ExoSnap: a Modular Approach to Semantic Synchronization and Snapshots

Liuba Shrira and Siying Dong
Department of Computer Science
Brandeis University
Waltham, MA 02454
liuba@cs.brandeis.edu

Abstract

Type-specific synchronization and *consistent snapshots* are transactional storage system approaches for dealing with transaction contention. Current approaches suffer from a limitation that precludes the use of commodity storage servers because they require to run the type-specific synchronization code at the servers. This is a problem for "cloud computing" systems that must rely on commodity components to exploit economies of scale. We describe a new approach that overcomes the limitation. The new approach splits the transaction protocol code, the type-specific code runs outside the server, on the client side, and the generic performance-critical code runs in the server. New type-specific synchronization protocols and new type-specific consistent snapshot protocols can be developed without modifying the servers, providing the ability to use commodity servers. Moreover, no new type-specific schemes need to be invented, because the approach derives client-side type-specific synchronization and snapshot schemes from existing server-side schemes.

1 Introducation

"Cloud computing" systems that provide applications as services and store application data in data centers pose a challenge for storage systems. They need to be flexible to support application needs but must rely on commodity servers to exploit economies of scale. This position paper describes a novel flexible way to handle transaction contention in a storage system using commodity servers. Consider a "bargain hunting" application running *speculative transactions* that explore alternatives to optimize the committed choice. Such computations may run for a long time and may query large amounts of data, including historical states. Workloads that contain long-running update transactions and large queries suffer from contention. Contention interfers with transaction commits in optimistic and pessimistic concurrency systems, leading to loss of work and resource waste.

Type-specific synchronization and *consistent snapshots* are known approaches to handle transaction contention. Type-specific synchronization (semantic locking [7]) reduces contention for update transactions. Snapshot isolation [4] eliminates contention for read-only queries. Back-in-time queries running over persistent snapshots allow to access historical states in real-time on-line [18, 17] without interfering with the current state. Current approaches suffer from a limitation because they implement type specific synchronization and snapshots on the server side, requiring to run the type-specific synchronization code at the servers. This precludes the use of "commodity" servers, a "showstopper" limitation in cloud systems. To allow commodity servers, type-specific code needs to run outside the servers so that new synchronization protocols can be developed, and new applications can be deployed, without modifying the servers. On the other hand, for efficiency reasons, snapshot mechanisms such as cache-based copy-on-write used by high-performance database storage systems must run inside the server. Yet, snapshots and synchronization mechanisms can not be implemented independently since snapshot consistency semantics depends on the transaction synchronization semantics.

We describe ExoSnap, a new transactional storage approach that overcomes this limitation. The approach is based on encapsulated *revertable* reservation objects and *semantically consistent snapshots*. The reservation objects resemble semantic locks (reservations) [7]. They provide tentative revertable operations that allow to synchronize contention-prone transactions in a type-specific way. Reservation objects are implemented using a low-level transaction system supported by the storage system engine running in the servers. The approach implements type-specific synchronization in a modular way, without modifying the servers. Instead, the type-specific code, encapsulated in the reservation objects, runs in the client on cached state, and generic read/write concurrency control runs in the low-level transaction system at the server, treating reservation objects as ordinary persistent objects, provid-

ing the ability to use commodity servers.

The snapshot system reflects the synchronization levels. Low-level, type-independent snapshot system, implemented efficiently on the server side, captures snapshots that are consistent with respect to the low-level transactions. A low-level snapshot, however, may reflect modifications that eventually get aborted by the speculative transactions. A query accessing such a snapshot could observe inconsistencies. The high-level *semantic snapshot* system guarantees that no inconsistent snapshot states are observed by queries by "repairing" the low-level snapshots "just-in-time" when a snapshot query accesses a revertable object. The repair is type-specific and runs on the client side. This approach supports short-lived and long-lived snapshots providing what amounts to *semantic snapshot isolation* queries and *semantic back-in-time execution*.

