
UNICOM Global

solidDB high availability

Originally released in 2011 IBM Redbook sg247887 “IBM solidDB Delivering Data with Extreme Speed”
Chuck Ballard, Dan Behman, Asko Huumonen, Kyösti Laiho, Jan Lindström, Marko Milek, Michael Roche, John Seery, Katriina Vakkila, Jamie Watters, Antoni Wolski

Updated 2025. Unicom System Inc.

This whitepaper describes the solidDB high availability (HA) database. It includes information about the principles of HA, shows how the solidDB HA solution works, and shows how it can be used to its full potential.

High availability helps to make a system more redundant, and more tolerant against hardware problems, software problems, and human errors.

solidDB and its HotStandby technology (solidDB HSB) are used in many areas, including carrier-grade telecommunication networks and systems, railway systems, airplanes, defense applications, TV broadcasting, banking and trading applications, web applications, and point-of-sale systems.

HA is more than a replication solution. You must also consider how to manage HA, and how to execute automated recovery after failures.

High availability (HA) in databases

HA makes system failures tolerable.

Tolerability can be measured by calculating the ratio of the time the service is *actually* operational to the total time the service is *supposed* to be operational. In the following formula, MTBF (mean time between failures) is the frequency of outages, and MTTR (mean time to repair) is the maximum duration of an outage:

$$A = \frac{MTBF}{MTBF + MTTR}$$

The higher the value of A, the better is the availability of a system.

When A is close to 1, the value of MTTR becomes small. For example, if A is 0.99999 (referred to as “five 9s availability”), the total yearly MTTR is approximately 5 minutes. In a six 9s system, the MTTR is about 30 seconds. These high levels of availability can be achieved only by automated systems.

To deal with failures, an HA system uses redundancy in both hardware and software. Redundant system parts mask the failures. Various *redundancy models* can be applied. For example, the simplest redundancy model, called *2N*, or *hot standby*, comprises two units, *active* and *standby*. If a failure

occurs, the failed active unit (which can be hardware or software) is replaced with the standby unit. This operation is called a *failover*. Failover maintains the required availability level.

Other redundancy models are, for example, N+1 (several active and one standby unit) and N*Active whereby all units are active, and the failure is masked by redistributing the load over the surviving units.

The availability of the database services is maintained by using similar approaches. This white paper focuses on the solidDB HotStandby DBMS.

The solidDB HotStandby solution

The HotStandby (HSB) solution is included with solidDB. The same server binary is used for both stand-alone and HotStandby operation modes; HotStandby is enabled by using configuration parameters.

HSB architecture

In solidDB HSB, a stream of transactions is continuously sent from the primary (active) server to the secondary (standby) server by using a replication protocol. The connection between the servers is called an *HSB link*.

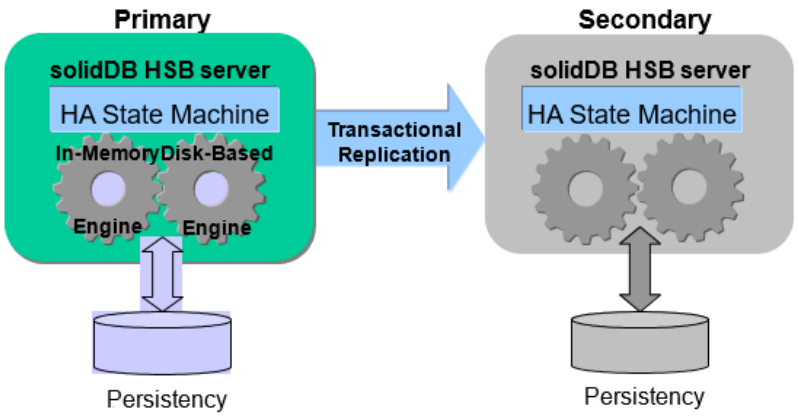


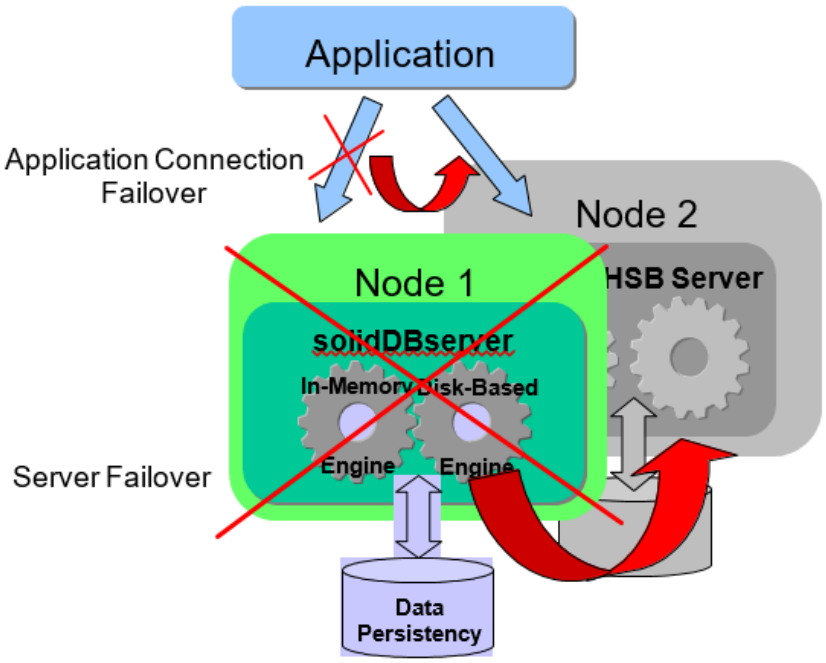
Figure 1 Hot-standby database

Figure 1 represents a *shared-nothing HA DBMS*. In a shared-nothing system:

- All the components of a HA DBMS are redundant, including the persistent data storage. Even in the case of an in-memory database, there is persistent storage, which allows the database to be recovered.
- You are also protected against media failures. (A *shared-disk* or shared-storage system assumes that the common storage never fails.)

An HA DBMS is different from a non-HA system because it has an HA state machine in the database server. The HA state machine makes the server aware of the HA state. The HA state machine preserves the database consistency. For example, when the server is in the secondary state, receiving the transaction stream from the primary, updates to the secondary database are disabled.

If the primary site fails, the failover takes place as depicted in Figure 2.



In a failover, two events happen:

1. The secondary server takes over as a new primary.
2. Applications reconnect to the new primary.

This process is described in the following sections.

State behavior of solidDB HSB

solidDB implements an internal HA state machine to allow consistent approach to well defined HA Management. Thanks to clearly defined states and transitions from one state to another, HA Management can be built reliably, for the purposes of health monitoring, failure handling and recovery actions.

The following diagram shows the HA state machine of solidDB HSB. The operational hot-standby states are shown on the right: PRIMARY ACTIVE (active) and SECONDARY ACTIVE (standby). Other states come into the picture when taking care of failure, startup, and reconfiguration scenarios.

