

Compensation-Aware Data Types in RDBMS

Aravind Yalamanchi

Dieter Gawlick

Oracle USA

{firstname.lastname}@oracle.com

ABSTRACT

In a traditional database system, the transaction management protocols and mechanisms are constrained by the fundamental properties of atomicity, consistency, isolation, and durability (ACID). A transaction management system with strict ACID properties typically employs read and write locks, held for the duration of the transaction, to protect its uncommitted data from being seen and modified by some other transaction. While this approach is effective for applications involving short execution times and relatively small number of concurrent operations, it is too restrictive for applications that involve reactive, long-lived, and complex transactions. The common denominator of such applications is the need for transactions to read and possibly modify uncommitted data values [1] and for the database system to still retain the ability to abort a transaction and the ability to recover from failures. This paper proposes a Business Transaction framework that allows long lasting, discontinuous, and resumable transactions to perform *shared updates* to common data by holding semantic locks on the modified rows. Under this framework, basic SQL data types are made compensation-aware by associating domain-specific shared update semantics with them. These semantics ensure that each data modification operation is compatible with other uncommitted activity on the same data and that the operation can be undone, if needed, without resorting to cascading aborts. This paper describes the key concepts and presents our approach for supporting shared updates in Oracle RDBMS.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems – *concurrency, relational databases, transaction processing.*

General Terms

Algorithms, Design, Management.

Keywords

Business Transactions, Compensation, Concurrency control, and Constraints.

1. INTRODUCTION

In a traditional database system, the transaction management protocols and mechanisms are constrained by the fundamental properties of atomicity, consistency, isolation, and durability (ACID). The transaction atomicity, which is the cornerstone of a

transaction management system [1], indirectly impacts a transaction's ability to view the effects of another transaction while it is executing (visibility) and the database system's ability to recover in case of a failure (recoverability). In a typical transaction management system, access to another transaction's uncommitted data is restricted using read locks or multi-version read consistent views of data. Write locks protect a transaction's uncommitted data from being seen and modified by some other transaction. In such implementations, all operations on the database data are characterized as primitive and uninterpreted read or write operations and the concurrent operations on common data are blocked based on the compatibility of these primitive operations – a write operation prevents other write or read operations (not considering multi-version concurrency control implementations). The transaction management systems employ 2-phase locking techniques [2] for supporting strict ACID properties, which result in holding the locks acquired during the transaction until the end of the transaction. Such systems are effective for database applications involving short execution times and small number of concurrent operations on common data.

The limitations of traditional transaction management systems in handling applications involving interactive, long-lived and complex transactions are well understood [1][3][4][5][6][7]. The solutions proposed for such applications characterize the data manipulation operations as semantically meaningful steps rather than uninterpreted read and write operations. The solutions broadly fall into two categories; those that optimize the operations known to be commutative [3][4] and those that decompose long-transactions into small nested transactions, each having a pre-programmed compensating transaction [1][5][6]. The former allow concurrent execution of operations that are known to be commutative, such as escrow operations incrementing and decrementing the value, with a guarantee that, in the event of a transaction abort, the effects of these operations can be undone without impacting others. With the use of optimistic locking and commit time validation [3][4], these solutions continue to support strict ACID properties. On the other hand, the solutions based on compensating transactions selectively relax the transaction atomicity and isolation properties by modularizing a long-transaction into semantically meaningful open-nested transactions that can be compensated for. Although the nested transactions themselves are atomic and support full ACID properties, the parent transaction exposes its unconfirmed state after each nested transaction and only attempts to preserve application-dependent atomicity with preconditions [1] and compensation [5][6]. A compensating transaction, which semantically undoes the actions performed by a specific nested transaction, is application dependent and must be developed independently. Comprehensive meta-models have been proposed [6][7] to manage the dependencies between various nested transactions and ensure that conflicting operations across nested transactions are serialized. Unfortunately, the complexity of these meta-models makes them

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'09, June 29–July 2, 2009, Providence, RI, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06...\$5.00.

difficult to implement and impractical for mainstream applications, specifically involving relational database systems.

