Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 5 Issue 05, May-2016

Comparison of Concurrency Control and Deadlock Handing in Different OODBMS

Sonal Kanungo
Smt. Z. S.Patel
College Of Computer,
Application,Jakat Naka, Surat

Dr. Rustom. D Morena
Department Of Computer Science,
Veer Narmad South Gujarat University, Surat

***Abstract* - Object oriented databases are being widely used in variety applications of industries including telecommunications, banking, manufacturing, insurance, shipping, etc. These applications are characterized by having complex data model. Object databases are very good at storing the complex data model. Many applications can runs efficiently with many concurrent users. Transactions involving complex objects can continue for several hours, or even several days. Longer-duration transactions need to use different set of protocols from those that are used in traditional database applications where duration of transactions is very short.**

The purpose of this paper is to explore various techniques that can be used in object oriented databases to achieve high concurrency.

1. INTRODUCTION

Concurrency control is the technique used to maintain the consistency and isolation properties of transactions and it is required when two concurrent transactions try to simultaneously perform Read or Write operations on the same objects. Read consistency is defined as requiring all Reads in a transaction are performed against the same state of the database, while Write consistency guarantees that the order of the operations on objects in the database doesn't affect the final outcome.[13]OODBMS supports both pessimistic and optimistic concurrency controls. Traditional concurrency control protocols can slow down the overall system performance of an OODBMS.OODBMS has complex nature therefore complex concurrency control mechanisms are needed and also that this mechanisms should not affect the performance of OODBMS. Concurrency control mechanisms should improve the throughput of the system by improving concurrency and without affecting the performance, so that maximum number of transactions can run in parallel.

2. CONCURRENCY CONTROL TECHNIQUES IN DIFFERENT OODBMS

In this paper we are going to discuss various concurrency control techniques; their advantages and disadvantages in the following OODBMS.

1. Versant 8.0
2. GemStone
3. Objectivity
4. ObjectStore

2.1Versant Version 8.0

To maximize concurrent user access, Versant uses fine-grained object-level locking. Versant supports both pessimistic and optimistic locking protocols which allow the application to control the access. To ensure that objects in the database server are in sync with client access, Versant by default uses a pessimistic locking strategy, which is done by using a combination of locks against both schema and instance objects.[13]

2.1.1 Pessimist Locking In a multiple user environment, Versant's Read, Update and Write locks provide consistent and coherent access guarantee. Under most circumstances these guarantees are necessary and appropriate. Locking can be set at the object level but can also be applied to classes or instances. Versant gives different mechanism for long and short transactions. Persistent locks support long transactions which are often necessary in OODBM systems, while short locks are provided for shorter transactions as well as 'nolocks' are provided for optimistic locking.
The Versant system also provides the functionality to allow users to define their own locks. An entity is locked for the entire time in pessimistic locking approach. A Read lock indicates that the object can be read but cannot be modified or deleted by the other transactions. Only the current transaction can read, modify or delete the object if Write lock is there. This also indicates that the object is locked exclusively.
Versant also provides mechanisms for upgrading and downgrading locks on specific objects for writing on database. Ending a session ends the current transaction with either a transaction commit or rollback. Locks also have precedence. The precedence of locks is: Write>Update >Read>Null. If locks are compatible using multiple processes in the same session, different processes can place different locks. For example, different processes can request and receive a Read lock and an Update lock on the same object, but the net effect is the same to outside users, the highest lock prevails. Strict two-phase locking is used where locks are set before work starts rather than at commit time. Strict locking prevents other applications from modifying an object. Versant's implicit locking strategy can be overridden in several ways. Upgrading of locks, change of the default lock, and/or change the lock wait time, can be done explicitly. [13]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

Transactions involving changes in the definition of a class will require pessimistic concurrency control. In such conditions, exclusive locks are obtained not only on the classes being modified but also on all of their descendant classes. [12]Versant also supports IntentionalRead, IntentionalWrite, and Read/IntentionalWrite Locks. Intentional locks are relevant only to class objects. The purpose of intention locks is to prevent changes to class objects while you are using an instance of the class.[13]

