT Press, 1985.
Revised version of his Ph.D. thesis, 1981, MIT.

[26] D.C. Oppen and Y.K. Dalal.
The Clearinghouse: A decentralized agent for locating named objects in a distributed environment.
ACM Transactions on Office Information Systems, 1(3), July 1983.

Addison-Wesley Publishing Company, first edition, 1987.

[5] P.A. Bernstein, M. Hsu, and B. Mann.
Implementing recoverable requests using queues.
In *Proceedings of 1990 SIGMOD International Conference on Management of Data*, pages 112–122, May 1990.

[6] K. Birman and T. Joseph.
Exploiting virtual synchrony in distributed systems.
In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 123–138. ACM/SIGOPS, November 1987.

[7] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder.
Grapevine: An exercise in distributed computing.
Communications of ACM, 25(4):260–274, April 1982.

[8] S.B. Davidson, H. Garcia-Molina, and D. Skeen.
Consistency in a partitioned network.
ACM Computing Surveys, 17(3):341–370, September 1985.

[9] W. Du and A. Elmagarmid.
Quasi serializability: a correctness criterion for global concurrency control in InterBase.
In *Proceedings of the International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[10] A. Elmagarmid and W. Du.
A paradigm for concurrency control in heterogeneous distributed database systems.
In *Proceedings of the Sixth International Conference on Data Engineering*, pages 37–46, Los Angeles, February 1990.

[11] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem.
Coordinating multi-transation activities.
Technical Report CS-TR-247-90, Department of Computer Science, Princeton University, February 1990.

[12] H. Garcia-Molina and K. Salem.
Sagas.
In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.

[13] H. Garcia-Molina and G. Wiederhold.
Read-only transactions in a distributed database.
ACM Transactions on Database Systems, 7(2):209–234, June 1982.

[14] D. Georgakopoulos and M. Rusinkiewicz.
On serializability of multidatabase transactions through forced local conflitcs.
In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[15] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger.
Graunlarity of locks and degrees of consistency in a shared data base.
In *Proceedings of the IFIP Working Conference on Modeling of Data Base Management Systems*, pages 1–29, 1979.

[16] M. Hsu and A. Silberschatz.

Because ESR is defined in terms of logs, its properties can be precisely analyzed. Also, asynchronous operations facilitate autonomy and can improve both performance and availability. At the same time, ESR is "compatible" with standard SR notions of correctness. That is, ESR maintains "strong" consistency in several senses. Objects are guaranteed eventual (SR) consistency; by tuning the divergence control method, inconsistency can be controlled, and reduces to SR in the limit. As result, transactions operating under SR are not affected by epsilon-transactions (ETs) running under ESR.

Because of these properties ESR is a useful correctness criterion for important practical applications. Examples include replicated distributed systems, federated databases, and airline reservations. Such systems are characterized by a need for asynchronous query and update propagation (for autonomy, performance and availability) together with a need for guaranteed eventual consistency. Another application of ESR is in time-critical systems, since ESR offers immediate (even if potentially inconsistent) data access to priority users. In comparison, existing asynchronous systems such as versions and quasi-copies offer only older data. Finally, ESR's ability to tune the degree of temporary inconsistency desired (possibly down to zero) is useful in a large distributed system with varied needs in the trade-offs between consistency and availability.

We also show that ESR properties can be realized with concrete and low overhead divergence control methods: moreover, these methods do not require explicit reference to ESR criteria. One advantage of these methods is that, although they are semantics-based they are not application-specific. As a result, ETs can use them without reference to the specifics of the divergence control method. ETs thus serve as a high-level interface (analogous to transactions) to systems maintaining ESR. Four such methods are described and analyzed. Use of these methods is not for "free". They work by placing restrictions on the set of possible concurrent executions (but offer more concurrency than SR). The methods described in this paper implement the common specification, which limits the *amount* of consistency and ignores the *kind*. ESR, however, allows other user specifications of inconsistency, including some that are dependent on semantics.

References

[1] D. Agrawal and S. Sengupta.
Modular synchronization in multiversion databases: Version control and concurrency control.
In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, 1989. ACM/SIGMOD.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina.
Data caching issues in an information retrieval systems.
ACM Transactions on Database Systems, 15(3):359–384, September 1990.

[3] D. Barbara and H. Garcia-Molina.
The case for controlled inconsistency in replicated data.
In *Proceedings of the Workshop on Management of Replicated Data*, pages 35–42, Houston, November 1990.

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman.
Concurrency Control and Recovery in Database Systems.

inconsistent updates. ETs can be used to implement each of these classes. While immediate updates are done within standard transactions (ETs with no divergence), deferred updates correspond to ETs with deadlines. Similarly, independent updates correspond to ETs applied periodically, and potentially inconsistent updates to ETs with backward DC. Wiederhold and Qian proposed a temporal constraint resolver [34] to implement transaction processing satisfying these specifications. A shortcoming of this approach is the considerable overhead to test satisfiability of constraints. In contrast, DC methods offer more efficient execution at less concurrency.

Sheth and Rusinkiewicz [32] have proposed a taxonomy for interdependent data management. They separate data consistency criteria into temporal and spatial dimensions. The temporal consistency has two kinds. Eventual consistency refers to the temporal constraints specified by identity connections. Lagging consistency refers to asynchronous updated copies, in the same style of quasi-copies [2]. The spatial consistency criteria are divided into three cases. Inconsistency is controlled by limiting either (1) the number of data items changed asynchronously, (2) the data value changed asynchronously, or (3) the number of allowed asynchronous operations. An implementation of interdependent data management is described in [31]: it essentially corresponds to ORDUP. In this paper, we have described the ETs and divergence control in terms of limiting the number of asynchronous operations. In order to implement the other spatial consistency criteria, divergence control methods can be extended to include those factors.

Another specification approach is Controlled Inconsistency proposed by Barbara and Garcia-Molina [3], which generalizes their work on quasi-copies. Controlled Inconsistency specifies arithmetic consistency constraints, similar to the data value limit in interdependent data management. They constrain updates to be *safe* operations, defined by some semantic correctness criteria appropriate to the application. In contrast, ESR and divergence control methods discussed in this paper are application independent.

