Consequences of Operational Transformation

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

This article discusses some of the consequences of using Operational Transformation as a basis for replications and synchronisation of data in CEDA.

1 Overview

The Operational Transformation (OT) technique [12] is quite radical in its approach to concurrency control in a multi-user system. Before comparing it with the pessimistic locking approach used by most of today's DBMS products we first review the main characteristics of these conventional systems.

2 Conventional DBMS systems

The conventional approach to multi-user data management normally involves a central server which contains all the data, and many thin clients that only cache small amounts of data on the client machines. All changes to the database are made with *transactions*. A transaction provides the ACID properties:

- *Atomicity* : the transaction will either be performed in its entirety or else not at all.
- *Consistency* : transactions must comply with all integrity constraints defined on the schema.
- *Isolation* : transactions performed concurrently by different users behave as though they were in fact performed one after the other (this is referred to as serialisability of transactions). In other words the changes made by transactions are not visible to each other until after they commit.
- *Durability* : when a transaction commits, the changes will be made durable (i.e. won't be lost, typically by flushing data to secondary storage).

Most databases today use *Two Phase Locking* (2PL) in order to ensure serialisability of the transactions. This means that variables are protected with locks and a transaction must lock a variable before access is permitted. According to 2PL, for each transaction there are two phases. In the first *expanding phase* locks are granted but not released. In the second *shrinking phase* locks are released but not granted. It can be shown that 2PL is sufficient to guarantee serialisability of transactions.

The most common approach is to use *strong strict 2PL* (SS2PL). The expanding phase occurs throughout the execution of the transaction and the shrinking phase (releasing both shared read and exclusive write locks) is performed at the end when the transaction commits or aborts. Unfortunately there is inevitably a risk of *dead-lock* scenarios (also called the *deadly embrace*) where two or more transactions cannot proceed because each transaction is waiting on another transaction to complete in order to release a lock that it requires. The only way to break out of the dead-lock is to select a so called victim which is aborted - meaning that all changes must be rolled back.

This strategy is also called pessimistic locking because transactions are blocked on access to a variable on the (pessimistic) assumption they would violate isolation rules. This can be contrast to optimistic locking where a transaction is allowed to proceed to its end on the (optimistic) assumption that conflict won't be a problem [1].

Unfortunately there are some serious downsides to this approach:

- A centralised remote server means that clients tend to be exposed to network latency during a client side executed transaction. Synchronous messages over the wire have an overhead that is easily a *million* and may even approach a *billion* times greater than in-process function calls.
- Lock contention can occur, limiting scalability. This is where many transactions require the same lock. Even if locks are held for a very short time, it creates a bottleneck that's analogous to a multi-lane highway feeding into a single lane bridge.
- Long term blocking occurs when a transaction holds a lock for an extended period, blocking all other transactions that need access to that resource. For example, this could arise if a long term transaction is held while a user edits information in a dialog window. This has the advantage of isolation for users, but also can prevent concurrent data entry - perhaps in unexpected ways (as can happen with page level locking) or if transactions access common data structures.

In practice transactions may need to be executed server side to avoid exposure to network latency. This imposes constraints on client side transaction logic.

Fortunately, the Relational Model allows for set based processing: A client can issue a query (e.g. in SQL) and the set based query evaluation is performed entirely on the server, greatly reducing the number of synchronous messages between client and server.

2.1 Distributed transactions

A distributed transaction operates on multiple physically separated databases. The challenge is to develop an *Atomic Commit* (AC) protocol where all databases either agree to commit the changes or not, even in the presence of failures such as lost messages or network partitions.

In the *Two Phased Commit* (2PC) protocol each participating database is asked by a coordinator to "prepare to commit". A participant can either vote "yes" to indicate readiness to commit or "no" to unilaterally abort. Before voting yes it must flush log records to secondary storage. In the second phase the coordinator only commits the transaction if all sites voted yes. The coordinator sends either a "commit" or "abort" message to each participant and the participants responds with an "ack" [2].

If a site fails and needs to perform a recovery, any locks that were held as part of a prepared distributed transaction need to remain locked until it is known whether the transaction is to be committed or aborted.

A *blocking protocol* is where a site may be forced to block indefinitely while it waits for another site to recover and become accessible again. 2PC is especially vulnerable to failure of the coordinator. When that occurs, participants must block indefinitely until it recovers [3]. A major problem is that any locks acquired by the transaction cannot be released. Since 2PC is not resilient to a random single site failure, a *Three phase commit* (3PC) protocol was proposed. However 3PC is more costly and ultimately still subject to blocking under certain failure conditions, and therefore not commonly implemented.

It turns out that no protocol exists that is safe in the presence of network partitions when messages are lost [3]. This is rather discouraging.

Since distributed transactions involve synchronous messaging on potentially high latency and unreliable networks, it is not surprising to see comments like the following:

> It comes as a shock to many, but distributed transactions are the bane of high performance and high availability [6]
>
> ...Yes they're evil. But are they a necessary evil – that's the real question. ...I believe they are indeed the worst sort: necessary. We're all doooooooomed..... [7]
>
> ...a number of companies haven't a clue how much danger they are in. They trust to 2-phase transactions and independent backups. They spend huge sums of money on HA paired servers to try and make the physical system failure-proof. At the end of the day, however, it is impossible. Nature finds a way to make your life miserable, and all because of these horrid distributed transactions. [7]

2.1.1 Single Source of Truth

Conventional database systems emphasise the concept of a *Single Source of Truth* (SSOT) for every data element. The most straightforward way to achieve this is to store every data element exactly once. This aligns well with conventional client-server architectures - i.e. where the SSOT is enforced by storing all data on a *single* server.