The existing works on the compensating transactions regard a set of database operations grouped into atomic transaction programs as the unit of compensation. Effectively, the burden of defining the transaction programs and the associated compensating transactions falls on the application developer and often such implementations are not reusable. This is especially true when the transaction programs and the compensating transactions are developed for a specific schema and/or an application.

Typically, when the primitive write operations are interpreted as semantically meaningful operations on the corresponding data, the ability to serialize a set of concurrent operations is a characteristic of the data and the nature of the operations. Often this logic can be built into the data type support, as serialization tests, for validating specific operations. For instance, the serialization test for escrow operations on `NUMBER` data type may ensure that all possible outcomes of unconfirmed operations leave the value above 0 and this may be built into the extended data type support. Our work focuses on this key observation and introduces a notion of compensation-aware data types. These data types enforce application-specific serializability by embedding domain-specific logic for validating concurrent operations and for performing compensation when needed. With this approach, each data-manipulation operation becomes a unit of compensation and the decomposition of long transactions into atomic sub-transactions is not dependent on the programs having clearly defined compensation logic. This paper introduces a flexible business transaction framework, which leverages from the traditional transaction management, to support nested and distributed long-transactions with the added ability to synchronize across transactions. The data manipulation operations on compensation-aware data, when performed within the context of a business transaction, are recognized as *shared updates*. All the shared update operations performed within a business transaction are either made permanent or compensated for when the business transaction confirms or aborts respectively. Compensation-aware data types can be developed independently using Oracle's extensibility framework and be reused for multiple applications.

The rest of the paper is organized as follows. Section 2 discusses the Business Transaction framework and Section 3 covers the key concepts related to the Compensation-aware data types. Section 4 discusses our approach of integrating this support into Oracle RDBMS.

2. BUSINESS TRANSACTIONS

A Business Transaction is an abstract representation that models a real-world business transaction and keeps its state consistent with the state of that transaction. In a database system, a business transaction is modeled using a sequence of traditional database transactions that may span multiple sessions. The database transactions within a business transaction are atomic and are expected to be short and in some extreme cases they can be self-committing database activity. An active business transaction has a persistent representation in the database and it survives all types of database failures and restarts.

The possible external states of a business transaction are analogous to that of a database transaction: Begin, Confirm, and

Abort. Additionally, a business transaction could enter an inactive or suspended state. Programmatic interfaces are used to begin or activate a business transaction and set the session context for subsequent data manipulation operations. Following is the command, as implemented in Oracle RDBMS, for starting a business transaction in a database session.

```
exec dbms_btx.begin_transaction (
    trans_key => 'Itinerary123456');
```

All data manipulation operations in a database management system are executed within boundaries of some database transaction. Such transactions are arranged into database sessions, which maintain the client's context information including any information pertaining to an active business transaction. The active business transaction's context is presented to each successful data manipulation operation performed on the data marked for compensation (See Section 3) and this information is recorded in the business transaction journal. This is used for validating concurrent shared update request and to perform compensation when necessary. All data manipulation operations performed within the boundaries of a database transaction acquire traditional row-level locks on the impacted rows, which are only held for the duration of the database transaction. Between two database transactions, the state of a business transaction is persistently stored in the database. In the event of failure within a database transaction, it can be retried while the business transaction preserves its last known state. Following a database failure, the traditional recovery process restores the state of the business transaction to the last successful database transaction.