2.1.2 Optimistic Locking Sometimes locking becomes too costly when we want to Read a large number of objects but update only a few of them. Or, there may be situations where there is only a small chance that the objects you want to work with will be updated by others. In such cases the optimistic approach is useful.[13]In the real world there are many systems where the collisions are not very frequent, optimistic locking will give reasonable solution. In optimistic locking it is accepted that collisions may occur, but instead of trying to prevent them, the system tries to detect and then solve them. While updating unlocked objects, optionally event notification methods are used. When objects are held with optimistic locks, this will provide a shorter duration of locking, which in turn will improve performance and improve concurrency of access among multiple users. Time stamp validation also checks at commit, delete, and group Write time. This will keep databases more consistent. [13]

2.1.3 Deadlock Handing

Deadlock problems can be avoided by automatically denying locks that would result in a deadlock situation. When the deadlocks are detected, a 'would-cause-deadlock' error is sent to the user immediately. For example, if two users have Read locks on an object and then both request a Write lock, the first user to request a Write lock is blocked, and the second user receives a 'would-cause-deadlock' error message. Versant directly handles complex, single-database deadlocks with any number of clients and any number of locked objects. An example of a *complex deadlock* is "A waiting for B waiting for C waiting for D waiting for A". Versant uses timeout mechanisms to detect multiple database deadlocks. [13]

2.2GemStone

GemStone provides both approaches of optimistic and pessimistic locking for managing concurrent transactions. The default mechanism in GemStone is optimistic concurrency control. An application can use either or both approaches. A transaction that fails to commit leaves the user's workspace intact with all of the new and modified objects which it contains. The user may then either take some action to save the values of those objects in a file outside of GemStone, or abort the transaction altogether. [9]When you tell GemStone to *commit* the current transaction it tries to merge the modified objects in your view with the shared object store. Conflict sets are cleared at the beginning of a commit or abort. *Write-Write* conflict occurs when same object in your Write set is also in the Write set of another transaction. Write-Write conflicts can involve only a single object. An object in your Write set is also in another session's dependency list - a *Write-dependency conflict*. An object belongs to a session's *dependency list* if the session has added, removed, or changed a dependency for that object. The transaction cannot commit if a Write-Write or Write-dependency conflict is detected. This mode allows an occasional out-of-date entry to overwrite a more current one. [15]

*2.2.1 Optimistic concurrency control :*The method is "optimistic" that means mechanism is "hoping" that conflicts between transactions will not occur. [12] Optimistic protocol can give greater degree of concurrency where probability of conflicts is low. A transaction means that you simply Read and Write objects as if you were the only session. Gemstone detects conflicts with other sessions only at commit.[13] Each GemStone session manages a local copy of the object store and adds changes made to the end of a transaction. This process is called optimistic locking. [9]Optimistic concurrency control is easy to implement in an application, but when conflicts are detected, commit does not take place and there is always a risk of re-do the work which is done already. [15]As GemStone looks for conflicts only at commit time, the chances of being in conflict with other users increase with the time. Under optimistic concurrency control, GemStone detects conflict by comparing your Read and Write sets with those of all other transactions committed since your transaction began. [13]

2.2.2 Pessimistic concurrency control: When there is a lot of competition for shared information in an application, Pessimistic locks are best choice. [15]This helps to prevent conflicts as early as possible by explicitly requesting locks on objects before modifying them. When an object is locked, other users will be unable to lock the object or commit changes to it. [13]With the pessimistic approach, conflicts are prevented by explicitly requesting locks on objects before modifying them. When an object is locked by another user, you can wait, or abort your transaction immediately. GemStone supports Read, Write and Exclusive locks. Locks are held across transaction boundaries and must be released explicitly. A session may hold only one kind of lock on an object at a time. You can remove the Read lock you currently hold on an object and then immediately request a Write lock. There is no facility to wait for a lock to be released which is hold by another transaction and therefore no deadlock detection is necessary. GemStone currently provides no built-in support for upgrading locks. [9]