6.4 Distributed Systems Models

There have been some recent work on asynchronous transaction processing preserving SR. One example is Unilateral Termination Protocols (UTPs) by Ramarao [30]. However, despite a careful definition of the properties of UTPs, the author did not suggest concrete UTPs. Their focus was on the Cooperative Termination Protocols such as two-phase commit. A second example is Hsu and Tam's Unilateral Commit Protocol [17], which describes some situations where sub-transactions may commit unilaterally, making a commit agreement protocol unnecessary. This is similar to forward DC methods. A third example is Korth and Speegle's work on compensation-based recovery, also called Optimistic Commit [21], which uses compensation transactions to rollback if the global transaction outcome does not coincide with the local decision. This is similar to backward DC methods. ESR is similar to these works in the asynchronous nature of DC, but differs in allowing bounded inconsistency in ETs.

7 Conclusion

We introduce *epsilon-serializability* (ESR), a correctness criterion that regulates inconsistent accesses to a database. In order to allow temporary and bounded inconsistency, ESR permits asynchronous operations, but ensures that data will eventually converge to consistency.

6.2 General Weak Consistency Criteria

ESR allows temporary and bounded inconsistency. The investigation of correctness criteria weaker than SR (typically allowing some particular kind of inconsistency) is not new. In this section, we omit the semantics-dependent models and focus on semantics-independent work. Historically the earliest, Gray et al [15] have defined 4 degrees of consistency, numbered from 0 to 3. Degree 0 only protects a transaction's uncommitted updates from being overwritten. Degree 1 offers the same protection for recoverable transactions, but a transaction may still read uncommitted values. Degree 2 isolates a transaction from the uncommitted data of other transactions, however, if it reads the same object twice, it may read two different values. Degree 3 is equivalent to SR. The particular specification (ImpLimit and ExpLimit) used in this paper does not distinguish degree 2 from degree 1. Update ETs are SR, so they are at degree 3. Query ETs may see other ETs' intermediate states, so they are at degree 1 and 0 (the two are the same for queries). We note that our 2PL-based DC methods (e.g., tables 2 and 3) already implement each degree of consistency for a more detailed specification.

Garcia-Molina and Wiederhold [13] have introduced the *weak consistency* class of read-only transactions, defined as "each local query has a consistent view of the data, but the union of all schedules may be inconsistent". Their Wait-for List Centralized Algorithm (WLCA) gives each local query a consistent view of the data, but the union of all schedules may be inconsistent. This is more restrictive than ESR. ESR maintains global (eventual) consistency and allows bounded inconsistency in (local and global) query ETs. Kogan and Garcia-Molina introduced Bakunin Networks [20], which supported *virtual serializability*. Bakunin Networks require a reliable message delivery mechanism. That algorithm is similar to ORDUP.

Du and Elmagarmid [9] have introduced *quasi-serializability* (QSR) as a correctness criterion for federated databases. QSR may be seen as a generalization of read-only weak consistency, since QSR also maintains locally consistent views, but does not guarantee global SR. QSR allows both updates and queries that may be globally inconsistent, but locally consistent. QSR queries may be modeled with ETs, but QSR updates may introduce unbounded inconsistency between autonomous databases. Therefore, QSR differs from ESR due to the bounding of global as well as local inconsistency.

Korth and Speegle [23] have introduced a formal model that allows (nested) transactions to specify pre-conditions and post-conditions to be satisfied before and after the transaction execution, respectively. These input and output conditions are predicates defined over the states of data objects. Their model includes multiple versions, nested transactions, and the consistency predicates. ESR is independent of versions and nesting. More importantly, their consistency specification is on data object states, while ETs specify the amount of inconsistency tolerated by the ET.

6.3 Specification of Inconsistency

ETs provide an interface to ESR, which is a correctness criterion that allows bounded inconsistency. Specification of inconsistency allowed is an important topic of further ESR development. Although some of the proposed specification approaches were proposed in the context of replication, they may be applied more broadly.

Wiederhold and Qian [33] have introduced the notation called *identity connection* to specify the constraints binding the replicas of an object. They classify the update propagation into four classes: immediate updates, deferred updates, independent updates, and potentially

	No Lock	R^{ET}	W^{ET}
ImpLimit= 0 read/write	—	—	—
ImpLimit> 0 R^{ET}	AOK	AOK	LOK–1
ImpLimit> 0 W^{ET}	AOK	LOK–2	Commu–1

Table 5: 2PL Compatibility for Dirty Objects

Table 5 shows the lock compatibility for dirty objects. The backward DC uses Table 3 to handle clean objects. In Table 5, R^{ET} includes the read requests from Q^{ET}. Note that we have identified the places where potential inconsistency may arise and made explicit the references to these situations. This type of analysis is facilitated by ESR because it is semantics-independent. This does not prevent ESR from incorporating the explicit specification of semantics-dependent inconsistency. In contrast, sagas as proposed are implicitly dependent on application semantics for the maintenance of database consistency.

5.4 Distributed Backward DC Methods

The extension of backward DC methods to distributed databases uses the same techniques outlined in Section 4.1. With backward DC, we have an additional method to handle inconsistency. Namely, once a non-SR situation is detected we can use backward DC methods to decrease the amount of inconsistency.

6 Related Work

Due to space constraints we will only list a few representative examples of the many papers related to this topic. A survey [8] on merging network partitions describes some techniques on replication. Since a companion paper [28] describes the application of ESR to replicated data, we omit further references on replication.

6.1 Eventual Consistency

One characteristic of ESR is eventual consistency, i.e., the database eventually converges to a consistent state. Grapevine [7] and Clearinghouse [26] are early examples of replicated directory systems that achieve eventual consistency using periodic updates. Another example of application-specific databases supporting eventual consistency is the battle management system proposed by Jajodia and Meadows [18], where a specialized algorithm is used for maintaining eventual consistency of track files. Unlike previous work, ESR (because of the ET interface and DC methods) is independent of particular applications and application semantics.

	ImpLimit= 0	ImpLimit> 0
ExpLimit= 0	Ts (SR)	Q^{ET}, comp.step
ExpLimit> 0	U^{ET}	saga step

Table 4: New Kinds of ETs

5.2 Operation-Level Backward DC

The main idea of backward DC methods is represented by Time Warp-style compensations (COMPE).

Operations handling: Each message (operation) has an anti-message, which can undo the effects of the original. As operations are executed, they are saved on a log. If an anti-message arrives, the general mechanism of the previous section is used to recover database consistency. Messages older than the Global Virtual Time can be safely garbage collected.