Semantic snapshots are a new concept but semantic synchronization is an old idea, explored extensively in the mid 80's. Remarkably, despite many (server-side) proposals no general commercial implementations exist. We believe, the need to modify the concurrency control and recovery systems in the legacy server has been the primary barier. This problem is even more acute in cloud systems. By solving the problem, our approach allows a cloud storage system to handle contention in the most effective way for each application, taking advantage of new and existing synchronization schemes, while providing the ability to use commodity servers.

We have prototyped an ExoSnap storage system, based on reservation objects and semantic snapshots, and are evaluating the prototype performance.

2 A Data Type-based Approach

Consider a regional tour service used by visitors for planning trips on a limited budget, and by commerce organizations for capacity planning. A visitor needs to secure reservations in 2 or 3 tours to make the trip worthwhile. He has several alternative tours he would be happy to explore. The visitor puts together an initial plan and fine-tunes it in the course of a couple of days. To do this, he executes a "long-running bargain hunting transaction" that updates his trip by adding and removing reservations. The regional commerce organization may monitor visitor traffic by running read-only "audit transactions" that alert vendors of increased traffic. Similarly, it may analyze historical reservation traffic to predict future capacity needs.

Our goal is to provide effective support for long-running contention-prone speculative transactions accessing such services storing persistent data. Specifically we require:

1. Ability to acquire a service reservation so that a client can carry out speculative booking transactions and be sure the transactions will be able to commit without conflicts.

2. A proper outcome at the end. For example, the client should be able to commit only the bookings he ultimately decided to finalize.

3. A proper outcome in case of failure. For example if the client never finalizes the booking, the reservation should be released.

We have developed a new approach based on specialized data types to support these requirements. Importantly, our approach requires no special processing on the storage server nodes. This is attractive because it keeps the server code as simple as possible, enabling commodity servers. Because we run no special code at the storage servers, our work differs from earlier approaches.

Earlier work also made use of specialized data types to avoid concurrency control conflicts and developed a number of implementations of different data types [20, 9]. However, all these approaches involved the use of specialized code running at the servers. Our approach makes it possible to use the same implementations but without running them at the servers; all earlier implementations can be supported by our approach. Our approach thus allows a type-specific server-based scheme to be transformed into a client-based scheme.

In our scheme, the persistent storage for objects resides on storage servers while clients cache and access local copies of these objects. A client runs top-level transactions that contain within them special smaller *revertible* transactions (called in literature *open nested* transactions). The revertible transactions perform modifications to objects that are cached on a client and are used to commit changes, e.g. trip reservations, that may be cancelled later. They allow clients to coordinate their changes so that conflicts are avoided. Our requirement to not run any special code at the storage server implies that the storage servers do not know anything about the revertible changes. Instead, storage servers process all commit requests, including revertible transactions, identically. Our approach, instead, has special processing performed at the client. These computations run on cached copies of data from the storage server, and these copies will reflect the changes made by other committed transactions, including both committed top-level transactions and committed revertible transactions. Thus the computations can observe the revertible changes of other speculative transactions and take these into account.

Our approach makes use of special types of objects. Such an object provides the normal operations, including obtaining or releasing a reservation for a resource. Additionally, these objects are prepared to handle the changes committed by revertible transactions. When the user calls a modification operation on such an object, the operation performs the modification and records the execution of the operation in a log along with a lease. The lease stores the time at which the revertible operation will expire. The information about the revertible modifications and their leases is part of the representation

of the object, and thus is written to the storage server when the mobile client reconnects and the application commits the revertible transaction. Other clients, upon connecting to the shared storage server, will observe the revertible modification on the special object.

When the client reconnects and is ready to commit the top-level transaction, it must first call special *confirm* operation on all objects on which it wants the revertible change to become permanent. This operation updates the status of that change so that it *no longer* appears revertible. Additionally, the transaction can call a special *release* operation to undo the modifications that are no longer of interest to it. Thus when the top-level transaction commits, all of the objects whose modifications have been confirmed will be stored with those changes having really happened, and objects whose changes have been released will have those modifications dissapear. Note that the application need not explicitly cancel (release) the changes that are no longer needed, since these modifications will be undone automatically when those objects are used by other transactions after the leases expire. However, cancelling is desirable since it can release the resource earlier, before their leases expire.