Locking is possible for individual objects or collections of heterogeneous objects. GemStone treats that only one user is working in same session, it detects conflicts only at the time of commit. GemStone provides **Reduced-conflict classes** which can be used instead of their regular counterparts in applications with which too many unnecessary conflicts are experienced.
The class Counter encapsulates a number of the counter but it also incorporates other intelligence. The class RcIdentityBag provides the same functionality as IdentityBag, including the expected behavior for add:, remove:, and related messages. The class RcQueuegives

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

the functionality of a first-in-first-out queue, including the expected behavior for add:, remove:, size, and do:, which evaluates the block provided as an argument for each of the elements of the queue. The class RcKeyValueDictionary provides the same functionality as KeyValueDictionary, including the expected behavior for at:, at:put:, and removeKey.
Using these classes allows a greater number of transactions to commit successfully and in improving system performance. [15]

2.2.3Deadlock Handling
No matter how long you wait you may never succeed in acquiring a lock. GemStone does not automatically wait for locks so it does not attempt deadlock detection. It is user's responsibility to limit the attempts to acquire locks in some way. For example, when we Write in a portion of application there is an absolute time limit on attempts to acquire a lock. Or it let users know when locks are being waited and allow them to interrupt the process if needed. [15]

2.3Objectivity
In Objectivity/DB federated database, the objects are shared by sessions. If two sessions perform actions simultaneously one of the updates would be overwritten by the other update. Objectivity/DB use locking mechanism to prevent incompatible operations.[14]Containers are the fundamental unit of locking within Objectivity/DB. When any basic object in a container is locked, the entire container is locked. Objectivity/DB applies the session's concurrent access policy to determine the lock is requested for a container is compatible with any existing locks. Locks are granted by a lock server to ensure that data remains consistent. [14]
The transaction model supported is a pessimistic (locking) using Read, Write and Update locks and additionally by Multiple Readers One Writer (MROW) locks. The locks are implicitly provided in Read mode when objects are accessed in a transaction, but may also acquire explicitly. [14]
Objectivity/DB uses the standard two-phase commit protocol and provides database commit either to save objects to disk and free there sources or commit-with-hold that is acting as a checkpoint and holding resources or abort transaction.[14]
Versioning and application-defined version control is also available. Objectivity/DB stores each version as a separate object and users can simultaneously access multiple versions. In the versioning of a composite object, only the changed objects within the composite are versioned and any associations are automatically managed in this process. Work-group support is also provided through check-out and check-in.
Objectivity/DB automatically applies the locking and it is maintained until the application commits or aborts the transaction and after that all locks obtained during the transaction are released. When Objectivity/DB is not able to obtain a lock, the method that generated the lock request will throw an Exception. The mode of the session is set to wait for locks. Objectivity/DB records modifications to the object in a journal file. While the transaction is aborted or abnormally terminated, Objectivity/DB uses the journal file to restore a federated database to its previous state. [14]
Lock can be upgraded from Read to Write. You can downgrade all the Write locks to Read locks. Downgrading Locks is done by committing or aborting a transaction and releases *all* the locks held by the session. Check pointing makes all changes permanent and allows other applications to access those changes. However, the transaction remains active and all previously held locks are still held by the session. [14]

2.3.1 Exclusive concurrent access policy prevents any application user from being misled by viewing data that may be in the process of being altered by another session. If a session has a Read lock on a container, any other session may also obtain a Read lock on the same container, but no other session can obtain a Write lock on it. If a session has a Write lock on a container, no other session may obtain a lock neither Read nor Write on the same container. However, this policy becomes too restrictive. Such applications would rather access potentially out-of-date data, than not access the data at all.

2.3.2 Multiple Readers, one Writer (MROW) relaxes the restriction that a container may not be simultaneously updated and Read. Write locks are still mutually exclusive. If a session has a Write lock on a container, no other session may obtain a Write lock on the same container. A container can be updated by two sessions simultaneously with MROW status. MROW is used to allow sessions to Read the last committed or check pointed version of a container being updated by another session as well as a session can update a container if all sessions that are Reading the object obtained their Read locks with MROW enabled. Objectivity/DB has session with MROW enabled can get a Read lock for a container that is locked for Write by another session. A session can get a Write lock for a container that is locked for Read by another session with MROW enabled. A session can update a container that is being Read by an MROW session. An MROW session can Read a container that is being updated by another session.[14]