Divergence control: Time Warp rolls back at the message granularity (the anti-messages). Time Warp can preserve SR by rolling back every operation after any divergence. In ESR, we need only to roll back Ts and update ETs. If a query ET's inconsistency counter has not exceeded its ImpLimit, there is no need to roll it back.

Fuzziness bounding: Each time a rollback happens, inconsistency may be introduced into some query ETs that read the previous value. To bound the divergence, we can choose from three alternatives. One, increase the inconsistency counter of an affected query ET, and leave it as is. Two, rollback the query ET to avoid increasing its inconsistency counter. Three, block the rollback in order to avoid both query rollback and more inconsistency.

5.3 Transaction-Level Backward DC

Backward DC can also be done at the transaction level, as proposed by Sagas [12] and its generalization [11]. Sagas can be seen as a special case of ESR, where each *step* in their model is an ET. The results of each successful step is visible to subsequent steps as well as other sagas. When all the steps of a saga finish, the database becomes consistent.

Operations handling: The execution of each step is an ET. Some ETs may produce results that are corrected later by a compensation step. We capture the total amount of inconsistency with the specification of ImpLimit and ExpLimit. An update ET that may be compensated later is an ET with ImpLimit> 0 and ExpLimit> 0. A compensation ET has ImpLimit> 0 but ExpLimit= 0. Table 4 shows the new kinds of ETs. A "saga step" may introduce unbounded inconsistency, but a "comp.step" (compensation step) restores consistency explicitly.

Divergence control: Sagas as proposed [12] do not support ESR, since the amount of inconsistency (revealed before a compensation) may propagate and persist in the database. To guarantee ESR in general, we need to remember the "dirty" objects that contain potentially inconsistent information (those updated by saga steps subject to later compensation). Only ETs with ImpLimit> 0 should be allowed to read these objects. The dirty objects are cleaned by the compensation steps or the end of the saga.

Divergence control: The CDB execution of local operations follows the same rules (Table 3) as described in Section 3.2. Superdatabase validation can guarantee global serialization order.

Fuzziness bounding: The same as distributed ORDUP (Section 4.2).

4.4 Read-Independent Timestamped Updates

Operations handling: The same as local RITU.

Message delivery: Messages can be delivered in any order.

Divergence control: The main problem is the assignment of timestamps in such as way that each sub-ET does not contradict another. One way to achieve this is to assign timestamps at DET generation time, so that all data affected by a given transaction has the same timestamp. In the absence of such global ordering, Superdatabase validation works.

Fuzziness bounding: The same as distributed ORDUP (Section 4.2) and COMMU.

5 Backward DC Methods

5.1 Difficulties in Compensation

The backward convergence method repairs divergences after it has happened. The divergence may have happened at the operation level, as in Time Warp [19], or at the transaction level, as in Sagas [12]. To take into account a model such as Sagas, we extend the ET model described in Section 2.2 to include ETs with ImpLimit>0 and ExpLimit>0. In the Sagas model, the application semantics maintain the global consistency of the database, and inconsistency propagation is minimized by compensations in each saga. Here, the term "inconsistency" refers to any visible information that is incorrect or has become incorrect. Concretely, each compensation transaction corrects some inconsistency in the database.

The main difficulty with inconsistency repair based on compensations is that the entire history needs to be redone in the worst case. Consider the following example in the standard model. Let transactions $F_1(a) = Inc(a)$ and $F_2(a) = Mul(a)$. Their compensation transactions are respectively $F_1^{-1}(a) = Dec(a)$ and $F_2^{-1}(a) = Div(a)$. It is easy to see that the following naive compensation does not work:

$$Inc(a,200)Mul(a,10)Dec(a,200) \neq Mul(a,10).$$

Instead, we need to restore the history:

$$Inc(a,200)Mul(a,10)Div(a,10)Dec(a,200)Mul(a,10).$$

In general, to compensate for $F_1(a)$ followed by $F_2(a)\cdots F_n(a)$ we need

$$F_1(a)F_2(a)\cdots F_n(a)F_n^{-1}(a)\cdots F_2^{-1}(a)F_1^{-1}(a)F_2(a)\cdots F_n(a).$$

Although the general case is difficult, since we have to remember not only the compensation transactions but also the recent history of transactions, some situations are simpler. In particular, we can see that if all F_i and F_i^{-1} are commutative with respect to each other, then we would not need to rollback the intermediate Ts. This observation suggests that in ESR forward DC may be combined with backward DC. The combinations are beyond the scope of this paper.

since it is a general framework for guaranteeing global ordering without having to specify a particular distributed concurrency control.

Similar to guaranteeing the SR properties of update DETs, another problem in distributed DC is to maintain the inconsistency limit of the query DETs. We have two general strategies. The optimistic method allows each CDB to run up its own inconsistency counter and add them up at the end. If the ImpLimit has been exceeded then the entire DET is rerun. The pessimistic method distributes quotas of inconsistency to the CDBs, each of which enforces the local limit (see Section 3). Since the sum of quotas is equal to the global limit, local enforcement maintains global inconsistency limit. The local quota may be negotiated between participating CDBs during the DET processing, if one sub-ET is exceeding its quota but others are not. We note that we can obtain additional information from the CDBs (a generalization of O-vectors) to bound the divergence due to different serializations, but this optimization depends on the particular DC method and is beyond the scope of this paper.

4.2 Ordered Updates

Local operations handling: A DET is created by some client program and sent to the participating CDBs for execution. The update DETs go through the global validation, while the query DETs enforce the inconsistency limit as described above.

Message delivery: ORDUP does not require the reliable delivery of update messages in order they were sent. We only need that the messages arrive eventually. The important thing is to execute the updates in the right order in each CDB. This can be accomplished by reliable and ordered message delivery (e.g., stable queues [5] and reliable broadcast [6]), which are sufficient but not necessary.

Divergence control: The CDB execution of local operations follows the same rules of local execution as described in Section 3.1. If we have ordered message delivery, the R/W and W/W dependencies are preserved at each CDB. In the absence of such ordered message delivery, Superdatabase validation can guarantee global serialization order.

Fuzziness bounding: The total fuzziness of a DET is the combination of fuzziness from each sub-ET. To control the divergence of a DET, we use either the optimistic validation or pessimistic quotas, strategies outlined in Section 4.1.

