



“Achieving Century Uptimes” An Informational Series on Enterprise Computing

**As Seen in *The Connection*, An ITUG Publication
December 2006 – Present**

About the Authors:

Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, have a combined experience of over 90 years in the implementation of fault-tolerant, highly available computing systems. This experience ranges from the early days of custom redundant systems to today’s fault-tolerant offerings from HP (NonStop) and Stratus.

Gravic, Inc.
Shadowbase Products Group
301 Lindenwood Drive, Suite 100
Malvern, PA 19355
610-647-6250
<http://www.gravic.com/shadowbase>

Achieving Century Uptimes

Part 3: Avoiding Data Collisions

March/April 2007

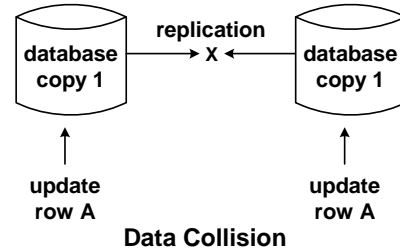
Dr. Bill Highleyman
Dr. Bruce Holenstein
Paul J. Holenstein

The Data Collision Problem

The most common way to keep database copies synchronized in an active/active network is via asynchronous replication. With this technique, changes made to one database copy are queued for replication to the other database copies in the application network. It is the job of data replication engines to propagate updates from these queues to the other database copies and to apply them.

Asynchronous data replication engines are decoupled from the application. That is, they replicate independently from the source application. There is therefore a delay from when an application updates one database copy to the time that the data replication engine applies that update to the other copies. This delay is known as the replication latency of the data replication engine.

A critical issue with asynchronous replication is data collisions. Data collisions come about should two or more users update the same row at substantially the same time in different database copies. If this time difference is within the data replication latency, each user is unaware of the updates made by the other users. Each of their updates will be replicated to the other database copies and will subsequently overwrite the original updates. As a result, the database copies will all be different; and all are wrong.



Of course, data collisions can be avoided if synchronous replication is used since all data items across the network are locked before any are updated. Thus, they will either all receive the same update or none will. However, there are ways to structure an active/active system which uses asynchronous replication to avoid data collisions. Some of these techniques are described in this article.

If collisions cannot be avoided, they must be detected and resolved. Data collision detection and resolution are discussed in our next article.¹

¹ Data collision avoidance, detection, and resolution are discussed in detail in the book “*Breaking the Availability Barrier: Survivable Systems for Enterprise Computing*,” by Dr. Bill Highleyman, Paul J. Holenstein, and Dr. Bruce Holenstein, AuthorHouse; 2004.

Avoiding Data Collisions

In some applications, collisions are simply not possible. In some other applications, data collisions can be ignored.

However, if data collisions are possible, they can be avoided at the data level by partitioning the database; or they can be avoided at the system architecture level by declaring a master node

Application

Some applications are immune to data collisions. Either they cannot occur, or they can be tolerated.

Single Entity Instance

In some applications, there can be only one physical entity instance that is represented by a collection of data items in the database. Therefore, there can only be one transaction outstanding at any one time; and collisions are not possible. A good example of this type of application is gift cards.

Gift cards are identified by a unique number. There is only one physical instance of each gift card, and they are not rechargeable; so there is no chance of a colliding administrative action. When a gift card is used, the system which receives the transaction authorizes the sale and replicates the new gift card amount to the other database copies in the application network. The transaction response to the point-of-sale device allows it to update the new gift card's balance on the card's magnetic stripe.

Since there is only one instance of the gift card, it is impossible for there to be simultaneous transactions. Thus, data collisions are not possible (short of the fraudulent use of gift card copies).²

Insert Only

If all database operations are simple unique inserts, there can be no data collisions. An example of such an application is the recording of call detail records (CDRs) by telephone companies for later billing purposes.

² When payment cards first were introduced some three decades ago, there was often no central authorization system. The card's magnetic stripe held all of the data, and the transaction was authorized based only on the data recorded on the card. It was then found that thieves could easily buy magnetic stripe readers and writers and forge the magnetic stripe on stolen cards. As a consequence, they could use them multiple times without detection.

Collision Tolerance

There are some applications in which collisions will occur but can be tolerated. For example, those which are simply accumulating statistical information for later data mining may be deemed to be tolerant of collisions if these collisions will have little effect on the statistical results. Other applications may become resynchronized at a later time due to normal transaction activity, and this period of database divergence may not be considered a problem.

In cases such as these, though data collisions can occur, no collision detection or resolution facilities are necessary.

Partitioning

In many applications, it is possible to partition the database so that all updates made to a particular partition are made to a specific database copy. This precludes the possibility of data collisions. There are several strategies for database partitioning.

User Locality

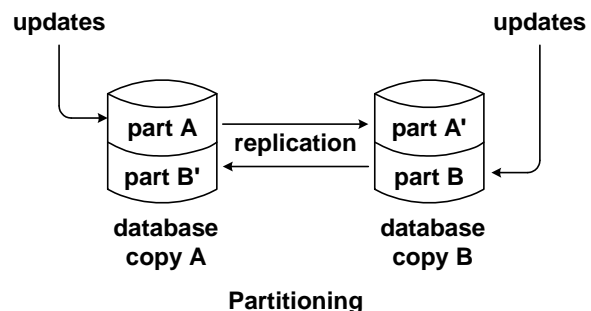
The simplest form of partitioning is a natural partitioning based on user locality. In these applications, a specific community of users will only update a specific set of data items. If the users who are members of a specific community all use the same database copy, there will be no collisions.

This might be the case, for instance, for a sales tracking application in a regional sales office. All data entered for a particular salesperson would always be entered by the same sales office into the database copy to which it is assigned.