Although we have described the system as if the applications must use custom-made special objects, in fact our approach allows application developers to avoid this work. Instead we have defined some built-in data types that provide the needed type-specific leases and transaction support. We have implemented two such classes, fragmentable objects and collections.

3 ExoSnap System

We have developed a storage system for speculative transactions called ExoSnap. The storage system runs high-level and a low-level transactions. High-level transactions are the speculative transactions running computations meaningful to the application. For example, a high-level transaction may include renting a car, assembling several trip alternatives, and committing a final trip transaction. High-level transactions synchronize in a type-specific way using the special typed reservation objects. The correctness condition for high-level transaction is *semantic serializability*.

The high-level transactions are implemented using low-level transactions. Low-level transactions update the persistent state. They synchronize using *read/write* concurrency control providing strict serializability. A high-level transaction may commit and abort multiple low-level transactions before comitting or aborting. A specialized protocol, described in this section, ensures the atomicity (all-or-nothing) property of the high-level transactions.

To avoid terminology confusion between transaction levels (high-level and low-level) and the structure of transactions within a level (top-level and nested) we refer to high-level transactions as *application* and refer to low-level transactions as *base*.

Base transactions Base transactions in our system are provided by the Thor client/server object storage system [11], though we could use any caching client-server storage system such as a SQL server replication. We have extended Thor to support client-side nested transactions (transparent to the server). A client runs transactions accessing the local copies of the cached objects stored persistently in storage servers. To commit a transaction, the client connects to the server. The server validates the transaction using an optimistic concurrency control scheme OCC [1]. The scheme provides efficient validation of client transaction read and write sets using object based invalidations and invalidation acknowledgements. A transactions that passes server validation is committed, and its results are stored persistently at the server (without re-executing it).

Semantic Synchronization on the Client Side Top-level application transactions run at the client and synchronize using reservation objects. A reservation object implements type-specific synchronization scheme on the client side. We explain using an example how we derive such client-side scheme from a type-specific server-side synchronization scheme.

Consider the *available-slots* object representing available bookings in the tour service, and consider the write/write conflicts that occur when concurrent speculative transactions add or remove reservations. These conflicts are superfluous in the sense that, as long as there remain available slots, no matter in what order the reservations are interleaved they produce the same available-slots balance. A type-specific concurrency control scheme called *escrow locking* [14, 10] avoids these unneeded conflicts by exploiting the semantics of the *escrow* type. An object of *escrow* type provides two commutative operations: *split(delta)* and *merge(delta)* . A transaction calls the split operation to make a reservation for specified (delta) escrow amount, and calls the merge operation to return the unused escrow amount. As long as the in-stock balance is positive, the escrow locking protocols allows concurent transactions to interleave the split and merge operations without conflicts.

Many type-specific concurrency schemes have been developed that allow additional concurrent operation interleaving compared to the strict read/write concurrency control(e.g. [3, 20, 21]). In all cases the type-specific code that avoids conflicts runs on the server side. Any such type-specific server-side concurrency control scheme can be transformed into a derived scheme that performs the type-specific synchronization actions at the clients yet allows the same operation interleavings as the server-side based scheme. In the derived

scheme the server has no type-specific code, running a simple read/write concurrency control scheme. We use escrow synchronization as our running example. The escrow type is a representative of a general class of *fragmentable* objects [19]. Objects of this class have commutative operations that can be exploited by type-specific concurrency control schemes like escrow locking to avoid conflicts. The ubiquitous collection types represent another large class with similarly exploitable properties. There are many others. We use escrow because it is simple.

Consider the server side implementation of a toor-booking service using escrow synchronization. The service is implemented by an object (service object) that exports a collection of methods. The methods include *acquire*, *release*, and *expire* operations that can be overriden by object implementation.