Theorem 6 *ORDUP/2PL–Distributed guarantees ESR.*

Proof: Since we are using Superdatabase validation to guarantee uniformity of global serialization order, the only remaining part of the proof is the maintenance of global fuzziness limit. We have outlined two strategies for fuzziness bounding. The first is optimistic validation. If the sum of local fuzziness exceeds the global limit, we invalidate the DET. This strategy obviously maintains the global inconsistency limit. The second is pessimistic quotas. Each local CDB maintains its own local quota. Since each sub-ET is guaranteed to be within the local quota, and the sum of local quotas is equal to the global limit, the global limit is observed.

4.3 Commutative Operations

Local operations handling: The same as centralized COMMU.

Local operations handling: The same as centralized COMMU.

Message delivery: Messages may be delivered in any order.

Theorem 5 *There are logs allowed by ORDUP but not by COMMU; logs allowed by COMMU but not by ORDUP; and there are logs allowed by both DC methods.*

Proof: Again serial logs are always in the intersection. Log (4) is in COMMU but not ORDUP. An SRlog without commutative operations is in ORDUP but not COMMU.

The added concurrency allowed by DC methods is due to the interplay of two factors: operation semantics such as commutativity and the controlled inconsistency in Q^{ET}.

The intuitive reason DC methods do not allow all of SR logs is that they work under some additional restriction (in addition to that of standard concurrency control) on the execution of operations to control the divergence. This restriction may happen during the message delivery time, which is the case of ORDUP (Section 3.1). Alternatively, the restriction may be on the kind of operations allowed, during the operation processing time, as in the case of both COMMU (Section 3.2), and RITU (Section 3.3). Finally, the restriction may be on the operation processing as a whole, which is the case of backward DC methods based on compensations to be described in Section 5.

4 Distributed Forward DC Methods

4.1 Distributed Divergence Control

In applying ESR to distributed databases, the problem of recovery becomes specially difficult. The general problem of distributed database recovery is reaching a uniform abort/commit decision. DETs must be able to reach this decision in a asynchronous fashion. Some papers have suggested solutions to asynchronously reaching agreement, such as Unilateral Commit Protocol [17] and Optimistic Commit [22].

Although DC is clearly related to the implementation of recovery, we focus here on the problem of concurrency, i.e., the task of serializing each sub-ET the same way in all participant CDBs. We apply and extend some of the existing techniques used in heterogeneous transaction processing, such as Superdatabase [29].

In a Superdatabase, each CDB is allowed to execute its subtransaction independently. Each CDB is assumed to guarantee SR, and will return the subtransaction ordering. (This ordering may be sent back by the local concurrency control in an O-vector [29] or by the sub-ET itself in a ticket [14]). At commit time, the Superdatabase checks and validates a global transaction by making sure that all its subtransactions have been serialized the same way. This prevents inter-site cycles in the global SG. Due to the lack of commit protocols, there may be no "commit time" in a distributed database supporting ESR. Nevertheless, the creator of a DET must collect the results from each sub-ET. When all the results are in, the originator of the DET validates their serialization. As we will see later, this is important for ORDUP, but not necessary for COMMU.

Some other alternatives to Superdatabase exist. For example, one could distribute an ordering at transaction creation time [10]. This solution depends on the ability of CDBs to enforce the ordering, which is not the case of 2PL-based CDBs, to give one example. Another possibility arises when all the CDBs use strict 2PL and support some kind of commit protocol. In this case, no global checking is necessary since the commit protocol also marks the global lock point for each global transaction. In general, if all participating CDBs know about a distributed concurrency control method, then we have global SR. We use Superdatabase

3.3 Read-Independent Timestamped Updates (RITU)

Operations handling:

The third forward DC method is *read-independent timestamped updates* (RITU). The goal is the same: increase concurrency through operation semantics, this time explicitly eliminating the R/W dependencies on that object. There are two ways an RITU update stores its results: append-only or overwrite. In an overwrite RITU, there is only one version at any given time. An update with a newer timestamp erases the old version and an older version is simply ignored. In an append-only RITU, each update generates a new immutable version.

Divergence control: Since RITU writes commute between themselves and with reads, we can apply the results from the COMMU method.

Fuzziness bounding: Since RITU assumes the existence of timestamps, we can modify timestamp-based methods of concurrency control. For version-based databases, we use the Modular Synchronization Method [1], which makes versions of objects visible to queries in such a way that no smaller version can be created by any active or future transactions. This visibility control, called a visible transaction number counter (VTNC), produces SR queries. Query ETs may read versions newer than VTNC, knowing that the newer value may introduce inconsistency. Each time a query ET reads such a version its "inconsistency counter" is increased by one. (This is a conservative approach since it is possible that value will be committed and become valid.) The DC may prevent a query ET from reading newer data when the query has reached its inconsistency limit.

Theorem 3 *RITU/2PL produces ESRlogs.*

Proof: Eliminating the R/W dependencies means that read operations must know what to do with data. For overwrite RITU, by definition read operations will read the "most recently available" version of data, whatever it is. Period signals, such as radar scan, could be stored this way. For immutable versions, read operations are also timestamped and get data with the most recent timestamp before the read timestamp. Intuitively, although the RITU operations are performed in any order, the subsequent reads sort the versions to obtain the right version. Analogously, RITU operations are free of W/W dependencies. The strict ordering of updates in overwrite RITU eliminates the W/W conflicts. For immutable versions, since each RITU creates its own version, there are no W/W conflicts. In summary, RITU operations have no R/W or W/W dependencies, so they are COMMU by construction. So we can apply theorem 2.

3.4 Analysis of Forward DC

Theorem 4 *For any method* $\in$ {ORDUP, COMMU, RITU}*, there are (1) logs allowed by the method but not by SR; (2) logs allowed by SR but not by the method; (3) logs allowed by both the method and by SR.*

Proof: It is clear that the intersection between each method and SR is non-empty (serial logs are good examples). Log (5) is an example of ORDUP and COMMU (including RITU) that is not in SR. Since locking methods are more restrictive than SR, there are SRlogs that ORDUP/2PL and COMMU/2PL cannot produce.