2.3.3 Deadlock Detection
When two or more sessions are queued, and each is waiting for a lock that will never become available leads to deadlock circular condition. If a deadlock condition is detected, Objectivity throws an exception. Whenever infinite lock waiting is requested, Objectivity/DB checks whether queuing the request would result in a deadlock situation. If the deadlock condition is found, an error is returned to the requesting application. When finite lock waiting is requested, no deadlock checking is done. In this case, Objectivity/DB assumes that any deadlock condition that occurs will be broken when lock waiting times out.[14]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

2.4 ObjectStore
ObjectStore provides facilities for multiple users to share data in a cooperative fashion. A user can check out a version of an object or group of objects ,and then project the changes so that they are visible to other members of the cooperating team. In the interim, other users can continue to use the previous versions, and therefore there is no concurrency conflicts on their shared data. [16]
Workspaces are used for accessing control to configurations. A configuration will be checked-out of a global workspace into a local one and then modified. When it is checked-in, a new version of the configuration will be created. [5]

2.4.1Strict two-phase locking: ObjectStore conforms to the strict two-phase locking. It gives the guarantee of serializability; that is, it guarantees that the results of the schedule will be just the same as the results of non-interleaved scheduling of the transactions' operations. [16]

2.4.2Multi-Version Concurrency Control (MVCC): This control performs non-blocking Reads of a database and allows another ObjectStore application to update M*ulti-version Concurrency Control* (MVCC) and performs non-blocking Reads of a database, allowing another ObjectStore application to update the database concurrently, with no waiting by either the Reader or the Writer. If a transaction only performs Read access on a database, and does not require a view of the database that is completely up to date, but can keep a snapshot of the data. Even though the snapshot might be out of date by the time some of the access is made, the multi-version concurrency control retain serializability. [5]
The database can be accessed concurrently, with no waiting by the Reader or the Writer. MVCC can be applied selectively to individual databases. Application with transactions of a Read-only fashion follows MVCC. This allows clients to Read objects even if they are locked for writing. If other users want to make concurrent parallel changes, they can check out alternative versions of the same object or groups of objects, and work on their versions in private. There will be no concurrency conflicts as the users are operating on different versions of the same objects. Alternative versions can later collate together to reconcile differences resulting from this parallel development.[16] Private workspaces helps user to control which versions to use for each object or group of objects. These workspaces could be shared by different users. That specifies the desired version. This version might be the most recent version, or a particular previous version such as the previous release, or even a version on an alternative branch. Users can also use workspaces to selectively share their work in progress. Workspaces can inherit from other workspaces and also add individual new versions as changes are made, overriding this default. [16]There might be multiple sub-workspaces within the workspace, subgroups of the design team or individual team members. In the database opened through MVCC, the Reader may never update objects. [7]

For non-workgroup applications, multiple concurrent users are supported by the traditional lock-based approach. Locking and Multi-Version Concurrency Control (MVCC) with multiple Readers and one Writer are also available.[7]

2.4.3 Deadlock Handling
When there is a deadlock, the server must pick the victim for aborting transaction. One of the criteria is the transaction priority. Every client has a transaction priority which is a value in the specific in a deadlock situation. The ObjectStore server compares the transaction priorities of clients involved in the deadlock. If the lowest transaction priority is held by only one client, this client is the victim. [16]

3. RELATED WORK

Daniela Rosu: A Model of Concurrency in Object-Oriented Databases
This paper represents the classical serializability theory extended with an abstract model for computations allowing for arbitrary operations and non-deterministic programs. Transaction management allows nested computations and a technique for proving the correctness of the schedules designed by the concurrency control mechanism. All objects in the database are assumed to have their own concurrency control mechanisms that schedule the operations local to the objects according to the order devised by the central transaction manager.

Robert Bretl: Achieving high concurrency in object oriented database
This paper provides an overview of transactions and techniques that can be used to achieve high concurrency in object oriented databases. Database features that make it easier to develop concurrent applications include: Guaranteed view – consistent Reads, Optimistic concurrency control – backward validation', support for multiple concurrency approaches in the underlying database, Active databases – supports execution of object behaviors.