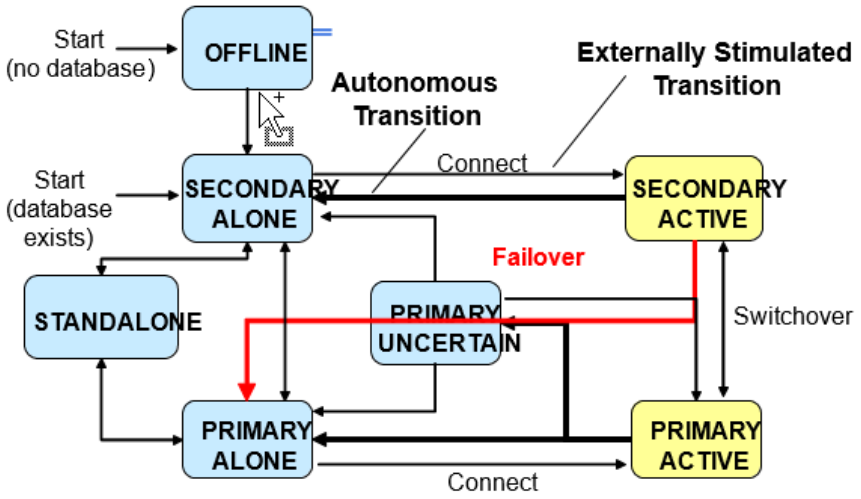


Figure 3 State transition diagram of solidDB HSB

The state behavior is externalized in the form of commands that invoke state transition and query the state. The commands are available to applications or a dedicated HA controller as extensions of the SQL language. For more information about the states and transitions, see the solidDB: High Availability User Guide.

In the diagram, **bold** indicates transitions that are executed autonomously by the database server process. They relate to falling back to a *disconnected* state (that is, PRIMARY ALONE or SECONDARY ALONE), both on the primary and secondary side, if a communication failure occurs between primary and secondary. This behavior is made possible by a built-in heartbeat function. All other transitions are invoked with administration commands.

This means that the crucial failover transition is invoked by an external entity, such as a dedicated HA controller or a general-purpose HA framework. It is performed with a single solidDB admin command, **hsb set primary alone,** which can be issued in both the SECONDARY ACTIVE and SECONDARY ALONE state (because the Secondary server might have fallen back to the ALONE state already). The resulting state is PRIMARY ALONE; it is HA-aware in the sense that it involves collecting committed transactions to be delivered later to the Secondary, on reconnection and the resulting *catchup* (that is, resynchronizing the database state).

If a failure occurs, no reconnection is likely to happen in the near future; there is a possibility to move to a *pure* STANDALONE state that has no HA-awareness. In that case, future actions might include restarting a secondary database server without a database, and sending a database over the network (referred to as *netcopy*). For this purpose, a startup state OFFLINE exists. Its only purpose is to receive the database with a netcopy. After the successful netcopy, the state SECONDARY ALONE is reached, and the admin command **connect** brings both servers back into the operational hot-standby state. However, if a secondary database already exists, the startup state is SECONDARY ALONE.

The auxiliary state PRIMARY UNCERTAIN is meant for reliably dealing with Secondary failures. If the internal heartbeat alerts the Primary server that the communication with the Secondary has failed, and there are transaction commits that have been sent but not acknowledged by the Secondary, the

resulting state is PRIMARY UNCERTAIN. In this state, the outstanding commits are blocked until resolved. The resolution may happen automatically when the Secondary becomes reconnected. Alternatively, if the Secondary is assumed to become defunct for a longer period of time, command-invoked transitions are possible, that is, to PRIMARY ALONE whereby the outstanding commits are accepted.

The PRIMARY UNCERTAIN state is not mandatory; you can by-pass it by using a configuration parameter setting.

In addition to the transitions dealing with failures, you can perform a role switch for maintenance purposes. This operation is called a *switchover*. You can invoke it by using the following commands:

```
hsb switch [to] primary
hsb switch [to] secondary
```

solidDB HSB replication and transaction logging

The HSB replication protocol carries the transaction results safely from the primary and secondary, over the HSB link. When delivered, the data allows for the following events:

- Failover, preserving the database state.
- Read-only load, to be applied to the Secondary.

The replication protocol “lives” in a symbiosis with the local transaction logging.

Transaction logging

Both the replication and transaction logging move the committed transactions out of the volatile memory, as shown in Figure 4.

The primary DB and secondary DB represent persistent storage of the data:

- The primary DB is a live database updated by the transactions running on the active server.
- The secondary DB is kept up-to-date by a replication protocol. The secondary DB can be subjected to read-only load, if necessary.

Logger is a thread that writes the transaction log (*Log*), which is one or more persistent files to store the effects of transactions as they are executed on the server. The Log is instrumental in making it possible to perform a startup database recovery. Startup recovery happens when any server is started as a process. It is assumed that a *checkpointed database* file and log files exist. A checkpointed database file is a materialization of a consistent snapshot of a database state stored in a file. The state typically represents some point in the past. In solidDB, the database file always contains a valid checkpoint.

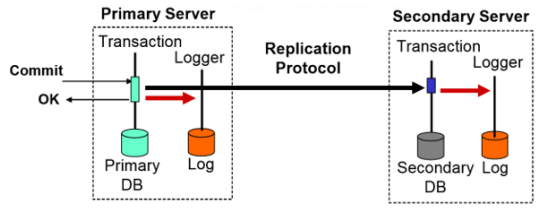


Figure 4 Replication and logging in solidDB HSB

In the recovery process, the Log brings the database to the latest consistent state, by doing the following:

1. Remove the effects of uncommitted transactions
2. Re-execute committed transactions that have not been checkpointed to the database.

If we deal with a stand-alone database system (not hot standby), the recovery process preserves the *atomicity* and *durability* characteristics of the database over system failures and shutdowns:

- Atomicity means that no partial transaction results are persistently stored in the database, and they are not visible to other transactions.
- Durability means that a transaction, when committed, will never be lost, even if a failure immediately follows the commit.

In enterprise databases, the standard level of durability support is called *strict durability*. It requires that the commit record of a transaction is written synchronously to the persistent medium (disk) before the commit call is returned to the application. The technique is often referred to as write-ahead logging (WAL). WAL processing is resource-consuming and it often becomes a bottleneck in the system. Therefore, when the durability requirement can be relaxed, it is done. Especially, in the telecommunication environment, in some applications such as call setup and session initiation, a service request is occasionally allowed to fail (and be lost) if the probability is not high. In such a case, relaxed durability may be applied whereby the log is written asynchronously, which means that the commit call can be returned without the need to wait for the disk write. The result is significant improvement in both the system throughput and response time.

Replication

In an HSB database, transactions are also sent to the secondary server by a replication protocol. To preserve the database consistency in the presence of failovers, the replication protocol is built on the same principles as physical log writing. That is, the transaction order is preserved, and commit records demarcate committed transactions. If a failover happens, the Standby server performs a similar database recovery as though a transaction log was used. The uncommitted transactions are removed and the committed ones are queued for execution.