At any given instance a database session may only have one active business transaction associated with it. However, the active business transaction can be suspended and another business transaction may be started/resumed within the session. The calls to resume/start a business transaction may not coincide with any database transaction boundaries and hence a single database transaction may group data manipulation operations that effectively map to more than one business transaction. So, unlike [1][5][6] the current system does not impose a nested transaction relationship between the business transaction and its components. This added flexibility allows multiple business transactions to perform a set of operations atomically. For instance, when two itineraries with some unconfirmed reservations are modeled as separate business transactions, some resources can be transferred from one business transaction to the other atomically using a single database transaction.

```
-- start a database transaction --
SET TRANSACTION name 'transfer_resources';

-- resume a business transaction --
exec dbms_btx.resume_transaction (
    trans_key => 'Itinerary123456');
UPDATE Reservations SET capacity = capacity-2
WHERE flightId = '123' and depart = '12/01/08';

-- suspend active txn and resume another one
exec dbms_btx.resume_transaction (
    trans_key => 'Itinerary654321');
UPDATE Reservations SET capacity = capacity+2
WHERE flightId = '123' and depart = '12/01/08';

-- end of database transaction --
COMMIT;
```

The effects of a business transaction are made visible immediately after each database transaction. A concurrent business transaction

can view and edit the unconfirmed data, thereby allowing shared updates to common data. Application-specific serializability for the concurrent operations is enforced using the shared update logic (Section 3.1) implemented in the data type and the accompanying operational constraints (Section 3.2).

A business transaction can be “prepared” to confirm or abort by invoking a corresponding programmatic interface within a database transaction. Such transaction ending operation can be made permanent by committing the database transaction. This allows multiple business transactions to be confirmed or aborted, within a single database transaction, atomically. Open nested business transaction may be created so that transaction-ending operation on a child transaction simply involves either merging its journal entries with the parent transaction or purging them, after applying necessary compensation.

A business transaction involving related services may be distributed across database instances to support a notion of distributed business transaction. In this scenario, a coordinating business transaction starts instances of the transaction on remote sites using a programmatic interface. The remote transactions are either confirmed or aborted, atomically, with the coordinating transaction. From a traditional transaction management point of view, only the transaction ending operations, such as Confirm or Abort, are truly distributed and thus employ a multi-phase commit protocol [2]. All other operations can be performed by directly connecting to the corresponding database instance and resuming the appropriate business transaction. This allows multiple database instances to independently perform their operations and either confirm or abort atomically.

3. COMPENSATION-AWARE DATA TYPES

Compensation-aware data types are specific usages of SQL data types that have associated logic to validate shared update requests on the corresponding data and to perform compensation when needed. The shared update logic is encapsulated in an object type[10] that directly interacts with the business transaction framework for various maintenance tasks. These object types, also called the *data classification* types, are implemented to identify and process semantically meaningful operations on specific SQL data types. An instance of such type is associated with a column of a relational table to make the traditional data types compensation-aware. For example, the escrow logic may be implemented as an `Escrow` data classification type and an instance of this type may be associated with a `Number` column defined to capture the quantity of some resources in a table. The following Oracle code sample associates the escrow logic with the `Capacity` column in the `Reservations` table.

```
exec dbms_btx.set_constraint (
    table_name => 'Reservations',
    column_name => 'Capacity',
    data_class => Escrow());
```

Ideally, a compensation-aware data type should allow only the operations that are semantically meaningful and well understood by the shared update logic. For instance, for an `Escrow` data type, the increment and the decrement are the only operations that are meaningful and all other operations must be restricted. Although this can be achieved using object relational features in the RDBMS, the inability to use standard data manipulation

languages (such as `UPDATE` statement) intuitively makes this approach unappealing. So, in the context of compensation-aware data types, it is understood that the shared update logic is able to correctly infer the exact type of operation from the application context and the old and the new values of the data. For example, an update statement operating on an `Escrow` data type may reserve half the capacity using “`SET capacity = capacity/2`”, in which case, the default shared update logic recognizes it as a decrement operation for specific number of units as computed at the time of executing the update statement.

