Integrity: Do You Know Where Your Objects Are?

Andrew E. Wade, Ph.D.
Objectivity, Inc.
drew@objy.com

Abstract

ODBMSs can support wonderful features, but what does it matter how fancy the versioned, replicated, composite objects are if they are corrupted? In production users require integrity constraints to be guaranteed, a far more important requirement than any other. We'll examine the types of integrity constraints, compare ODBMS integrity support to that of RDBMSs, and discuss specific ODBMS integrity issues in light of: pointer dereferencing, network and node failures, replication, transaction modes, and user-defined priorities, constraints, and trade-offs.

Keywords

Object reference, relational database, object oriented database, replication, client/server, referential integrity

1. WHAT IS INTEGRITY?

Object database systems do for objects the same thing that traditional database systems have always done for simple, flat data; i.e., the ODBMS supports access to many objects, by many users, with recovery and integrity. In addition, the ODBMSs often provide support for transparent access to objects distributed over heterogeneous operating systems, networks, and languages, as well as features such as object versioning, composites with propagating methods, many-to-many bi-directional relationships, etc. All of these are valuable to many users, but the primary motivation for the ODBMS, just as for any DBMS, is the basic capability: accessing many objects, by many users, with integrity. If objects are lost or damaged or other integrity constraints violated, then the features are of no use.

Integrity fundamentally means that the “goodness” of your objects is preserved. This includes a broad range of possible violations, including lost or damaged objects, dangling references to objects, inconsistent attribute values, inconsistent replicas, concurrency race conditions, stale objects due to schema evolution, etc. Integrity can work at various levels, broadly divided into:

- System Level
- Application Level

The former applies to situations in which the damage, loss, invalid or inconsistent information appears at the level of the DBMS itself. If an object is inaccessible due to corrupted identity mechanism or indices, this is system level. Referential integrity is violated if a bi-directional relationship becomes inconsistent so that the two sides do not point to each other. Even simple (unidirectional) references might become invalid if they, at times, do not point to the correct object.

Such system-level integrity violations are sometimes caused by weaknesses in the DBMS architecture or implementation, possibly triggered by application bugs, operating system failures, network failures, or other computer failures. In many computer systems such errors would just be accepted as the natural, if unfortunate, result of such bugs or failures. However, with a DBMS users often expect the system to avoid, prevent, or at least recover from such failures, preventing loss or damage of information. In fact, this is often the primary motivation for use of a DBMS, to ensure validity of critical information.

Even when the DBMS-level mechanisms are intact, the user may find information lost or damaged at the application level. There may, for example, be dependencies or constraints on values of attributes or results of operations on objects. If such application constraints are violated, the information is indeed damaged, from the user's point of view. Maintaining these integrity constraints is therefore critical, too.

2. RDB VS. ODB INTEGRITY

Integrity is not new with the advent of ODBMSs, but it does have different aspects. In traditional databases, including RDBMSs, the level of the integrity support is limited to data only, and only simple, flattened data, while ODBMSs address objects at various levels of abstraction. Also, traditional and relational DBMSs are typically built on server-centered architectures, while ODBMSs often support distribution, multi-tiers, and local caching, all of which have different integrity support implications. We'll consider this latter architectural difference first.

In typical RDBMSs the user, or client, interacts with the DBMS by sending a request (e.g., a query) to the DBMS server. That might happen in many ways; e.g., by using a client-side graphical (GUI) tool, by typing in a string of SQL, or by executing a program that makes a programmatic call to an ODBC interface that

then transmits the request to the server. All the DBMS functionality is executed on the server side, and the user's answer, if any, is returned to the client.

Figure 1. Client-Server "Firewall" Protection

This narrow communication link, and separation of client and server into different processes, and often different computers, provides a well-defined integrity boundary. Nothing the application or client does can affect the database state except via the well-defined SQL (and often procedural) interface. In particular, many operating systems provide some protection mechanism to ensure that the client cannot otherwise access any of the data in the other (server) process. Application bugs cannot, for example, run rampant through the DBMS buffers corrupting the data. This process boundary "firewall" provides a comfort level of safety and integrity.

In a typical ODBMS, the same mode of client/server, safe, firewall operation is available, often through an ODBC interface, just as in RDBMSs. However, the ODBMS also offers an additional interface (e.g., in C++ or Smalltalk) in which some of the objects are cached in the application's address space, and some of the database operations occur in that address space. For applications that use that interface, the operating system cross-process, cross-address space firewall is no longer in place. An application could, in fact, run rampant and damage locally cached objects. Approaches to minimize such problems are discussed in the next section.

Actually, most current RDBMSs have begun to violate the strict client/server split described above. By providing *stored procedures*, they allow users to write

(application) code that is installed in and executes on the server. In that case, they, too, lose the firewall protection of the operating system.