The replication protocol can be asynchronous or synchronous. To picture that, we use the concept of a safeness level where *1-safe* denotes an asynchronous protocol and *2-safe* denotes a synchronous one. The two safeness levels are illustrated in Figure 5.

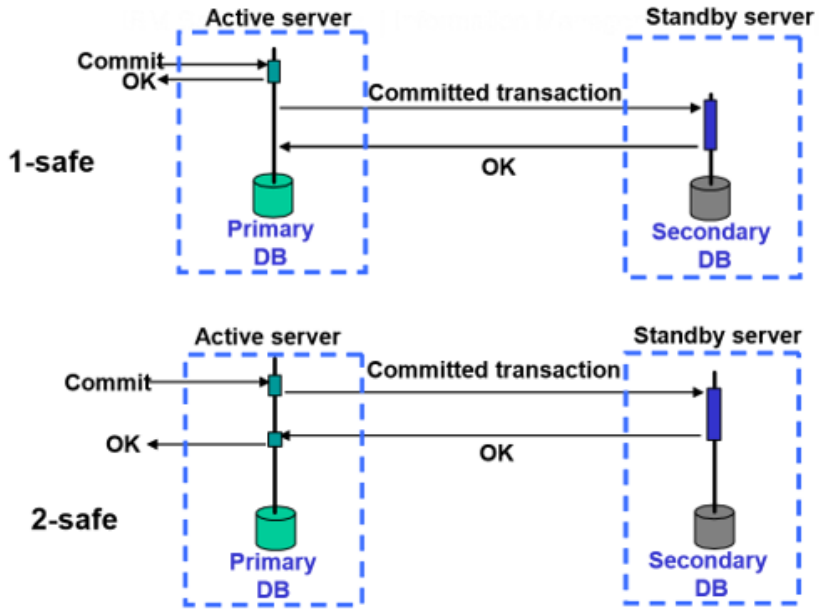


Figure 5 Illustration of the 1-safe and 2-safe replication protocol

The benefit of 1-safe replication is similar to that of relaxed durability: that is, the transaction response time is improved, and the throughput might be higher. On the other hand, with 2-safe replication, no committed transactions are lost upon failover. You might call this transaction characteristic standby-based strict durability, as opposed to log-based strict durability of a traditional DBMS. One immediate observation is that the log-based durability level has no effect on actual durability of transactions in the presence of failover. It is the standby-based durability that counts. The traditional log writing is relegated to the role of facilitating the database system recovery in the case of a total system failure. All other (more typical) failures are supposed to be taken care of by failovers. If a total system failure is unlikely (as builders of HA systems want to believe), a natural choice is to replace strict log-based durability with strict standby-based durability, which is the 2-safe protocol. Here, the gain is a faster log processing without really losing strict durability (if only single failures are considered).

Adaptive durability

To take the full advantage of the possibility to use the standby-based durability, the solidDB HA DBMS has an automated feature called *adaptive durability* (depicted in Figure 6). With adaptive durability, the Active server's log writing is automatically switched to strict durability if a node starts to operate without a standby.

Otherwise, the Active server operates with relaxed durability.

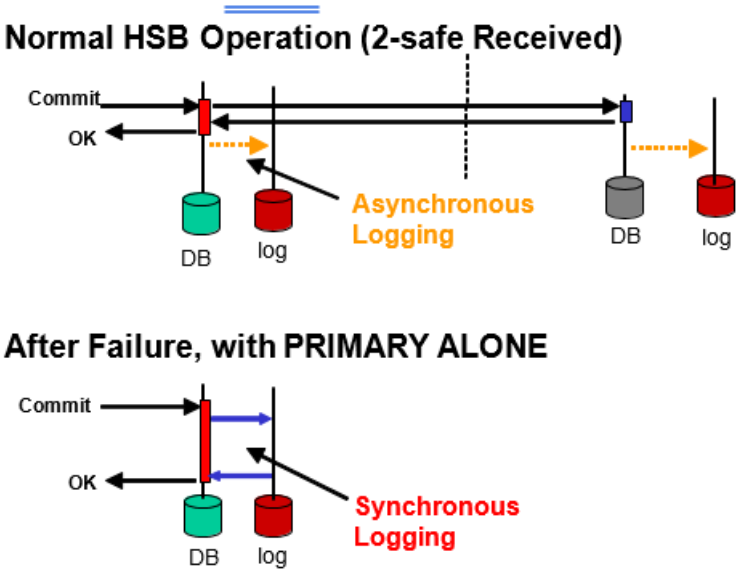


Figure 6 Adaptive durability

On low latency networks adaptive durability offers some performance benefits. It can be expected that network response is up to 2x faster than strict logging into a fast SSD disk.

Performance benefits depend on the disk speed and the network latency. The following graph demonstrates this.

With solidDB HSB running 2-safe protocol, the log-based durability was switched between strict and relaxed, the network latency was also compared with local switched LAN (0.15ms latency) and routed network (1.1ms latency). The results are shown in Figure 7, for two read/write mix ratios being 80/20 and 20/80. The results were obtained with the [Telecom Application Transaction Processing (TATP) Benchmark].

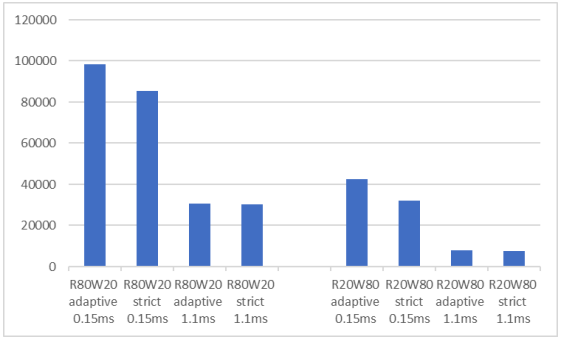


Figure 7 Impact of logging synchrony and latency on performance

Levels of 2-safe replication

In addition to the choice between 1-safe and 2-safe replication, 2-safe protocols can be implemented with various levels of involvement of the Secondary server in the processing of the commit message.

The following 2-safe policy levels are defined:

2-safe received: the Standby server sends the response immediately upon receipt.

2-safe visible: the Standby server processes the transaction to the point that the results are externally visible (in-memory commit).

2-safe durable: the Standby process processes the transaction to the point that it is written to a persistent log (strictly durable commit).

The three policy levels are illustrated in Figure 8.

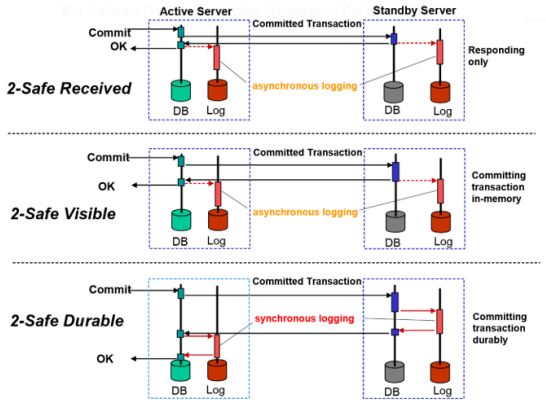


Figure 8 2-safe replication: policy levels

Of the three 2-safe policy levels, *2-safe received,* is intuitively the fastest and *2-safe durable* the most reliable. In a system with 2-safe durable replication, the database can survive a total system crash and, additionally, a media failure on one of the nodes. This comes, however, at a cost of multiple synchrony in the system.