The data classification type has specific methods to perform various business transaction related tasks such as maintain journal information, validate a shared update request, and compute a compensated value in the presence of concurrent updates. Often the logic to validate a shared update operation is external to the type implementation and it may be desirable to specify the validation criteria at the time of instantiating the data classification type. For example, a specific usage of `Escrow` data type may have minimum and maximum bounds that are stored in additional columns for each table row. For this purpose, the type externalizes the information maintained in the journal through pseudo variables and operators, which can be used to form declarative Operational Constraints (See Section 3.2) that span multiple columns.

Specific relational tables in a database schema may be enabled for shared updates with a programmatic interface. This step automatically assigns a default data classification type called `Ordinal` to all the columns in the table.

```
exec dbms_btx.enable_shared_updates (
    table_name => 'Reservations');
```

The `Ordinal` data classification type uses a default operational constraint that prevents concurrent changes to the corresponding values by ensuring that a business transaction can modify a value only if there is no prior unconfirmed change to the value. An `Ordinal` type can be configured to accept concurrent changes by specifying an appropriate operational constraint (Section 3.2). Alternately, the `Ordinal` type can be replaced with a domain-specific data classification type (`Escrow` or other) as discussed below.

3.1 Shared Updates

Shared update is defined as a data manipulation operation performed within the context of a business transaction, using standard DML (`INSERT`, `UPDATE` and `DELETE`) statement. Such operations can view the effects of other concurrent business transactions and further update the values following domain-specific logic. When a table with compensation-aware columns is operated on without a specific business transaction context (Section 2), a system generated business transaction is assumed to exist for the duration of underlying database transaction. Within a database transaction, the shared updates can be interspersed with DML on tables that are not enabled for shared updates. Such operations fall outside the context of business transactions and they follow traditional transaction management principles.

A standard DML operation modifying a value configured for compensation undergoes a series of validations to ensure that it is compatible with other unconfirmed operations. The purpose of these validations is to allow only the shared updates that are semantically serialized and to ensure that some deterministic

compensation exists for these operations. These validations may be done procedurally with the logic embedded in the data classification type or declaratively using Operational Constraints specified at the time of instantiation (Section 3.2). While the operational constraints are expressed as Boolean conditions using factors that describe the net effect and the summaries of all unconfirmed shared update operations, the procedural approach can iterate over each concurrent shared update operation to perform complex validations, if necessary.

The effects of shared updates performed within the boundaries of a database transaction are visible to concurrent transactions immediately after the commit of the database transaction. An INSERT operation performed within a business transaction, although visible, cannot be updated by concurrent transactions until the owning business transaction is confirmed. Similarly, a business transaction can DELETE a row only if there are no unconfirmed shared updates to any of its column values by other business transactions.

A traditional SQL query on a table with compensation-aware columns returns the version of data that includes the effects of all concurrent business transactions, including the one that is active in the session issuing the query. This is also the version of data that is physically stored in the database. For example, the following query on the `Reservations` table returns a value for `Capacity` that is adjusted for all unconfirmed shared updates.

```
SELECT Capacity FROM Reservations
WHERE FlightNo = 'UA123' and
      DepartDate = '03-JAN-2009'
```

A confirmed version of the same data, which excludes the effect of all active business transactions, can be obtained with the following query using `BTX_CONFIRMED` operator [11]. A `BTX_PROJECTED` operator can be used to see the effect of all concurrent transactions except the one that is issuing the query.

```
SELECT BTX_CONFIRMED (Capacity)
FROM Reservations
WHERE FlightNo = 'UA123' and
      DepartDate = '03-JAN-2009'
```

3.2 Operational Constraints

Operational constraints are Boolean conditions used to validate the shared update requests on the table columns marked for compensation. They simulate semantic locks on the data and are used as a concurrency control mechanism, which restricts the data manipulation operations that do not satisfy the conditions. The operational constraints may be used in conjunction with the procedural validation logic embedded in the data classification type. For a column being modified, the operational constraint is expressed in terms of the new column value, its last known confirmed value and its projected value, which includes the effects of all the shared updates performed by other business transactions. Additionally, the operational constraint defined for a specific column can refer to values corresponding to other columns in the same row. For example an operational constraint may be defined to ensure that all possible outcomes of the unconfirmed shared update operations leaves a column value above a limit stored in another column (See example below).