2.1.2 Distributed transactions on replicated data

A distributed database system can replicate data and yet comply with the SSOT principle by applying updates atomically to all copies using distributed transactions. One advantage of replication is for load balancing read only queries across the available copies. The isolation property of transactions implies that other transactions cannot see divergence in the replicated data.

However this approach is impractical and mostly only of academic interest. The problem is poor performance. A non-distributed system able to achieve tens of thousands of short update transactions per second will slow to perhaps hundreds of transactions per second when using replication and distributed transactions [5].

2.1.3 Commitment ordering

Fortunately S-S2PL together with Atomic Commit guarantees global serializability in a distributed system, without the need to share concurrency control information between databases apart from that needed for the AC protocol.

In [4] a new concept called *Commitment Ordering* is introduced which generalises this result, providing alternative ways to ensure global serializability. This may provide dead-lock free executions, although the price to pay is cascading aborts. Unfortunately cascading aborts is a very significant problem.

2.1.4 CAP theorem

The CAP theorem [8] states that in general it is not possible to have all three of the following at the same time in a shared data system:

1. **C**onsistency;
2. **A**vailability; and
3. **P**artition tolerance.

If there are no partitions then consistency and availability is easy to achieve - such as in a centralised database. Also systems that run on LANs often assume there will be no network partitions.

Distributed databases may make minority partitions unavailable in order to achieve consistency and partition tolerance. Atomic Commit protocols such as 2PC make some nodes unavailable (i.e. block indefinitely) when the network partitions. A message forwarding proxy to a centralised database inherits the consistency and availability of the central server, but the proxy necessarily becomes unavailable when connectivity to its central server is lost.

In [9], consistency is formalised as *atomic consistency*. A distributed object must behave as though all operations are performed in some total order and each operation looks as though it is completed in a single instant.

OT rejects atomic consistency and instead only aims to achieve consistency at quiescence. Since consistency can *eventually* be achieved, OT is able to achieve both availability and partition tolerance (which are conventionally incompatible). Therefore OT can be seen as a way to sidestep the very discouraging limitation inherent in the CAP theorem.

2.2 Conflicts

The isolation requirement on transactions is associated with their serialisability. 2PL implies conflict serialisability which implies view serialisability. Here, a conflict relates to non-commutative access such as two writes or a read and a write to the *same variable*, not to higher level semantic dependencies that may hold across variables. Therefore it is misleading to say that 2PL eliminates all manner of conflicts in multi-user data entry applications.

In this article the terms *syntactic conflict* and *semantic conflict* by definition refer respectively to low and high level conflicts depending on whether they can be avoided automatically by the database system (typically using 2PL as well as integrity constraint checking which can cause a transaction to be aborted). Therefore, we say that the database system only prevents syntactic conflicts.

Syntax relates to formalised integrity constraints. A database schema (i.e. datatype definition) imposes constraints by specifying the possible values that may be recorded in the database. By contrast semantic constraints are not enforced, perhaps because they are too difficult to formalise.

Consider a centralised database recording source code files edited by a group of programmers using client applications over a network. Let the files be locked pessimistically with 2PL. We will use this example to compare short lived and long lived transactions.

2.2.1 Short lived transactions

With short lived transactions associated with individual, fine-grained edits the users will be able to see each other's typing in real-time (subject to network latency). As a result of brief locking of files, characters inserted into a file cannot be lost because the system avoids dirty reads and lost updates.

This has the advantage of supporting concurrent and highly interactive work. It is well suited for users collaborating on the *same task*. However, users working on more independent tasks require *isolation* so they can go through the edit-compile-run cycle independently. Having another user's edits appear can be a serious impediment when they are irrelevant to the task at hand.

Unfortunately 2PL inevitably exposes the users to network latency – even with sophisticated algorithms that cache data and retrieve locks optimistically. For example, a user may sometimes notice a few hundred milliseconds delay between hitting a key on the keyboard and seeing the inserted character appear on the screen. Research has shown that a latency exceeding about 150 msec negatively impacts productivity because users instinctively slow down, perhaps because they are less sure that they have typed correctly [11].

When multiple users edit the same file there can be problems because the displayed text or their caret position jumps around. On the positive side, semantic conflicts tend to be reduced because users become aware immediately of what other users are doing, allowing rapid decision making that avoids incompatible or wasted work. However, that may not happen if another user's edits don't happen to be visible on the screen, such as when they edit a different file – i.e. the introduction of higher level semantic conflicts goes unnoticed. For example, two users may inadvertently add a function with the identical name in different files, causing an error that is only detected later by the linker.

With short lived transactions the system cannot enforce strong integrity constraints. That isn't necessarily a problem – evidently programmers work quite well with editors that happily persist source code files that don't compile. Instead, integrity constraints can be checked *manually* when a user compiles the code or runs a unit test and uses the reported errors to make ongoing improvements and corrections.

2.2.2 Long lived transactions

Consider that the database enforces complex integrity constraints (e.g. source code must compile, link and perhaps even pass some user-defined unit tests) when each transaction commits. These constraints cannot be applied during a transaction because the user wouldn't be able to insert or delete individual characters with a text editor.