The *2-safe visible* level is meant to increase the system utility by maintaining the same externally visible state at both the Active and Standby servers. Thus, if the transactions are run at both the Active and Standby servers (read-only transactions at Standby), they see the database states in the Primary DB and Secondary DB as mutually snapshot-consistent. The cost of maintaining this consistency level involves waiting for the transaction execution in the Standby server, before acknowledging the commit message.

To summarize, the intuitive rules for choosing the best trade-off between performance and reliability are as follows:

To protect against single failures, while allowing for some transactions to be lost on failover, use 1-safe replication with relaxed log-based durability.

To protect against single failures, with no transactions lost on failover, use 2-safe received replication with relaxed log-based durability.

To protect against single failures, with no transactions lost on failover and a possibility to use the Primary and Secondary databases concurrently, use 2-safe visible replication with relaxed log-based durability.

To protect against total system failure (in addition to single-point failures), use any 2-safe protocol and strict log-based durability in the Active server.

To protect against total system failure and a media failure, use 2-safe durable replication with strict log-based durability in both the Active and Standby servers.

Another worthwhile item to note is that a third dimension in assessing various replication protocols is the failover time. The further transactions are processed in the Standby server at the time of a failover,

the faster the failover. The protocols may be ordered by the failover time, from the shortest to the longest, in the following way: 2-safe durable, 2-safe visible, 2-safe received, and 1-safe.

In the performance testing experiments, we study the effect of all the parameters (we listed) on the system performance. We take advantage of the fact that solidDB HSB has all the necessary controls, both in the form of configuration parameters and dynamic administrative commands.

The performance results displaying the impact of the protocol synchrony on performance are shown in Figure 9.

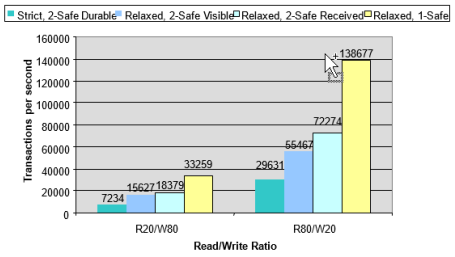


Figure 9 Impact of protocol synchrony on performance

You can see that the more asynchronous the protocol is, the more performance that can be delivered.

You may tune the system to reflect the needed trade-off. In most cases, the default replication and logging settings reflected with Adaptive Durability using the 2-safe received protocol is the best match.

Uninterruptable system maintenance and rolling upgrades

To run a database system for a longer period of time, you must be able to change the configuration, satisfy *ad hoc* monitoring needs, and perform software updates without shutting down the system. These possibilities are available in solidDB.

Dynamic reconfiguration

Configuration parameters are set in the configuration file, which is usually called `solid.ini`.

Many of the solidDB configuration parameters can be changed online and take effect immediately; these parameters are indicated in teh documentation as having *access mode* being *RW*, in the documentation. All crucial HSB configuration parameters are of this type.

Included is also a parameter telling what is the HSB connect string pointing to the mate node. By using the possibility to change that value, and the switchover capability, you can move the HSB processes, for example, to more powerful hardware, if needed. Assume that the current configuration includes the systems P1 and S1 serving as the Primary and Secondary nodes, respectively. Say, we have two other more powerful computers, P2 and S2 that we want to move the HSB operation to, in an uninterruptable way. The way to do that is as follows:

1. Disconnect the servers (P1 runs as Primary Alone).
2. Install solidDB on S2.
3. Move the Secondary working directories from S1 to S2 and update the configuration file.
4. Start solidDB at S2 (it starts as Secondary Alone).

5. Update the HSB connect string in P1 to point to S2.
6. Connect the servers (after the catchup is completed, P1 runs as Primary Active).
7. Switch the servers (S2 runs as Primary Active).
8. Disconnect the servers (S2 runs as Primary Alone).
9. Install solidDB on P2.
10. Move the solidDB working directories from P1 to P2.
11. Start solidDB at P2 (it starts as Secondary Alone).
12. Update the HSB connect string in S2 to point to P2.
13. Connect the servers (after the catchup is completed, S2 runs as Primary Active).
14. To arrive at a preferred configuration, switch the servers (P2 runs as Primary Active and S2 as Secondary Active).

These interactions with the solidDB servers are performed with admin commands. The sequence can also be automated with a script, or a program.

On-line monitoring

When a system runs for a longer time, non-anticipated monitoring needs may appear, having to do with performance tuning or problem tracking. In solidDB, various traces can be enabled and disabled dynamically, and one-time performance reports can be produced. For more information about performance monitoring, see the solidDB manual.

Uninterruptable software updates (rolling upgrades)

In a continuously running system, an obvious need to upgrade the software to a new version might appear. In solidDB, that is possible with the capability of *rolling upgrades*. Similar to the case of online hardware reconfiguration, advantage is taken of the fact that the Secondary server can be shut down temporarily. Additionally, an important feature used here is the possibility of the Secondary server to run a newer solidDB version than that of the Primary server.

Briefly, a rolling upgrade of an HSB system to a newer software version is performed with the following sequence of steps. Assume there are two nodes, A and B, running originally the Primary and Secondary database servers, respectively, at a version level V1:

1. Node A runs a Primary Active database, node B runs Secondary Active.
2. Disconnect the servers (A runs now as Primary Alone), shutdown solidDB on node B.
3. Upgrade solidDB to a new version V2 on B.
4. Start solidDB at B (it starts as Secondary Alone).
5. Reconnect the servers (after catchup, A runs as Primary Active).
6. Force a failover to B (by killing A, or with an admin command -- now B runs as Primary Alone).
7. Upgrade solidDB to version V2 on A.
8. Start solidDB at A (it starts as Secondary Alone).
9. Reconnect the servers (B runs as Primary Active).
10. To arrive at the original configuration, switch the servers (A runs as Primary Active and B as Secondary Active).

Upgrading the ODBC and JDBC drivers can be done at any point in time by deploying the new library / JDBC jar file and restarting the application instance. If the database server and the client side are upgraded at different times, the database server should be upgraded first.

Database schema changes can be done online also. These are (DDL) SQL statements, and should be executed against the Primary database. solidDB replicates these SQL statements to the Secondary database, keeping the databases in sync also for these changes. Use proper planning for online schema changes to avoid any problems to applications using the database at the same time. Also, mass data migrations need to be planned, for example to avoid the effects of adding a column and populating a new value to a large-sized table in one transaction, because it can cause unexpected load peaks.