Alexander Thomason: Concurrency Control: Methods, Performance, and Analysis
In this paper, the performance of the locking model is analyzed. This article is to provide ideas of factors leading to performance degradation. It also summarized the conclusions of previous simulation and analytic studies regarding the relative performance of concurrency control methods and survey methods applicable to the analysis of standard locking, restart-oriented locking methods, and optimistic concurrency control.

Wai Lam, Yalin Wang, and YongbingFeng: Concurrency Control in Object-Oriented Databases
This report introduced two classes of semantics approaches for transactions in object-oriented database systems one is the transaction approach, and the other is the data approach. Models of the transaction approach define concurrency properties on transactions according to the semantics of the transactions and the data they manipulate, while models of the data approach define concurrency properties on abstract data types according to the semantics of the type and its operations.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

Jorge F. Garza, Won Kim: Transaction management in an object-oriented database system
The Objective of this paper is to describe two aspects of transaction management in ORION. First is the enhancements to the performance of transactions for application environments of ORION, second is the concurrency control and recovery mechanisms used in ORION to support the conventional model of transaction ORION satisfies the concurrency control requirements of the rich object-oriented data model.

4. DISCUSSION

Versant uses pessimistic strict locking to ensure database compatibility in a multi-user environment and guarantees that changes that are made in the database will be consistent and safe. All types of sessions use a two-phase commit protocol to ensure consistency among multiple databases. Versant's Optimistic locking sessions are compatible with strict locking sessions run by other users. Nested transactions are supported by Versant, while the check-in/check-out mechanism establishes a persistent lock on objects to allow long transactions to complete. Versant provides concurrency control for composite object for different levels. In Versant, persistent locks support long transactions which are often necessary in OODBM systems. The short locks are also provided for shorter transactions as well as "no-locks" are provided for optimistic locking. Upgrading and downgrading of locks are also possible. Intentional locks are supported by Versant which is relevant only to class objects. The purpose of intention locks is to prevent changes to class objects while you are using an instance of the class. Versant handles deadlock problems by avoiding automatic denying of locks that would result in a deadlock situation. Versant detects deadlocks and a would-cause-deadlock error is sent to the user.

Pessimistic protocol leads to deadlock and starvation. When the serializability order is not pre-decided, relatively more transactions will have to be rolled back.

GemStone provides two approaches to managing concurrent transactions; optimistic and pessimistic. Optimistic approach of GemStone is easy to implement. If GemStone detects conflicts then it does not permit you to commit your transaction. Reduce conflict classes help to resolve conflicts. They are designed for specific circumstances. Using these classes allows a greater number of transactions to commit successfully which will improve system performance. GemStone provides Reduced-conflict classes which can be used instead of their regular counterparts in applications with which too many unnecessary conflicts can be experienced. GemStone does not permit you to commit your transaction. The risk of having to re-do the work you've done increases. While GemStone looks for conflicts only at commit time, the chances of being in conflict with other users increase. More time is required to determine conflict resolution in the optimistic approach which leads to greater overheads. If other users have committed changes to the same data the locking mechanism is not automatically carried out. However, to make sure that changes are committed, users are permitted to lock an object that has not been previously locked. Reduced-conflict classes can increase the overall costs as these classes use more storage than their ordinary counterparts. When using instances of these classes, the application may take longer to commit transactions. Under certain circumstances, instances of these classes can hide conflicts from you that you indeed need to know about. They are not always appropriate. These classes are not exact copies of their regular counterparts. In certain cases they may behave slightly differently. "Reduced conflict" does not mean "no conflict." These classes use different implementations or more sophisticated conflict-checking code to allow certain operations that human analysis has determined need not conflict. They do not allow *all* operations. GemStone does not support locking, copying or manipulating a collection of composite objects. While with GemStone you may never succeed in acquiring a lock, no matter how long you wait and because GemStone does not automatically wait for locks, it does not attempt deadlock detection.