Often, applications such as this take advantage of data locality by having one of the nodes of the active/active network in close proximity to each community of users. This arrangement promotes full capacity use and maximizes performance by minimizing the communication delays during transaction execution since users are generally local to their assigned database partition.

Data Content

Perhaps the most common form of partitioning is by data content. In this architecture, the database is partitioned according to some piece of data common to each data item. Each partition is “owned” by a specific node (the partition’s “primary” node), and all updates for a partition are always routed to its primary node.



For instance, the database might be partitioned by customer account number. Account numbers beginning with 0 through 4 might be resident in one database copy (let us call this partition A), and account numbers beginning with 5 through 9 might be resident in another database copy (partition B). Partition A is owned by node A, and partition B is owned by node B.

In this case, transaction distribution could be done in two different ways. One way is for any transaction to be received by any node. If node A receives a transaction that will update partition A, it will process that transaction. However, if node A receives a transaction that will update partition B, it will send that transaction over the network to node B for processing.

Alternatively, intelligent routers which can be driven by the data content of messages can be used to route transactions to the desired node.

With this strategy, it is straightforward to rebalance the system should the loads on the nodes become unbalanced. For instance, in the example given above, if node A (which is handling accounts beginning with 0 through 4) should become more heavily loaded than node B (which is handling accounts beginning with 5 through 9), it may be desirable to “move” those accounts beginning with 4 from node A to node B. This can be done by simply sending new routing tables to the nodes or routers, as appropriate, which will direct that all transactions for accounts beginning with 4 now be routed to node B.

This partitioning scheme loses the advantage of data locality since transactions will generally be routed over the network and will accrue a round-trip communication delay, thus affecting response-time performance. To the extent that there is locality of the data partitions to the users most likely to update them, the performance penalty will be minimized.

There is a further complexity if a transaction can update data in multiple partitions. In this case, the node processing the transaction must either include all partitions in the scope of the transaction (thus incurring a round-trip communication delay for each action, such as an update) or break the transaction into subtransactions, one for each partition. In the latter case, it will usually be necessary for the database manager to have the capability to handle subtransactions as it is generally not possible for the application to manage several subtransactions and still guarantee transaction atomicity (that is, either all subtransactions are committed or none are).

Node Ownership

An alternate to partitioning by data content is partitioning by node ownership. In this technique, each data item (for instance, a row or a record) is “owned” by a particular node. This is implemented by including in the key for the data item the identification of the node that owns that data item.

The node identification is generally added when each data item is created (that is, when the data item is inserted into the database). The node on which the insert occurs notes its ownership of this data item by adding its node id to the data item's key. Thereafter, any updates for that data item must be made to the database copy controlled by the owning node.

This is a form of data partitioning by data content (the node id in the key) and carries most of the same advantages and disadvantages of data content routing, as described above. However, load rebalancing is not so simple since the keys of the data items whose ownership is to be relocated must be modified by replacing the node id field in the key with the id of the new owning node.

How Many Database Copies?

With partitioning, there need not be more than two copies of the entire database in the application network. For instance, if there are four nodes in the network, the database can be partitioned into four partitions. Each node can be the primary node for one partition and can be the backup node for another partition. Therefore, each node will carry one-half of the database. Thus, among the four nodes, there are only two copies of the database.

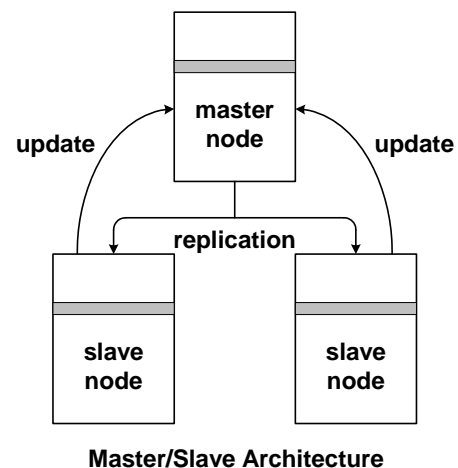
As an extension to this, there can be multiple copies of the database in the application network. This provides the opportunity to achieve a compromise between cost and availability (and performance as well if more effective data locality is realized).

Architecture

An active/active system can be structured to avoid data collisions by declaring one node to be the master node. All updates must be made to that node, and it will replicate database changes to all of the other (slave) nodes, including the node that originated the change. Since all updates are made to only one database copy – the master node's copy – there can be no data collisions.

Should the master node fail, any one of the slave nodes can be promoted to be a master node until the original master node is returned to service. This is simply done by notifying the new master node that it is to handle updates and by notifying all of the other surviving nodes of the new master so that they can route their updates to it.

One downside of this technique is the effect of communication latency. Since transactions must be sent across the network, they will suffer a delay due to the time that it takes for each interaction to move across the network to reach the master node and for



the response to be returned.

There are two ways in which transactions can be handled. In one case, the originating node owns the transaction and makes every update to the master node across the network. In this case, each update will incur a round-trip communication latency.

The other technique is to send the entire transaction to the master node so that it can be executed locally at that node. In this case, there is only one round-trip communication latency time – that required to send the transaction and to receive its response.

Note that this master/slave configuration can be used either for symmetric or for asymmetric capacity expansion. In a symmetric architecture, all nodes are performing the same function except that the slaves must send updates to the master node.

In an asymmetric architecture, the nodes are providing different functions. A common use for an asymmetric architecture is query processing. In this case, the slave nodes provide support for complex queries. The master node provides transaction processing functions and sends all updates to the query nodes to maintain query result consistency.

Data Collision Resolution

If data collisions under asynchronous replication cannot be avoided, they must be detected and resolved. There are many automatic methods for doing this. Data collision detection and resolution will be described in our next article.