The other major difference in integrity between RDBMSs and ODBMSs arises from the modeling level. In an RDBMS all modeling is restricted to a flat level, data fields in tables. Operations (join, insert, update, etc.) operate directly and separately on this flat data, so it is difficult to provide a means for integrity management at any other level. For example, data values, called foreign keys, are used to implicitly represent relationships, and the addition to SQL2 of a proviso to establish that they be maintained with bi-directional referential integrity was late in coming and difficult to express and coordinate with other parts that allow independent direct modifications of all fields, including these.

In an ODBMS information is modeled as objects, each with attributes, operations, and relationships. The explicit definition of relationships, and access to them only by ODBMS-supported operations, makes definition and support of bi-directional referential integrity straightforward. The availability of operations, and ability to hide internal structure by limiting access to desired operations, allows the ODBMS users to define integrity constraints that the ODBMS then guarantees are respected in all object access. For example, a limitation that restricts or relates certain attribute values within a single object or even different objects can be directly and easily supported with such operations.

Also, the ODBMS model supports higher level of abstractions. By defining arbitrarily complex objects, and composite objects consisting of other objects, users can directly define, at the DBMS level, the entities of importance in their application. Although an RDBMS user might well translate such application entities down to flat tables, expressing constraints on these flattened views might be very difficult, and is in general not possible with uncontrolled, independent access to the flattened level. This can be a major advantage to ODBMS users.

This advantage of abstraction can be viewed as effectively moving the enforcement of constraints from the application to the ODBMS. With an RDBMS the user is forced to talk to the DBMS only at that flattened level, so all higher levels of abstractions and operations must be part of the application. The complexity, the higher levels, the constraints do not go away; they are determined by the application. However, left in the application they open the door to integrity violations if two different applications effect them in different ways. Although the DBMS claims everything is fine, user-level constraints are violated across different applications due to this application-level enforcement.

With the ODBMS approach, the user (or some subset of users) must still define, and in general program, the constraints, just as he did at the application level for the RDBMS. However, now these constraints can be installed into the ODBMS, and it will enforce them uniformly across all applications. Also, via encapsulation,

it can enforce them across all object operations. These advantages bring the DBMS function of integrity management to a much broader level.

Next, we'll turn to some of the issues specific to ODBMS integrity maintenance, detailing four system-level issues first, followed by a discussion of application-level integrity and how the user may achieve his desired level of integrity.

3. OBJECT REFERENCE ISSUES

The ability, mentioned above, of ODBMSs to cache objects in the application's address space has both a positive and a negative. The positive is speed of access. Operations across address spaces take milli-seconds, while operations within address spaces take machine cycles, tenths or hundredths of micro-seconds for 10MIPs or 100MIPs machines. This difference, over four orders of magnitude, can produce dramatic performance improvements. On the negative side, an application reference bug can result in damage to the objects and integrity loss.

There are two approaches that are commonly used for object references, each with very different integrity behavior. The first, and simplest, is to give the application a direct pointer to the object. Eventually, though, such pointers become invalid; e.g., the object moves. After commit, the pointer is necessarily invalid, because commit semantics require the object to be available for moving to other application's address spaces. If the programmer is careful and never uses the pointer when it's invalid, there is no problem. Attempted de-reference of such invalid pointers, though, will either cause a crash (segment or access violation) or, worse, will dereference into the middle of some other object that happens to occupy that address. Reading values from that (wrong) object can violate integrity, and certainly writing values into the wrong object will violate integrity.

Unfortunately, such pointer errors are extremely difficult to eliminate, as any virtual memory programmer will attest. Further, once the DBMS gives the pointer to the application, the application may store it in variables, pass it to other functions, and there is no way for the DBMS to get it back. These errors can be subtle, because they may not appear at all in the prototype. When the application is deployed, however, and there are many objects, and transactions, and swapping, the result can be corrupt databases.

Figure 2. Object Reference via Indirection Guarantees Integrity

The other approach provides an extra level of integrity by inserting one level of indirection between the application's pointer and the actual object location. Then, if the object is moved, the ODBMS runtime object manager can fix the second pointer. Even if the application has made many copies of its pointer, since the dereference will go through the second pointer, too, it will always go to the correct object. The cost may be one or a few cycles, tenths or hundredths of micro-seconds, which is usually negligible in practice, compared to application processing such as method execution, and always negligible compared to the major DBMS performance overhead of disk I/O and network transfer, both of which are in the milli-second range. The advantage is that every object reference is guaranteed correct, even after commit.