For each column defined in the table, the current, the projected, and the confirmed versions of the column values are accessed using the pseudo variable-extended column names such as

`btx_current.Capacity`, `btx_projected.Capacity`, and `btx_confirmed.Capacity` respectively. The logic defined in the data classification type is used to derive these values for each column being modified. Additionally, the data classification type may compute summaries over all unconfirmed shared update operations and externalize them as domain-specific operators that can be used in the operational constraints. For example, the `Escrow` type implementation may maintain a sum of all unconfirmed increments and decrements to specific data and externalize them as `BTX_ESCROW_INCR` and `BTX_ESCROW_DECR` operators respectively. Using these operators, the default operational constraint for an `Escrow` type, which ensures that the value resulting from all possible outcomes of unconfirmed shared updates is equal to or above 0, is defined as follows.

```
btx_current.Capacity >=
      BTX_ESCROW_INCR (Capacity)
```

This Boolean condition, when associated with the `Capacity` column, ensures that a shared update request is accepted only if the resulting value is greater than or equal to the sum of all unconfirmed value increments for the same data. If the `Capacity` column should maintain the values between a fixed range that is captured in `MinCapacity` and `MaxCapacity` columns in the same table, the following operational constraint may be used.

```
btx_current.Capacity between
      (MinCapacity + BTX_ESCROW_INCR (Capacity)) and
      (MaxCapacity - BTX_ESCROW_DECR (Capacity))
```

Each instantiation of a data classification type may specify a different operational constraint to enforce the application-specific logic. For instance, a specific use of `Escrow` type may allow a negative capacity value to simulate a waiting list. The size of the waiting list may be dynamically adjusted to be less than or equal to a value that is proportional to the number of unconfirmed reservations.

```
btx_current.Capacity >=
      BTX_ESCROW_INCR (Capacity) +
      (btx_confirmed.Capacity * -0.1)
```

The operational constraints can make use of user-defined functions and application context to handle special cases. For example, the `Ordinal` data classification type, which uses the arrival order of update operations to determine the compensation order, may be configured to allow new updates only if the resulting value dominates all unconfirmed and confirmed values.

```
SA_DOMINATES (btx_current.AccessLabel,
              btx_projected.AccessLabel) = 1
```

The previous example demonstrates a situation in which the security access labels associated with the rows in a table are updated by concurrent business transactions and a monotonically increasing order is enforced for the shared updates. This models a desirable lost-update scenario as the last update that is confirmed always prevails over all other updates preceding it.

This example may be further extended to demonstrate the use of application context in the operational constraints. The following operational constraint makes exceptions for a privileged user and allows a shared update even if it does not hold the `Dominates` relationship in the previous example.

```
SA_DOMINATES (btx_current.AccessLabel,
              btx_projected.AccessLabel) = 1
OR SYS_CONTEXT('appctx','userrole') = 'VP'
```

A common data classification type implementation can be used with different operational constraints to assign specific shared update semantics to each compensation-aware column. The following Oracle code sample assigns the `Escrow` type to the `Capacity` column in the `Reservations` table and also assigns a specific operational constraint to it.

```
exec dbms_btx.set_constraint (
  table_name      => 'Reservations',
  column_name     => 'Capacity',
  data_class      => Escrow(),
  constr_expr     =>
    'btx_current.Capacity >=
     BTX_ESCROW_INCR (Capacity)');
```

The operational constraint associated with a table column is evaluated only when an update statement modifies a value in the specific column. This technique can be used to enforce column-level semantic locks with business transactions. However, if some constraint should be enforced for all update operations, independent of the columns modified, a table-level operational constraint may be defined. Such constraints can be expressed using confirmed, projected, and current values for any of the columns in the table. Unlike the operational constraints that are assigned to specific columns, a table-level constraint is evaluated for all update operations modifying some rows in the table and the operation is restricted if the constraint fails.