	Q^{ET}	R^{ET}	W^{ET}
Q^{ET}	AOK	AOK	LOK–1
R^{ET}	AOK	AOK	Commu–2
W^{ET}	LOK–2	Commu–2	Commu–1

Table 3: 2PL Compatibility for COMMU

Divergence control: For concreteness, we describe the modification on two-phase locking, a method called COMMU/2PL. The standard locks R^T and W^T have the usual compatibility and are omitted from the table. The three new classes of locks (Q^{ET}, W^{ET}, and R^{ET}) are the same as the previous Table 2, so we omit the upper and left sides of the table. Table 3 shows the details of conflicts (lower right corner of Table 2), where "AOK" means always compatible, LOK–1 limited compatibility (under control) by reading uncommitted data and LOK–2 limited compatibility by overwriting data already read. Commu–1 means compatible when commutative, the usual case, and Commu–2 means that sometimes it is commutative and therefore compatible. This is usually implemented as a separate class of objects. For example, intention-exclusive locks in ARIES [24] are not compatible with either normal reads or writes, but are compatible with themselves. (There are many examples of commutative W^{ET} operations among themselves but few examples of commutativity between W^{ET} and R^{ET}). Finally, R^{ET} locks are compatible with Q^{ET}, R^T and commutative W^{ET}, but not the W^{ET} including W^T.

Fuzziness bounding: In situations where the system does not guarantee the delivery order of messages or the execution order of transactions, commutative operations serve well. To control the amount of inconsistency in query ETs caused by interleaving with execution, we refine the lock management in a way similar to ORDUP. The DC algorithm maintains an "inconsistency counter" for each query ET. Note that neither Commu–1 nor Commu–2 introduces any inconsistencies. However, each time a query ET obtains a read lock despite a W^{ET} (through LOK–1), its inconsistency counter is incremented by one. If an update ET obtains a write lock despite some Q^{ET} reading (through LOK–2), all the involved inconsistency counters are incremented. When ImpLimit is reached LOK–1 is no longer compatible with W^{ET}; in other words, query ET can no longer cause another non-SR conflict, since doing so would exceed the specified ImpLimit. For the same reason, an update ET can no longer overwrite (W^{ET}) a value read by that query ET (LOK–2). Since compatibility between Q^{ET} and W^{ET} is the main source of inconsistency, if we use an inconsistency limit of zero, table 3 reduces to the standard SR 2PL lock compatibility table except for the commutativity squares.

Theorem 2 *The COMMU/2PL DC method allows only ESRlogs.*

Proof: We have to show that COMMU/2PL DC produces logs that are ESRlog. The proof is essentially the same as theorem 1. The only addition is on the Commu–1 and Commu-2 squares, which by definition do not introduce additional inconsistency.

In our notation, I is an increment and D a decrement. There are no dependencies between increments and decrements. In log (5), transaction $U_1^T = R_1^T(a)W_1^T(a)$ precedes both update $U_2^{ET} = I_2^{ET}(b)D_2^{ET}(a)$ and query $Q_3^{ET} = R_3^{ET}(a)R_3^{ET}(b)$.

Fuzziness bounding: To control the amount of inconsistency allowed in a query ET, we further refine the lock management at LOK–1 and LOK–2. The 2PL DC algorithm maintains an "inconsistency counter" for each ET. An update ET's counter stores the amount of inconsistency the update ET has exported, and a query ET's counter stores the amount of inconsistency imported. Each time a Q^{ET} obtains a read lock under LOK–1, we increment the inconsistency counter of the conflicting Q^{ET} and U^{ET}. Similarly, each time an U^{ET} obtains a write lock under LOK–2, all the query ETs holding the conflicting read lock have their inconsistency counters incremented by one. If either the update ET's ExpLimit or a query ET's ImpLimit has been reached, then we must stop the information flow and disallow that particular access. In this case, we can choose to make the requester wait as in standard 2PL, or to abort and restart by returning a lock failure or breaking a lock already held.

Theorem 1 *The ORDUP/2PL DC method allows only ESRlogs.*

Proof: We have to show that ORDUP/2PL DC produces logs that are equivalent to an ESRlog. We use that fact that standard 2PL has been shown to guarantee SR. Table 2 shows that without the squares LOK–1 and LOK–2 we would have a basic 2PL producing an SRlog. So for the case ImpLimit= 0 and ExpLimit= 0, we have no cycles in the Serialization Graph (SG). For the other cases, we must check and make sure that the ExpLimit of update ETs and the ImpLimit of query ETs are maintained.

First we observe that the definition of ESRlog says that a limited number of cycles can be allowed in the SG. As long as the number of cycles for each ET is below its specification, then the log remains ESRlog. ORDUP/2PL DC says that each query can acquire inconsistency by importing uncommitted data under LOK–1 and each update ET can cause inconsistency by overwriting data under LOK–2. In case 1, only one new cycle is possible since only one arc is being added, and the possible cycle is counted in the inconsistency counters. In case 2, multiple arcs may be added, but all of them are counted in inconsistency counters. This is the case because the cycles form a bipartite graph, with update ETs in one partition and query ETs in the other. Since ORDUP/2PL DC prevents cycles to form when the specified limits are reached, the resulting SG has less or equal number of cycles as specified. Thus the log is an ESRlog.

3.2 Commutative Operations (COMMU)

The second forward DC method uses only *commutative* operations (COMMU). If a log contains only commutative operations, then the log is an ESRlog. The reason is that commutative operations can be conveniently rearranged on the log to produce a serial log.

Operations handling: Although the added concurrency of commutative operations is well known, one of the major difficulties in applying it to SRlogs is the lack of commutativity between writes and reads. As a result, in practice, queries tend to limit the amount of available concurrency. In contrast, query ET operations are commutative with any update ET operation, since they are allowed to see the intermediate results of update ETs. Complications arise when the update ETs include operations (such as simple reads) that do not commute with some updates, even though the update operations may commute among themselves.

	R^T	W^T	Q^{ET}	R^{ET}	W^{ET}
R^T	AOK	—	AOK	AOK	—
W^T	—	—	—	—	—
Q^{ET}	AOK	—	AOK	AOK	LOK–1
R^{ET}	AOK	—	AOK	AOK	—
W^{ET}	—	—	LOK–2	—	—

Table 2: 2PL Compatibility for ORDUP

a cycle to form in the Serialization Graph. The extension stage isolates the identification part of concurrency control and the relaxation stage modifies the cycle prevention part so as to permit specific inconsistencies. The exact modification varies for each concurrency control method. There are many known concurrency control methods, many possible modifications, and many ways to bound fuzziness. In this paper, we only show an existential proof of such techniques, without attempting a comprehensive coverage.