It remains true, of course, that applications can damage objects cached in their address space. Without an operating system firewall to prevent this, such damage is always possible. In practice, though, the most common problems are due to pointer bugs, and those are eliminated by indirect reference approach. For cases where users want the firewall level of protection, they must move from the application-with-cache interface to a client/server (ODBC) interface, just as in RDBMS they must move from stored procedures to client ODBC interfaces. Mixtures are, of course, possible, too; some trusted users may be allowed to implement via the direct interface (or stored procedures), while other, untrusted, users are restricted to the client ODBC interface.

4. NETWORK AND NODE FAILURES

In the "good old days" there was only the mainframe computer with terminals. Any failure brought down the entire system, and its attentive cadre of administrators

repaired it appropriately. Any failure is a complete failure, so no subtle or complex integrity violations are possible. Today, however, many environments consist of multiple computers, PCs, workstations, servers, etc. This means many more things can fail, and the rest continue to function. Networks connecting nodes may fail, and nodes (computers themselves) may also fail, but in each case, other nodes may continue to function and communicate via the remaining operational network links. This raises new integrity issues for distributed databases, including ODBMSs.

Figure 3. Network Partitions with Replicated Critical Resources

In a distributed ODBMS, users may be on any nodes, objects may be on any nodes, and access to those objects is transparent. Whether accessing via name or object ID or query or relationship traversal or key, etc., the users simply access objects. When a network or node fails, users may be prevented from accessing certain objects, which seems reasonable. We'll return to this in the next section. Worse, however, the loss of a network connection or a certain node might prevent some or all users from using *any* ODBMS functionality. For example, the lost node might contain the schema information necessary to create a new instance object. Similarly, a failed network connection might prevent access to such schema information. Even worse, lost access to some critical resources, such as locking information, logs, catalogues, etc., can effectively block all DBMS usage for all users. This is non-intuitive; i.e., users wonder why all work is stopped by the loss of only one machine. Of course, they may also be unhappy that all work is stopped!

There are various ODBMS approaches that can continue to provide services even with some node or network failures. To accomplish this, the ODBMS must effectively replicate the critical resources, so that, if the primary is unavailable, the

replica can be used. For example, when a network link fails, it effectively divides (or partitions) the previous configuration into two parts or partitions. Any critical resources that are required by the DBMS must be replicated in each such partition for continued service. In general the ODBMS might provide for any number of such partitions, thereby effectively allowing the DBMS to continue to provide services with any number of node or network failures.

When the ODBMS is functioning under such a failure situation, further integrity issues may arise. If users in two different partitions wish to modify the same object, this could violate integrity. The ODBMS can avoid this violation by replicating sufficient concurrency (locking) information in each partition to ensure that locks obtained before the failure are known to other partitions, or at least that other partitions know that they cannot obtain a conflicting lock.

In a more subtle example, a user makes a change in one partition, and a second user makes a change in a second partition, with the result that those two changes, together, violate an integrity constraint. It is important, then, that the replicated partition information includes enough information about constraints to disallow such changes. For example, if there is a bi-directional relationship between two objects, and only one of those is available to a user in his partition, the system must not allow that user to update the relationship, just as it would not allow him to update attributes on an inaccessible object.

So, by replicating all critical system resources in each partition, the ODBMS can continue to provide functionality even when some networks or nodes fail. By ensuring that those replicas are kept in synch, and by appropriately managing lock conflicts in separated partitions, the ODBMS can avoid inconsistencies and maintain integrity even during continued service under such failure situations.

5. REPLICATION ISSUES

The above discussion covered critical system resources, but there may be other resources that are important to the users. For example, even though my partition continues to provide DBMS services, it might not allow me to access some of my (or others') objects, because they are in a different partition, and it is currently inaccessible. The ODBMS can continue to provide availability of all services and access to all resources by replicating these other resources, such as user objects, so they are available even when node or network failures partition the environment.

With replication, however, arise new integrity issues. As soon as there is more than one copy of some information, there is the question of how the copies are kept in synch. Various approaches have been offered, with various levels of resulting integrity...and violation of integrity. We'll describe three levels to illustrate the types of approaches available.

Figure 4. Master-Slave Asynchronous Copying

The simplest approach is master-slave asynchronous update. A single master copy is considered to capture the correct state of all objects at all times. Periodically, through manual or other mechanisms, this master is copied to various slaves. Users may then access the slave copies. However, if users change information in the slave copies, that leaves them inconsistent with other slave copies, not to mention the masters. So, either integrity is sacrificed, or update permission is denied, limiting availability of the DBMS services. Similarly, if a change is made to the master, it is not reflected in the slaves until the next copy occurs; i.e., replication update is asynchronous. In the intervening time, the slave copies are inconsistent. Integrity is sacrificed.