Objectivity supports flexible transactions of any duration, including long transactions through checkout/check-in. Transactions are allowed to wait for a user-defined period for access to an object that is already locked by another transaction. The time that objects are locked for Write can be minimized. Multiple-Readers-One-Writer (MROW) improves concurrency by allowing multiple readers to Read the most recent state of an object that is locked for update by another transaction. Object-level versioning allows an application to selectively operate on specific versions of objects. Two-phase hierarchical locking controls Read and Update access to objects. Containers are the fundamental unit of locking within Objectivity/DB. Nested transactions are not currently supported and an application may only have a single transaction active at a time, although multiple top-level transactions are supported. Objectivity supports a pessimistic locking, using Read, Write and Update locks and additionally, Multiple Readers One Writer(MROW). Versioning and version control is also provided. When lock conflicts are raised the operation is immediately given-up and throws an exception. The operation will wait until the desired object is available. Two-phase hierarchical locking controls Read and Update access to objects.

If the deadlock condition is found in Objectivity/DB, an error is returned to the requesting application. When finite lock waiting is requested, no deadlock checking is done. Objectivity/DB works on assumption that any deadlock condition that occurs will be broken when lock waiting times out.

Wait and deadlocks are the conditions that are still bound to be occurred. For long session if the lock server is remote, and network access is slow, this can have a negative impact on performance.

In Objectstore when an application is opened for MVCC, it never has to wait for locks to be released in order to Read the database. Reading a database opened for MVCC also never causes other applications to have to wait to update the database. An application accesses a database opened for MVCC, would see a snapshot which may contain all

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

changes committed before the transaction started. It is internally consistent. It might not contain changes committed during the transaction by other processes. Even two databases both of which are opened for MVCC might not be consistent with each other, because updates might be performed on one of the databases in between the times of their snapshots.If many users want to update the same data concurrently then they can create alternate versions of their own. With Check-out/Check-in, user checks out a version of an object, makes changes and then check changes back in to the main development project so that they are visible to other members of cooperating team.

ObjectStore supports traditional lock-based system for non-workgroup applications. Locking is on a Read, Write or Update basis. To support Multi-Version Concurrency Control (MVCC) Multiple Readers and One Writer (MROW) is also available. ObjectStore supports workspaces for access control to configurations. For non-workgroup applications strict two phase locking and for workgroup applications where less updating is needed MVCC is used, that does not mind Reading snapshot which may not be an updated versions.

ObjectStore supports composite object by making a class the type of an attribute of another class. Composite objects can be treated as one unit for the purposes of locking or deletion. A configuration can be nested and it is a unit of locking that is composed of objects from one or more classes. ObjectStore also supports nested transactions. Workspaces can inherit from other workspaces and also add individual new versions as changes are made, overriding this default.

While using a greater number of transactions, there will be increase network overhead, because each transaction commit requires the client to send a *commit message* to the Server. This extra network overhead is often outweighed by the savings from shorter waits for locks to be released. ObjectStore abort the transaction if deadlock is found and have different criteria to select the victim.

Tabular representation of characteristics of different OODBMS

OODBMS	Support for Nested transactions	Support for Complex Object	Occurrence of Deadlock	Concurrency Control
Versant	Yes	Yes	Yes	Strict 2PL and Optimist
Gemstone	No	Yes	No	Optimist
Objectivity/Db	No	Yes	Yes	MROW
ObjectStore	Yes	Yes	Yes	MVCC

5. RESEARCH GAPS

Based on the study carried out in the paper, a number of issues and evident gaps are identified which may be considered to enhance the methods. The research gaps to be filled are

5.1 Versant supports pessimistic locking for long transactions and optimistic locking for short transactions. Although both methods are supported in Versant, still a new method can be searched which may work with both type of transactions while keeping the resource utilization on optimum without affecting the efficiency.

5.2 Deadlocks and rollbacks are bound to occur with versant therefore a new algorithm to be prepared to overcome from such cases.

5.3 With GemStone, there is always risks of having to re-do the work you've done because GemStone looks for conflicts only at commit time. It increases the chances of being in conflict with other users .This area must be considered for more research in future.

5.4 Objectivity/ DB works on assumption that any deadlock condition that occurs will be broken when lock waiting times out. Avoidance of deadlock before it enters in transaction to be studied with new methodologies.