The object implementation consists of the procedures implementing the operations and the representation for the shared state they manipulate. The representation includes a set of outstanding reservations and an internal available-slots balance object that implements the escrow operations. The *split(delta)* operation is called by the acquire request to obtain the reserved escrow amount, and the *merge(delta)* operation is called by the release request to return the unused escrow amount. The *merge(delta)* operation is also called by the expire method that is invoked internally by the service system when a reservation expires.

The acquire request atomically commits the modifications to the in-stock-balance object and inserts a record describing the reservation into the reservation set. The reservation record specifies the reservation expiration time, and the actions that need to be performed if the reservation expires. These *reconciler* actions are type-specific, they perform the inverse of the operation invoked by the acquire request. The release and expire requests atomically commit the effects of the corresponding merge operation and remove the reservation.

The synchronization code described above resembles a concurrent object with a type-specific lock manager implemented using a monitor where monitor procedures implement the reservation requests, and monitor state encapsulates the internal available-slots-balance object and the outstanding reservation set. Within the monitor, the procedures use a simple mutex to serialize accesses to the shared monitor state.

We obtain our special encapsulated reservation objects by running the concurrent object on the client side. That is, by storing the persistent monitor state at the server, caching at the client the monitor code and state, running the monitor procedures on the cached state, and replacing the mutex synchronization with a concurrency control protocol (OCC) that coordinates access to cached state by validating read/write conflicts at the server.

When the client issues a reservation acquire request, the corresponding monitor procedure updates the client's cached state (the reservation set and the state of the in-stock-balance object) to reflect the reservation and sends the modified state to the server. If the state sent to the server is not stale, the server can commit the request making the updated state persistent. If the cached state is stale because another client has committed a reservation request, the server aborts the request and informs the client. The client obtains from the server the up-to-date monitor state, re-runs the request, and re-tries the commit with the new state. Eventually the request will succeed.

Application Transactions The top-level application transactions run at the client invoking operations on regular cached objects and the cached encapsulated reservation objects. The reservation operations (e.g. acquire, release and expire) are run as base transactions *nested* inside the top-level transaction. As mentioned, since the system serializes the base transactions using optimistic concurrency control, the server will abort a nested base transaction if the cached reservation object state is stale, i.e. has been modified by another client. In such case, the client refetches the new state of the reservation object, re-executes the nested transaction on the fresh state, and retries the commit of the nested base transaction. The committing nested transaction is retried without undoing the top-level transaction.

A top-level transaction that commits at the server a nested transaction (say, a nested transaction that acquires a leased escrow reservation) without also committing itself, exposes uncommitted effects by updating the *durable copy* of the escrow object. Such transaction nesting structure, called open nesting [21], violates the strict read/write serializability of the enclosing top-level transaction but allows to coordinate the top-level transactions among concurrent clients in a type-specific way that avoids conflicts.

A top-level transaction may crash or abort. To ensure the all-or-nothing property for top-level transactions, the open nested transactions have associated compensation actions that accompany the commit or abort of the top-level transaction. The goal of the abort compensation is to revert the exposed effects of the open nested transactions committed by the aborted enclosing top-level transaction. The goal of the commit compensation is to ensure that exposed effects are not reverted.

The reservation objects define operations called *reconcilers*, that revert the effect of their operations. The reconcilers are stored in the part of the reservation object representation, called the *reconciler log*. For example, the reconciler for an operation that acquires an escrow reservation for one item, is an escrow operation that releases the item. A reconciler is written durably to the reconciler log when the open nested transaction that runs the associated reservation operation commits at the server.

The reconciler entry in the reconciler log can be *active, deactivated*, or *timed-out*. The open nested transaction commits

an *active* reconciler that includes the lease expiration time. The commit compensation of the enclosing parent transaction *deactivates* all the *active* reconcilers that have been recorded by its enclosed open nested transactions.