Figure 5. Hot Backup Server

Another approach improves this one step. Often called "hot backup," it works by having the primary server copy every action to a secondary, or backup, server. Since this copying is synchronous, or occurs with each change, it has the advantage that the backup server can take over should the primary fail. So, it provides some measure of continued of a single server, at the cost of extending the time for every server operation, which must wait for secondary server confirmation of execution. However, if the network between the two servers fails, each server must assume it is the only one left; i.e., they cannot distinguish between a network and a node failure. In such a network failure, each server will allow user updates, without any concurrency control, resulting in conflicting updates that cannot be resolved even after the network failure is repaired. Integrity in the face of network failures is lost.

Figure 6. Distributed Replicas Guarantee Integrity

The third approach we'll describe guarantees integrity and provides higher level of continued availability, even during network and node failures. This approach, called distributed replicas, assumes, first, the replication of system critical resources in each partition, as described in the previous section. In addition, it allows users to specify objects to be replicated and in which partitions they should be replicated. The system resources in each partition, including the catalogue, are aware of which objects are replicated and where. Update of replicas is done synchronously, with each commit. In general, then, there is extra overhead for updates, because they must be propagated to replicas, though implementations can minimize that by batching such updates at commit time. Further, propagating updates in parallel to all replicas allows any number of replicas with no cost beyond that of one. Also, read operations need have no extra overhead, and in fact can be accelerated if the DBMS is able to locate a replica that is closer (in access time) to the user.

In the distributed replicas approach integrity is maintained by having the servers communicate to determine with which other servers they can communicate. When there are network or node failures, each server determines whether it is in the majority or minority (a voting process). Those in the majority grant users update permission, and the usual concurrency (locking) mechanisms guarantee integrity among them. The minority servers grant only read permission, thereby avoiding conflicting updates. Although this limits some users from some services (update), all users have minimum services, read, and the majority of users have all services. If certain users desire continued access to certain objects, they can be given higher weight in the voting process, to adjust the result to that which best fits their needs and maximizes this availability. As with the partitions described above, users may achieve any desired level of continued availability, in the face of any number of network or node failures, by creating any number of replicas. Most importantly, integrity is guaranteed even while access is maintained during failure conditions.

6. TRANSACTION MODE ISSUES

Another integrity issue arises in relation to various transaction modes. Although this is at the system level, it also relates to application desired semantics.

Traditional transaction semantics is very strict, in order to support the ACID (atomicity, consistency, integrity, and durability) requirements and to support serializability so that the interleaved operations of concurrent users do not effect the results of their operations. Usually, this is effected via two-phase transaction with exclusive locking. Objects with read locks will not permit write locks until the read is released, though they will permit other read locks. Those with write locks will not permit any read locks. For example, if one user is withdrawing money from his bank account, and another user tries to check the account balance, that second user will be suspended until the first user's withdrawal completes, at which time he'll see the new, lower balance.

Figure 7. MROW Transaction Mode

In practice, however, there are users who prefer to trade off such rigid restrictions in favor of higher performance, higher concurrency, transaction rates, and throughput. To accomplish this, several different techniques have been offered, including MROW, or Multiple-Readers-One-Writer. In this transaction mode, an object allows up to one writer along with any number of readers. All of these run exactly in parallel, which, for some applications, can dramatically improve performance. It differs from dirty read, in which readers are allowed to see the object as it's being changed, leading to potential integrity problems due to partially changed state, inconsistent attribute values, and dangling references. Instead, the DBMS maintains a pre-image of the object state as of the previous commit, so the readers see a completely consistent state, albeit with less strict serializability as described above.

Although maintenance of transaction-level integrity is clearly a system-level integrity issue, both of the above-described approaches are well-defined, and it is ultimately the user (or the application) that must choose its preferred semantics. The same is true for other possible transaction modes, including locking and unlocking in mid-transaction, locking with timed waits, etc.

7. HOW TO ACHIEVE INTEGRITY

The above issues are mostly system-level, but often application-level constraints are critically important. As discussed above, the ODBMS allows the user to define

those, at any desired level of abstraction, in terms of any operations, whose implementation effects the desired constraint. Once defined and installed, the ODBMS can ensure they are respected.

The user's desired integrity result, then, requires a combination of system- and application-level constraints. The application-level ones are, by definition, entirely at the discretion of the application, and help it achieve its desired semantics. At first, the system-level ones may seem to be required. However, even these may be at the discretion of the user. For example, there are many users of file systems without any integrity constraints. These users consciously choose the trade-off of simpler or cheaper systems without recovery, without guarantees of atomic transaction commits, referential integrity, etc. Other users have decided they require these and then choose to use a DBMS. The various levels in between, then, can also be user choices for some.

Those who demand a high level of integrity guaranteed generally choose a DBMS to supply it. For those users, it's important to understand the above integrity issues, to analyze the systems they choose, and to ensure that they achieve the desired level of integrity. Far more important than feature differences, these integrity issues often determine the level of success in deployment.