3 Centralized Forward DC Methods

3.1 Ordered Updates (ORDUP)

Operations handling: The first forward DC method is *ordered updates* (ORDUP). In this case, update ET operations are run under some standard concurrency control method (probably the same governing the Ts) to keep them SR. Query ETs are allowed to interleave with update ETs through some modification of the concurrency control method. The idea is to add a marker to update ETs so that query ETs can distinguish them from the normal Ts. Each concurrency control method would be modified differently. For example, the modification for two-phase locking (2PL) is summarized in Table 2.

Divergence control: In Table 2, R^T means read lock by T, W^T write lock by T, Q^{ET} read lock by query ET, R^{ET} read lock by update ET, and W^{ET} write lock by update ET. The standard 2PL lock compatibility appears on the upper and left sides of the lock table. The squares marked AOK are always compatible, dashes are always incompatible. The ET extension of 2PL, called 2PL DC, starts with the read compatibility between Q^{ET} and R^{ET}. The difference between ORDUP and SR is between Q^{ET} and U^{ET}. LOK–1 (for *L*imited OK, case 1) allows query ETs to read uncommitted data, while LOK–2 allows update ETs to overwrite data that query ETs have read. (In the table, columns represent locks held and rows locks requested.) Log (4) is an example of LOK–1 and log (3) is an example of concurrency allowed by LOK–2:

$$D_1^{ET}(a)D_2^{ET}(b)Q_3^{ET}(b)Q_3^{ET}(a)I_1^{ET}(b)I_2^{ET}(a) \tag{4}$$

$$R_1^T(a)W_1^T(a)I_2^{ET}(b)Q_3^{ET}(a)D_2^{ET}(a)Q_3^{ET}(b) \tag{5}$$

against increased concurrency. For example, the commutative operations method (described in Sections 3.2 and 4.3) restricts ET operations to those that are commutative. Another DC method, ordered updates, is similar to SR in the centralized case (described in Section 3.1), but is more restrictive in the distributed case (described in Section 4.2). A third DC method, read-independent timestamped updates (described in Sections 3.3 and 4.4) also restricts update operation semantics. But, in all cases, DC methods offer a higher degree of concurrency and asynchrony than standard concurrency control methods, and also allow more concurrency for queries.

We divide the DC methods into two groups. The above three are *forward* DC methods (Sections 3 and 4), which prevent inconsistency from occurring at all. The second group is based on compensations (Section 5), that rollback operations or ETs that cause inconsistency. When operating asynchronously (without commit protocols), forward DC methods assume the ETs have decided to commit. Aborts have to be handled by backward DC methods.

An important question in ESR is the amount of inconsistency (called *fuzziness*) a query ET may see, and the techniques which DC methods can use to bound it. For simplicity of presentation, we consider only the numerical values in objects. Observe that inconsistencies arise from non-SR conflicts of query ETs and update ETs on the log. If update ETs can set new values arbitrarily large, we need to enumerate all the conflicting updates to calculate the query ET's fuzziness. Fortunately, many practical databases and applications have an upper bound on their update values (e.g., bounds are standard banking practice). In such cases, we have the upper bound of fuzziness for each query ET:

$$Q^{ET}\ \text{UpperBound} = \sum_{\text{conflict}}^{\text{non-SR}} |\text{Update Values}| \leq |\text{conflicts}|^{\text{non-SR}} * \text{UpdateBound}$$

Since it is often impractical to collect the missing update ETs in a database, especially if it is distributed, we need to estimate the amount of fuzziness. If we restrict the maximum update value to K, then fuzziness is bound by $N * K$, where N is the number of non-SR conflicts in update ETs.

One of the goals of DC methods is the control of each query ET's fuzziness according to a user specification. Despite the differences in the design of DC methods describe in this paper, they all implement the ImpLimit and ExpLimit specification introduced in Section 2.2. Note, that although other DC methods can be designed to implement alternative user specifications (for example, see Section 6.3), the basic idea of limiting the amount of non-SR conflicts a query ET may have is common to all DC methods. For example, DC methods described in Sections 3, 4, and 5 take care of both DC for ETs and concurrency control for Ts. In these cases, a DC method modifies an existing concurrency control method to allow non-SR conflicts. To bound fuzziness we count the number of conflicts that are non-SR.

This paper describes divergence control only through the operations of each type (divided in sections); that is, we do not consider the issue of "mixing" methods. Each DC method is divided into three steps. The first step is *operations handling*, which generates ET operations and synchronizes updates. The second step is *divergence control*, which schedules the interleaving of query ETs with update ETs. The third step is *fuzziness bounding*, which controls the amount of inconsistency allowed by the divergence control step.

The design of DC methods follows a two-stage methodology. In the first stage, existing concurrency control methods are extended to include ETs. In the second stage, the extension is "relaxed" so as to allow ETs more concurrency. The underlying idea is that (conflict-based) concurrency control methods must be able to (1) identify the conflicts and (2) prevent

a disjoint part of the set. A distributed ET (DET) contains several sub-ETs, each running on a CDB. Each CDB has its local log. The union of local logs is the global log. By the Serializability Theorem, a local SRlog has an acyclic SG. To achieve global SR, we merge the sub-SGs from the CDBs built from the precede relations of sub-Ts. In the following example of distributed ESRlog, we have log (2) in CDB X and log (3) in CDB Y.

$$R_3^{ET}(a)Q_4^{ET}(a)W_3^{ET}(a) \tag{2}$$

$$R_3^{ET}(b)W_3^{ET}(b)Q_4^{ET}(b) \tag{3}$$

The union of logs (2) and (3) is an ESRlog (in fact, an ϵ-serial log) for U_3^{ET} with ExpLimit≥ 0 and Q_4^{ET} with ImpLimit≥ 0. It is easy to see that Q_4^{ET} precedes U_3^{ET} in CDB X because of an R/W conflict on object a and U_3^{ET} precedes Q_4^{ET} because of another R/W conflict in CDB Y on object b. Thus the sub-history formed by U_3^{ET} and Q_4^{ET} is not SR. However, the non-SR interleavings are within the specified limits, so the union of logs (2) and (3) is an example of distributed ESRlog.