HA management in solidDB HSB

The primary role of any HA architecture is to maintain system availability in the presence of failures. In a hot-standby system such as solidDB HSB, a failure of the Primary process or Primary node is followed by a failover to the Secondary server. In executing a failover, a governing principle is that no single solidDB server can make that decision on its own because a single server does not have sufficient information for doing that. If the failover was the responsibility of a server, then, in the case of network partitioning (HSB link failure), the two servers might decide that each becomes a Primary Alone (a split-brain scenario). Dual Primaries are not allowed because there are no consistency-preserving methods to merge two databases having different histories into one Primary database. For this reason, the responsibility to decide about a failover, and execute it, is deployed within a component (or components) outside of the HSB servers. The functionality in question is called *HA control*. In this section, we introduce three ways to implement HA control in solidDB HSB.

HA control with a third-party HA framework

If solidDB becomes a part of a broader HA system, capacity to execute HA control might already be in that system. The necessary functionality can be a part of a cluster management software or other generalized HA management software that can be abstracted as an *HA framework*. In that case, solidDB is integrated with the rest of the HA system, such as depicted in Figure 10.

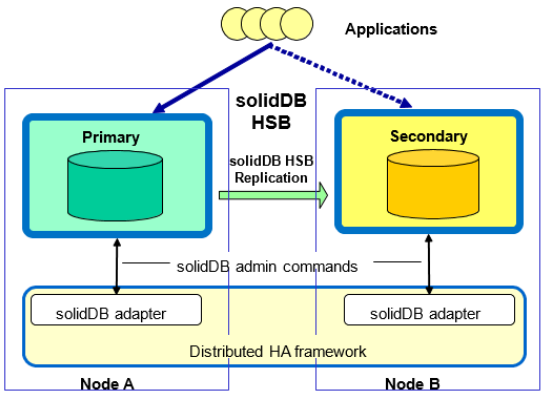


Figure 10 Using an external HA framework

An HA framework is typically a distributed software having an instance running on each node of the system. First, it is responsible for failure detection. Detecting hardware and operating system failures is unrelated to solidDB. Detecting solidDB process failures is implemented by a heart beat method, possibly based on polling. If there is a failover, the HA framework commands all relevant components with state change primitives. If there is an operator interface, the operator commands are passed to system components also. The reporting of the component state travels the other way around. The task of integrating solidDB into such a system involves developing a *solidDB adaptation* (of scripts or interface calls) that translates the primitives of the HA framework to solidDB commands.

HA control with the watchdog sample

The solidDB product package has a sample program called Watchdog (in the HSB sample set) that implements a rudimentary HA control. Watchdog is used in a configuration such as the one shown in Figure 11.

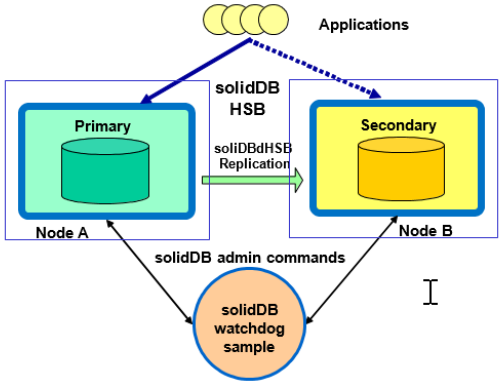


Figure 11 Using the Watchdog sample program

Watchdog includes both the failure detection and failover logic. The former is based on time-outs of active heart beat requests. If the Primary does not respond, it is considered to have failed. This method is prone to false failures or of having too long a failure detection time, depending on the timeout settings. Another deficiency is that the Watchdog instance is not backed up in any way. Thus, the sample is not meant for production use as such. However, it can be used as an area for experimenting with solidDB HA commands.

Using solidDB HA Controller (HAC)

A production-ready solution to solidDB-only HA control is called *HA Controller* (HAC). HAC binaries are distributed with the product. HAC is used in the form of two instances running on both the Primary and Secondary nodes, as shown in Figure 12.

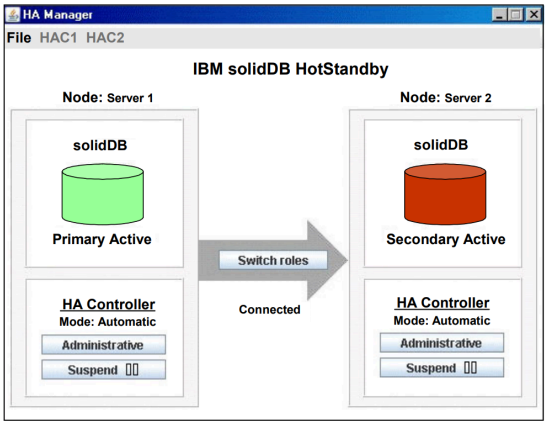


Figure 12 Using solidDB HA Controller

Failure detection in HAC is based on solidDB events. A surviving server notifies HAC of the failure of the mate node or server. The event-based method is fast and allows for sub-second failovers. The failover logic guarantees an unambiguous failover decision in all failure cases other than HSB link failures. For a remedy to deal with those, see the information about External Reference Entity, in "Preventing Dual Primaries and Split-Brain scenarios"

HAC can also be configured to start and restart solidDB servers automatically.

In connection with HAC, a basic GUI-based administrator tool is available, called HA Manager (HAM), as shown in Figure 13.

HAM is a Java-based, cross-platform tool where any number of instances can be active in a system. The GUI of HAM displays the HA state of the servers, the direction of replication, the state of the HSB link, and the state of HAC.

The *Automatic* mode of HAC means that both failure detection and failover execution is enabled. The mode can be changed to *Administrative* whereby the failover execution is disabled. Additionally, the execution of HAC can be suspended and resumed. The GUI controls included allow the administrator to perform server switchovers and change the HAC states.

The HAC binary is available in the HAC sample directory, in the product package.

Preventing Dual Primaries and Split-Brain scenarios

A HAC configuration shown in Figure 12 can be used only if the HSB link is assumed to be failure-free. That might well be the case in blade systems having backplane network wiring and redundant network interface controllers.

In all other cases, HAC should be configured to use a HSB link failure detection method called External Reference Entity (ERE). A configuration using ERE is shown in Figure 14.

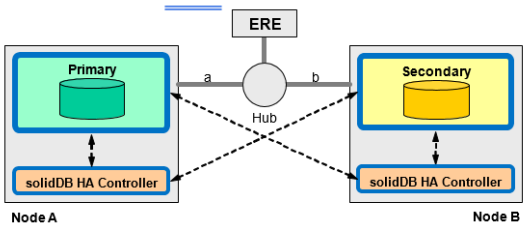


Figure 14 HA Controller used with the External Reference Entity

ERE represents a system component having a network address. It is assumed that a device suitable for ERE function already exists in the system. Such a device can be, for example, a network switch that is capable of responding to ping requests (most are). Another possibility is to use any other computer node available in the system, for that purpose. ERE does not require any additional software instance to be installed in the device. A capability to respond to standard ping requests is sufficient.