5.5 Objectstore also applies different methodologies and abort transactions which causes deadlocks, a new method can be find as a solution which shall prevent aborting.

6. CONCLUSION

An object-oriented database system can directly support the needs of the applications that create and manage objects. In this paper, concurrency control in major commercial ODBMS has been described and evaluated.It is evident that there are substantial differences between them. When OODMS follows pessimistic locking for database systems there are limitations of having deadlock and starvation. Also, while in pessimistic locking the wait time of lock to be released can be too long. While with optimistic approach conflicts are checked at the time of commit which may leads to the chance of redo work.

As evaluated above, the different OODBMS have their preferred set of locking even though they support more than one set of locking; Versant prefers Pessimistic locking while GemStone prefers Optimistic locking. Objectivity/ DB and ObjectStore provides additional feature of MROW. The attempt by each OODBMS is to reduce the overhead and resources but have their own limitations in providing optimum resource utilization and subsequently an efficient output.

Therefore new reliable model for expressing the full spectrum of the possible solutions of concurrency controlisneeded,which can reduce the excess resource utilization, provide reduction in deadlocks and gives an efficient and concurrent output.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

REFERENCES

[1] ROBERT BRETL :Achieve high concurrency in Object Oriented Databases GemStone Systems, Inc.,JOOP 10 (8): Pages 40-44 ,1998

[2] WOOCHUN JUN AND LE GRUENWALD :An integrated concurrency control in object-oriented databases

[3] DANIELA ROSU :A Model of Concurrency in Object-Oriented databases McMaster University, September 2001

[4] ALI BARFATANI,DAVID BUCHANAN,MATTHEW BUCHANAN, MARK DOWNING : Comparison of Object Oriented Database Systems

[5] AKMAL B. CHAUDHRI AND PETER OSMON :A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs: GemStone, O2, Objectivity/DB, ObjectStore, VERSANT ODBMS, Illustra, Odapter and UniSQL, Computer Science Department, The City University, London

[6] PAUL BUTTERWOTH,ALLENOTIS ,JACOB STEIN : The Gemstone Object Database Management System , October 1991/Vol.34, No.10/COMMUNICATION8 OF THE ACM

[7] CHARLES LAMB,GORDON LANDIS,JACK ORENSTEIN,DAN WINERB : ObjectStore Database System, October 1991,1.34, No.10/COMMUNICATIONS OF THE ACM

[8] JOHN V. E. RIDGWAY : Concurrency Control and Recovery in the Mneme persistence object store ,Object Systems Laboratory Department of Computer Science University of Massachusetts, Amherst, January 3, 1995

[9] GemStone Systems, Inc.:Introduction to GemStone Transactions and Concurrency Control, July 1996

[10] WAI LAM, YALIN WANG, AND YONGBING FENG :Concurrency Control in Object-Oriented Databases

[11] EVERTON G. DE PAULA, CAPTAIN, BRAZILIAN AF MICHAEL L. NELSON, MAJOR: Clustering, Concurrency, Crash Recovery, Garbage Collection, and Security in Object-Oriented Database Control Management Systems, USAF , Naval Postgraduate School Department of Computer Science, Code CS Monterey, California, February1991

[12] H.T. KUNG and JOHN T. ROBINSON: On Optimistic Methods for Concurrency Control, ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, Pages 213-226.

[13] Versant Object Database Fundamental Manual - release 8.0.1.0 2010http://www.versant.com/developer/resources/objectdatabase/documentation/database_fund_man.pdf

[14] Objectivity Inc. Technical Overview, Release 9, January 2006. http://www.objectivity.com/Misc/docs/oodb_techOverview.pdf

[15] GemStone/S 64 Bit 3.0 Programming Guide, version 3, September 2011http://www.support.GemStone.com

[16] Advanced C++ API User Guide ObjectStore Release 6.3 for all platforms, October 2005ObjectStore Architecture Introductory, Dirk Butson

[17] Slides by Kalpesh Kapoor http://www.cse.iitb.ac.in/dbms Data/ Courses/CS632/1999/odbs/odbs.html

(This work is licensed under a Creative Commons Attribution 4.0 International License.)