Long lived transactions are necessary to move the data atomically from one valid state to the next. Transactions are very expensive to commit because of the high validation cost, but that may not be a problem because a transaction is typically associated with a high level task and the rate at which these tasks are performed is very low (e.g. only a few per day).

Integrity checking can be made very convenient by highlighting parts of the source code that contain errors, similarly to word processing software that highlights spelling and grammar errors within the editor. This idea is appropriate to many other domains such a electronic circuit design, CAD etc.

When a transaction fails to meet integrity constraints, the conventional approach is to abort it and roll back all changes. That is hardly appropriate if a user has spend hours or days on a complex task. Aborting a transaction also seems inappropriate in a dead-lock scenario - say when two users have each locked a file needed by the other user to complete their task.

Long lived transactions often prevent concurrent work - i.e. a user may be blocked from editing a file because it has been locked for hours or days by another user. On the upside this reduces the risk of conflicts – however it only does so to the extent that it prevents concurrency in the first place. Even so semantic conflicts are still possible despite locking, when users change *different* files where there are semantic dependencies between them. In [10] it is pointed out that locking is restrictive and often becomes a roadblock for users:

- Locking may cause administrative problems: A user may lock a file and then forget about it or go on vacation, preventing other users from doing work. An administrator is needed to release the lock.
- Locking can cause unnecessary serialisation: E.g. locking a single text file serialises all edits to that file, even though the edits may be quite independent.
- Locking can give users a false sense of security, because they may assume that locking ensures they can make changes without introducing incompatibilities, and therefore don't communicate with each other adequately.

Very strong integrity checking can detect many more conflicts. However, it typically does so by accessing a very large proportion of the data in the database. For example the validation of a transaction on source code may require access to *all* source code files in a project to verify the linking step. The effect is that a transaction cannot commit until it has (read) locked every single file! This makes dead-lock almost inevitable because when a transaction is about to commit it will require read access to all files that are currently accessed with exclusive locks by other users. The only solution is to disallow concurrent development completely.

2.3 Locking versus merging

As a result of the problems with 2PL, most developers prefer a form of optimistic concurrency control to manage their source code – using automated merging to deal with concurrent editing of the same text file. For example, Subversion is a typical *Revision Control System* that promotes edit and merge workflows. Many RCS products (e.g. Bazaar, CVS, Git and Mercurial) don't support exclusive locks at all.

The main reason for supporting locking is to deal with *binary files* (e.g. images or videos) because adequate merge tools don't exist.

An interesting question is whether other forms of data (besides text files) would also benefit from edit and merge workflows. For example it is expected this would be valuable for CAD teams. However a software tool supporting merge on CAD models is much more challenging that the case of simple text files.

3 Data replication in CEDA

CEDA allows for decentralised data management within a peer-to-peer network. Clients or peers on the network store a completely independent copy of the data. The data is *replicated*, and there is no need for a centralised server.

3.1 Operations

Data replication depends on the ability to synchronise the replicated data asynchronously by sending database changes (called deltas or *operations*) in the background. Operations can be serialised as a stream of bytes - which means they can persist in the database or be sent over the wire.

In principle operations can be calculated as a difference between two database states. However that is inefficient compared to a system that generates the operations directly from the user edits. The disadvantage is that it is more difficult to convert existing single user applications to use CEDA because it is necessary to hook into the edit functions rather than simply the loading/saving of the data.

Individual edits to the data are represented as distinct operations. For example, as a user types on a keyboard, every character inserted or deleted is represented as a distinct operation. Similarly as a user manipulates an object with the mouse, every mouse move event may generate a distinct assignment operation. This might generate 50 operations per second.

3.2 Operational transformation

Operations are sent asynchronously between peers on the network in order to synchronise the replicated databases. Operations are transformed as required so they may be applied in different orders at different sites whilst ensuring the distributed databases reach the same final state and the original intentions of the operations are preserved.

Keys features include:

- Avoids pessimistic locking of data and the complexity and pitfalls with distributed locking, dead-lock detection, and multiphase commit protocols.
- True peer-peer based replication
- Synchronisation of peers only sends relevant deltas and is extremely fast and efficient.
- Local editing operations are always applied immediately without being delayed or blocked - even in the presence of network failure.
- Robust to network outages, dropouts or changes to network topology.
- Editing of complex data models off line and merging changes afterwards.
- Allows efficient tagging, branching and merging (i.e. configuration management) of the entire database.

Developers that have used configuration management tools such as CVS or Subversion are familiar with the idea of automated merging of concurrent edits to text files. OT is very similar in nature except that it is based on a more robust mathematical approach. For example it is not line based and it can correctly merge edits on a single line of text. Furthermore, unlike Subversion, CEDA utilises OT for merging complex data models encoded in binary form.

There are three requirements that must be met by the system using OT [13]:

1. All sites must converge at quiescence - meaning that when all operations have been applied to all sites, the replicated data will have reached the same final state.
2. Operations should preserve their original intentions
3. Causal ordering relationships between operations must be respected.

The intuition behind OT is comparatively easy. The implementation however is not! For example, in the literature there is a defined total ordering on characters in a shared text document called the *effects relation* [14]. This makes it straightforward to see how all sites can in principle converge at quiescence. Actually implementing an efficient, correct solution is not so easy. There are subtle problems that only become apparent when there are three sites or more.