Top-level transactions implement compensations by registering callbacks to handlers, called reconciler handlers, provided by the reservation object. The handlers are invoked at the client before the top-level transaction about to commit or abort at the server, sends all its tentative uncommitted modifications to the server for validation (invoking a base transaction).

If a reservation operation observes a timed-out reconciler in the reconciler log, it registers a handler for the timed-out reconciler with the enclosing top-level transaction. This way, commit or abbort compensation at the observing client will execute the reconciler. The execution of such a reconciler results in the deactivation of the corresponding durable reconciler in the reconciler log. Note, there is no problem with concurrent (duplicate) invocations of the same timed out reconcilers at multiple observing clients because the base concurency control scheme serializes the nested transactions, and therefore ensures that the first reconciler invocation that commits will deactivate the timed-out reconciler, thus invalidating the corresponding reconciler log (and the reservation object) in the other observing clients.

A top-level level transaction may be committing updates to multiple reservation objects. The commit compensation runs a reconciler handler for each object as a regular *nested* transaction that commits (or aborts) atomically with the top-level transaction. The commit compensation resembles releasing locks at transaction commit time in read/write locking schemes but there is an important difference. Where the release of read/write locks only affects performance, "semantic locks" must be released atomically with the parent commit to maintain correctness. This is because, if the enclosing transation commits and crashes without releasing, the time-out of the reservation will revert its effects, thus violating the all-or-nothing property of the top-level transaction whose commit depends on acquiring the reservation.

To commit the nested transaction as part of the top-level transaction commit, the client simply includes its read/write sets with the parent read/write sets. If the server can not commit the joint transaction because the reservation object was stale, the client receives an invalidation for the reservation object, refetches the new state of the object, and retries the joint commit without aborting the enclosing transaction. If other data (not a reservation object) was stale, the application will need to resort to after-the-fact reconciliation for that particular data.

4 Semantic Snapshots

ExoSnap creates low-level and high-level snapshots corresponding to the transaction synchronization levels. The low-level snapshots are created in two steps. The SNAP split snapshot system [18] creates persistent high-frequency (after every transaction) snapshots at each storage server node by retaining pre-states of updated objects. A variation of Fast Read-only Transaction protocol (FROP) by Liskov and Rodrigues [12] assembles a consistent *distributed* low-level snapshot out of individual server snapshots "just-in-time" when read-only queries access snapshot objects.

The snapshots assembled by the FROP/SNAP protocol reflect transactionally consistent states in the low-level transaction system. However low-level snapshots do not necessarily reflect consistent states in the high-level transaction system. This is because the snapshot system that runs inside the storage engine, oblivious to the revertable object state, could capture states that reflect tentative modifications, that eventually get aborted by the high-level transaction system. A snapshot query, observing aborted modifications, could violate the high-level transaction consistency.

Semantically consistent snapshots (semantic snapshots) provide the consistency guarantees for snapshot queries and back-in-time execution. The semantic consistency condition that captures the correctness of semantic snapshots requires that a snapshot reflects no tentative (uncommitted) states. The key observation underlying the creation of semantically consistent snasphots is that, given a state captured by a consistent low-level snapshot, any revertable modification recorded in a reconciler log of some revertable object corresponds to a high-level transaction that was uncommitted at the low-level snapshot serialization point. This is because a high-level transaction commit makes all tentative modifications "non-revertable". Therefore, "repairing" the state of a revertable object captured by a low-level snapshot by reverting (undoing) all the revertable modifications (expired or not) recorded in its reconciler log brings the object to a state that reflects a comitted high-level transaction system state at the execution point corresponding to the low-level snapshot.

A simple and efficient mechanism enforces semantic snapshot consistency. The mechanism is type-specific and therefore runs outside the storage system server. The mechanism requires the revertable objects to provide a special *snapshot* method. This method transparently "recovers" a consistent snapshot state in a type-specific way "just-in-time", so that semantically consistent high-level snapshot state is observed each time a query accesses a revertable object in the snapshot. The recovery does not affect the persistent snapshot state at the server.