3.3 Compensation Through Semantics

When a business transaction is aborted, the effects of all the DML operations performed over the lifetime of the transaction should be undone. Since the outcome of these DML operations was exposed and possibly updated by concurrent business transactions prior to the abort, a traditional log based undo where the current value is replaced with a prior copy of the same value may not be appropriate. Instead, a logical undo, based on the semantics of the data, should be performed. Logical undo for an operation originally performed in the transaction is modeled as a compensating action that should be performed on the current version of data. As is the case with most compensation-based systems [1] the database state reached from aborting a business transaction may not be identical to the state that would have been reached, had the business transaction never happened.

The data classification methodology specified for a table column determines the exact compensation logic used for the corresponding data. This logic is embedded in the data classification type implementation. The `Escrow` and `Ordinal` data classification types have well defined semantics for compensation and they are pre-defined in the framework.

Escrow and Change-based Compensation

The `Escrow` data classification methodology is used for columns storing measurable data such as capacity or quantity of some resource. The SQL data type for such columns is `Number` and the update operations on the corresponding data either increase or decrease the value by a specific number of units. For an update operation modifying a value by δ , the compensating action is to adjust its current value by $-\delta$. By ensuring that the original update operation satisfied the operational constraint at the time it was performed, the compensation is guaranteed to leave the value within the valid range.

Ordinal and Time-Ordered Compensation

The `Ordinal` data classification methodology is used for table columns that can take values from a finite set of independent categories. The key characteristic of such columns is that the values are not usually derived or derivable from other column values in the same or the other tables and the arrival order of operations that manipulate the values is significant. For example, a `Varchar` column in a table may capture the status of a collaborative project. The value may be updated by multiple business transactions, provided they satisfy the operational constraint associated with the column. The order in which these shared update operations manipulate the value is significant in that last successful operation that is confirmed by the corresponding business transaction prevails over the updates that precede it. Also, when a business transaction is aborted, compensation is necessary only if the corresponding shared update operation is the last, in arrival order, to update the value. The compensated value in such case is the value set by the update operation that immediately precedes it or the confirmed value if no such update exists.

Extensibility Framework for Custom Compensation

New data classification types with domain-specific logic for validating concurrent shared updates and compensation may be developed using the Business Transactions extensibility framework. The new types are implementations of an abstract type that has specific methods to perform maintenance for significant events in the lifecycle of a business transaction. These methods are summarized below.

- `getProperties`: The properties associated with the data classification type determine the SQL data types it can be used with, the names for any auxiliary operators, and the types of shared update validation implemented (procedural and/or declarative).
- `isValidChange`: (Optional) This method is required if the validation for shared updates on a specific compensation-aware data type cannot be modeled as a declarative operational constraint. When a procedural shared update validation is chosen for the type, the business transaction framework invokes this method with all the necessary metadata to validate each DML operation.
- `prepareJournalRecord`: Upon successful validation of a shared update operation (using operational constraints and/or procedural logic), the business transaction framework invokes this method for each modified column to prepare the journal record with all the necessary information. The user implementation for this method maintains the projected version for the value as well as any summaries needed by the corresponding operational constraints.
- `confirmChange`: When a business transaction is confirmed, all the associated shared update operations are considered confirmed. This in turn performs some maintenance on the journal to adjust any summaries computed for the shared updates. The `confirmChange` method allows application specific journal maintenance operations at the time of transaction confirmation.
- `undoChange`: This method is invoked for each shared update operation associated with a business transaction that

is currently aborting. This method, in addition to making necessary changes to the journal, computes the compensated value for the specific data modification that is being undone (logically) and returns this value to the business transaction framework. The framework uses these compensated values to make necessary changes to the table data.