3.3 Interactive collaboration

Real-time, interactive collaboration occurs when a group of users all edit some replicated data and the locally generated operations are sent to peers over the network so that users see each other's edits in real-time. For example, a user may see characters appear in a paragraph as they are being typed by another user.

CEDA is efficient enough to allow hundreds of users to interactively collaborate on the same data, sending, receiving, transforming and applying many thousands of operations per second. Operations are *streamed* such that throughput is constrained by network bandwidth, not latency.

Local operations are applied immediately when they are generated – i.e. always independently of network latency. This guarantees that the application is always as responsive as a single user application. Therefore the problem of delays affecting data entry (discussed in section 2.2.1) never occurs in CEDA. This is probably immaterial on a reliable LAN, but for fine grained interactive collaboration amongst users on opposite sides of the globe it is a significant improvement over any locking scheme that exposes the user to network latency.

If the network fails then all users are able to continue working. The only thing a user notices is that edits by other users appear to stop. Later when the connectivity is reestablished merging will be performed efficiently and reliably. The only potential issue is the risk of semantically incompatible or wasted work.

As stated in section 2.2.1, interactive collaboration is only appropriate when users work closely together on the same task. Therefore this mode of real-time data synchronisation is *optional* in CEDA. In fact it is expected that most of the time users will require isolation as they work on distinct tasks.

3.4 Configuration Management

CEDA supports Configuration Management in a similar way to RCS tools like Subversion. A repository (either local or remote) records any number of branches of the data. A user typically edits a local *working copy* in isolation and chooses when to perform an *update* from a repository which will merge changes into the local working copy. Alternatively the user can *commit* changes to a repository. Distributed repositories are supported such as in Git or Mercurial.

CEDA is able to enforce linearity of a given branch even though users are able to work on tasks in isolation. This involves separately supporting updates and check-ins, and not allowing a check-in to be performed without a prior update.

This also provides an effect similar to transactions. If users only check-in changes that meet strong integrity constraints then the check-ins can be seen as moving the database atomically from

one valid state to the next. This makes this comparable to long lived transactions as described in section 2.2.2.

Distinguishing local working copies and repositories, allows CEDA to get both the benefits of short lived transactions (on a working copy) and long lived transactions (on a repository).

3.5 Changes to network topology

CEDA allows the computers to reconnect in an entirely new topology and operations are exchanged as required so they always *converge at quiescence*.

3.6 Site identifiers to break symmetry

It is assumed that each site is uniquely identified by a *site identifier* and there is a system wide total ordering on the site identifiers. This total ordering is used to allow all sites to agree on how to merge concurrent edits whenever there is an ambiguity due to *symmetry*. In CEDA site identifiers are 128 bit guids.

3.7 Merge using OT

Under OT a merge of concurrent operations *always succeeds* and the result is always well defined. Therefore a merge can always be performed automatically without human intervention.

Syntactically the result of a merge is straightforward and doesn't do anything nasty like put the database into a corrupt state, or somehow lose or mess up large amounts of work from one or more users. It is perhaps worth pointing this out to users that are skeptical of automated merging (which for example is quite common with developers that have never used RCS tools before)

3.7.1 Values and variables

A *value* is abstract and mathematically defined, eternal, immutable and decoupled from the computational machine (i.e. doesn't exist in time or space). An example of a value is the integer 5. It is immutable meaning that it can't be modified - e.g. it makes no sense to say that the value 5 can be changed to become the value 6. A value is eternal meaning that there is no point in time where it sprang into existence or later ceases to exist.

A *datatype* is a set of values plus a set of read only operators on those values. Like values, a datatype is abstract and mathematically defined, eternal and immutable and doesn't exist in time or space. An example is the set of integers with the arithmetic and comparison operators.

A *variable* is a holder for the appearance of an encoded value and is regarded as existing in time and space as part of a computational machine.

An *update operator* can be applied to a variable to modify its value. Examples are direct assignment, insert and delete operators on string variables, and increment and decrement operators on integer variables.

A variable has a *vtype* which specifies both the variable's datatype (which defines the set of possible values which the variable can hold) and the available update operators on the variable.

Different vtypes may exist for the one datatype. For example for the signed 32 bit integer datatype, one vtype may support assignment and another may only support increment and decrement operators. This influences the way merging is performed under OT, since OT is concerned with intention preservation of update operators.

The application programmer specifies vtypes when defining a database schema in CEDA. It is important to understand the various repercussions when alternative vtypes are available for the same datatype.

3.7.2 Information content

The amount of information in a value of a given datatype can be quantified using entropy which is the logarithm of the number of possible states, and this relates to the amount of storage space required to encode the value. This makes it meaningful to speak informally about low versus high entropy values. For example it requires a lot less space to record a 32 bit integer than a typical CAD drawing or text document.

Input devices like the mouse and keyboard can only generate information at a limited rate. It is assumed that individual key presses and mouse events can be used to generate small operations, and it takes time to accumulate enough operations to specify a value with high entropy. This should be kept in mind when considering the appropriate vtypes and hence update operators for a given datatype.

3.7.3 Assignment

Every datatype has an associated vtype that supports direct assignment to the variable. Obviously concurrent assignments of different values to a variable always conflict and there is nothing that can be done about that. i.e. it isn't possible to preserve the intention under OT.