Using an ERE configuration with HAC prevents both nodes from becoming a Primary (Alone) database at the same time, preventing a split-brain scenario.

The ERE method works as follows:

The HSB link consists of two sublinks: a and b.

When a failure occurs in any component included in the HSB link, at most one of the HACs can access ERE. In that case the solidDB server of that node will become the Primary server. For example, if the sublink a, in Figure 14, fails, the Node B becomes the Primary node, and HAC will switch the database to Secondary Alone mode on the node that cannot ping the ERE.

To enable the ERE method, it is enough to enter the network address of ERE into the HAC configuration file, hac.ini.

Use of solidDB HSB in applications

This section describes how the applications use the underlying database pair. As examples, which database connects to, how the failures and failovers are seen by the application, and how to handle the failovers are typically in the error handling part of the application code.

The way the application should connect to the solidDB HotStandby database pair might vary depending on where the application resides (compared to the database it is using), and depending on the preferred level of automation in failover situations.

In the classic sense, when an application uses a database, it connects to one database, and uses that database for all the queries, reads, and writes. If that database becomes unavailable, the connection is broken and the application has to wait until it can reconnect again to the same database.

Now, with the solidDB HotStandby pair, there are two databases, both live and running, mirroring each other and having the same content. Therefore, the application has more options available, and more responsibilities of making sure that the service the application provides to its users is able to continue even if one of the database nodes fails.

Location of applications in the system

In terms of where the applications run, in relation to the nodes where the databases run, two basic topologies exist.

In the first one, the application runs on a node separate from the databases, as illustrated in Figure 15.

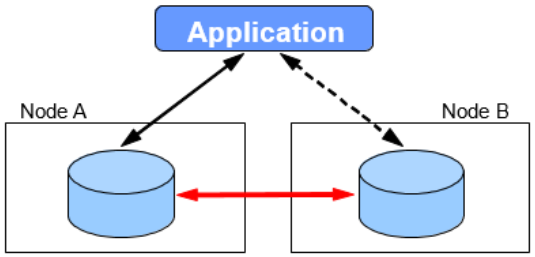


Figure 15 Application runs in a separate node from the databases

With this topology, the application normally continues to run regardless if one of the database nodes fails, and regardless of the reason for that failure (for example, the database, the underlying hardware or operating environment, or the network between the application node and the database node). The application may get an error message, and is unable to continue using the failed database. However, the application continues to run, and it can decide how to handle the error and what to do next. In the following sections, we describe in more detail how this failover handling can be resolved.

In the second topology, the application runs in the same node as the database, as illustrated in Figure 16.

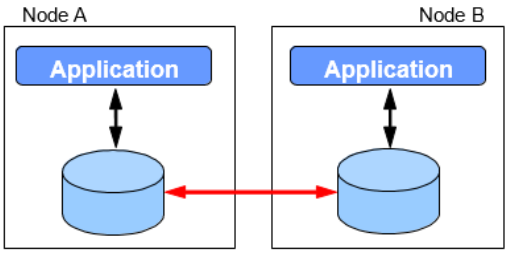


Figure 16 Application runs on the same node as the database

In this case, a node failure may, and most likely will, result in both a database and application failure, especially if the failure is because of a severe hardware problem. Therefore, the application must also be made redundant to ensure continuous application/service availability. Typically, with this topology, the entire node and all the related components and services are failed over to the standby/secondary node. If this is the case, the application may not need to be aware of the second database. It will always be using only one database, the local one.

More granular approaches can be used also, where each system component can have its own redundancy policies and individual components can be failed over separately, but they typically require more sophisticated high availability management and component monitoring.

As an example, illustrated in Figure 17, both applications can be active and use either database, or both at the same time, making full use of the available hardware and database resources. In the case

of a component failure, such as a database software failure, both applications can still continue and use the remaining database.

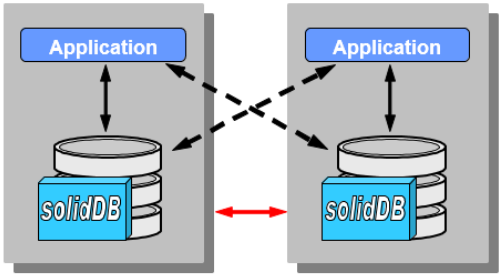


Figure 17 Two-node service cluster

Failover transparency

With topologies where the application should survive a database failure, the application can use the optional *transparent failover* functionality built into the solidDB JDBC and ODBC drivers.

When the built-in transparent failover mechanism is used, the application takes one logical connection to the database, but gives two (or more) connection strings (or URLs) to the solidDB driver. The solidDB JDBC/ODBC driver then chooses automatically the then-current HotStandby Primary database, and uses a physical connection to that database for query execution. The logic for the database selection is built into the driver.

If the then-current Primary database (or database node) fails, the physical connection is broken, and all pending query and transaction data of the last, ongoing transaction, is lost. To ease the recovery from the situation, and to enable the application to continue after the failure, the solidDB JDBC/ODBC drivers contain the following functionality:

1. The driver returns a 25216 native error code to the last SQL execution, to which the application should respond with a rollback request, as part of its error handling sequence.

2. As part of the rollback execution, the solidDB driver then finds, automatically, the new HotStandby Primary database.

3. After the rollback call returns, the application can continue to use the same logical connection handle (without a need to re-connect to the database); and internally the driver now uses a new physical connection to the new Primary database.

4. The application can ask the driver to additionally preserve several connection level attributes, such as the used database catalog, schema, and timeout settings, and also all the prepared statements. This way the application can continue (after the rollback) directly with a re-execute of the last transaction, without having to re-prepare all SQL statements and without having to set any specific connection attributes.

5. To speed up the failover handling, the solidDB JDBC/ODBC driver is also

(internally) listening to HotStandby system events from the then-current Secondary database. The built-in event listening mechanism notifies the driver right away about a database failure, making the reaction time shorter.

To activate the solidDB transparent failover functionality, the application must set the TF_LEVEL connection attribute (with ODBC) at the time, taking the database connection, or set a database connection property (with JDBC) called SOLID_TF_LEVEL.

The value NONE means no transparent failure handling by the JDBC/ODBC driver (default). The value CONNECTION automates the reconnection to the new Primary database, and the value SESSION automates the session (on top of the reconnect) and preserves the connection level attributes and the prepared statements.

If the transparent failover functionality is not used, the application's error handling code must implement many parts of the reconnection to the database, verifying its hotstandby state and re-preparing the SQL statements.

Each application connecting to the database can choose to use any of the available options.

For more information about the supported functions and examples of the connection strings and URLs, see the *solidDB High Availability User Guide*. The solidDB installation package contains sample C and Java program code for both ODBC and JDBC applications using the transparent failover handling.

Load balancing

In addition to the transparent failover built-in functionality, the solidDB JDBC/ODBC drivers also contain support for load balancing of the query execution.