The data classification type implementation maintains the confirmed and projected versions for the impacted values with the help of the business transaction journal. After a successful shared update operation, the new projected value is computed by the domain-specific logic in `prepareJournalRecord` method. In some cases, the projected value may not match any of the values set by the concurrent shared update operations. For example, a data classification type may use a lattice structure in a multilevel security application [9] to define valid data transformations for concurrent shared update operations. The lattice structure represents the sensitivity labels as the nodes and the relative strength of these labels as directed edges between these nodes. In this case, the shared update validation may enforce that an updated value dominates the last confirmed value and the projected value for a set of unconfirmed changes may be the Supremum, or the least upper bound, of all the unconfirmed values. For such data classification type, the operational constraint can ensure that a shared update request is accepted only if the new value dominates the last confirmed value. The `prepareJournalRecord` method may compute a new supremum for each shared update request after consulting the pending changes to the same value and set this computed value as the new projected value.

The flexible extensibility framework allows independent development of the data classification types by the domain experts. These types can be customized for each use with an appropriate operational constraint.

Compensating Insert and Delete Operations

A row inserted or deleted within the context of a business transaction may not have any shared updates from concurrent transactions. So, INSERT and DELETE operations may resort to log based undo for compensation. The compensation activity for an INSERT performed in a business transaction is a DELETE operation and vice versa.

Abort Callback

In addition to the compensation logic associated with each shared update operation, the business transaction may have an abort-callback routine, which can perform additional housekeeping during transaction abort. So, any compensation activity that does not directly belong in a data type implementation may be embedded in a callback routine that is associated with the business transaction through transaction profiles. For example, a travel reservation application may maintain a waiting list for pending reservations and may choose to consult this list and confirm waitlisted reservations when some resources are released with a business transaction abort operation. Such logic may be implemented as the abort callback routine, which is automatically executed for each successful transaction abort operation.

3.4 Handling Integrity Constraint

In a normal environment any attempt to abort a business transaction should go unobstructed and all its compensation activity should happen automatically with no user intervention.

This implies that the data modifications performed during compensation should uphold any integrity constraints defined on the corresponding tables. For example, an unconfirmed delete operation could remove a key on which a uniqueness constraint is enforced and if a concurrent insert or update operation is allowed to claim the removed key, the delete operation cannot be undone. To avoid such situations, the business transaction framework imposes certain restrictions on table columns involved in Referential, Primary Key, and Uniqueness constraints. Such columns may only be configured with `Ordinal` data classification type, which guarantees a finite set of values for all possible outcomes of unconfirmed shared update operations. In addition to enforcing the operational constraints for these columns, the business transaction framework validates the shared updates for potential conflicts involving integrity constraints. This guarantees unobstructed compensation when the corresponding transaction is aborted.

4. INTEGRATION WITH ORACLE RDBMS

The business transaction framework and the support for compensation-aware data types are implemented in Oracle RDBMS using its Object-relational [10] and extensibility features [11]. The business transaction journal is maintained in a system table and the data classification types are implemented as object types in the database. For each compensation-aware column, an instance of a data classification type and its operational constraint, in the compiled form, are stored in the extended column metadata. A row trigger is defined on the table to capture all the DML activity as shared updates and process them accordingly. The business transaction framework consults the journal entries for the row and columns being modified and evaluates the operational constraints for validating the shared update request. If necessary, the appropriate methods in the data classification type are invoked to validate the update operation procedurally and to compute the projected versions for the values. For example, if a business transaction modifies a value multiple times, the `undoChange` method is invoked to compute the net effect of the transaction with an appropriate projected value.