The situation isn't as bad as it first appears. Assignment by its very nature is lossy and so it's actually reasonable for merging to be lossy as well. Assignment only tends to be used for simple datatypes that are edited using GUI controls that generate assignment operations with minimal effort from the user. For example, in the CEDA jigsaw demo application, simply moving a jigsaw piece can be made to generate an assignment operation.

By contrast it wouldn't normally be reasonable to use assignment on a datatype that records a value with high entropy, that is very expensive to edit (like a CAD drawing or a text document). Under merging this would arbitrarily keep the edits from one user and drop everyone else's contribution. In any case, as noted in section 3.7.2, input devices like the mouse and keyboard cannot generate high entropy assignment operations in the first place.

As a result of symmetry the system needs to pick one of the assignments on a rather adhoc basis. The total ordering on site identifiers is used to pick a unique "winner" amongst all competing sites. This winning assignment dominates all concurrent and causally preceding assignments. The total ordering on the site identifiers can be seen as an imposed ordering on the concurrent assignments, where all the "loser" assignments are deemed to have been applied before the "winner" assignment.

The lossy characteristic of assignment operations is an important consideration in their use. Curiously there are situations where it is exactly what is needed! The CEDA jigsaw demo application provides a good example of where assignment operations work well in practice.

3.7.4 vtypes involving subvariables

An *accessor operator* on a variable returns a reference to another variable called a *subvariable*. Updates on the subvariable indirectly cause updates on the containing variable.

A database is essentially a single variable. Accessor operators allow a database to be regarded as being composed from subvariables. Accessor operators often compose, allowing one to drill down through nested subvariables.

In each of the following four cases, there is a sense in which values of that datatype consist of a fixed set of elements that are uniquely identified (i.e. indexed) with immutable keys in some fashion:

1. *Tuples*: A tuple type is parameterised by a set of attributes where an attribute has a name and a type. The names are unique. A tuple variable records the values of all the attributes.

2. *Fixed sized arrays*: An array type is parameterised by the size of the array and the type shared by all the elements. Elements are indexed by integer position.

3. *Dynamic arrays*: A dynamic array is like a fixed size array except that it can grow arbitrarily large. It is formalised as an infinite mapping from the natural numbers to the element values. The array's type is parameterised by the type shared by all the elements and an initial element value. When an array is created, all the element values have the initial element value. The implementation only physically stores a finite prefix of the array, assuming that all elements in the infinite suffix have the initial value. The stored prefix can grow or shrink over time.

4. *Maps*: A map associates every key value (of the given key type) to an element value (of a given element type). As for dynamic arrays, the map type is also parameterised by an initial element value, and an implementation avoids recording elements that have this special value. This allows a map to be used when the cardinality of the key type is very large or infinite. Note that logically there is no concept of inserting or deleting elements even though physically the implementation will do that. The CEDA implementation persists a map using a B+Tree.

For each of these four cases, there are two possible vtypes that are available:

1. One supports (and only supports) direct assignment to the entire variable as described in section 3.7.3. The elements do not represent subvariables and it is only necessary to specify the types of the elements as datatypes (not vtypes – because the elements are not independently updateable as variables).

2. The other doesn't support any update operators that directly modify the variable including direct assignment. Also, there are no operators for insertion and deletion of elements. Instead all updates can only be performed indirectly by using an accessor operator parameterised by an immutable key value to retrieve a reference to an *element* as a subvariable. The vtype of the tuple, array or map is parameterised in the vtype(s) of its elements. This in turn affects what update operators are available on the elements when they are treated as variables. It is assumed that preserving the intention of the operation means we preserve the identity of the element that is updated. Therefore the key value used in the accessor operator to index the element doesn't require transformation under OT. It is also assumed that the subvariables are independent so therefore merging is performed on each subvariable independently.

3.7.5 Vectors

The vector and array vtypes represent the same underlying datatype – a list of element values indexed by ordinal position. However as vtypes the update operations are different. A vector can grow and shrink using insert and delete operations whereas the size of an array is fixed. The elements of a vector are immutable - i.e. they cannot be modified whereas the elements of an array are treated like subvariables with their own update operators. The vector vtype is well suited to recording text (where the elements are characters) that can be edited by multiple users and supports merging of insertion and deletion operations.

In the case of an array, an element is forever identified by a fixed index position. By contrast, in a vector an element is considered to have an index position which may change over time. A given element is inserted by a particular user at some point in time and may shift left or right as other elements are deleted or inserted to the left of its current position. Eventually the element may be deleted (and once deleted that element is gone forever).

Consider the following definitions of *intention* of insert and delete operations:

- *Insert*: The inserted string must appear after the elements on the left and before the elements on the right of the insertion position.

- *Delete*: The elements to be deleted must not appear in the result.

Insertions don't conflict with other insertions – i.e. it is always possible to preserve the original intentions of every insertion for every user. It is easy to simply take the union of all the insertions in the result – for each insertion preserving both the string to be inserted as well as the insertion position relative to the elements that existed at the time the operation was originally generated.

The only ambiguity exists with insertions at identical positions. For example consider a string variable that is initially empty and one site inserts "x", while another site concurrently inserts "y" at the same position. At quiescence should the string variable become "xy" or else "yx"? CEDA makes use of totally ordered site identifiers in order to break this symmetry and ensure all sites agree on the order of the insertions.

Elements inserted by one user cannot be concurrently deleted by a different user (they can't delete what they haven't got). Therefore a concurrent delete never interferes with the preservation of an insert, and vice versa.