The basis for the load balancing is the fact that there are two (HotStandby) databases running at the same time, and the two databases are in sync regarding the content. Therefore, any read query will provide the same result regardless of whether it is executed in the Primary or the Secondary database.

When the load balancing is activated, the JDBC/ODBC driver uses two physical connections, one to each database, and allocates the query load to the *workload connection*. The workload connection is selected based on query type (such as read or write), and the then-current load in the database servers.

Several main principles in the solidDB HotStandby load balancing implementation are as follows:

Read-only queries (at the read committed isolation level) can be executed in either database.

Read queries needing higher isolation level (repeatable read, select for update) are executed in the Primary database.

Write queries are executed in the Primary database.

Read queries after any write operation within the same transaction are executed in the Primary database (to ensure that updated rows are visible for subsequent reads).

Internal read/write level consistency of the databases is ensured so that after a write transaction is committed, the secondary database is not used for reading from the same connection until the

secondary database is up-to-date for that write. This way eliminates the possibility that if the *1-safe* or *2-safe received* HotStandby replication protocol is used, the next read transaction would not see committed data from the previous write transaction.

The selection of the workload connection is automated by solidDB, and the load balancing is automatic and transparent to the application.

As a result, especially read-centric applications can easily balance out the load between the two database servers, and use the full CPU capacity of both servers.

The load balancing is activated with ODBC driver by setting the

PREFERRED_ACCESS connection attribute to value READ_MOSTLY; or with

JDBC driver by setting the property called solid_preferred_access to value

READ_MOSTLY.

Each application connecting to the database can choose to use the load balancing functionality, or choose to use the Primary database for all queries (default).

The *solidDB: High Availability User Guide*, and the *solidDB: Programmer Guide* contain more details about the supported functionality, and also samples of the connection strings/URLs to use. The solidDB installation package contains sample C and Java program code for both ODBC and JDBC applications using the load balancing functionality.

Linked applications versus client/server applications

Most of the discussion regarding transparent failover and load balancing are related to *client/server* use of solidDB, that is, where the client applications connect to solidDB database server with a TCP/IP connection or similar socket based communication mechanism.

With the linked library mode, and with the shared memory mode, the application is more tightly and directly linked to one solidDB database instance. Although this may be enhanced in future solidDB releases, in the current release, the transparent failover and load balancing functionality is not yet supported for these applications.

Usage guidelines, use cases

solidDB HotStandby comes with configuration parameters preset to offer the best trade-off of performance, availability and data safeness, in most common cases. In this section, we describe those trade-offs and show how other choices can be made if needed.

Performance considerations

Several aspects of performance must be considered when looking at the best performance for a HA system.

In addition to the topics discussed in the following text, the main performance optimization effort should be, as always with relational databases, targeted to optimize the SQL query and schema design. Poor performance is always best corrected by looking at and enhancing the SQL queries, indexes, and other basic RDBMS performance elements.

With a HA (redundant) system, the effect of mirroring/replicating the changed data, reliably and consistently, to another node plays a role also. Consider the following performance elements:

Latency or response times: How quickly a single read or a write operation is completed?

Throughput: How much total query or transaction volume the two-node system can handle?

Data safeness: Does the system guarantee that every transaction is safely persisted, on the same node (to disk) or to the next node (over the network)?

Failover times: How quickly a loaded system can continue to provide its service after a single-node failure, including the error detection time?

Recovery times: How quickly (and how automatically) a system recovers to an HA state after the failure has been resolved?

Optimizing one of the areas might be easy with the configuration parameters available, but the need is often to find the best balance of all factors.

Behavior of reads and writes in a HA setup

Read queries execute only in one database node (databases are identical, having the same content for HA reasons), so there is no need to involve both databases for executing a single read query. Therefore, the response times are expected to be the same in a single db mode and in a HA setup.

Write operations are applied to both databases (insert, update, and delete, and schema changes). Therefore, effects on the latency exist that also depend heavily on the *safeness* level used. With safeness, we mean whether the write operation is safely persisted at the time of commitment, either to the persistent media (disk) or to the second database node over the network. This approach is to guarantee that a committed transaction is not lost in case of a failure.

The safeness level can be set with two controls, independently configured. The configuration can further be done at a system level (global), a connection level and a transaction level. Given this flexibility, there are plenty of options to optimize the trade-off of latency versus data safeness for each part of the application using the database.

The two controls are as follows:

Durability level, which can be strict or relaxed, determines whether the log writing is synchronous or asynchronous, respectively.

Hot standby replication protocol safeness level, that can be 1-safe or 2-safe. When using 1-safe, the replication is asynchronous, and with 2-safe it is synchronous. The 2-safe level has also more granular options (received, visible, durable).

The response time, throughput, and data safeness of write operations are interrelated in terms of how the used configuration affects performance, and hence they are discussed in a combined fashion in the following sections.

Using asynchronous configurations with HA

As might be obvious, asynchrony leads to better performance overall, with the risk of losing data (a few of the latest transactions) in case of failures.

The most asynchronous approach is by using relaxed log writing in each database node (or no transaction logging at all) and 1-safe hot standby replication protocol, which also leads to shortest response times because the Primary database does not have to wait for local disk I/O or the Secondary database before it can complete the transaction commit. It also leads to best throughput, because transactions can be sent to the Secondary in groups, making the overall throughput higher.

Applications needing the fastest possible performance and tolerating the risk of losing some of the latest transactions with a failure situation should consider this configuration.

The risk of losing transactions can be reduced (but not totally eliminated) by shortening the maximum delays of the log writes or hot standby replication. The configuration parameters are called Logging.RelaxedMaxDelay and HotStandby.1SafeMaxDelay, respectively. Setting these parameters to a lower value than the default forces solidDB to persist or replicate the transactions earlier.

We must also mention one special feature here. Because the solidDB HotStandby technology is targeted for HA purposes, solidDB must guarantee, even with the asynchronous replication mode, that the Secondary database is not *too far behind* in executing transactions received from the source of the replication. This way can ensure a reasonable and up-to-date database state for the Secondary database at all times. Therefore, there is an internal mechanism in the Primary database to start *throttling* the write operations if the defined limits of staying behind are reached. As a result, the throughput in the Primary can be limited by the throughput in the Secondary.

Using default solidDB HA setup

The default solidDB HotStandby configuration is using Adaptive Durability. In a normal state, that translates to relaxed (asynchronous) log writing and synchronous (2-safe received) replication. This way ensures that all committed transactions are always replicated synchronously to the Secondary database at the time of the commit, and thus located at least in the memory of both database nodes. Compared to the fully asynchronous mode, the latency is longer for each write commit, because the Primary database has to wait for the Secondary database to at least acknowledge receiving the changes.

With this configuration, no data is lost in case of a single-node failure. In case of two-node failure, the risk is still there. This configuration is considered the best of both worlds because it eliminates the typical biggest performance problem, which is the (synchronous) disk I/O, but provides data safeness against any single node failures. Given normal maturity of proper server operating environments, a single node failure is rare, and the probability of a two-node failure at the same time, while always there, is even less likely to happen.