Once a shared update operation is validated, the business transaction framework, with aid of data classification type, prepares the journal record for subsequent use. This journal information facilitates additional row specific (shared updates to the same row by other transactions) and business transaction specific operations (confirm or abort). The journal accommodates all forms of user data by marshalling them into a homogeneous (`AnyData` [10]) type instance. All journal operations are performed within the context of the database transaction that initiated the shared update operation or the business transaction abort or confirm operations. So, if the database transaction is rolled-back the journal leaves no trace of the operation.

The standard data manipulation operations on a table enabled for shared updates undergo a series of validations and maintain relevant information in the business transaction journals, when successful. The current implementation uses triggers to capture the shared update requests and SQL commands to record the information in the journals. The trigger logic and the additional DML activity for maintaining the journal contribute to an approximate 120% performance overhead for single row

UPDATE operations on the tables enabled for shared updates. However, a similar overhead can be expected from any custom business transaction logic implemented in the application tier. A tighter integration planned with the database kernel aims to reduce this overhead considerably with the use of internal drivers for capturing the DML activity on the tables and for maintaining the business transaction journal.

5. CONCLUSIONS

The paper discussed a flexible business transaction framework with the support for compensation-aware data types. This framework leverages from the traditional transaction management, to support nested and distributed long-transactions with the added ability to synchronize across transactions. A compensation-aware data type allows domain-specific logic for validating concurrent shared update operations and for performing compensation when needed. Reusable data type definitions may be developed using an extensibility framework and they can be customized for each use with declarative operational constraints. Shared update operations validated using the embedded data type logic and the operational constraints ensure that they are semantically serialized and that some deterministic compensation exists for these operations. The flexible transaction framework, the encapsulation of well defined shared update semantics into the data type implementation, and the ability to model each data manipulation operation as a unit of compensation makes this approach viable for wide-range of applications. The ability to apply shared update and compensation semantics to traditional data types allows legacy applications to make use of this improved databases functionality with minimal application changes.

6. ACKNOWLEDGEMENTS

Our thanks to Jayanta Banerjee, Peter Fischer, Donald Kossmann, and Garret Swart for many helpful discussions and their support.

7. REFERENCES

- [1] Korth, H. F., Levy, E., and Silbershatz, A. "A Formal Approach to Recovery by Compensating Transactions", *Proceedings of 16th VLDB Conference, 1990.*
- [2] Bernstein, P. A., Hadzilacos, V., and Goodman, N. "Concurrency Control and Recovery in Database Systems", *Addison Wesley Publishing Company, 1987, ISBN 0-20110-715-5*
- [3] Gawlick, D., and Kinkade, D. "Varieties of Concurrency Control in IMS/VS Fast Path", *Tandem Technical Report, June 1985.*
- [4] O'Neil, P. E "The Escrow Transaction Model", *ACM Proceedings of Transactions on Database Systems, Vol 11, No. 4 Dec 1986.*
- [5] Garcia-Molina, H., and Salem, K. "Sagas", *Proceedings of ACM-SIGMOD International conference on Management of Data, 1987.*
- [6] Chrysanthis, P. K. and Ramamritham, K. "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior", *Proceedings of ACM SIGMOD International conference on Management of Data, 1990.*
- [7] Barga, R. and Pu, C. "A Practical and Modular Method to Implement Extended Transaction Models", *Proceedings of the 21st VLDB Conference, 1995.*
- [8] Biliris, A. et. al, "ASSET: A System for Supporting Extended Transactions", *SIGMOD Record 23(2) June 2004.*
- [9] Clark D. D, and Wilson, D. R. "A Comparison of Commercial and Military Computer Security Policies", *In Proceedings of IEEE Symposium on security and Privacy*, pp. 184-194, 1987
- [10] Oracle Database Object Relational Developer's Guide, 11g Release 1, Oracle Corp., *Part# B-28371-03 Jun 2007.*
- [11] Oracle Database Data Cartridge Developer's Guide, 11g Release 1, Oracle Corp., *Part# B-28425-03 Jun 2007.*