Finally deletes don't conflict with other deletes. Deletes are preserved by deleting the *union* of all the deletes in the merge result. An element can be concurrently deleted any number of times – in the merge result it is deleted only once.

So both these intentions can always be preserved under merging. In that sense operations on vectors *never conflict* (where syntactically the vector is modeled as just an ordered sequence of elements with no higher level semantics).

The merge result can be expressed as *the union of all the inserted elements minus the union of all the deleted elements*. This is simple and intuitive for users to understand, which means they can easily predict the result of a merge and there will be no surprises.

3.7.6 Sets

Consider a vtype for a set in which the elements are immutable. Therefore the only way the set can change over time is to insert new elements and remove existing elements.

We use the following definitions of *intention* of insert and delete operations:

- *Insert*: The element to be inserted must appear in the result.

- *Delete*: The element to be deleted must not appear in the result.

Note that inserts never conflict with inserts, and deletes never conflict with deletes. Inserts and deletes only conflict on the same elements. In CEDA we resolve this conflict by giving precedence to the insertion. i.e. insertions dominate concurrent deletes.

3.7.7 Bags

A bag is like a set except that the multiplicity of each element value is recorded. The vtype for a bag offered in CEDA considers the elements to be immutable and the bag only changes though delete and insert operations. The following apply to elements of a given key value:

- In the face of concurrent edits we favour inserts over deletes
- Concurrent inserts are assumed to have inserted distinct elements. For example if one site inserts 3 elements and another site inserts 2 elements then after the merge we assume 5 elements were inserted.
- Concurrent deletes are assumed to have deleted as many elements in common as possible. For example if one site deletes 3 elements and another site concurrently deletes 2 elements then under merge we assume 3 elements were deleted.
- A delete can only remove an element that was inserted by an operation that causally preceded the delete. i.e. a delete can never be thought of deleting an element inserted by a concurrent insert.

3.8 Reject SSOT

A crucial difference with conventional systems is that CEDA doesn't subscribe to to the usual formulation of the SSOT principle. Putting it another way, every peer is regarded at all times as an independent and more to the point a *valid* version of the data. This reflects reality much more closely. If data is stored separately in space, why pretend it's the same data, and why pretend it can be updated simultaneously? The latter conflicts with Einstein's Theory of Relativity where no information can go faster than the speed of light. A distributed transaction tries to make all observers agree on the simultaneity of events distributed in space, which is like the old fashioned Newtonian view of global consistency and global time.

Rejecting SSOT allows each peer to continue operating no matter how badly the network becomes partitioned. i.e. availability and partition tolerance are both maximised. Peers are allowed to continue working because we don't impose an artificial constraint (SSOT) on the system.

OT allows users to make concurrent edits that are automatically merged without syntactic conflicts (but like locking, OT cannot hope to prevent semantic conflicts). This is achieved by allowing operations to be performed in difference orders at different sites, and yet crucially, all sites converge to the same final state. This agreed final state represents a kind of non-localised SSOT (admittedly an oxymoron)! Interestingly this has an analogue in Relativity where the absolute ordering of space-time events is only partial, being relative to the observer for events that are outside each other's future or past light cones, and yet there is an underlying, shared reality (spacetime).

A *vector time* can be used to unambiguously identify any causally valid version of the data, even though it is shared and edited by any number of sites without any distributed Atomic Commit protocols. One can define any SSOT one likes by simply specifying a vector time.

3.9 Reject global serialisability

Conventional distributed database systems emphasise *global serialisability* of transactions as a criterion for correctness. OT provides a very different perspective and completely rejects Global serialisability! Instead transactions are always generated locally in a way that only obeys *local serialisability*. Obviously a transaction is meaningful in the context of the local database on which the operation was originally generated. Under OT, operations are only propagated to sites in a way that preserves causality relationships between operations. An operation is only executed on another site in a context that includes all the operations that had been performed when the operation was originally generated. This helps to ensure that the operation will be meaningful in a different execution context when it is propagated and applied remotely. An important criterion for OT algorithms is to preserve the original intention of the operation. For example, if an operation deleted some characters from a string then that operation should always delete the same characters when it is applied on another site. The intention of an operation is by definition low level or syntactic, and cannot account for higher level semantic intentions because that is too difficult to formalise for a definition of the Inclusion Transformation – which is where one operation is transformed to an execution context that includes another concurrent operation, whilst preserving its original intention. The aim of OT is to support the merging of concurrently performed operations that don't really conflict according to the intentions of the users, even though the operations would be regarded as conflicting in a conventional database using pessimistic locking.

3.10 Reject distributed transactions

In the CEDA implementation each site persists all operations in its local Persistent Object Store with proper attention to atomicity.

As for conventional systems, transactions are used to parenthesise access to a database - particularly mutative operations. However access is always local to the database on the same machine and therefore avoids any effects of network latency or network partitioning. The main role of transactions is to define boundaries for atomicity and rather less to do with consistency, isolation or durability. In CEDA they employ *conservative 2PL*. The data is partitioned into coarse mutually exclusive *pspaces*. Each pspace has a single lock supporting exclusive write or shared read modes. All required locks on the applicable pspaces must be granted before a transaction begins. Locks are requested in a consistent total order so that dead-lock is impossible.

3.11 Applications where OT is inappropriate

OT is inappropriate where the data needs to be centralised because it maps very directly back to something that physically exists in the real world. In these case the SSOT principle is required for correctness.