Although the focus of ESR is divergence control, we consider briefly the issue of DET recovery: All the sub-ETs of a DET must asynchronously agree whether to commit or abort.[1] A *synchronous* way to reach agreement among the sub-ETs is by running a commit protocol such as two-phase commit (2PC) that make the decision for the entire DET. Note, however, that an important motivation for the development of ESR is to increase database concurrency when a commit protocol is unnecessary or undesirable. For example, if we know that a transaction is going to commit, then each sub-ET can execute asynchronously and commit independently. This is the case with the Unilateral Commit Protocol [17], in which sub-ETs know that they must commit: they therefore repeat their execution until successful. If we do not know the transaction outcome *a priori* but we want high concurrency, we can allow the sub-ETs to commit and run a compensation transaction if the sub-ET should be aborted later. This is the case with Optimistic Commit [22]. Integrating recovery with divergence control is a subject of future research.

The only assumption we make about the communications facility between CDBs is that all messages will *eventually* arrive. Examples of stronger reliable communication mechanisms in the context of distributed databases include stable queues [5] and persistent pipes [16]. In general, we do not require messages to be delivered in order.[2]

2.4 Divergence Control

By analogy to concurrency control methods that support SR efficiently, we want to design *divergence control* (DC) methods that guarantee ESR. The goal is to produce execution logs that are ESRlogs (allowing more concurrency than SR) by using some rules to restrict the update ET execution. Since DC methods are concerned with the concurrent (consistency preserving) execution of ETs, we factor out the problem of message losses due to site crashes by encapsulating it in reliable communications mechanisms.

In comparison with concurrency control methods, a DC method may impose more stringent restrictions on a specific system parameter. The additional restrictions "trade-off"

[1] This is a standard assumption made in early distributed transaction models. In contrast, most nested transaction models (e.g. Moss [25]) allow a DET to commit despite sub-ET aborts.

[2] Even the ordered update DC method (Section 3.1) can receive operation-level messages out of order, since it only has to run the execution of update ETs in the right order.

	ImpLimit= 0	ImpLimit> 0
ExpLimit= 0	Transaction	Q^{ET}
ExpLimit> 0	U^{ET}	unbounded inconsistency

Table 1: Three Kinds of ETs

data read by other transactions and reading uncommitted data called degree 2 and degree 1 [15], respectively. One important observation is that when ImpLimit= 0 and ExpLimit= 0, the ET must be SR (degree 3), so ETs include the standard transactions as a boundary case. We continue to use T as a short notation for ETs that are SR. In the rest of this paper, the term ET implies the other two kinds of ETs shown in Table 1: query ETs denoted by Q^{ET} and update ETs denoted by U^{ET}.

When there are no inconsistencies allowed (ImpLimit= 0 and ExpLimit= 0), ETs are the same as standard Ts. For ImpLimit> 0, query ETs may import some inconsistency, up to ImpLimit. For ExpLimit> 0, update ETs may export some inconsistency, up to ExpLimit. However, the case of an ET both importing and exporting inconsistency is postponed to Section 5, since this may introduce new and unbounded inconsistency into the database.

A log containing query ET and update ET operations is called an ϵ-serial log if:

- we delete all Q^{ET} operations and the remaining update ETs form a serial log;
- the interleaving of query ETs and update ETs is such that their non-SR conflicts are less than the specified limits for each ET.

One immediate property of ϵ-serial logs is that they preserve database consistency, since all update ETs are serial.

An ESRlog is a log that produces results equivalent to an ϵ-serial log. A database that execute ETs producing an ESRlog is said to support epsilon-serializability (ESR). An example of ESRlog is:

$$R_1^{ET}(a)R_1^{ET}(b)W_1^{ET}(a)Q_2^{ET}(a)Q_2^{ET}(b)W_1^{ET}(b) \tag{1}$$

In our notation, R_1^{ET}(a) means a *R*ead operation in ET_1 on object a. W_1^{ET}a denotes a *W*rite, Q_2^{ET}(a) a query ET_2 reading object a. Log (1) is an ESRlog (in fact, an ϵ-serial log) for U_1^{ET} with ExpLimit$\geq$ 0 and Q_2^{ET} with ImpLimit$\geq$ 0. It is easy to see that Q_2^{ET} precedes U_1^{ET} because of an R/W conflict on object b and U_1^{ET} precedes Q_2^{ET} because of another R/W conflict on object a. Thus the sub-history formed by U_1^{ET} and Q_2^{ET} is not SR. However, the non-SR interleavings are within the specified limits, so log (1) qualifies as an ESRlog.

2.3 Distributed ET (DET)

So far we have described ESR in a centralized database. Distributed databases also can support ESR. For simplicity we consider only a partitioned distributed database; the application of ESR to replicated databases is the subject of a companion paper [28]. Our model of a distributed database is still of a set of objects, but each component database (CDB) manages

2 Epsilon-Serializability (ESR)

2.1 Standard SR Model

A database is a set of objects, which support sophisticated operations. For compatibility with SR, we classify these operations into two groups: *read* operations that do not change object state and *write* operations that do. A transaction (denoted by T) is a sequence of operations that take a database from a consistent state to another. Transactions may be *updates* that contain at least one write or *queries* that are read-only. Queries do not modify database state.

Our terminology follows the standard model of conflict-based serializability [4]. A history or *log* is a sequence of operations such as read and write. A *serial* log is a sequence of operations composed of consecutive Ts. A log of transaction operations (typically reads and writes) is said to be serializable (SRlog) if it produces results equivalent to some serial schedule, in which the same transactions executed sequentially, one at a time. *Concurrency control methods* that preserve SR (e.g., two-phase locking) are algorithms that restrict the interleaving of operations in such a way that only SRlogs are allowed.

Intuitively, in the standard model a log is shown to be an SRlog by rearranging its operations according to certain rules. Two operations are said to *conflict* if at least one of them is a write, so we have read-write (R/W) and write-write (W/W) conflicts. Each pair of conflicting operations establishes a *dependency* and we cannot move one past the other in a log rearrangement, since the resulting log would have produced a different database.

A more formal way to specify concurrency control uses serialization graph (SG), where each arc represents the *precede* relation [4], denoted by $\rightarrow$. Transaction T_1 precedes T_2 ($T_1 \rightarrow T_2$) when one of T_1's operations precedes and conflicts (R/W or W/W) with T_2 operations. Since the Serializability Theorem [4] says that a log H is SR iff its serialization graph SG(H) is acyclic, an acyclic SG implies an SRlog.