The approach applies to short-lived snapshots, providing *semantic snapshot isolation* queries, and also to long-lived his-

torical snapshots, providing *semantic back-in-time execution*.

5 Related work

Database studies in mid 80's, Weikum [21], Ramamritham [2], and others, have explored multi-level and semantic concurrency schemes. Weikum [21] uses "physical" low-level transactions, serialized with read/write locks, and high-level "logical" transactions, serialized with semantic locks. The open nesting permits physical updates in the low-level system, without committing the containing high-level transactions. The logical conflicts and logical recovery, are handled ("compensated") on the server side. ExoSnap uses the same multi-level transaction idea, albeit with an optimistic low-level transactions, but moves to the client side the type-specific code for conflict compensation, crash recovery, and lease expiration. Moreover, the client handles type-specific snapshot support, not considered by Weikum. A recent middleware-based implementation of the J2EE Activity Service [16] uses a multi-level system and specific semantic locks. Instead, we provide a general framework that exploits existing server-side schemes. We exploit escrow schemes by O'Neil [14], demarcation protocols [3], concurrent abstract objects [20], and collection classes [13]. Recent work by Fekete et al [8] uses a *promise* a mechanism similar to reservations, to exploit an interesting new semantic property taxonomy, that we could exploit in ExoSnap. In on-going work we using our client-side approach for transfering reservations between disconnected appliances.

Semantic snapshots generalize the concepts of snapshot isolation queries [4] and time-travel queries [15, 18], to handle semantic synchronization. Graefe and Zwilling [6] propose to combine snapshot isolation queries and server-side escrow locking to reduce contention for indexed summary views in a SQL database in a data wearhouse. Binder et al [5] describe a multi-version generalized search tree data structure that supports snapshot isolation queries for web directories.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1995.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Trans. Database Syst.*, 17(1), 1992.

[3] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: a technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, 1994.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, San Jose, California, United States, 1995.

[5] W. Binder, S.Spicher, I. Constantinescu, and B. Baltings. Evaluation of multiversion concurrency control for web service directory. In *Proceedings of International Conference on Web Services (ICWS 2007)*, Salt Lake City, Utah, 2007.

[6] G. Graefe and M. J. Zwilling. Transaction support for indexed views. In *Proceedings of ACM SIGMOD Conference*, 2004.

[7] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. 1993.

[8] P. Greenfield, A. Fekete, J. Jang, D. Kuo, , and S. Nepal. Isolation support for service-based applications: A position paper. In *Proceedings of Conference on Innovative Data Systems Research (CIDR'07)*, Asilomar, CA, January 2007.

[9] M. Herlihy. Replication methods for abstract data types. Technical Report MIT/LCS/TR-319, 1984.

[10] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *ACM SIGMOD Record*, 17(3):117–125, June 1988.

[11] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, June 1999.

[12] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[13] Y. Ni, V. Menon, A. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the PPOP*, November 2007.

[14] P. O'Neil. The escrow transaction method. *ACM Transactions Database Systems*, 11(4):406–430, June 1986.

[15] G. Ozsoyoglu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knoweldge and Data Engineering*, 7(4):513–532, August 1995.

[16] F. Prez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and J. Vuckovic. Highly available long running transactions and activities for j2ee applications. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.

[17] L. Shrira, C. van Ingen, and R. Shaull. Time travel in the virtualized past: cheap fares and first class seats. In *Wirtualization Workshop, Haifa Systems and Storage Conference, Haifa, Israel*, 2007.

[18] L. Shrira and H. Xu. Snap: a non-disruptive snapshot system. In *Proceedings of the 21st International Conference on Data Engineering*, Tokyo, Japan, 2005.

[19] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Symposium on Reliable Distributed Systems*, pages 31–40, 1995.

[20] W. E. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM TOPLAS*, 11(2):249–283, 1989.

[21] J. Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of ACM PODS*, 1986.