For example, an airplane reservation system should be dealt with centrally and use pessimistic locking, because there is a real airplane with real seats on it, and double booking isn't a good idea. Replicated data synchronised using optimistic concurrency control allows for divergence, and hence allows for double booking of seats.

In the case of financial systems, when an ATM provides a cash withdrawal there is a transaction involving real money and somewhere the withdrawal is made on an account in a database. In a sense the account balance recorded in a physical database represents real money. Therefore transaction atomicity with respect to

the real world is required. This explains the need for transaction durability (the 'D' of ACID).

Financial systems also seem to need pessimistic locking. However It can be argued that it's possible for a distributed system to support transfer of money between bank accounts without any need for distributed transactions.

3.12 OT and integrity constraints

The mathematics of OT is subtle enough that numerous incorrect algorithms have appeared in the literature for the simple case of edits on text ([14] discusses some of these). Fortunately OT solutions exist on a wide enough set of datatypes to make it practical in general.

OT imposes significant restrictions on what integrity constraints can be directly enforced on the editable data. In order to support strong integrity constraints it is generally necessary to impose them *indirectly*. Therefore the database only enforces comparatively simple syntactic validation on the editable data and the stronger integrity constraints that are incompatible with OT are treated as semantic constraints. These can be pinpointed or highlighted in a data entry application - either by having the user manually run a validation check or else having the validation check run continuously – say as a background task. This allows users to correct problems over time (and normally the elimination of all semantic errors would be a high priority).

This provides an elegant methodology to deal with merge conflicts. As we have seen, under OT, a merge always succeeds without syntactic conflicts. However, a merge could easily cause a semantic conflict. Such merge conflicts could be automatically pinpointed, highlighted then corrected by a user with a suitable software application. It is even possible for a group of users to take part in an interactive session aimed at fixing the merge conflicts.

There are two variations on this theme: Either the validated version of the data only becomes available once all errors and omissions have been corrected, or else it is possible at all times to *calculate* a pristine version of the data that obeys the stronger constraints, typically subtracting away the parts of the data that break integrity constraints. This appears straightforward in the Relational Model. For example,

- meet a foreign key constraint by throwing away records with a foreign key that cannot be resolved.
- meet a no-null constraint by throwing away records that have a null
- meet a uniqueness constraint by throwing away duplicates

An interesting example of using a semantic model on top of the syntactic one appears in section 3.13.

3.13 Jigsaw example

Since the meaning of OT can appear rather mysterious it is worth studying some specific examples to find out what it actually means in practice. Consider a CEDA jigsaw application using the following schema:

```
$tuple TPoint2d
{
  int32 X;
  int32 Y;
};

$tuple TJigsawPiece
{
  TPoint2d Position;
  bool JoinedOnRight;
  bool JoinedOnBottom;
};

$tuple TJigsaw
{
  int32 NumRows;
  int32 NumCols;
  TJigsawPiece Pieces[];
};
```

The completed jigsaw is comprised of pieces arranged in a two dimensional grid with the given number of rows and columns. The jigsaw piece at row `r` and column `c` has an index `i = r*NumCols + c`. Information about the pieces is recorded in a *dynamic array*, indexed by i. Each piece records its (x,y) position and two boolean flags for whether it has been joined to the piece immediately on its right or below as they appear in the completed jigsaw. Initially each piece has `JoinedOnRight = JoinedOnBottom = false` allowing all the pieces to have an independent (x,y) position on the screen.

The jigsaw application allows many users to concurrently move the pieces around. Occasionally two people may drag the same piece with the mouse in different directions, and everyone sees it rapidly flicker between the two alternative positions as incoming operations to a given computer successively reassign the (x,y) location. This is an example of a "conflict" that is visually obvious and its resolution simply involves one person giving up before the other.

The surprising feature supported by OT is that some people can work on the jigsaw off-line and it is always possible to silently merge in their work. At no time do dialogs appear asking users to resolve conflicts. This is achieved despite a model that keeps track of what pieces have been snapped together, and in fact the OT is applied in such a way that pieces that had been snapped together can never come apart after merging.

Although OT sometimes needs to make arbitrary decisions in order to resolve ambiguities because of symmetry, it is possible at a higher level to *interpret* the shared "syntactic" data in such a way that some activities (like snapping pieces together) take precedence over others (like moving pieces around). The trick is to realise that integrity constraints typically reduce the degrees of freedom in the model, and therefore there is an opportunity to resolve conflicts in favourable ways. In the case of the jigsaw example, the syntactic model records the (x,y) of every piece, as well as information about which pieces have been snapped together. The integrity constraint (which can be regarded as applicable in a "semantic model" of the jigsaw) means that some of the recorded (x,y) positions of the pieces need to be ignored! In a way, as the jigsaw is being put together the number of degrees of freedom in the semantic model is steadily decreasing. The OT itself only cares about the syntactic model which is directly expressed in the schema. In this case the high level semantic is expressed in the implementation of methods like `GetPiecePosition(int i)`, which returns the logical position of the ith jigsaw piece. The implementation simply offsets from the physical (x,y) of the top left most piece in any given set of connected pieces as a basis for calculating a logical position compatible with the integrity constraints.

Note that the semantic view of the data is not ncessarily read only. E.g. it is possible to implement a semantic version of SetJigsawPiecePosition(i,pos) that ensures that all pieces in the group containing piece i move together as expected. This means it syntactically needs to assign the position of the top left piece in the group!