2.2 Epsilon-Transaction (ET)

To take advantage of operation semantics and to introduce controlled inconsistency, we introduce a new grouping of database operations. An epsilon-transaction (denoted by ET) is a sequence of operations that maintain database consistency when executed atomically. ETs extend standard transactions in two ways. First, by using operation semantics, ETs support more kinds of rearrangements on the log than R/W and W/W dependencies. For example, a log containing only commutative operations produces a result equivalent to a serial log of consecutive ETs. By themselves, the new operations do not change the notion of an SRlog: a log containing ET operations is an SRlog iff the operations can be rearranged to become a serial log.

Second, ETs include a specification of the amount of permitted inconsistency. Abstractly, this specification is divided into two parts, *imported* inconsistency and *exported* inconsistency. There are many kinds of inconsistency and consequently the specification may take several forms (Section 6.3 contains some examples). In this paper we use the number of non-SR conflicts as a concrete measure. Each ET has two parameters, *ImpLimit* that denotes the maximum number of non-SR conflicts in which the ET reads data and *ExpLimit* that denotes the maximum number of non-SR conflicts in which the ET writes data. For simplicity of presentation our ImpLimit/ExpLimit specification lumps together all sources of inconsistency, although the divergence control methods described later distinguish at least two: overwriting

1 Introduction

Serializability [4, 27] is the standard notion of correctness in transaction processing. Serializability (SR) maintains database consistency: when a database only runs SR transactions, then (if the database starts out in a consistent state) the database is guaranteed to remain in a consistent state. Another advantage of SR is its efficient implementation by concurrency control algorithms that require low run-time overhead and relatively high transaction throughput. Despite these advantages, serializability has some limitations, first of which is on the concurrency allowed in the transaction processing. Second, in distributed databases serializability requires tight coupling of participant databases. This results from the synchronous nature of SR transactions: sites must agree atomically to update objects in a given order. From the point of view of performance, this requirement decreases both system availability and throughput, as the system size increases. From the point of view of autonomy, sites may not wish to support this kind of tight coupling.

We define the notion of *epsilon-serializability* (ESR), which permits execution logs with temporary and limited differences from an SR history. Instead of *weakening* SR consistency, ESR *stretches* it. In addition, ESR allows users to bound the amount of temporary inconsistency in ESR. We analyze this divergence, and show that at the limit, ESR becomes SR. Finally, we describe divergence control methods that preserve ESR in a way analogous to concurrency control for SR. Divergence control methods automatically restrict database operation execution sequences to guarantee ESR properties. This enables users to benefit from ESR through a high-level and simple specification of inconsistency tolerances.

Many important applications can use ESR. Such applications can and must tolerate some amount of inconsistency, due to the natural imprecision in their data (e.g., caused by data entry errors) or performance and availability considerations (e.g., real-time applications). The point of ESR is that inconsistency can and should be modeled independently of application semantics, since the important issue is the *amount* of inconsistency, not what kind. Extending the specification of inconsistency in ESR to include semantics-specific constraints is beyond the scope of this paper. A concrete example of ESR application is data replication. During network partitions, asynchronous mechanisms to maintain mutual consistency among replicas are preferred. Asynchronous updates increase availability and introduce temporary inconsistency. A companion paper [28] explains how ESR can solve that problem. Another example is airline reservation systems, which reportedly do not maintain SR. Because airlines make allowance for overbooking, they tolerate temporary inconsistency. ESR can model such trade-offs between degrees of consistency and concurrency requirements.

In Section 2 we introduce the idea of ESR in the context of both centralized and distributed databases, including a general framework for divergence control methods. We then describe some concrete instances of divergence control methods. Section 3 describes centralized forward divergence control methods that prevent inconsistency through various types of restrictions; Section 4 describes the distributed cases. Section 5 summarizes the backward divergence control methods: these methods compensate for inconsistency *after* it occurs. Section 6 gives a few representative examples of related work.

Contents

1 Introduction **1**

2 Epsilon-Serializability (ESR) **2**
- 2.1 Standard SR Model . . . 2
- 2.2 Epsilon-Transaction (ET) . . . 2
- 2.3 Distributed ET (DET) . . . 3
- 2.4 Divergence Control . . . 4

3 Centralized Forward DC Methods **6**
- 3.1 Ordered Updates (ORDUP) . . . 6
- 3.2 Commutative Operations (COMMU) . . . 7
- 3.3 Read-Independent Timestamped Updates (RITU) . . . 9
- 3.4 Analysis of Forward DC . . . 9

4 Distributed Forward DC Methods **10**
- 4.1 Distributed Divergence Control . . . 10
- 4.2 Ordered Updates . . . 11
- 4.3 Commutative Operations . . . 11
- 4.4 Read-Independent Timestamped Updates . . . 12

5 Backward DC Methods **12**
- 5.1 Difficulties in Compensation . . . 12
- 5.2 Operation-Level Backward DC . . . 13
- 5.3 Transaction-Level Backward DC . . . 13
- 5.4 Distributed Backward DC Methods . . . 14

6 Related Work **14**
- 6.1 Eventual Consistency . . . 14
- 6.2 General Weak Consistency Criteria . . . 15
- 6.3 Specification of Inconsistency . . . 15
- 6.4 Distributed Systems Models . . . 16

7 Conclusion **16**

Epsilon-Serializability

Calton Pu and *Avraham Leff*
Department of Computer Science
Columbia University
New York, NY 10027

Technical Report No. CUCS-054-90

calton@cs.columbia.edu

Revised January 31, 1991

Abstract

We introduce and formalize the concept of *epsilon-serializability* (ESR). ESR ensures eventual consistency in the serializability (SR) sense, allowing temporary and bounded inconsistency. Furthermore, ESR is compatible with SR since users may control the precise amount of inconsistency allowed, which is zero in the limit when ESR reduces to SR.

ESR is useful for solving problems such as data replication that require *asynchronous* query and update propagation for reasons of autonomy, performance, or availability. Low overhead divergence control methods maintain ESR the same way concurrency control methods maintain SR. They work by placing restrictions on potential concurrent executions, but are application independent. Divergence control is more restrictive in some ways, but offers (1) more concurrency than SR, and (2) user control of the amount of inconsistency.

Because ESR is a semantics-independent correctness criterion, it provides a uniform interface to user specification of inconsistency. Concrete specifications of inconsistency that depend on semantics are implemented by semantics-specific DC methods.