In answer to the always-asked question "*How much performance overhead does this synchronous replication cause?"* the answer is "it depends".

With modern NVME ssd drives the local synchronous writing might be faster than sending and receiving responses from the Secondary database over network. However the network performance heavily affects the comparison. With modern low latency networks the network I/O is likely faster than even standard NVME devices.

Before deciding, experiment with your environment.

The solidDB HA setup for best data safety

If your application needs maximum safety for the write transactions, consider using the 2-safe durable solidDB HotStandby protocol configuration.

This configuration uses synchronous replication and synchronous log writing in both database nodes. No data is lost even if both nodes fail at the same time, because every transaction is committed all the way to the disk in both nodes. The performance is approximately the same as with a single (stand-alone) server that uses synchronous (strict) logging.

Failover time considerations

In all configurations the error detection time is typically the same (a health check timeout, or similar). After the error has been detected and recognized as an error (with possible retry operations), the remaining task is to switch the other database server to be the (new) primary database. If the original Primary fails, the Secondary can be switched to the Primary role after it has executed all received transactions. With *2-safe visible* and *2-safe durable,* the transactions have been executed already and the failover is practically immediate. With 1-safe and 2-safe received, there is a queue of transactions waiting to be applied to the Secondary database, and there is a small delay because of this. Because the queue on the secondary side is not very large, this delay is in most cases short.

During the downtime of one database server, the system continues to operate with the remaining database. The performance (of write transactions) is based on the configuration for that server, mostly importantly dependent on the log writing mode. If the Logging.DurabilityLevel is *adaptive*, the transaction log writes change from asynchronous to synchronous, ensuring the data safeness for committed transactions but most likely affects the speed also. If the requirement is rather to maintain the speed, even with the risk of losing few transactions, then keeping the logging as relaxed also in the single node state (PRIMARY ALONE) should be considered.

Recovery time considerations

The recovery time depends on the following factors:

- how many changes occurred during the downtime (the delta)
- how long it takes to restart the database
- the database checkpoint interval: how far back the database has to go to find a proper sync point, from which the delta needs to be applied between the databases (the catchup phase). To minimize the recovery time, make relatively frequent checkpoints.

However, if the database is small and the entire content changes quickly because of high speed writes, it might be faster to copy the entire database over (by using `hsb netcopy`) instead of recovering all transactions that took place during the downtime.

Given normal operating environment expectations, the usual recovery means a database restart and a catchup (apply the delta). In addition, the HA management solution should always be prepared to also execute the full sync, that is, copying the whole database from Primary database to Secondary database, to re-initialize it. In certain situations, catchup can fail or is not possible; you must consider these situations in the automated recovery implementations.

Example situation

The following example shows a setup for solidDB HotStandby configuration and manual SQL admin commands to set the database in the mode where hot standby replication is automatic and continuous.

This example assumes two computers, nodeA and nodeB, and any operating system. The configurations and commands are similar on all of them:

NodeA solid.ini configuration

```
[Com]
Listen = tcp 1315
[HotStandby]
HSBEnabled = yes
Connect = tcp nodeB 1315
```

NodeB solid.ini configuration:

```
[Com]
Listen = tcp 1315
[HotStandby]
HSBEnabled = yes
Connect = tcp nodeA 1315
```

Start solidDB on nodeA:

```
>solid -Udba -Pdba -Cdba
```

Start solidDB on nodeB:

```
>solid -Udba -Pdba -Cdba
```

Manual SQL commands to connect databases

The following SQL statements are given with the solsql utility (in the solidDB bin directory). Because these are part of solidDB SQL syntax, they can also be given from any tool or application.

The following lines or commands are given in solsql in nodeA. That is, connect first to nodeA database, with operating system terminal or command prompt:

>solsql “tcp nodeA 1315” dba dba

The lines starting with two hyphens (--) are comments that explain what is done:

-- switch node A to primary mode admin command ‘hsb set primary alone’;

-- initialize the secondary db based on primary db content admin command ‘hsb netcopy’;

-- wait until netcopy is finished, repeat until ‘success’ admin command ‘hsb status copy’; -- connect, i.e. start active replication admin command ‘hsb connect’;

-- check that this succeeded

-- you should now have ‘primary active’ state admin command ‘hsb state’; -- complete hsb actions commit work;
-- exit from solsql exit;

Summary

You now have an active solidDB HotStandby setup. Any writes (that are committed) to the primary database will be replicated automatically to the secondary database, including schema changes. Create a table, insert rows and commit them, and then connect to the secondary database and read the data from there.

In the initial situation (before executing **hsb set primary alone**) both databases have a secondary role, and any write attempts to either database fail. This is because of the following solid.ini configuration parameter:

[Hotstandby]
HSBEnabled=yes

When this parameter is set, the database has to be made a primary database before any write operations.

Application failover

Application failover is the act of moving the service offered by an application from a failed application instance to another one. It is a hot standby scheme applied to applications. The need for it depends on the overall configuration. If an application runs in a configuration such as the one shown in Figure 15, a failure of the database server does not imply the application failover. Thanks to the transparent failover (TF) capability in solidDB drivers, the application can continue over the server failover. However, that application itself may fail, and that might require a failover to the standby instance of the application.

If the application is located with the server as shown in Figure 16, the application can fail at the same time as the database server, say, in the case of a computer node crash. In that case, we might have to perform both the server failover and the application failover. In general, the application failover is required in the following cases:

- A failure of an application instance on a remote node (if a restart, on the same node or another one, is not sufficient).
- A failure of a collocated application instance, together with the database server (if restart of the application on the new primary node is not sufficient).

Application state preserving

When the service is moved from one application instance to another, the service state must be preserved to the extent required by the service continuity. With non-database applications, that requires complex application checkpointing protocols or other solutions to move the state between the

instances. A database, and especially an HA database, offers a perfect opportunity to transfer the service state through the database. What is needed is only that the application is designed to be stateless, because all the relevant state is stored in the database. Especially, when using an HA database as solidDB HSB, no application state will be lost in any single failure in the system. The applications, whether restarted or failed-over, can recover the service state from the database at any time.

Controlling application failover

Application failover requires the same type of HA control that an HA database requires. For that purpose, the applications can be integrated with an external high availability framework, or they can use the HA control that is already available with the database. In the case of a collocated application, the standby instance of the application can be connected to the Secondary server and it can monitor the HA state of the Secondary server. That can be done either by polling (checking periodically the HA state) or by subscribing to a system event that would indicate the state change. When the server role changes from secondary to primary, the application reacts by changing its state from standby to active, effectively performing the application failover. Switchover, for example, the intentional role switching between the solidDB servers can be taken care of in the same way.

More information

solidDB website

https://soliddb.com

solidDB 201.0 manuals

https://support.unicomsi.com/manuals/soliddb/201

solidDB sales

soliddb.sales@unicomglobal.com

solidDB support

soliddb.support@unicomglobal.com