OO classes that decorate an underlying syntactic model are able to express a higher level model that is expressed in terms of

the lower level syntactic model.

In many respects the jigsaw example seems an unlikely candidate for OT, even though it works well in practice! The jigsaw model contains assignable variables that are implicitly lossy because they can only record their last assigned value. Under merging the system is forced to disambiguate by ensuring all sites agree on a unique "winner". The saving grace is that the cost of manual conflict resolution isn't commensurate with the original effort of assigning the value. Therefore users seem very happy to put up with lossy merges.

Dialogs reporting a merge conflict would be undesirable. With a jigsaw containing 500 pieces, it's simply not feasible to expect a user to resolve each and every merge conflict manually. It only seems reasonable to manually resolve conflicts on entered data that involved a lot of work in the first place. Moving a jigsaw piece takes too little effort to justify manually prompting the user to disambiguate a merge conflict.

In many examples (but not in the case of the jigsaw), data entry often involves creation of *new data*. For example, when a user works on a text document, most of the effort involves inserting new text. Therefore merging off-line work doesn't tend to discard edits - basically because the shared data can always grow to contain everyone's contribution.

3.14 Semantic conflicts

As far as actually locking out users from off-line work, the most important criterion is the level of semantic incompatibility (or else overlap) in the high level tasks, and the system typically cannot formalise that. It can't even necessarily determine whether substantial conflict exists during the merge! Consider the following users that are all working off-line doing some work on a shared text document:

User	Task
1	Restructure the chapters
2	Write the introduction
3	Write the conclusions
4	Write the introduction
5	Correct the spelling
6	Correct the grammar and diction
7	Insert figures

Under OT, after merging we end up with a reasonably faithful union of all their efforts. The biggest problem is that there will be two introductions. The problem is not so much with the integrity of the document - because it is easy to delete one of the introductions afterwards. Rather it is in the wasted effort!

It seems that in this case, locking in any shape or form will add nothing of value, and only interfere with the ability for the users to work in isolation. Furthermore, a locking protocol is unable to detect the semantic overlap in the high level tasks between users 2 and 4. In fact these two tasks are probably the least likely to create syntactic conflicts that could be detected by locking!

Most conflicts between users seem to occur at a semantic level that is inaccessible to the system. In a way locking implicitly accounts for that (because if the system was able to understand the presence of a conflict it would probably be clever enough to take corrective action as well). Instead locking simply forces users to take turns. Unfortunately there are many semantic conflicts that can go undetected, unless locking is at a very coarse level, but that comes at a heavy price - lack of concurrency.

A premise of CEDA is that in practice it is more important to detect integrity constraint violations than conflicts. To the extent that the presence of such violations can be *calculated* we have a more robust way of finding and isolating problems with the entered data. For example, in a database supporting software development, an enormous amount of validation is performed by the compiler. In addition good developers write extensive automated unit tests that make it very hard for bugs to go undetected in the formal releases.

References

[1] Wikipedia, *Concurrency Control*. `http://en.wikipedia.org/wiki/Concurrency_control`.

[2] J.N. Gray, *Notes on Database Operating Systems*, Operating Systems: An Advanced Course, Lecture Notes in Computer Science, Vol 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[3] D. Skeen, M. Stonebraker, *A Formal Model of Crash Recovery in a Distributed System*, IEEE Transactions on Software Engineering, Vol 9, Issue 3, May 1983, pages 219-228.

[4] Yoav Raz, *The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment*, Proceedings of the Eighteenth International Conference on Very Large Data Bases (VLDB), pp. 292-312, Vancouver, Canada, August 1992.

[5] Kenneth P. Birman, *Reliable distributed systems: technologies, Web services, and applications*, 2005, ISBN: 978-0-387-21509-9.

[6] Dan Pritchett, *2PC or not 2PC, Wherefore Art Thou XA?*, December 2006, `http://www.addsimplicity.com/adding_simplicity_an_engi/2006/12/2pc_or_not_2pc_.html`

[7] *Distributed Transactions Are Evil*, `http://c2.com/cgi/wiki?DistributedTransactionsAreEvil`.

[8] Eric Brewer, *Towards Robust Distributed Systems*, Keynote speech, ACM Symposium on the Principles of Distributed Computing, 2000 July.

[9] Seth Gilbert, Nancy Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT News archive, Volume 33, Issue 2 (June 2002), pages 51–59.

[10] Ben Collins-Sussman, C. Michael Pilato, and Brian W. Fitzpatrick, *Version control with Subversion*, O'Reilly Media, ISBN-13: 978-0596004484, June 2004.

[11] James R. Dabrowski, Ethan V. Munson, *Is 100 Milliseconds Too Fast?*, Conference on Human Factors in Computing Systems, ISBN:1-58113-340-5, pages: 317–318, 2001.

[12] Wikipedia, *Operational Transformation*. `http://en.wikipedia.org/wiki/Operational_transformation`.

[13] C.Sun, X.Jia, Y.Zhang, Y.Yang, D.Chen, *Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems*, ACM Transactions on Computer-Human Interaction (TOCHI), Volume 5, Issue 1, March 1998, Pages 63–108.

[14] Du Li and Rui Li, 2004. *Preserving Operation Effects Relation in Group Editors*, Proceedings of the ACM CSCW'04 Conference on Computer-Supported Cooperative Work, ACM Press New York, NY, USA. pp. 457–466.